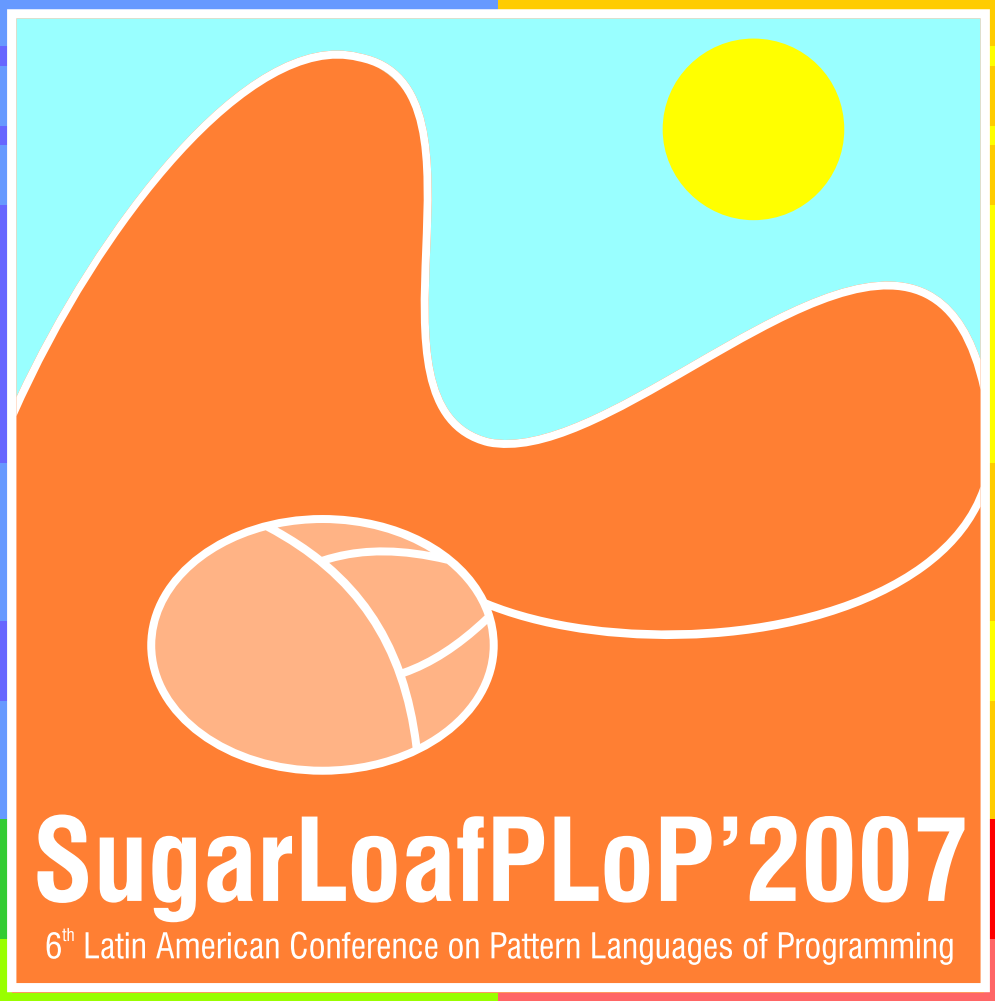
SugarLoafPLoP'2007 Proceedings



SugarLoafPLoP'2007

6th Latin American Conference on Pattern Languages of Programming

May 27-30, 2007 - Porto de Galinhas, Pernambuco, Brazil
sugarloafplop.dsc.upe.br

Editors
Sérgio Soares
Jerffeson Teixeira de Souza (UECE, Brasil)
Richard P. Gabriel (Sun Microsystems, Inc.)

Organized by
DSC - UPE
UECE

Sponsoref by the Hillside Group
Supported by SBC (Brazilian Computer Society)

Sponsors

Supporting Organizations

Organization

SugarLoafPLoP´2007 Organizing Committee

Conference Chair

Sérgio Soares (DSC/UPE, Brazil)

Program Committee Co-Chairs

Jerffeson Teixeira de Souza (UECE, Brazil)
Richard P. Gabriel (Sun Microsystems Inc., USA)

Program Committee

Claudia Werner (COPPE/UFRJ, Brazil)
Eugene Wallingford (U. Northern Iowa, USA)
Fabio Kon (IME/USP, Brazil)
Jerffeson Teixeira de Souza (UECE, Brazil)
Jorge L. Ortega Arjona (UNAM, Mexico)
Joseph Yoder (U. Illinois / The Refactory, Inc, USA)
Linda Rising (Independent Consultant, USA)
Lise Hvatum (Schlumberger, USA)
Marcos Cordeiro d'Ornellas (UFSM, Brazil)
Neil Harrison (Utah Valley State College, USA)
Paulo Borba (CIn/UFPE, Brazil)
Paulo Cesar Masiero (ICMC/USP, Brazil)
Richard P. Gabriel (Sun Microsystems Inc., USA)
Robert Hanmer (Lucent Technologies, USA)
Rosana Braga (ICMC/USP, Brazil)
Rossana Andrade (DC/UFC, Brazil)
Sérgio Soares (DSC/UPE, Brazil)

Local Organization

Emanoel Francisco Spósito Barreiros (DSC-UPE, Brazil)
Liliane Sheyla da Silva (DSC-UPE, Brazil)
Márcio Lopes Cornélio (DSC-UPE, Brazil)
Ricardo Massa Ferreira Lima (DSC-UPE, Brazil)
Sérgio Castelo Branco Soares (DSC-UPE, Brazil)
Thaysa Suely Beltrão Paiva (DSC-UPE, Brazil)
Maria Lencastre (DSC-UPE, Brazil)
Tiago Massoni (DSC-UPE, Brazil)

SugarLoafPLoP´2007 Organizing Committee

Shepherds

Alexandre Sztajnberg (UERJ, Brazil)
Ed Fernandez (CSE/FAU, USA)
Eugene Wallingford (U. Northern Iowa, USA)
Francisco José da Silva e Silva (DEINF/UFMA, Brazil)
Jerffeson Teixeira de Souza (UECE, Brazil)
Jorge L. Ortega Arjona (UNAM, Mexico)
Joseph Yoder (U. Illinois / The Refactory, Inc, USA)
Lincoln S. Rocha (DC/UFC, Brazil)
Linda Rising (Independent Consultant, USA)
Lise Hvatum (Schlumberger, USA)
Marcio Barros (UNIRIOTEC, Brazil)
Marcos Cordeiro d'Ornellas (UFSM, Brazil)
Maria Lencastre (DSC/UPE, Brazil)
Neil Harrison (Utah Valley State College, USA)
Paulo Borba (CIn/UFPE, Brazil)
Paulo Cesar Masiero (ICMC/USP, Brazil)
Robert Hanmer (Lucent Technologies, USA)
Rohit Gheyi (CIn/UFPE, Brazil)
Rosana Braga (ICMC/USP, Brazil)
Rossana Andrade (DC/UFC, Brazil)
Rute Castro (DC/UFC, Brazil)
Sérgio Soares (DSC/UPE, Brazil)
Tiago Massoni (CIn/UFPE, Brazil)

Table of Contents

I – Writers' Workshop

II – Pattern Applications

Foreword

Once again, pattern community members have got together to discuss and share pattern experiences. This year, as in 2003, the stage was the beautiful Porto de Galinhas in Pernambuco, Brazil. During unforgettable four days, participants had the chance to learn and teach patterns, and about them.

In this SugarLoafPLoP´2007 edition, conference participants had the chance to hear from several pattern experts in tutorials and invited talks. On the first morning of the event, we had an inspiring 4-hour lesson on how to write patterns, lead by Joe Yoder. With his undeniable experience as a pattern writer, Joe showed us "The Straight Scoop" on writing good patterns. During the evening of that day, Gibeon Aquino entertained us and taught us about patterns and software metrics. In the next evening, we heard from Richard Gabriel about Ultra-Large-Scale Systems in the tutorial "Design Beyond Human Abilities". Finally, in the final morning of the event, Rosana Braga presented and discussed OO Analysis and Design Patterns.

This year, we had a record number of participants and submissions. In total, 46 pattern enthusiastic old and new members of our pattern community have attended SugarLoafPLoP´2007. For the number of submitted paper, we had 38 of them, where 19 were sent to the Writers´ Workshop track, 13 to Pattern Applications and 6 to the Writing Patterns track.

In these proceedings, we share with the world a little of our SugarLoafPLoP´2007 experience. Here, you will find 14 papers describing new patterns (the ones discussed during the Writers´ Workshop sessions) and 6 discussing Pattern Applications. As in previous years, the papers dealt with a great variety of topics, including: Aspect and OO-based Software Development; Requirement, Analysis, Design and Architectural Patterns; Organizational Patterns; Educational Patterns; Patterns for Mobile Development; Documentation Patterns; Patterns for Software Testing and Quality Assurance; and more.

Several persons deserve our acknowledgment for making SugarLoafPLoP´2007 such an enjoyable conference. Among them, we emphasize Sérgio Soares, the conference Chair. He made it seem really easy to organize an event of this magnitude. Thanks Sérgio for the flawless organization.

That is it !! Another SugarLoafPLoP has passed. But don´t be sad, others will came. 8-)

Thanks, Gracias, Obrigado!

Jerffeson Teixeira de Souza and Richard P. Gabriel
SugarLoafPLoP´2007 Program Committee Chairs

SugarLoafPLoP´2007

Writers' Workshop

Padrões para apoio ao desenvolvimento de Políticas de Privacidade

Luanna Lopes Lobato, Sérgio Donizetti Zorzo

Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – CEP: 13565-905 – São Carlos – SP– Brasil

{luanna_lobato,zorzo}@dc.ufscar.br

Abstract. *This paper presents patterns for privacy policies to be used in web sites, mostly by e-commerce and e-business sites. In those transactions, because of their financial aspects, the users need to provide personal information, and expect integrity, security, and privacy. The patterns are derived from a study of the 33 most accessed e-commerce sites in Brazil, where it was possible to observe that they do not use a systematic approach to develop policies which are clear and friendly and with relevant contents.*

Resumo. *Este artigo apresenta uma proposta de padronização para as Políticas de Privacidade utilizadas pelos sites, principalmente pelos sites de e-commerce e e-business. Nessas transações os usuários disponibilizam suas informações pessoais, desejando-se a sua integridade, segurança e privacidade, pois há valores financeiros em compras realizadas na Internet. Propõe-se essa padronização a partir da realização de um estudo de caso, onde foram analisados 33 dos sites mais acessados pelos usuários no comércio eletrônico brasileiro, em que foi possível observar que não são utilizados parâmetros para o desenvolvimento de políticas claras, amigáveis e de conteúdos relevantes.*

1. Introdução

A partir do Estudo de Caso (Lobato e Zorzo, 2007) de avaliação por inspeção em 33 sites de comércio eletrônico brasileiro mais acessados, de acordo com uma pesquisa divulgada pela Info Exame[1] e e-bit[2], observou-se que a maioria desses não utiliza uma Política de Privacidade com regras e deveres claros aos usuários, e, quando as fazem, não seguem uma padronização.

O Estudo de Caso analisa, por inspeção manual, se alguns itens considerados relevantes, eram apresentados pelos sites, buscando equacionar as características de privacidade e personalização contempladas. Para cada site avaliado foi registrado os itens contemplados, acrescidos de observações particularizadas e sumarizadas em uma tabela. Ao final de toda a análise foi sumarizado o que os sites apresentavam como vantagens e desvantagens aos usuários, mostrando a porcentagem dos itens contemplados, não contemplados e os itens que não puderam ser aplicáveis nos sites por motivos de verificação da inspeção manual.

[1] http://www.infoexame.com

[2] http://www.e-bit.com

Esses itens são as características relevantes que devem ser contempladas nas Políticas de Privacidade, de forma que os usuários se sintam esclarecidos quanto ao que os sites disponibilizam como benefícios e problemas em sua utilização.

As políticas foram um dos tópicos de maior dificuldade de análise durante a realização do Estudo de Caso, pois essas são apresentadas das mais diferentes formas. No entanto, essas são, em muitas das vezes, as mais relevantes para obtenção de informações referentes à prática seguida pelos sites (LOBATO e ZORZO, 2007).

Uma pesquisa publicada por Turow (2003) mostra alguns dados sobre a relação dos usuários com as Políticas de Privacidade, ressaltando a insatisfação, a falta de compreensão e a necessidade de informação dos mesmos quanto às políticas disponibilizadas.

Já um trabalho realizado por uma equipe de pesquisadores da *North Carolina State University* identificou que dentre 40 Políticas de Privacidade examinadas, 12 requeriam um nível de escolaridade superior para seu entendimento e 7 requeriam o equivalente ao nível de pós-graduação (ANTON *et al*., 2004).

Com base nesses estudos, ressalta-se que tais políticas devem informar aos usuários sobre o que é feito para garantia da privacidade dos mesmos e quais métodos são utilizados para prover personalização, bem como, tratar dos assuntos referentes a manipulação dos dados coletados, utilização de entidades certificadoras, armazenamento de informações na máquina do usuário, dentre outras questões que abordam a privacidade, segurança e personalização.

Preocupados com isso, propõe-se neste artigo uma padronização para as Políticas de Privacidade a serem disponibilizadas pelos sites, de modo a tentar aproximá-las ao entendimento do usuário e englobar todos os pontos interessantes a serem ressaltados em uma política escrita de maneira objetiva e clara.

Para a padronização foram utilizados padrões (*patterns*), que, de acordo com Borches (2001), podem ser entendidos como uma forma de expressar conhecimento por meio de textos e esboços em um formato estruturado, cuja solução é de sucesso já que os mesmos podem ser utilizados e aplicados a outros problemas, os quais ocorrem frequentemente em um determinado contexto. Alexander, Ishikawa e Silverstein (1977) mencionam que padrão é uma solução de sucesso para um problema recorrente em um determinado contexto. Já Gamma *et al*. (1995) diz que os padrões de projeto capturam soluções que foram desenvolvidas e evoluídas ao longo do tempo. Coplien e Harrison (2004) apresentam o padrão como uma configuração estrutural recorrente que resolve um problema em um determinado contexto.

Os padrões são utilizados em várias abordagens e definidos por diferentes autores em suas respectivas áreas de atuação, no entanto todas as definições mostram um principal objetivo para os padrões: o seu reuso. Os padrões de Política de Privacidade definidos neste artigo abordam principalmente assuntos referentes à segurança, privacidade e coleta de dados dos usuários.

A privacidade pode ser entendida como a habilidade de um indivíduo ou grupo manter suas informações pessoais longe do conhecimento público ou como a capacidade de controlar o fluxo de informações que pode ser revelada (HAFIZ, 2006).

Seja no mundo eletrônico quanto no mundo real, a privacidade é algo que se almeja, de forma que as ações possam ser efetivadas sem que alguém esteja

monitorando-as. Os indivíduos devem poder viver sem serem perturbados e os usuários em interação com a web navegar sem serem identificados.

A privacidade pessoal on-line tem se tornado uma preocupação crescente durante a navegação na web. Organizações comerciais e governamentais estão sendo convocadas a implementar controles de segurança e políticas que dêem mais segurança ao usuário quanto à sua privacidade (ROMANOSKY *et al*., 2006).

A medida que os usuários utilizam serviços na rede, deixam rastros que podem ser utilizados pelas empresas que dispõem de tecnologias suficientes para registrar as páginas visitadas, bem como o que foi feito em cada uma durante a visita, criando-se perfis de usuários (LOBATO e ZORZO, 2006). Assim, da próxima vez que o usuário visitar o site, serão apresentados promoções e recomendações de acordo com o seu perfil.

Na web, o fato de muitas pessoas não saberem ao certo para que e o quanto de seus dados são coletados representa um grande risco à privacidade dos usuários que utilizam seus serviços (SPIEKERMANN, GROSSKLAGS e BERENDT, 2001).

Assim, são criadas e descritas Políticas de Privacidade, onde são dadas informações relevantes aos usuários. Uma das preocupações sobre privacidade é o nível de consciência do usuário, de forma que a política do site deve ser criada de maneira objetiva e bem definida, trazendo esclarecimento aos usuários e tornando-os conscientes sobre os problemas providos da navegação na web.

Para definição dos padrões, utilizados para embasar o desenvolvimento das Políticas de Privacidade disponibilizadas pelos sites, seguiu-se alguns princípios apresentados por Sadicoff, Larrondo-Petrie e Fernandez (2005). De acordo com esses autores, existem algumas forças que podem ser utilizadas de modo a tornar os usuários conscientes sobre as políticas seguidas pelos sites para a coleta e utilização de seus dados, antes dos usuários divulgarem suas informações pessoais, sendo elas:

- As Políticas de Privacidade devem ser exibidas aos usuários de maneira que sejam claramente entendidas;
- Os usuários devem ser capazes de decidir quais de suas informações poderão ser coletadas e utilizadas pelos sites;
- Pode haver modificações nas Políticas de Privacidade, e dessa forma, os usuários devem ser capazes de visualizá-las;

A seguir são apresentadas as diretivas para embasamento e os padrões de Política de Privacidade propostos, seguindo o modelo de estruturação para definição dos Padrões e conceitos de Meszaros e Doble (1996), Gamma *et al*. (1995) e Buschmann *et a*l. (1996). São também mostrados os Padrões desenvolvidos, bem como a definição de cada um deles e ao final, uma aplicação prática desses em uma Política de Privacidade tomada como exemplo.

2. Diretivas para Embasamento aos Padrões

Além da utilização das características observadas no Estudo de Caso apresentado por Lobato e Zorzo (2007) para a definição e validação da relevância dos padrões, também foram considerados alguns princípios impostos por duas organizações. Esses estudos foram seguidos de forma a definir um escopo de uma solução de sucesso que deva ser seguido para a criação das Políticas de Privacidade, facilitando o reuso para os demais projetistas e fácil entendimento aos usuários.

Existem duas organizações que destacam-se no cenário internacional, com objetivo de regularizar a proteção de privacidade dos usuários da web: *Organization for Economic Co-operation and Development*[3] (OECD) e *Federal Trade Commission*[4] (FTC), descritas a seguir.

A OECD trata da proteção de privacidade dos usuários, disponibilizando e retratando documentos específicos para sua segurança e a privacidade. Os princípios estabelecidos pela OECD especificam de que forma as informações pessoais dos usuários devem ser protegidas, sendo alguns desses apresentados a seguir:

- Princípio do Limite de Coleta: a coleta de dados pessoais deve ser limitada, e quando essa ocorrer, deve ser feita através de meios legais;
- Princípio da Qualidade dos Dados: os dados pessoais devem ser autênticos, completos e relevantes para os objetivos onde serão utilizados;
- Princípio da Especificação de Objetivo: o objetivo da coleta deve ser especificado antes da efetivação da ação e o uso dos dados devem ser restritos aos objetivos impostos e declarados nas políticas;
- Princípio da Limitação de Uso: os dados coletados não podem ser divulgados ou utilizados para outros propósitos além dos especificados, exceto por uma autoridade da lei ou com o consentimento do proprietário dos dados;
- Princípio da Segurança: devem ser utilizados mecanismos de segurança razoáveis para garantir a segurança dos dados;
- Princípio da Transparência: deve ser criada uma política geral que trate da divulgação sobre as práticas e políticas com respeito a dados pessoais;
- Princípio da Participação Individual: o dono dos dados deve ter acesso a seus dados, pesquisando, visualizando e modificando-os caso julgue necessário;
- Princípio da Responsabilidade: um gerenciador deve ser responsável por cumprir, colocando em prática todos os itens acima.

A FTC é uma instituição que tem por objetivo cuidar da privacidade e da vida econômica dos cidadãos, auxiliando no reforço de leis a favor da segurança dos dados pessoais, vasculhando criminosos de forma a evitar fraudes em bancos e também possibilitando aos consumidores tomarem decisões de compras, possibilitando assim que estejam esses melhores informados. Sob o ato da FTC, a comissão zela contra a deslealdade e a decepção por reforçar promessas de privacidade de companhias sobre como elas coletam, usam e asseguram informações pessoais dos consumidores (PITOFSKY *et al*., 2000).

Pela FTC são definidos alguns princípios de Práticas Justas de Privacidade, baseados e desenvolvidos sob uma legislação para as práticas recomendadas de privacidade que protegem as informações pessoais de serem coletadas e mantidas pelo

[3] http://www.oecd.org

[4] http://www.ftc.gov

governo (PITOFSKY *et al*., 2000). Esses princípios sintetizam os 8 princípios apresentados pela OECD, e incluem:

- Notificação: os sites devem manter os usuários informados sobre a coleta de seus dados;
- Escolha: devem ser fornecidas aos usuários opções para escolher como seus dados pessoais podem ser utilizados;
- Acesso: os usuários devem ter acessos às suas informações pessoais coletadas, podendo atualizá-las, corrigir e apagar caso seja necessário;
- Segurança: os sites devem ser responsáveis e proteger com segurança as informações coletadas sobre os usuários.

É possível observar que as propostas da OECD e da FTC se baseiam na idéia de que a privacidade está relacionada ao consentimento dos usuários, sobre o que está sendo feito com seus dados, e ambas visam trazer mais segurança sobre as formas de uso de dados: coleta, processamento, manutenção, responsabilidade, divulgação e controle.

A seguir são apresentados os padrões para Políticas de Privacidade definidos, seguidos dos objetivos de seu desenvolvimento.

3. Coleção de Padrões Definidos

À medida que cresce o uso da tecnologia também aumenta a preocupação em relação às novas formas de comércio eletrônico e à comunicação eletrônica, por isso, é preciso construir proteções adequadas que assegurem interações confiáveis aos usuários.

Para isso, foram definidos alguns itens que as políticas devem apresentar, chamados de padrões, de forma a aumentar a segurança oferecida ao usuário e o conforto na utilização dos sites.

Nessa seção são apresentados os padrões definidos para Política de Privacidade, sendo descritos o porquê desses, os problemas observados e a motivação encontrada para sua definição.

O formato e estilo de escrita dos padrões foram baseados na “Linguagem de Padrões para escrita de Padrões” de Meszaros e Doble (1996), onde é especificado que os padrões são mais fáceis de compreender e aplicar quando alguns elementos estão presentes no formato utilizado, como:

- **Nome (numeração)**: permite uma referência rápida e comunica a idéia principal do padrão. Pode-se utilizar uma numeração para facilitar a localização do padrão;
- **Contexto**: descreve o problema encontrado para se ter a necessidade de padronização e a solução implantada;
- **Problema**: apresenta a problemática a qual o padrão se aplica;
- **Forças**: informa os aspectos que influenciam a utilização do padrão;
- **Solução**: apresenta a mensagem para a solução do problema;
- **Conseqüências**: aborda os resultados decorrentes da aplicação da solução;

- **Usos Conhecidos**: mostra exemplos bem reconhecidos da aplicação prática do padrão.

Neste trabalho, além dos elementos base para a definição dos padrões, também utilizou-se o elemento Padrões Relacionados, o qual foi considerado importante para o entendimento dos padrões formalizados. Esses devem ser nomes de outros padrões que tenham alguma relação de contexto com os padrões propostos.

Com a utilização desses padrões a satisfação dos usuários tende a ser maior, já que terão informações bem definidas e claras nas Políticas de Privacidade sobre os serviços oferecidos pelo site e sua segurança.

Os padrões definidos neste artigo são organizados hierarquicamente em uma coleção, formando a base para uma futura formalização em linguagem de padrões. São apresentados agrupados por níveis de abstração, do nível 1 ao 4, e retratados através de nós, no modo por largura na estrutura de árvore e analisados da esquerda para direita, seguindo a numeração atribuída a cada nó.

Cada nó apresenta uma particularidade especial, sendo que apenas o conjunto desses dão sentido aos padrões apresentados neste artigo. A partir da raiz, foi utilizada uma distância de um nó para a apresentação dos padrões relacionados, como pode ser visualizado na Figura 1.

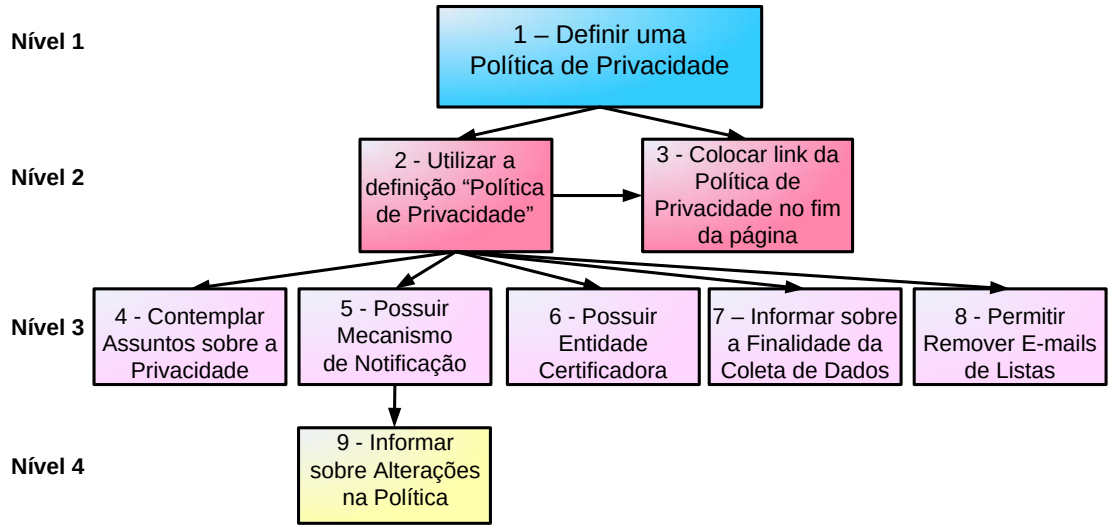


Figura 1. Definição dos Padrões para Política de Privacidade

Descrição dos Padrões Propostos

A seguir são apresentados os padrões, seguindo a numeração atribuída na Figura 1. A numeração evidencia a relevância do padrão dentro do conjunto de padrões, sendo esse conjunto a linguagem de padrões desenvolvida, seguido das definições, justificativas e características de cada padrão definido.

É recomendado utilizar os padrões em conjunto, observando também o uso com outros padrões já desenvolvidos para a web, de forma a criar uma arquitetura de soluções eficientes e testadas, já que os padrões representam a solução para um problema recorrente.

No detalhamento dos padrões são mostrados alguns exemplos de sua aplicação nas Políticas de Privacidade e no capítulo 4 do artigo é apresentado um exemplo de uma Política de Privacidade desenvolvida, onde todos os padrões são empregados.

---

1 - Nome: Definir uma Política de Privacidade (nível 1)

O padrão “Definir uma Política de Privacidade” é considerado o principal dentro do conjunto de padrões propostos, pois deve necessariamente existir para que os demais possam estar disponíveis, dividindo um problema genérico em um grupo de sub-problemas solucionados pelos padrões que o completam.

Contexto: Atualmente a web juntamente com mecanismos eletrônicos de comunicação estão sendo amplamente utilizados pelos mais diferentes perfis de usuários. Com isso a segurança dos usuários se torna cada vez mais necessária, já que dados pessoais podem ser requisitados para muitas operações, como por exemplo, no comércio eletrônico onde é necessária a coleta de dados para efetivação de negócios.

Problema: Nem todos os sites disponibilizam uma Política de Privacidade, quanto mais uma política de fácil entendimento aos usuários e que aborde assuntos relevantes.

Além da coleta de informações pessoais feitas durante a navegação dos usuários na web, podem ser feitos rastreamento de navegação e outras ações para saber as preferências dos usuários e identificá-los. Isso pode levar a uma invasão na privacidade do usuário e conseqüente diminuição de segurança, já que o usuário, em muitas das vezes não tem consciência do que possa estar acontecendo.

Dessa forma essa coleta vem provocando insegurança aos usuários que necessitam saber claramente como são armazenadas e distribuídas as suas informações que são coletadas durante sua navegação pelo site.

Forças: Sem a disponibilização de uma Política de Privacidade os usuários podem sentir-se inseguros em relação às práticas feitas pelo site, às regras seguidas e principalmente em relação a manipulação de seus dados pessoais coletados durante a navegação, o que pode diminuir a utilização de serviços web por esses usuários.

Além disso, devem ser considerados aspectos legais, pois sem a definição da Política de Privacidade os sites podem estar sujeitos a processos jurídicos. Como por exemplo, se a privacidade do usuário for violada sem seu consentimento e sem qualquer aviso prévio com notificações, esse pode recorrer a seus direitos constitucionais.

Solução: É necessário que os sites definam suas Políticas de Privacidade de forma clara e explicativa, informando aos usuários sobre as práticas e as normas seguidas, o que é feito com os dados coletados, qual a segurança oferecida, quais serviços são disponibilizados.

As políticas são uma forma rápida de comunicação entre o site e o usuário evitando mensagens direcionadas e específicas, por isso deve ser criada com vistas a facilitar o entendimento dos usuários e ter relevância nos assuntos abordados. Essas devem ser criadas de acordo com os princípios estabelecidos pela OECD e FTC, principalmente utilizando os princípios da Transparência, da Responsabilidade, Notificação e Segurança.

No desenvolvimento dessas políticas deve haver preocupação com: i) usabilidade, para facilitar a utilização de informações e serviços; ii) acessibilidade, para permitir que usuários com deficiência possam também entendê-la; iii) questões que tratem sobre a privacidade dos usuários, de modo a trazer maior segurança a eles; iv)

informações sobre medidas seguidas para prover a personalização, se essa existir, com vistas a facilitar e minimizar o tempo de buscas dos usuários; v) informar sobre a última atualização da política, de forma que os usuários possam se manter informados caso alguma mudança venha ocorrer, dentre outras informações relevantes ao conhecimento dos usuários.

Conseqüências: Os usuários se tornam mais confiantes na utilização dos sites e principalmente em relação aos serviços que demandam, por exemplo, de coleta de dados pessoais, tendo assim menor receio e maior segurança durante a navegação pelo site.

Além da vantagem oferecida aos usuários, a Política de Privacidade é de grande relevância para a empresa, que oferecendo maior segurança ganha mais usuários e ainda pode até mesmo facilitar sua organização interna em relação aos aspectos que demandam funcionalidades referentes à segurança, como a coleta de informações.

Usos Conhecidos: As empresas Extra e Comprafacil disponibilizam em seus sites Políticas de Privacidade, as quais contemplam assuntos sobre a privacidade e segurança dos usuários durante a navegação, sendo essas políticas claras e objetivas.

Padrões Relacionados: 2 - Utilizar a Definição “Política de Privacidade”; 3 - Colocar Política de Privacidade no fim da página.

---

2 - Nome: Utilizar a Definição “Política de Privacidade” (nível 2)

Contexto: As palavras “Política de Privacidade” devem ser utilizadas para a referência feita pelos sites às suas políticas, de modo a facilitar a sua busca pelo usuário.

Aplica-se ao ambiente web na arquitetura de informação dos sites para construir uma navegação adequada aos usuários e uma nomenclatura bem definida para a referência à Política de Privacidade.

Problema: Muitos usuários podem acabar desistindo de verificar a Política de Privacidade do site pois a busca pela referência à política se torna cansativa, já que nem se sabe qual a nomenclatura utilizada para referenciá-la, acarretando na insatisfação do mesmo.

Alguns dos sites avaliados no Estudo de Caso, apresentavam palavras diferentes para identificar a Política de Privacidade, como: Política de Segurança, Segurança, Compre Seguro, dentre outras (LOBATO e ZORZO, 2007), o que dificulta substancialmente a localização por essas no site.

Forças: Muitos sites apresentam palavras diferentes para identificar a Política de Privacidade, não se preocupando em disponibilizar nomes sugestivos e de fácil entendimento pelo usuário.

Em relação a aspectos legais, diferentes definições para Política de Privacidade podem ser vistas com o propósito de tornar a busca pelas políticas mais difícil.

Solução: Ter um único nome para referenciar o texto referente às práticas seguidas pelos sites, às regras impostas, aos serviços e a segurança oferecida aos usuários. Nesse padrão é sugerido que sempre seja referenciada a política dos sites através da nomenclatura “Política de Privacidade”.

Conseqüências: Facilita a busca do usuário pela Política de Privacidade do site, acarretando em um aumento de sua satisfação durante a navegação, já que os serviços são dispostos de forma clara e fáceis de serem encontrados.

Usos Conhecidos: O site da empresa Gol Linhas Aéreas utiliza a expressão "Política de Privacidade" para referenciar sua Política de Privacidade (LOBATO e ZORZO, 2007).

Padrões Relacionados: 1 - Definir uma Política de Privacidade; 3 - Colocar Política de Privacidade no fim da página; 4 - Contemplar Assuntos sobre a Privacidade; 5 - Informar sobre Alterações na Política; 6 - Possuir Entidade Certificadora; 7 - Informar sobre a Finalidade da Coleta de Dados; 8 - Permitir Remover E-mails de Listas; Privacy-Aware Network Client Pattern –descrevem um mecanismo para implementar este padrão em sites web (SADICOFF, M.; LARRONDO-PETRIE, M. M. E FERNANDEZ, E. B, 2005).

---

3 - Nome: Colocar link da Política de Privacidade no fim da página (nível 2)

Contexto: Os sites disponibilizam o link que faz referência à sua Política de Privacidades nas mais diferentes posições, podendo estar presente no menu suspenso de serviços disponíveis, apenas na página inicial dos sites ou na parte inferior da página.

Problema: Como não há um lugar específico onde a Política de Privacidade possa estar, os usuários perdem tempo na busca pela localização do link que leva à política. Isso pode tornar a busca pela Política de Privacidade cansativa e frustrante quando essa localização é sem sucesso, podendo afastar o usuário do site, já que esse não pôde conhecer as regras seguidas e a política imposta.

Forças: Facilitar a busca do usuário pela Política de Privacidade.

Solução: Para facilitar a busca pela política do site, é definida uma posição onde a referência à Política de Privacidade deve estar. Essa posição deve ser estratégica para que de qualquer parte do site seja possível localizar a Política de Privacidade.

Dessa forma é proposto que a referência à Política de Privacidade seja colocada ao final da página, não atrapalhando o design do site e possibilitando que essa seja referenciada por todas as páginas pertencentes ao site. Ainda completa-se a esse padrão a utilização da referência à Política de Privacidade no fim da página no formato centralizado em relação ao site.

Conseqüências: Facilita a busca do usuário pela Política de Privacidade do site, tornando-o mais satisfeito quanto a interface apresentada, já que a usabilidade foi levada em consideração, e possibilitando que a qualquer momento da navegação essa política possa ser verificada.

Usos Conhecidos: O sites das empresas PontoFrio e ShopTime utilizam a Política de Privacidade na base inferior dos sites e ainda centralizada, tornando-a fácil de ser encontrada e suficientemente entendível (LOBATO e ZORZO, 2007).

Padrões Relacionados: 1 – Definir uma Política de Privacidade; 2 – Utilizar a definição "Política de Privacidade".

---

4 - Nome: Contemplar Assuntos sobre a Privacidade (nível 3)

Contexto: A Política de Privacidade deve abordar tópicos de privacidade referente a segurança dos usuários, de forma a mostrar como a segurança é oferecida, a privacidade é garantida, a confiabilidade e veracidade dessas informações, de acordo com os princípios estabelecidos pela OECD, Princípio da Segurança e, FTC, Segurança.

Problema: Muitos sites não disponibilizam informações relevantes a privacidade dos usuários, não informam aos usuários sobre quais medidas podem ser tomadas caso algo venha a ocorrer contra sua privacidade e segurança.

Não contemplando tais assuntos os usuários podem se sentir ameaçados e inseguros em relação a navegação pelo site, ocasionando em uma desistência na utilização dos serviços disponíveis.

Forças: Traz mais segurança aos usuários podendo esses interagirem melhor com os sites e principalmente com maior confiabilidade.

Solução: Disponibilizar nas Políticas de Privacidade informações referente a segurança e a garantia de privacidade do usuário. Essas devem ser escritas de forma clara e objetiva, tornando o usuário ciente do perigo em ter sua privacidade invadida durante a utilização pela web, se esse perigo existir.

Conseqüências: Com a disponibilização de informações referentes a privacidade o usuário torna-se mais esclarecido quanto a esses assuntos e consequentemente, torna-se mais seguro para a utilização do site durante sua navegação na web.

Usos Conhecidos: Na Figura 2 é mostrada uma Política de Privacidade que contempla assuntos referentes à privacidade dos usuários.

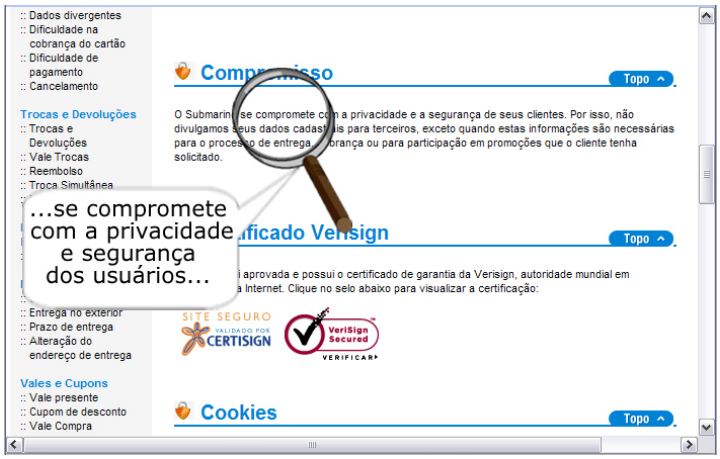


Figura 2. Exemplo da Aplicação do Padrão 4

Padrões Relacionados: 2 – Utilizar a definição “Política de Privacidade”.

---

5 - Nome: Possuir Mecanismo de Notificação (nível 3)

Contexto: A notificação é utilizada para tornar informações conhecidas pelos usuários. É importante notificar os usuários sobre o resultado de algumas de suas ações

e até mesmo, alertá-los durante sua navegação pela web, de acordo com os princípios estabelecidos pela OECD, Princípio da Especificação de Objeto e, FTC, Notificação.

Como, por exemplo, se uma página não possui ambiente seguro e solicita que o usuário informe seus dados pessoais, é saliente comunicá-los sobre isso. No entanto, é preciso ponderar para que tal notificação não seja cansativa e desnecessária, o que pode acabar incomodando o usuário.

Problema: Os sites não oferecem segurança e, na maioria das vezes, tentam esconder suas falhas ou a falta de serviços especializados disponíveis, não informando aos usuários sobre os perigos decorrentes de sua navegação.

Forças: Manter os usuários informados sobre os perigos e alguns benefícios provenientes de sua navegação aumenta a confiança depositada no site e consequentemente a utilização dos serviços disponíveis.

Solução: Notificar o usuário sobre vantagens e desvantagens oferecidas pelos sites, informá-los sobre as ações executadas com sucesso ou não. Se, por exemplo, o usuário for efetivar uma transação que deva ser confidencial, deve ser informado sobre a segurança oferecida ou a falta dela.

Essa notificação deve ser feita com a utilização de mensagens, podendo ser exibidas em janelas de alerta ou através de um tópico descrito na Política de Privacidade. A utilização de janelas de alerta são mais eficientes, pois chamam mais a atenção dos usuários, já que são exibidas no momento em que o usuário efetua a ação que deve ser notificada.

Conseqüências: Deixa o usuário sempre informado em situações adequadas sobre os perigos providos de sua navegação e em contrapartida, sobre a segurança oferecida.

Usos Conhecidos: Na Figura 3 é mostrado um exemplo de notificação, na qual o site informa aos usuários sobre alguns cuidados que devem ser tomados durante a interação com a web, de modo que sua privacidade não seja invadida, a segurança não seja violada e a oferta de serviços não seja prejudicada.

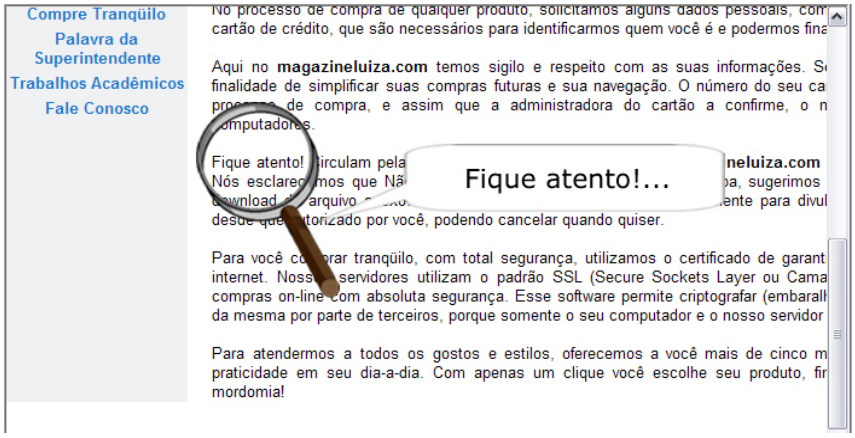


Figura 3. Exemplo da Aplicação do Padrão 5

Padrões Relacionados: 2 – Utilizar a definição “Política de Privacidade”; 9 – Informar sobre Alterações na Política; Privacy-Aware Network Client Pattern – utilizados como mecanismo para notificar os usuários sobre as mudanças no site (SADICOFF, M.; LARRONDO-PETRIE, M. M. E FERNANDEZ, E. B, 2005).

---

6 - Nome: Possuir Entidade Certificadora (nível 3)

Contexto: Para aumentar a confiança do usuário no site, garantindo que o site está em conformidade com as regras definidas em sua Política de Privacidade, podem ser utilizadas entidades certificadoras.

As entidades certificadoras são marcas de privacidade e de confiança, mostradas nas páginas dos sites, as quais informam aos visitantes que as práticas de segurança conduzidas pelos sites, estão de acordo com o que foi proposto em suas Políticas de Privacidade.

Esse padrão está de acordo com os princípios estabelecidos pela OECD, Princípio da Segurança, da Responsabilidade e, FTC, Segurança.

Problema: A não utilização das entidades certificadoras pelos sites pode fazer com que o usuário não se sinta seguro, principalmente em sites de comércio eletrônico onde transações são efetuadas envolvendo número de cartão de crédito e senhas.

Muitos sites não utilizam tais entidades devido ao preço ou por descumprimento do que é tratado em suas políticas e, em muitas das vezes, tais certificados são utilizados de maneira indevida (LOBATO e ZORZO, 2007).

Forças: Aumenta a confiança do usuário no site, pois provê uma maior segurança quanto aos serviços oferecidos, permitindo uma navegação tranqüila e a disponibilização de dados pessoais com maior segurança.

Solução: Os sites devem se preocupar em disponibilizar os serviços com garantias do nível de segurança. Isso pode ser obtido com a utilização de certificados de privacidade, sujeitando-se a passar por avaliação para receber um certificado de que está em conformidade com as práticas descritas em sua Política de Privacidade.

Após o recebimento do certificado deve-se utilizá-lo de forma correta, tendo como endereço a URL ao qual a certificação foi atribuída, observando a data de vencimento.

Conseqüências: Regulariza a situação do site com embasamento em entidades certificadoras reconhecidas e torna o usuário mais tranqüilo durante a navegação no site e efetivação de transações. No entanto, a utilização dessas deve ser descritas de forma clara nas políticas, pois muitos usuários não sabem em quais entidades certificadoras podem confiar.

Usos Conhecidos: Na Figura 4 é apresentada uma Política de Privacidade, onde é descrito sobre a certificação atribuída ao site, garantindo o cumprimento das regras descritas na política. Ainda é informado o nome da entidade certificadora a qual é responsável pela certificação dada.

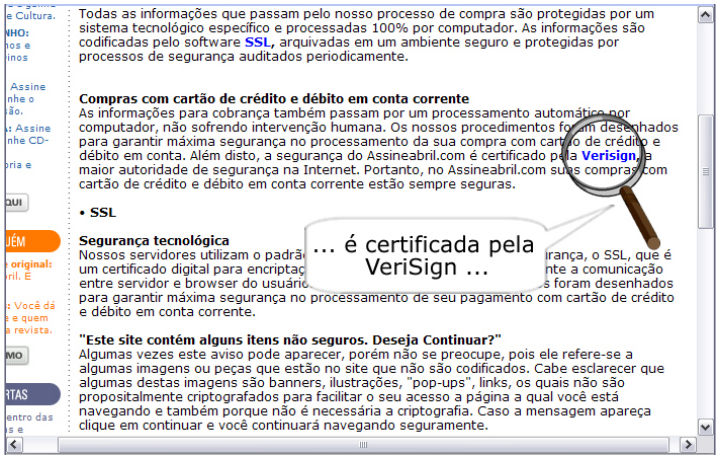


Figura 4. Exemplo da Aplicação do Padrão 6

Padrões Relacionados: 2 - Utilizar a Definição “Política de Privacidade”.

---

7 - Nome: Informar sobre a Finalidade da Coleta de Dados (nível 3)

Contexto: É importante informar aos usuários sobre a finalidade da coleta de seus dados para que esse se sinta mais tranqüilo durante a navegação pelo site, já que tem consciência do que é feito com seus dados coletados.

Esse padrão é proposto de acordo com os princípios estabelecidos pela OECD, Princípio do Limite de Coleta, da Qualidade dos Dados, da Especificação de Objetivo.

Problema: Nem todos os sites estão preocupados com o conforto do usuário no conhecimento do que é feito com seus dados coletados pelos sites.

Forças: Aumento na confiança do usuário em relação ao site.

Solução: Ter um tópico na Política de Privacidade sobre a coleta de dados, deixando claro aos usuários sobre quais são os dados coletados e sua finalidade. É relevante informar a vantagem da coleta de dados, como a oferta de personalização, facilitando suas buscas e otimizando os serviços disponíveis, e bem como as desvantagens com a coleta, como a disponibilização dos dados coletados para terceiros ou a identificação do usuário mesmo se ele não desejar.

Conseqüências: Torna o usuário mais seguro já que esse torna-se consciente sobre quais dados serão coletados durante sua navegação no site e sua finalidade.

Em sites de comércio eletrônico é muito importante informar aos usuários sobre a coleta, pois assim torna-os mais confiantes podendo fazer com que passem de simples visitantes para grandes consumidores.

Usos Conhecidos: A Figura 5 mostra que a Política de Privacidade definida contempla assuntos sobre a finalidade da coleta de dados pessoais dos usuários, abordando as vantagens e desvantagens que são oferecidas aos usuários.

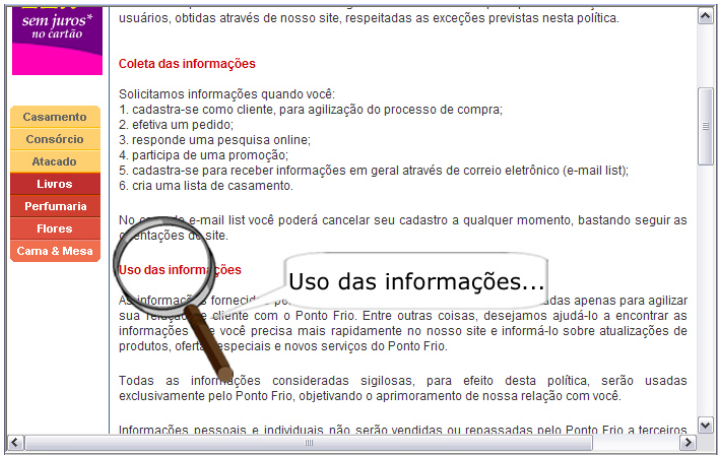


Figura 5. Exemplo da Aplicação do Padrão 7

Padrões Relacionados: 2 – Utilizar a definição “Política de Privacidade”.

---

8 - Nome: Permitir Remover E-mails de Lista (nível 3)

Contexto: Os usuários quando fazem o cadastro nos sites podem optar pelo recebimento de e-mails promocionais, cadastrando seu e-mail nas listas de promoções, alertas, novidades, mas com o passar do tempo, tal serviço pode se tornar cansativo e desnecessário ao usuário.

Esse padrão está de acordo com os princípios estabelecidos pela OECD, Princípio da Participação Individual e, FTC, Acesso.

Problema: Depois dos usuários terem cadastrado seus e-mails nas listas, alguns sites não permitem que os mesmos possam desfazer tal ação, podendo causar aborrecimento a eles que passam a considerar o envio de e-mails como envio de *spams*.

Forças: Diminuir as frustrações dos usuários deixando-os mais a vontade na utilização dos sites, possibilitando que tenham o controle do recebimento ou não de e-mails promocionais mesmo que já tenham cadastrado o e-mail nas listas.

Solução: Permitir que o usuário remova o e-mail das listas, caso julgue necessário. Isso pode ser feito apenas disponibilizando no site uma opção de seleção para esse propósito, onde pode ser disponibilizada uma mensagem, como por exemplo, *“desejo receber e-mails com promoções”*, e se o usuário não mais desejar o recebimento dos e-mails essa opção pode ser desmarcada.

Conseqüências: Possibilita que o usuário, ao sentir-se incomodado com o recebimento de e-mails, possa não recebê-los mais.

Usos Conhecidos: Na Figura 6 é apresentada aos usuários a opção de recebimento ou não de informativos pelo e-mail, podendo os e-mails serem removidos das listas aos quais foram cadastrados se assim o usuário desejar.

Figura 6. Exemplo da Aplicação do Padrão 8

Padrões Relacionados: 2 – Utilizar a definição “Política de Privacidade”.

---

9 - Nome: Informar sobre Alterações na Política (nível 4)

Contexto: É importante avisar aos usuários se alguma regra ou item imposto na Política de Privacidade for alterado, de forma que o usuário possa se manter informado e conscientizado sobre as normas e funcionalidades do site, de acordo com os princípios estabelecidos pela OECD, Princípio da Responsabilidade e, FTC, Notificação.

Problema: Os sites alteram suas Políticas de Privacidade sem informar aos usuários sobre isso sem consultar os usuários se concordam com as novas diretivas.

Forças: Permitir que o usuário esteja sempre informado sobre as novas diretivas definidas pelo site, de modo a conscientizá-los das regras seguidas, aumentando a confiança no site. Um gerenciador deve ser responsável por cumprir o que é descrito na política do site, colocando em prática todos os itens mencionados na política.

Solução: Para aplicar tal funcionalidade, colocar uma cláusula na Política de Privacidade informando de que se houver qualquer alteração nas regras seguidas pelo site, essas modificações serão expostas na própria Política de Privacidade. É importante ainda notificar a data da última atualização da Política de Privacidade, de modo que o usuário possa manter-se informado sempre que novas atualizações forem feitas.

Há também a opção de disponibilizar uma notificação na página inicial do site informando que houve mudanças nas diretivas da Política de Privacidade.

Conseqüências: Deixa o usuário mais esclarecido e aumenta sua confiança no site já que esse tem como saber quando alguma diretiva referente às regras impostas na Política de Privacidade do site for alterada. Assim o usuário poderá ter conhecimento dessa mudança e então verificar se aceita ou não as novas diretrizes impostas.

Com os usuários conscientes sobre as práticas definidas pelos sites, não haverá invasão de privacidade, já que a questão de privacidade está diretamente ligada ao consentimento ou não do usuário em relação às práticas seguidas pelos sites.

Usos Conhecidos: A Figura 7 mostra exemplo da notificação transcrita no site referente às possíveis atualizações que possam ocorrer na Política de Privacidade.

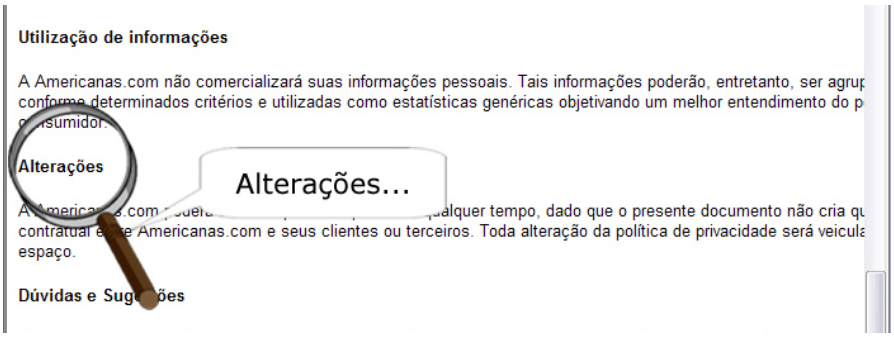


Figura 7. Exemplo da Aplicação do Padrão 9

Padrões Relacionados: 5 – Possuir Mecanismo de Notificação.

4. Resultado Final Aplicando os Padrões Definidos

Após formar a estrutura de base, ou seja, os padrões e explicar suas utilidades foi desenvolvida uma Política de Privacidade para ser tomada como exemplo.

Para desenvolvimento dessa política foram utilizados os padrões apresentados para definição de Política de Privacidade, onde foram observados os requisitos desejados, analisados as informações que se deve disponibilizar na definição das Políticas de Privacidade, tentando aproximá-la à política ideal.

Tabela 1. Exemplo de Política de Privacidade

Política de Privacidade

Atualizada em 05/03/2007.

Sobre esta Política de Privacidade
Esta Política de Privacidade foi estabelecida para o *"site Exemplo"* com o objetivo de assegurar a confiança e o sigilo das informações dos usuários coletadas.

Sabemos o quanto é importante para você conhecer e estar seguro sobre a utilização dos seus dados pessoais. Por isso, nos preocupamos em esclarecer e divulgar nossa política de utilização dessas informações. Assim, você poderá entender melhor quais informações obtemos e como as utilizamos.

Dados Coletados
Solicitamos informações quando você:

- Se cadastra no site (para agilização do processo de compra e para fins de estatísticas);
- Efetiva um pedido;
- Responde uma pesquisa on line;
- Participa de uma promoção;
- Cadastra-se em nosso boletim eletrônico ("*mail list*").

De forma automatizada, os seguintes dados também são coletados:

- Endereço IP;
- Data e horário do acesso;
- Tempo de leitura de cada página;
- Seqüência de páginas visitadas;

Cadastro
Não é necessário fornecer informações pessoais para navegar no site. Entretanto, para utilizar alguns dos serviços, será necessário identificar-se, fornecendo previamente alguns dados de caráter pessoal.

As informações serão armazenadas em um servidor seguro, e não são compartilhadas com terceiros.

Finalidade da Coleta
Inicialmente os dados coletados terão fins estatísticos. Para analisar, por exemplo, a quantidade de usuários que leram a Política de Privacidade, as diferenças entre as preferências de privacidade dos usuários e a freqüência de visita a cada página.

Os dados coletados serão também analisados para obter algumas informações sobre o perfil dos usuários que acessam o site, de modo a oferecer serviços personalizados.

Ainda utilizamos as informações coletadas por motivos de fins estatísticos, para efetivação da compra, andamento das operações e entrega de produtos.

Exclusão das Informações
O site possibilita que o usuário exclua e edite suas informações cadastradas no site, caso julgue necessário.

Segurança
Todos os dados coletados são armazenados em servidores internos e seguros, em um banco de dados reservado e com acesso restrito ao administrador deste site. Dessa forma, a manipulação dos dados se dá de maneira automatizada, não permitindo que pessoas não autorizadas tenham acesso aos mesmos.

Certificação
As práticas efetuadas pelo site seguem as diretivas definidas nessa política e são certificadas por uma Entidade Certificadora, chamada *XXX*, a qual garante que a Política de Privacidade está sendo seguida.

Confira o certificado de segurança clicando aqui.

Ambiente para Transações
Utilizamos um ambiente seguro para transações, fazendo a encriptação de dados, autenticação de servidor, integridade de mensagem e autenticação de cliente.

Tenha Cautela
É possível que nossas páginas contenham *hiperlinks* que o levem a sites de terceiros. Recomendamos a leitura da Política de Privacidade desses sites, uma vez que não temos nenhuma responsabilidade sobre os mesmos.

Algumas pessoas utilizam do nome de empresas de responsabilidade para enviar e-mails aos usuários e também podem ser enviados juntos a esses e-mails códigos executáveis. No entanto, em hipótese alguma, os aceite, pois tais e-mails e executáveis tem o intuito de coletar suas informações pessoais.

Esteja atento a esses e-mails, prestando atenção no endereço do remetente, e, se possível entre em contato conosco avisando sobre o ocorrido.

Envio de E-mails
Este site não envia e-mail de propagandas e promoções do site sem a autorização do usuário.

O site provê estruturas que permitem ao usuário selecionar o aceite ou não de seu e-mail nas listas de propagandas. Assim, você poderá cancelar o envio de e-mails a qualquer momento.

Cookies
Cookies são pequenos arquivos de texto enviados ao seu computador e que são armazenados no mesmo. Estes arquivos servem para reconhecer, acompanhar e armazenar a navegação do usuário na Internet.

O uso de *cookies* possibilita ao site oferecer um serviço mais personalizado, de acordo com as características e interesses dos usuários, possibilitando, inclusive, a oferta de conteúdo e

publicidade específicos para cada um.

Alterações nesta Política
Para assegurar regras claras e precisas, podemos eventualmente alterar essa política, e sendo assim, recomendamos sua leitura periodicamente.
Qualquer alteração na Política de Privacidade será transcrita na mesma.

No início da Política de Privacidade é indicada a data da última alteração, para facilitar ao usuário saber quando houve modificações.

Considerações Finais
Em caso de alguma divergência sobre nossa Política de Privacidade ou reclamações sobre os serviços prestados, sinta-se livre para entrar em contato conosco:

Nome Fantasia da Empresa ou Site
Nome de registro no CNPJ da Empresa
Endereço físico

Atendimento telefônico:
(0xxXX) XXXX.XXXX das XX:XXhs as XX:XXhs

Atendimento eletrônico:
http://www.empresa.com.br/antendimento
atendimento@empresa.com.br

Como já mencionado, esse modelo de Política de Privacidade foi desenvolvido baseado nos padrões apresentados. Dessa forma o uso desses padrões se torna viável durante a elaboração e construção de Políticas de Privacidade que apresente características relevantes aos usuários e que atendem às verdadeiras exigências que uma política deve apresentar, sendo essa uma política de sucesso.

Tal modelo de política pode ser utilizado de modo a trazer facilidades aos sites na definição de suas Políticas de Privacidade e principalmente, trazendo benefícios aos usuários, já que essas serão definidas de maneira mais clara e objetiva, disponibilizada em uma linguagem que o usuário entenda, de modo a aumentar sua satisfação na interação com o site, já que o mesmo se sentirá mais seguro. Ainda é referenciada por um nome sugestivo, “Política de Privacidade” e de fácil localização.

5. Agradecimentos

Este trabalho foi apoiado pelo Departamento de Computação da Universidade Federal de São Carlos (UFSCar) e Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brasil. Nossos agradecimentos especiais ao Prof. Eduardo B. Fernandez, nosso *shepherd*, pelos comentários e sugestões importantes que proporcionaram melhorias significativas em nosso trabalho.

7. Referências

ALEXANDER, C.; ISHIKAWA, S. e SILVERSTEIN, M. A Pattern Language. Oxford University Press, New York. 1977.

ANTON, A. et al. The lack of clarity in financial privacy policies and the need for standardization. IEEE Security & Privacy. 2(2): 36-45 p. 2004.

BORCHERS, J. A Pattern Approach to Interaction Design 2001. John Wiley & Sons, Inc. Disponível em: <http://portal.acm.org/citation.cfm?id=558433&coll=Portal&dl=GUIDE&CFID=3720139&CFTOKEN=24769028#>. Acesso em: 16 out. 2006.

BUSCHMANN, F. et al. Pattern-Oriented Software Architecture. vol.1: A System of Patterns: Chichester, Inglaterra: John Wiley & Sons Ltd. 1996. 476 p.

COPLIEN, J. O. e HARRISON, N. B. Organizational Patterns of Agile Software Development. Prentice Hall PTR. 2004. 419 p.

GAMMA, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

HAFIZ, M. A collection of Privacy Design Patterns. In: PLoP Pattern Languages of Programming Design. Potland, Oregon: 2006.Disponível em: <http://hillside.net/plop/2006/Papers/Library/PLoP2006_mhafiz0_0.pdf>. Acesso em: 14 fev. 2006.

LOBATO, L. L. e ZORZO, S. D. Avaliação dos Mecanismos de Privacidade e Personalização na Web. In: XXXII Conferencia Latinoamericana de Informática, CLEI, Agosto 2006. Santiago, Chile: 2006.Disponível em: <www.clei2006.org>. Acesso em: 23 fev. 2007.

LOBATO, L. L. e ZORZO, S. D. Estudo de caso da avaliação por inspeção em sites de comércio eletrônico. Universidade Federal de São Carlos. São Carlos, SP: 20/10/2006, p.94. 2007.

MESZAROS, G. e DOBLE, J. MetaPatterns: A Pattern Language for Writing Patterns. 1996. Conference on Pattern Languages of Programming PLoP. Disponível em: <http://www.hillside.net/patterns/writing/patternwritingpaper.htm>. Acesso em: 13 nov. 2006.

PITOFSKY, R. et al. Privacy online: Fair information practices in the electronic marketplace. 2000. Federal Trade Commission. Disponível em: <http://www.ftc.gov/reports/privacy2000/privacy2000.pdf>. Acesso em: 13 nov. 2005.

ROMANOSKY, S. et al. Privacy Patterns for Online Interactions. In: PLoP Pattern Languages of Programming Design.Potland, Oregon: 2006.Disponível em: <http://hillside.net/plop/2006/Papers/Library/romanosky_privacy_patterns_plop06.pdf> Acesso em: 22 fev. 2007.

SADICOFF, M.; LARRONDO-PETRIE, M. M. e FERNANDEZ, E. B. Privacy-Aware Network Client Pattern. In: Conference on Pattern Languages of Programming PLoP. 2005.Disponível em: <http://hillside.net/plop/2005/proceedings/PLoP2005_msadicoff0_0.pdf>. Acesso em: 15 fev. 2007.

SPIEKERMANN, S.; GROSSKLAGS, J. e BERENDT, B. E-privacy in 2nd Generation E-Commerce: privacy preferences versus actual behavior. In: Proceedings of the 3rd ACM Conference on Electronic Commerce.Tampa, Florida, USA: 2001. Pág. 38-47.Disponível em: <http://doi.acm.org/10.1145/501158.501163>. Acesso em: 24 jan. 2006.

TUROW, J. Americans and Online Privacy: The System is Broken. 2003. Disponível em: <http://www.appcpenn.org/04_info_society/2003_online_privacy_version_09.pdf>. Acesso em: 20 jan. 2006.

The Error Handling Aspect Design Pattern

Fernando Castor Filho[1] **, Alessandro Garcia**[2] **, Cecília Mary F. Rubira**[3]

[1] [1]Department of Computer Science - University of São Paulo
Rua do Matão, 1010. 05508-090, São Paulo - SP, Brazil

[2]Computing Department - Lancaster University
South Drive, InfoLab 21, LA1 4WA, Lancaster, UK

[3]Institute of Computing - State University of Campinas
P.O. Box 6176. 13083-970, Campinas - SP, Brazil

fcastor@acm.org, garciaa@comp.lancs.ac.uk, cmrubira@ic.unicamp.br

Abstract. *Exception handling is a well-known programming language mechanism for separating error handling code from the normal application code. One of the fundamental motivations for employing exception handling in the development of robust applications is to lexically separate error handling code from the normal code so that they can be independently modified. However, experience has shown that the exception handling mechanisms of mainstream programming languages fail to achieve this goal. In most systems, exception handling code is interwined with the normal code, hindering maintenance. Moreover, because of the difficulty in separating error handling code and normal code, the former is often duplicated across several different places within a system. In this paper we present a pattern,* Error Handling Aspect*, which leverages aspect-oriented programming in order to enhance the separation between error handling code and normal code. The basic idea of the pattern is to use advice to implement exception handlers and pointcuts to associate advice to different parts of the normal code in order to improve the maintainability of the normal code and the reuse of error handling code.*

1. Intent

To separate the error handling measures of a system from the code that implements its behavior when nothing goes wrong (normal code). The Error Handling Aspect design pattern leverages aspect-oriented programming (AOP) [Kiczales et al. 1997] techniques to improve the maintainability of the normal code and its reuse across different applications. The pattern also aims to reduce duplication of error handling code by making it easier to reuse within the same application.

2. Context

The Error Handling Aspect design pattern can be applied in everyday software development, mainly during the design and implementation phases of the software process, as a means to improve the flexibility of software systems. However, the benefits of the pattern are more tangible in situations where

- a software component is expected to be reused in several different contexts, associated to different error handling strategies.

- the same piece of error handling code is duplicated across several parts of a software system and it is desirable to localize this duplicated code in a single conceptual entity.

3. Motivation

Exception handling [Goodenough 1975] mechanisms have been conceived as a means to structure programs that have to cope with erroneous situations. These mechanisms make it possible for developers to extend the interface of an operation with additional exit points that are specific to error recovery. Moreover, they define new constructs for raising exceptions and associating exception handlers with selected parts of a program. Ideally, an exception handling mechanism should enhance attributes such as reliability, maintainability, and understandability, by making it possible to write programs where: (i) the code for error handling and the normal code are lexically separate and can be maintained independently [Parnas and Würges 1976]; (ii) the impact of the code responsible for error handling in the overall system complexity is minimized [Randell and Xu 1995]; and (iii) an initial version that does little recovery can evolve to one which uses sophisticated recovery techniques without a change in the structure of the system [Parnas and Würges 1976].

Separation of concerns is the overarching goal of exception handling mechanisms. However, the kind of separation promoted by the exception handling mechanisms of most mainstream object-oriented programming languages brings only limited advantages [Castor Filho et al. 2006, Cui and Gannon 1992, Lippert and Lopes 2000]. The following code snippet, extracted from an Eclipse plugin, illustrates this.

```
public class CRLFDetectInputStream extends FilterInputStream {
  ...
  protected CRLFDetectInputStream(InputStream in, ICVSStorage file) {
    super(in);
    try {
      this.filename = getFileName(file);
    } catch (CVSException e) {
      this.filename = file.getName();
    }
  }
  ...
}
```

The example above defines the constructor for class `CRLFDetectInputStream`. This class is responsible for detecting the carriage return and line feed characters in input streams. The constructor attempts to obtain the full name of `file` by retrieving it from the file system through method `getFileName()`. If something goes wrong,e.g. the file could not be found, and exception `CVSException` is raised, the handler simply gets the name stored in variable `file`. In order to reuse class `CRLFDetectInputStream` in a different system, it might be necessary to change this policy, for example, to interrupt program execution when the file cannot be accessed in the file system. To achieve this, it would be necessary to directly modify the `catch` block in the constructor. This kind of undisciplined reuse is generally considered a bad practice in the object-oriented development community. A much more desirable approach would be to simply “unplug” the error handling strategy associated to the constructor and “plug” the new one. However, this is currently not possible in any of the mainstream programming languages.

The following code snippet illustrates another undesirable situation:

```
public class EclipseSynchronizer implements IFlushOperation {
  public void endBatching(...)  throws CVSException {
    try {...} catch (TeamException e) {
      throw CVSException.wrapException(e); }
  } ...
  public IResource[] members(...)  throws CVSException {
    ...
    try {...} catch (CoreException e) {
      throw CVSException.wrapException(e); }
  } ...
}
```

In the example, two different methods within the same class, `endBatching()` and `members()`, implement identical exception handling strategies. `TeamException` is a subtype of `CoreException`. In Java, it is not possible to implement a single handler and associate it to both methods, to avoid code duplication.

4. Problem

In languages such as Java, Ada, C++, and C#, it is not possible to "plug" and "unplug" exception handlers. In these languages, the normal code and error handling code are entwined within fine-grained units (methods), making it hard to maintain the former independently from the latter. Also, this hardwiring of the exception handling code hinders reuse of normal code across different applications, as these applications often have different requirements pertaining to error handling.

Another problem is that the exception handling mechanisms of the aforementioned languages only support the definition of handlers that are local to specific parts of a program. Reuse of error handling strategies within an application is possible only to a certain degree, by extracting error handling measures to new methods. However, in most mainstream programming languages, the code that catches exceptions and initiates an exception handling measure has to be scattered throughout the application. As a consequence, most systems have a considerable amount of duplicated exception handling code.

5. Solution

The Error Handling Aspect design pattern promotes explicit separation between exception handling code and normal code. It leverages AOP techniques in order to: (i) localize error handling within units whose sole purpose is to implement this concern; (ii) reduce the amount of duplicated exception handling code; (iii) make it easier to reuse the normal code across different applications; and (iv) simplify the task of changing the exception handling strategies of a system. The overall idea of the pattern is to use advice to implement exception handlers and associate these "aspectized" handlers to different parts of a program by means of the composition mechanisms provided by AOP languages. For example, consider the following Java code snippet, extracted from an Eclipse plugin:

```
public class EclipseSynchronizer implements IFlushOperation {
  public void endBatching(...)  throws CVSException {
    try {...} catch (TeamException e) {
      throw CVSException.wrapException(e); }
  } ...
  public IResource[] members(...)  throws CVSException {
```

```
    ...
    try {...} catch (CoreException e) {
      throw CVSException.wrapException(e); }
  } ...
}
```

In the example, two different methods within the same class, `endBatching()` and `members()`, implement identical exception handling strategies. `TeamException` is a subtype of `CoreException`. In Java, it is not possible to implement a single handler and associate it to both methods, to avoid code duplication. The Error Handling Aspect design pattern leverages features of AOP languages to deal more elegantly with this problem. Figure 1 shows how the pattern solves this problem. It uses a slightly modified UML notation derived from the notation proposed by Chavez [Chavez 2004].

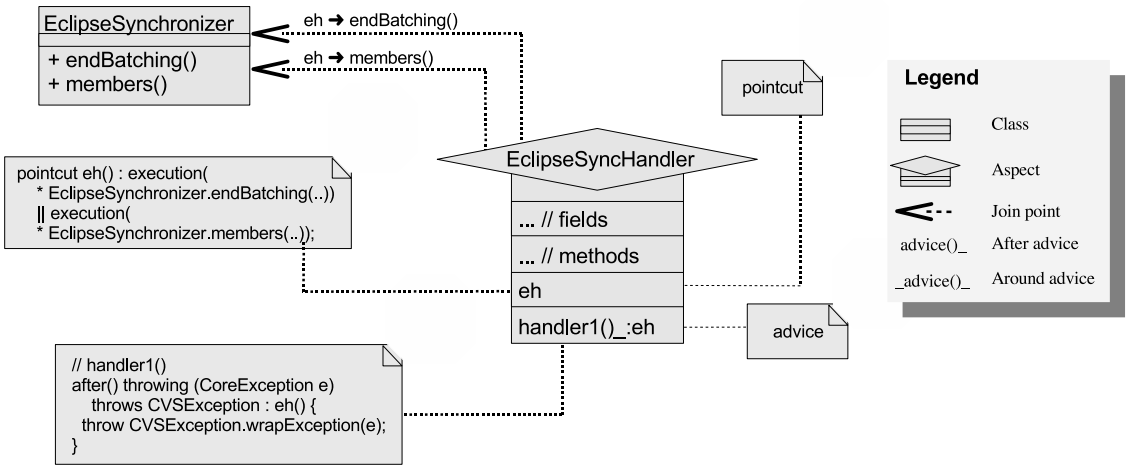


Figure 1. An example where the use of Error Handling Aspect avoids duplication of exception handling code.

In the figure, aspect `EclipseSyncHandler` defines a pointcut named `eh` that associates advice `handler1` to methods `members()` and `endBatching()`. This advice implements the exception handlers that would otherwise be scattered throughout the application code and is therefore callled a *handler advice*. The name of each advice in the diagram is followed by the name of a pointcut to which it is bound. The code snippets in the comments correspond to possible implementations written in the AspectJ [Laddad 2003] language. This approach separates the error handling code from the normal code and localizes it within a single program unit, namely, an error handling aspect implementing the different handler advice. As a consequence, code duplication is avoided and different error handling strategies can be easily plugged and unplugged to the normal code.

6. Background

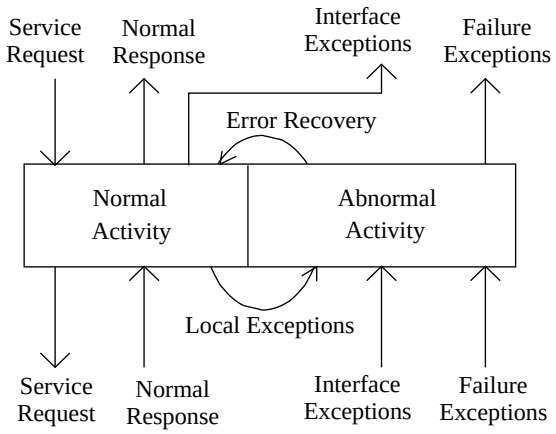
6.1. Exception Handling

Exception handling [Cristian 1989, Goodenough 1975] is a mechanism for structuring error recovery in software systems so that errors can be more easily detected, signaled, and handled. It is implemented by many mainstream programming languages, such as Java, Ada, C++, and C#. These languages allow the definition of exceptions and their

corresponding handlers. The set of exceptions and exception handlers in a system define its abnormal or exceptional activity.

When an error is detected, an exception is generated, or *raised*. If the same exception may be raised in different parts of a program, different handlers may be executed, depending on the place where the exception was raised. The choice of the handler that is executed depends on the exception handling context where the exception was raised. An exception handling context is a region of a program where the same exceptions are handled in the same manner. Each context has an associated set of handlers that are executed when the corresponding exceptions are raised. Typical examples of exception handling contexts in object-oriented languages are blocks, methods, classes, and exceptions [Garcia et al. 2001].

The *idealized fault-tolerant component* (IFTC) [Anderson and Lee 1990] defines a conceptual framework for structuring exception handling in software systems. An IFTC is a component (in a broader sense – an object, a software component, a whole system, etc.) where the parts responsible for the normal and abnormal activities are separated and well-defined, within its internal structure. The goal of the IFTC approach is to provide means to structure systems so that the impact of error recovery mechanisms in the overall system complexity is minimized. One of the most important goals of Error Handling Aspect is to promote the construction of systems where all the system components are IFTCs. The following figure presents the internal structure of an IFTC and the types of messages it exchanges with other components in a system.



When an IFTC receives a service request, it produces a *normal response* if the request is successfully processed. If an IFTC receives an invalid service request, it *signals* an *interface exception*. If an error is detected during the processing of a valid request, the normal activity part of the IFTC *raises* an *internal exception*, which is received by the exceptional activity part of the IFTC. If the IFTC is capable of handling an internal exception properly, normal activity is resumed. If the IFTC has no handlers for an internal exception or is unable to handle an exception, it *signals* a failure exception. Interface and failure exceptions are collectively called *external exceptions*. An IFTC might also *catch* external exceptions signaled by other IFTCs and attempt to handle them.

6.2. Aspect-Oriented Programming and AspectJ

AOP was proposed as a means to improve the separation of concern in systems that include *crosscutting concerns*. A crosscutting concern can affect several units of a software system and usually cannot be isolated by traditional OO programming techniques. A typical example of crosscutting concern is logging. The implementation of this concern is usually scattered across the modules in a system, and tangled with code related to other concerns, because some contextual information must be gathered in order for the recorded information to be useful. Other common examples of crosscutting concerns include profiling and authentication [Laddad 2003].

AspectJ [Laddad 2003] is a general purpose aspect-oriented extension to Java. It extends Java with constructs for picking specific points in the program flow, called join points, and executing pieces of code, called advice, when these points are reached. Join points are points of interest in the program execution through which crosscutting concerns are composed with other application concerns. AspectJ adds a few new constructs to Java, in order to support the selection of join points and the execution of advice in these points. A *pointcut* picks out certain join points and contextual information at those join points. Join points selectable by pointcuts vary in nature and granularity. Examples include method call and class instantiation. *Advice* can run *before*, *after*, or *around* the selected join points. In the latter case, execution of the advice may potentially alter the flow of control of the application, and replace the code that would be otherwise executed in the selected join point. AspectJ also allows programmers to modify the static structure of a program by means of static crosscutting. With static crosscutting, one can introduce new members in a class or interface, or make a checked exception unchecked.

Aspects are units of modularity for crosscutting concerns. They are similar to classes, but may also include pointcuts, advice, and static crosscutting. The code of an aspect-oriented application written in AspectJ consists of two parts: (i) base code, which is written in written in Java and implements the non-crosscutting concerns of the system; and (ii) aspect code, which implements the crosscutting concerns of the system and comprises a set of aspects and auxiliary classes. Aspect code is combined with base code by means of a process called weaving. Therefore, the tool responsible for performing weaving is called *weaver*.

The example below presents an aspect named `ConnectionPoolHandler`. Lines 2 and 3 declare a pointcut named `setManualCommitHandler` that captures calls to the method `setAutoCommit()` of class `Connection`, independently of return type ("`*`") or list of parameters ("`..`"). Line 4 *softens* `SQLException` for the join points selected by `setManualCommitHandler`. This means that `SQLException` is not statically checked by the Java compiler. At run time, if `SQLException` is raised in a call to `setAutoCommit()`, it is wrapped with an unchecked exception named `SoftException`, defined by AspectJ. Lines 5-8 declare an advice that is executed after the join points selected by `setManualCommitHandler` if their execution ends by throwing `SQLException` (Line 5). This advice captures contextual information on the selected join points by specifying that the target of the calls to `setAutoCommit()` can be referred to through variable `con`.

```
1 public aspect ConnectionPoolHandler {
2   pointcut setManualCommitHandler() :
3     call(* Connection.setAutoCommit(..));
4   declare soft : SQLException : setManualCommitHandler();
5   after(Connection con) throwing (SQLException e) :
6       setManualCommitHandler() && target(con) {
7     con.close();
8   }
9 }
```

7. Structure

In the rest of this paper, we call "exception-throwing statement" a statement that potentially throws an exception. Exception-throwing statements appear within "context methods". "Context" because, usually, these methods define exception handling contexts. In the figure, classes NormalClass1 and NormalClass2 define one context method each, contextMethod1() and contextMethod2(), respectively. We refer to a set of exception-throwing statements within the same context method as "exception-throwing code". Besides pointcuts and advice, error handling aspects can also include methods and fields that are specific to exception handling. A field in this case can be, for example, a hash table that stores temporary values that the handler advice use.

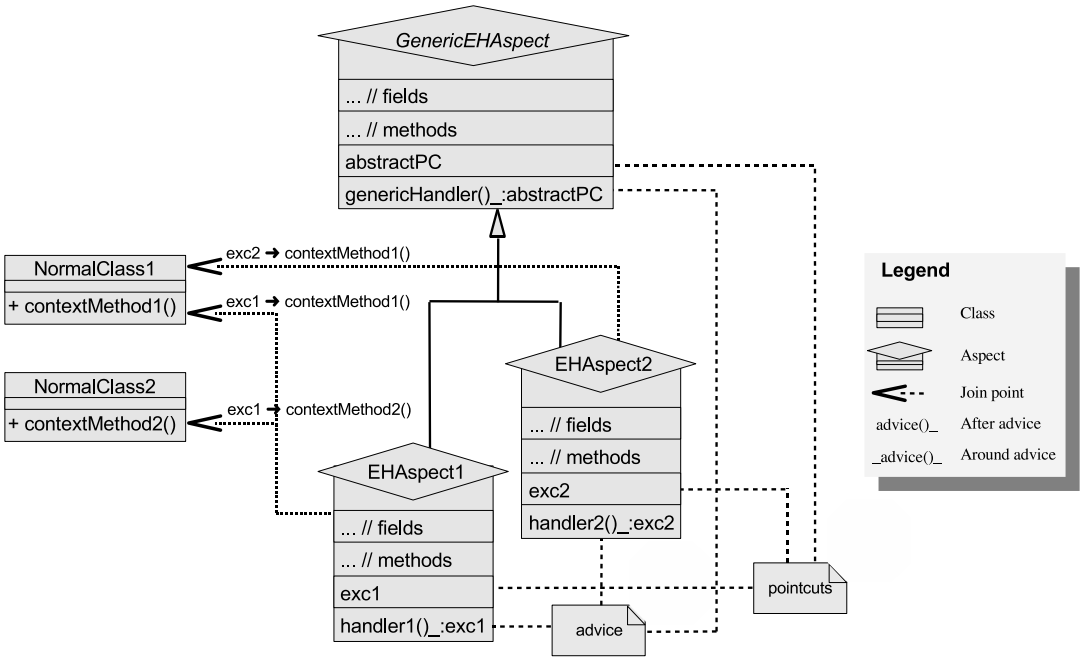


Figure 2. General structure of the pattern.

Normal classes. Classes NormalClass1 and NormalClass2 implement the normal code of an application. Each has one or more context methods, including one or more exception-throwing statements apiece.

Concrete error handling aspects. Figure 2 depicts two concrete error handling aspects, EHAspect1 and EHAspect2. Each one includes one or more handler advice. The handler advice implement exception handling code that is executed when exceptions are raised within the context methods. A handler advice may be bound to

several distinct context methods, in order to avoid duplicating exception handling code.

Abstract error handling aspects. If a handler advice is common to two or more error handling aspects, it is useful to move it to an abstract aspect and make the latter a super-aspect of the other error handling aspects. By binding the advice to an abstract pointcut and making it concrete in the sub-aspects, duplication of handler advice is avoided. In Figure 2, aspect GenericEHAspect is an example of abstract error handling aspect.

8. Dynamics

The following scenarios illustrate how the various components of the Error Handling Aspect design pattern interact at runtime.

Scenario 1. Figure 3 depicts the normal execution path when using an error handling aspect is present. In this scenario, a client invokes a method on a certain object and the execution of this method is a join point of interest for error handling. No exceptions are raised, though, and execution proceeds as if the aspect did not exist.

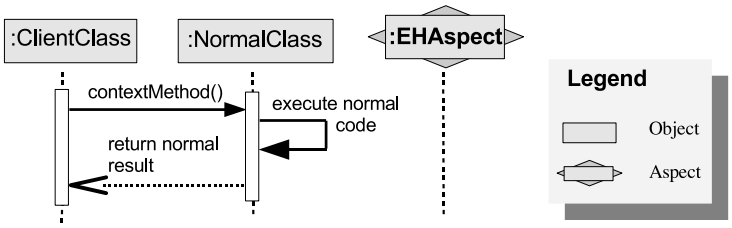


Figure 3. A scenario where no exceptions are thrown.

1. A client invokes `contextMethod()` on an instance of `NormalClass`. As implied by the name of the method, exceptions might potentially be raised within it.
2. Method `contextMethod()` is executed.
3. The method returns a normal result to the client object.

Scenario 2. Figure 4 depicts the scenario where a client invokes a method on a certain object, an exception is raised while the method is being executed, and an error handling aspect successfully handles the exception.

1. A client invokes `contextMethod()` on an instance of `NormalClass`.
2. While `contextMethod()` is being executed, exception `E` is raised.
3. Control is transfered to the error handling aspect `EHAspect`, which attempts to handle `E`.
4. The handler ends its execution normally, without raising any exceptions.
5. Control returns to the normal code, which resumes execution. Depending on the join point to which the handler advice is bound, either `contextMethod()` goes on executing or it immediately returns some normal (non-exception) response to the client object.

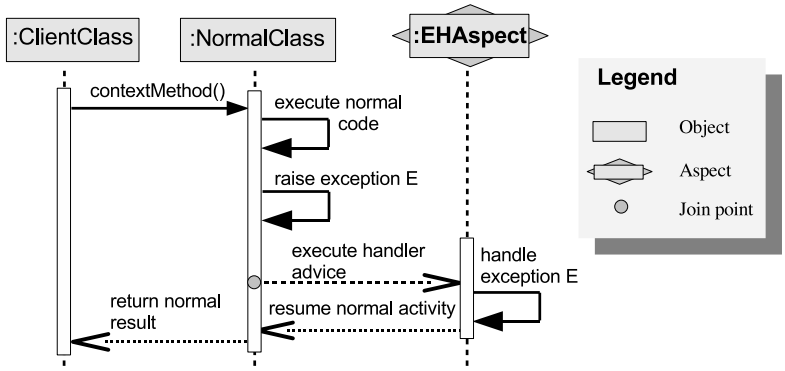


Figure 4. A scenario where a handler advice successfully handles an exception.

Scenario 3. This scenario shows the dynamics of error handling aspects that simulate two nested `try-catch` blocks. The scenario depicted in Figure 5 illustrates the case where the inner handler advice, after failing to handle an exception thrown within a context method, throws an exception that is caught by the outer handler advice. The latter then signals an exception and this exception is received by the client object. Handler advice `innerHandler()`, defined by aspect `InnerEHAspect`, is associated method `contextMethod()` or some part of it (e.g. a method call that appears in its body). Handler advice `outerHandler()`, defined by aspect `OuterEHAspect`, is associated to the same join point as `innerHandler()` or some 'outer' join point (e.g. the execution of or calls to `contextMethod()`).

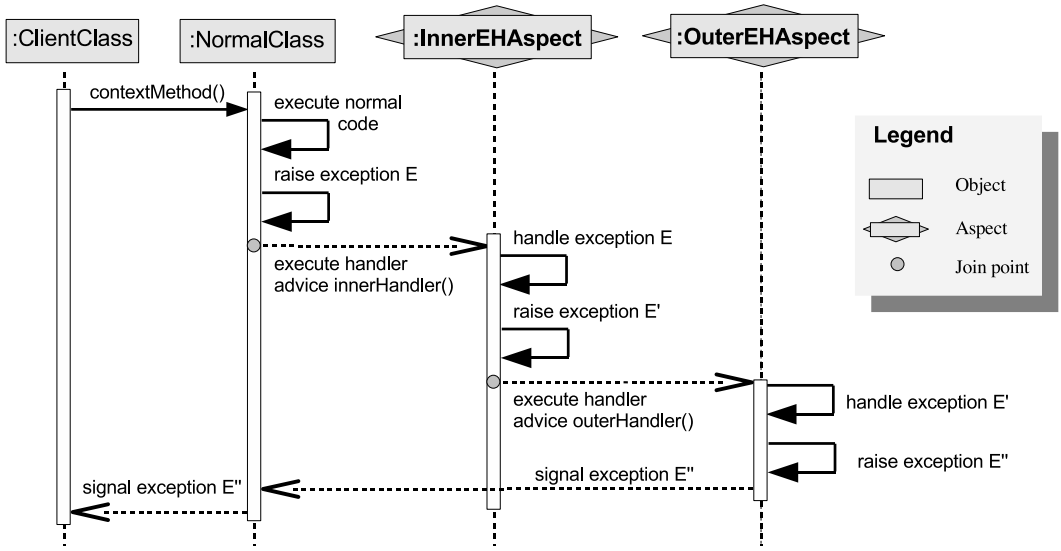


Figure 5. A scenario involving nesting of handler advice.

1. A client invokes `contextMethod()` on an instance of `NormalClass`.
2. While `contextMethod()` is being executed, exception `E` is raised.
3. Control is transferred to handler advice `innerHandler()`, which attempts to handle `E`.
4. Handler advice `innerHandler()` raises exception `E'`.
5. Control is transferred to handler advice `outerHandler()`, which attempts to handle `E'`.

6. Handler advice `outerHandler()` raises exception `E''`.
7. Exception `E''` is signaled to the instance of `NormalClass`, re-signaled by the latter, and finally received by the client object.

9. Consequences

The Error Handling Aspect design pattern has the following ***benefits***:

- *Localization of error handling code*. An important benefit of Error Handling Aspect is that it keeps all the exception handling code localized within program units whose sole purpose is to implement the exception handling concern. This localization simplifies system maintenance, as developers do not have to search through a whole program in order to change a certain exception handler. It also improves understandability, since it is possible to get an intuitive understanding of how error handling works in a given system just by looking at the error handling aspects.
- *Reduction of duplicated error handling code*. It is easy to encapsulate an exception handler that would otherwise appear in several parts of a system in a single handler advice. These parts of the system then become the join points of interest to which the handler advice will be associated. Notice, though, that this reduction of duplicated error handling code does not necessarily mean that the pattern will reduce the overall number of lines of code pertaining to error handling (see the last item under "Liabilities", below).
- *Arbitrary exception handling contexts*. The fundamental precept of the Error Handling Aspect design pattern is that advice implement exception handlers and are associated to exception throwing code through pointcuts. Because of this, the only limitation to the types of exception handling contexts that can be defined is the join point model of the employed aspect-oriented language.
- *Pluggability*. An error handling aspect can be easily replaced by another error handling aspect implementing different error handling strategies. This feature makes it easy to reuse the normal code of an application or part of it across different systems. The capability of reusing the normal code separately from the error handling code is desirable in cases where different systems require specific error handling strategies.
- *Textual separation*. Arguably, the textual separation promoted by Error Handling Aspect (and aspect-oriented techniques in general) makes it easier to understand how a system works. The rationale is that developers have to grasp smaller conceptual units that implement specific concerns.

Additionally, Error Handling Aspect has the following ***liabilities***:

- *Textual separation*. In spite of the advantages of textual separation, it makes it difficult for a developer examining the base code of an application to have a complete understanding about system behavior. Getting a complete picture requires an understanding about base code, aspects, and their often non-obvious interactions. In other words, this textual separation does not promote modular reasoning. It is often argued that tool support can help developers in overcoming this problem [Lippert and Lopes 2000], but current tools are still not mature enough.

- *Inapplicability in some scenarios*. Current aspect-oriented languages cannot, in some fairly common situations, simulate the exception handling mechanisms of existing programming languages. The design of the base code must take this into account and avoid these situations. Otherwise, Error Handling Aspect cannot be applied. When extracting error handling code from an object-oriented implementation in order to use Error Handling Aspect, this means that sometimes the system has to be refactored a priori, before the exception handling can be "aspectized". This subject is further discussed in the third and last items of the next section.
- *Limited integration with checked exceptions*. In languages that use checked exceptions, a method is required to either handle all the checked exceptions it encounters or explicitly declare those it does not in its interface. For example, the Java compiler statically checks whether programs adhere to this rule and complains if they do not. As a consequence, in these languages, Error Handling Aspect results in programs that are not valid, since the exception handlers are moved from methods to handler advice. Hence, some aspect-oriented languages, such as CaesarJ [Mezini and Ostermann 2003] and HyperJ [Tarr et al. 1999], have an inherently limited applicability for implementing Error Handling Aspect. They can only be used in situations where the "aspectization" of error handling results in programs whose base (non-aspect) code does not violate the language rules for checked exceptions. This is the case, for example, when a handler throws exceptions of the same type as (or a subtype of) the exceptions is catches. In this case, the context method would already indicate in its signature that it throws the exception. AspectJ provides a workaround for this problem called *exception softening*. This language feature makes it possible to suppress the checks conducted by the Java compiler in certain join points. Therefore, the use of Error Handling Aspect in AspectJ requires that almost all exceptions caught by handler advice become unchecked.
- *Increase in the overall program size*. In the early days of AOP, it was often claimed that its use for structuring exception handling code would result in a reduction in application size [Lippert and Lopes 2000]. However, more recent studies [Bartolomei 2006, Castor Filho et al. 2006] have shown that this is only true if error handling code is uniform and context-independent. If exception handling code in an application is non-uniform or strongly context-dependent, reuse of handler code becomes low and the number of lines of code in an application can grow due to the implementation overhead of AOP. Moreover, the number of operations (methods and advice) and components (aspects and classes) will almost always grow due to the use of Error Handling Aspect.

10. Implementation

In this section we discuss some implementation issues of the Error Handling Aspect design pattern. Our discussion revolves around the AOP mechanisms available in the AspectJ language and, consequently, the exception handling model of Java. As pointed out by Kersten [Kersten 2005], production-quality AOP languages and frameworks, such

as AspectJ, Spring AOP[1], AspectWerkz[2], and JBoss AOP[3], are similar in terms of the mechanisms they support.

1. Type of handler advice to use. Handler advice can be of two types: *after* and *around*. Whenever possible, we recommend the use of *after* advice for implementing handlers and clean-up actions, because they are simpler. *After* advice are not appropriate, though, for implementing exception handlers that have a masking behavior [Anderson and Lee 1990]. A handler has a masking behavior if it stops the propagation of the exceptions it catches. In other words, it does not end its execution by throwing an exception, neither the one it caught nor a new one. Several common exception handling idioms have a masking behavior, for example, a handler that logs an exception and ignores it. AspectJ requires that an *after* advice end its execution in the same way as the join point to which it is associated. Therefore, if the code of an *after* advice is executed following the throwing of an exception, the runtime system of the language assumes that the advice ends its execution by throwing an exception as well.

To implement handlers that have a masking behavior, *around* advice are the only possible choice. They are more powerful than *after* advice, but impose a larger implementation overhead. *Around* advice are also useful when an advice emulates the set of exception handlers associated to a single `try` block. In this case, using *after* advice requires much more handwork because each such advice is triggered by only one exception. Thus, the code to check the actual type of an exception, for the purpose of choosing the appropriate handler, has to be written by hand. This behavior is achieved automatically by implementing a `try-catch` block within an *around* advice.

2. Organizing error handling aspects. Various approaches are possible for organizing error handling aspects. Extreme alternatives include: (i) putting all the exception handling code in a single aspect; (ii) creating a separate aspect for each handling strategy; or (iii) creating one error handling aspect for each class implementing exception handling code. In the first case, error handling is contained within a single program unit and it becomes easier to combine similar handlers, as they are all located within the same place. However, for medium or large systems, the aspect might end up bloated and very hard to understand and maintain. The second case seems more reasonable but, in our experience, apparently similar error handling strategies often include subtleties that make it impossible to combine them in a single handler advice. Therefore, it might result in a very large number of very simple error handling aspects. The third case also typically results in a large number of very simple classes, as most classes in a system include only a few exception handlers.

A more moderate approach is to create one handler aspect for each type of exception and include in such aspects all the possible handling strategies for each exception type. This approach is conceptually sound and works well when a system employs a limited number of exceptions and for each such exception there are several possible handling strategies. For applications that use a very large number of exceptions, it does not scale up well. Another reasonable strategy is to create one handler aspect for each package in the application. Based on our experience, this approach scales up well in general.

[1] `http://www.springframework.org/docs/reference/aop.html`
[2] `http://aspectwerkz.codehaus.org/`
[3] `http://www.jboss.org/products/aop`

For a system where other concerns have been aspectized a priori, a feasible strategy is to create one exception handling aspect per aspectized concern. Each organization has pros and cons that revolve around the code size vs. system structuring trade-off. The extreme approaches mentioned in the previous paragraph are usually only beneficial for small systems or systems with very uniform error handling strategies.

It is commonplace for the same handler advice to be associated with different parts of a program. Depending on the way in which handler advice are organized amongst the error handling aspects, this may result in the same advice being necessary in two or more different aspects. The finer the granularity of these aspects, the higher the likeliness of this scenario arising. To avoid duplicating handler advice across different error handling aspects, one can define an abstract aspect from which these aspects inherit. The common advice is then placed in the abstract aspect and bound to an abstract pointcut that is made concrete by the inheriting aspects.

3. Association of handler advice to normal code. The ease of associating a handler advice to exception-throwing code depends on how the handler would be implemented using only the exception handling mechanism of an object-oriented language. In Java/AspectJ, it is straightforward to associate a handler advice to normal code when the advice emulates a `try-catch` block whose `try` part surrounds the entire method body, or a *whole-method* `try` *block*. It is a simple matter of binding the handler advice to the execution of the context method through an `execution` pointcut designator. Arguably, the resulting code is easy to understand and maintain, as it does not depend on the internals of the context method. Moreover, in some cases, due to the limitations of the join point models of existing aspect-oriented programming languages, binding handler advice to finer-grained program elements is not possible. In the rest of this section, we assume that, ideally, the implementation of handler advice should always aim to emulate whole-method `try` blocks.

Albeit easy to implement, whole-method `try` blocks are often the target of criticism [Papurt 1998]. The main argument against this idiom is that it makes it hard for a handler to establish the cause of an error when the same exception can be thrown from more than one place in the code. Therefore, developers often deal with exceptions more locally. It is commonplace for `try-catch` blocks to be tangled within the body of a method, surrounded by code that does not pertain to error handling. It is also a common practice to use nested `try-catch` blocks in order to define multiple exception handling contexts. These scenarios create some complications for the use of Error Handling Aspect. We discuss them in the rest of this item and in the next one, which addresses nesting of `try-catch` blocks.

When a `catch` block ends its execution by throwing exceptions or returning (executing the `return` statement), it is generally easy to implement a corresponding handler advice and associate it to the exception-throwing code. In this scenario, it does not matter if the `try` block surrounds a specific part of the context method or its whole body. After handler execution, control will be passed to whoever catches the exception thrown by the handler (in the former case) or to the calling method (in the latter). Therefore, once an exception is raised by the exception-throwing code, method execution will not be resumed and it is safe to assume that the handler advice has a whole-method `try` block behav-

ior. Care should be taken, however, in order to avoid catching exceptions unintentionally. There are two cases where this solution might not apply: (i) when the same exception can be raised by different points in the same context method and different error handling strategies are applicable for each such point; and (ii) when there are nested `try` blocks. In the situation described by item (i), the handler advice have to be associated to the specific exception-throwing statements. Otherwise, the handler advice would need to include additional logic with the purpose of distinguishing the point in the context method from where a caught exception was raised. Nesting of `try` blocks is discussed in the next item.

The following code snippet shows an example of the aforementioned scenario written in Java and a modified version using a handler advice. Notice that, in the pure Java version, the `try` block could as well surround the whole method body. Assuming that `m()` does not include any other `try-catch` block, the behavior of the program would be the same.

```
// A pure Java implementation.
public void m() throws E2 {
  ...
  try { doSomethingThatThrowsE1();
    ...
  } catch(E1 e) {
    throw new E2(e);
  }
  doSomethingAfterHandlingE1();
}
```

```
// An AspectJ implementation.
// in a class
public void m() throws E2 {
  ...
  doSomethingThatThrowsE1();
  ...
  doSomethingAfterHandlingE1();
}
// in an error handling aspect
pointcut pc() :
  execution(public void m());
declare soft :  E1 :  pc();
after() throwing (E1 e) :  pc() {
  throw new E2(e);
}
```

Exception handlers that do not execute any statement that alters the control flow of a program are called *masking* handlers, because they hide the occurrence of the exception from the rest of the program. A `catch` block that logs an exception and then ignores it is a typical example. When Error Handling Aspect is being introduced in an existing object-oriented system, masking handlers often hinder the use of the pattern because the code that textually follows a masking `catch` block cannot be ignored. The following code snippet presents an example. The three shaded method calls are exception-throwing statements that may raise exception `E` and the `catch` block masks the occurrence of exception `E`.

```
void m(){
  try{
     m1(); //throws E
     m2(); //throws E
     m3(); //throws E
  }catch(E e){ Logger.log(e); }
  doSomething();
}
```

For the example above, associating a handler advice implementing the `catch` block with each exception-throwing statement individually is not an adequate solution.

For example, if we associated a handler advice with the call to method `m2()`, after exception handling the call to method `m3()` would be executed. However, in the original implementation, control should be passed to the statement following the `try-catch` block, the call to `doSomething()`. Binding the handler advice to the execution of context method `m()` is also not adequate. After exception handling, control would return to the caller of `m()`. This implies that the call to `doSomething()` would not be executed. The bottom line is: we would like to associate a handler advice to a block containing more than one statement, just like a `try` block, instead of a single statement or a whole method. Unfortunately, no existing aspect-oriented language includes mechanisms for directly selecting a block of statements. If, nevertheless, it is necessary to transform the `try-catch` block into a handler advice, the code has to be refactored a priori. A possible solution is to extract the code within the `try` block to a new method and associate the handler advice to this new method.

4. Nested `try` ***blocks***. In order to use Error Handling Aspect to implement nested `try-catch` blocks, it is necessary to order the handler advice so that they simulate the hierarchical structure of `try` blocks. In AspectJ, this can be achieved by textually ordering handler advice that are associated to the same exception-throwing statements. The AspectJ weaver considers that the order in which advice appear in the body of an aspect indicates how they are to be woven into the join point. Advice that appear first are more internal. In aspect-oriented languages that include specific constructs to describe the order in which advice are associated to a join point of interest, such as Jasco [Suvee et al. 2003], the order of advice weaving can be indicated directly.

The case where all the handlers in a method either throw exceptions or return does not differ much from the situation described in the previous item. All the handler advice can still be associated to the execution of the context method, but they have to be ordered so as to simulate the nesting of `try` blocks. The following code snippet shows a simple example of nested `try` blocks in Java and a corresponding AspectJ implementation where two handler advice are associated to the same method. The lexical position of the two advice defines the order in which they are woven into the join point that `pc` selects.

```
// A pure Java implementation.
public void m() throws E3 {
  ...
  try { ...
    try { ...
      throw E1;
      ...
    } catch(E1 e) {
      doSomething();
      throw new E2(e);
    } ...
  } catch(E2 e) {
    doSomethingElse();
    throw new E3(e);
  }
}
```

```
// in a class
public void m() throws E3 {
  ...
  throw new E1();
  ...
}
...
// in an error handling aspect
pointcut pc() :
  execution(public void m());
declare soft :  E1 :  pc();
declare soft :  E2 :  pc();
after() throwing (E1 e) :  pc() {
  doSomething();
  throw new E2(e);
}
after() throwing (E2 e) :  pc() {
  doSomethingElse();
  throw new E3(e);
}
```

If any of the exception handlers does something other than throwing exceptions or returning, things get trickier. The same issues discussed above for masking handlers apply and are further complicated by nesting.

5. Exception softening. In languages that use checked exceptions, e.g. Java, it is often necessary to suppress the static checks performed by the compiler, in order to allow the error handling code to be moved to an aspect. In AspectJ, this is achieved by declaring some exceptions to be *soft* in the join points of interest. Exception softening affects not only the softened exception, but all of its subtypes.

An advice associated with a certain join point is implicitly considered part of that join point by AspectJ. Therefore, softening an exception E in an arbitrary join point JP will also soften any exceptions E', subtypes of E, thrown by advice bound to JP. In order to avoid softening exceptions by accident, as much as possible, developers should only soften leaves in the exception type hierarchy. When this is not viable, it is necessary to define an additional advice whose sole responsibility is to extract and throw softened exceptions wrapped within instances of `SoftException`. Such advice must be associated with join points where the exceptions it throws are are not softened. This issue only applies if E' is a strict subtype of E ($E' \neq E$), as it is not necessary to soften E if E' and E are the same exception. The following code snippet presents an example.

```
// in a class
public void m() throws SubTypeE {
  ...
  throw new SuperTypeE();
}
...
// in an error handling aspect
pointcut mHandler() :  execution(public void m());
declare soft :  SuperTypeE : mHandler();
after() throwing (SuperTypeE e) :  mHandler() {
  throw new SubTypeE(e);
```

```
}
after() throwing (SoftException se) throws SubTypeE :
    call(public void m()) {
  throw new (SubTypeE)se.getWrappedThrowable();
}
```

In the example, `SuperTypeE` is a supertype of `SubTypeE`. When an instance of `SuperTypeE` is thrown from within `m()`, the handler advice will catch the exeception, wrap it with an instance of `SubTypeE`, and throw the latter. However, `SuperTypeE` is softened within the execution of `m()`, the join point with which the handler advice is associated. Therefore, exceptions thrown by the handler, instances of `SubTypeE` in the example, will also be softened. This will result in `m()` throwing `SoftException` when it should actually be throwing `SubTypeE`. The second advice in the code snippet, associated to calls to `m()`, solves this problem by extracting the instance of `SubTypeE` from the instance of `SoftException` and throwing the former.

6. Implementing clean-up actions. Usually clean-up actions (`finally` blocks) are implemented using *after* advice. This is an appropriate solution in most of the cases, as the two constructs have similar semantics. There is a situation, however, where the two differ. According to the Java Language Specification [Gosling et al. 1996], if a `finally` block ends its execution with a `return` statement, the method of which it is part will return, independently of whether an exception was thrown or not from the corresponding `try` block. As pointed out previously, *after* advice executed after the throwing of an exception must also throw an exception. Therefore, `around` advice are a better choice for implementing `finally` blocks that return. This discussion also applies to `finally` blocks executing loop-specific commands, such as `break` and `continue`. These cases have some peculiarities, however, and are briefly discussed in the next item.

7. Unsupported error handling strategies. Sometimes handler advice cannot mimic the behavior of regular `try-catch` blocks. This is fairly common when reengineering the error handling code of an existing object-oriented application in order to use Error Handling Aspect. There are three main factors that hinder the use of the pattern: (i) the advice cannot simulate the flow of control of a regular `try-catch` block; (ii) uncaught exceptions in languages that use checked exceptions; and (iii) the exception handler depends on contextual information of the exception-throwing code in a way that the employed aspect-oriented language cannot capture. The second code snippet in Item 3 of this section portrays a situation where it is not possible to apply Error Handling Aspect without first redesigning the normal code. As pointed out previously, in order to simulate the flow of control of a `try-catch` block, aspect-oriented languages would need to support pointcut designators for selecting blocks of code.

In CaesarJ and HyperJ, there are no mechanisms for deactivating the static checks that the Java compiler performs for checked exceptions. Therefore, in these languages, Error Handling Aspect can only be applied directly if the raised exceptions also appear in the `throws` clause of the context method. Otherwise, the program has to be modified in order for the context methods to include the exceptions handled by handler advice in their `throws` clause. This limitation often implies in system-wide modifications that severely counterbalance the benefits yielded by the pattern.

Handlers that depend on the context of the exception-throwing code in certain

ways are also hard to implement as Error Handling Aspects. There are two specific situations that should be avoided at all costs if one intends to use the pattern to structure error handling in an entire application. The first situation occurs when a handler executes loop-specific statements, such as `break` or `continue`. To the best of our knowledge, no existing aspect-oriented language allows an advice to include a loop-specific statement related to a loop defined in the selected join point, i.e., outside of the advice. This is true even in the face of recent proposals for pointcut designators that select loops [Harbulot and Gurd 2006]. The second situation to be avoided is the use of exception handlers that depend on local variables defined by their corresponding context methods. To the best of our knowledge, no current aspect-oriented language supports the implementation of advice that access local variables visible at the join points to which they are associated. Moreover, this is arguably an undesirable feature, as it is a blatant violation of encapsulation.

11. Sample Code

In this section, we present sample code pertaining to the use of the pattern. To make the examples more concrete, we show the application of Error Handling Aspect to some portions of the Eclipse CVS Core Plugin. For each example of pattern use, we also show the original, pure Java, implementation. For completeness, we also present an example of Java code where Error Handling Aspect cannot be applied directly.

The code snippet below shows a situation where it is trivial (and usually beneficial) to apply the pattern. Since the entire body of method `fromString()` is surrounded by a `try` block, the join point of interest is the execution of the whole method. Moreover, the handler throws an exception, which makes it possible to implement it as an *after* advice in the aspectized version (in the bottom part of the code snippet). Another factor that makes this example simple is the type of the exception thrown by the exception-throwing code in the `try` block. Since it is the same as the exception thrown by the `catch` block, it is not necessary to soften it because it is already declared in the interface of `fromString()`.

```
/*** OBJECT-ORIENTED IMPLEMENTATION - ORIGINAL ***/
public class CVSRepositoryLocation extends PlatformObject
    implements ...  {
  public static CVSRepositoryLocation fromString(String location)
      throws CVSException {
    try { return fromString(location, false); // throws CVSException
    } catch (CVSException e) {
      MultiStatus error = new MultiStatus(...);
      ...
      throw new CVSException(error);
    }
  } ...
}
```

The following code presents a possible application of Error Handling Aspect to the example above.

```
/*** ASPECT-ORIENTED IMPLEMENTATION - REFACTORED ***/
public class CVSRepositoryLocation extends PlatformObject
    implements ...  {
```

```
  ...
  public static CVSRepositoryLocation fromString(String location)
    throws CVSException { return fromString(location, false); }
}
public privileged aspect CoreHandler {
  pointcut fromStringEH(String location) :  args(location) &&
    execution(public static * CVSRepositoryLocation.
        fromString(String));
  after(String location) throwing (CVSException e) :
      fromStringEH(location) {
    MultiStatus error = new MultiStatus(...);
    ...
    } ...
  }
}
```

The next example, presented in the code snippet below, is more convoluted. Neither the execution of method `deconfigured()` nor the exception-throwing statements within it are adequate join points for error handling. Because of the combination of a tangled `try-catch` block and a masking handler, simply associating a handler advice to either would result in a program that does not mimic the flow of control of the original program.

```
/*** OBJECT-ORIENTED IMPLEMENTATION - ORIGINAL ***/
public class CVSTeamProvider extends RepositoryProvider {
  public void deconfigured() {
    try {
      // when a nature is removed from the project, notify the
      // synchronizer that ...
      EclipseSynchronizer.getInstance().deconfigure(getProject(),
        null); // throws CVSException
      internalSetWatchEditEnabled(null); // throws CVSException
      internalSetFetchAbsentDirectories(null); // throws CVSException
    } catch(CVSException e) { CVSProviderPlugin.log(e); }
    ResourceStateChangeListeners.getListener().
        projectDeconfigured(getProject());
  } ...
}
```

The following code snippet shows the solution that we employed in order to make it possible to use the pattern. We created a new method, `notifySynchronizer()`, containing part of the code of method `deconfigured()` from the original implementation . The join point of interest for the error handling concern in this case was the execution of this new method. We then moved the exception handling code from method `deconfigured()` to an *around* advice.

```
/*** ASPECT-ORIENTED IMPLEMENTATION - REFACTORED ***/
public class CVSTeamProvider extends RepositoryProvider {
  public void deconfigured() {
    notifySynchronizer();
    ResourceStateChangeListeners.getListener().
        projectDeconfigured(getProject());
  }
  // when a nature is removed from the project, notify the
```

```
  // synchronizer that ...
  private void notifySynchronizer() {
    ...  // contents of the try block from the original version.
  } ...
}
privileged public aspect CoreHandler {
  pointcut notifySynchronizerEH() :
    execution(private void CVSTeamProvider.notifySynchronizer());
  declare soft :  CVSException :  notifySynchronizerEH();
  void around() :  notifySynchronizerEH() {
    try { proceed();
    } catch (CVSException e) { CVSProviderPlugin.log(e); }
  }
}
```

It is subject to debate whether the effect of aspectizing error handling in this example was beneficial or harmful. On the one hand, the new method makes sense by itself. The comment that appears in the `try` block, which refers to the part of the code of method `deconfigured()` that was extracted, clearly showed the intent of the lines that followed it and made it easy to name the new method. As discussed by Fowler [Fowler 1999], the ease of naming a new method created through the "Extract Method" refactoring is a good indicator of whether that method should have been created. On the other hand, the original method now comprises only two statements, the first one a call to the extracted method. This is a localized example of the "Middle Man" [Fowler 1999] bad smell, where method `deconfigured()` has no reason to be because it simply delegates what it should be doing to other methods.

The code snippet below presents an example that includes two complicating factors: (i) the `catch` block performs an assignment to one of the local variables of the containing method; and (ii) the `catch` block is a masking handler that is associated to multiple exception-throwing statements. Besides the complications introduced by the second factor, a handler that performs assignments to local variables is a strong obstacle to aspectization. Even for a simple example such as the one below, moving the error handling code to an aspect is infeasible unless the code is redesigned to remove any assignments to local variables from handlers and clean-up actions. To the best of our knowledge, there are no general solutions to this problem and workarounds involve knowledge of the inner workings of the system. Due to these complications, we do not apply Error Handling Aspect to this example.

```
/*** OBJECT-ORIENTED IMPLEMENTATION ***/
public class FileModificationManager implements
    IResourceChangeListener {
  private boolean isCleanUpdate(IResource resource) {
    if(resource.getType() != IResource.FILE) return false;
    long modStamp = resource.getModificationStamp();
    Long whenWeWrote;
    try {
      whenWeWrote = (Long)resource.
        getSessionProperty(UPDATE_TIMESTAMP); // throws CoreException
      resource.setSessionProperty(UPDATE_TIMESTAMP,
          null); //throws CoreException
    } catch(CoreException e) {
      CVSProviderPlugin.log(e);
```

```
      whenWeWrote = null;
    }
    return (whenWeWrote!=null && whenWeWrote.longValue() == modStamp);
  }
}
```

12. Known Uses

Lippert and Lopes [Lippert and Lopes 2000] were the first to report to a broader audience on the use of AOP to modularize error handling. They applied the pattern to an object-oriented framework called JWAM using an old version of AspectJ. Colyer and Clement [Colyer and Clement 2004] employed Error Handling Aspect to capture data about component failures in a commercial middleware infrastructure. Due to application requirements, they could encapsulate all the error handling strategies of the application within a single abstract aspect, maximizing reuse of handler code.

Soares and coleagues [Soares et al. 2002] used Error Handling Aspect to structure part of the exception handling code in a web-based healthcare information system named Health Watcher. This work distinguishes itself from the ones mentioned above because the authors targeted specifically the exceptions introduced in the system by distribution and persistence concerns. In Health Watcher, these two concerns were implemented as aspects.

Castor Filho et al [Castor Filho et al. 2006] used this pattern to structure error handling in four different systems: (i) a web-based traveller information system; (ii) Java Pet Store[4], a well-known demo for the Java Platform, Enterprise Edition; (iii) the CVS Core Plugin, part of the basic distribution of the Eclipse[5] platform; and (iv) Health Watcher [Soares et al. 2002]). The first three applications were originally object-oriented, whereas the fourth included some concerns that were implemented a priori as aspects. They also empirically analyzed the impact of the pattern in these four systems based on a set of metrics for quality attributes such as coupling, cohesion, and conciseness.

13. Related Patterns

Error Handling Aspect presents some improvements over the Handler pattern, proposed by Garcia and Rubira [Garcia and Rubira 2000]. Handler leverages a meta-object protocol in order to promote a complete textual separation between normal code and error handling code. One of the differences between Handler and Error Handling Aspect is that, in the latter, the use of aspects makes it possible to define arbitrary, both fine- and coarse-grained, exception handling contexts. The only limitation to what can be selected as an exception handling context is the join point model of the employed aspect-oriented language. Moreover, the quantification capabilities of aspect-oriented languages arguably make it easier to localize error handling code within the aspects. Also, using the Error Handling Aspect, the pointcut descriptions explicitly point out the locations where the classes and error handling aspects interact. In a reflective solution, these interactions are intertwined/hardcoded in the method body of meta-objects.

[4] http://java.sun.com/developer/releases/petstore/

[5] http://www.eclipse.org

The Exception Introduction pattern [Laddad 2003] leverages AOP to make new exceptions introduced by aspect-oriented implementations of crosscutting concerns transparent to the base code of an application. The pattern targets languages such as Java, which use checked exceptions, and makes the introduced exceptions temporarily unchecked so that they can be handled where it is more appropriate. Exception Introduction uses Error Handling Aspect to implement the exception handlers for the introduced exceptions.

Many authors [Diotalevi 2004, Laddad 2003, Lippert and Lopes 2000] propose the use of AOP for separating runtime assertion-checking code from the normal code. This pattern can be used in combination with Error Handling Aspect so that both error detection and error handling code become localized within well-defined program units. This combined solution results in normal code that is not cluttered by error detection and handling concerns.

Haase [Haase 2002] presents a comprehensive pattern language comprising eleven idioms to improve error handling in Java applications. We believe that Error Handling Aspect pattern can be combined with this pattern language, at the design level, in order to produce a system that is more flexible and maintainable.

14. Acknowledgements

The authors thank the shepherd for this paper, Robert Hanmer, by the many interesting comments. We are also grateful to the participants of the Writer's Workshop, specially Jorge Ortega-Arjona and Cassiano Becker, for the positive feedback. This work was conducted while Fernando was with the Institute of Computing, State University of Campinas, and supported by FAPESP/Brazil, grant #02/13996-2. He is currently supported by FAPESP/Brazil, grant #06/04976-9. Alessandro is partially supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. Alessandro is also supported by the TAO project, funded by Lancaster University Research Committee. Cecília is partially supported by CNPq/Brazil, grant #351592/97-0, and by FAPESP/Brazil, grant #2004/10663-8.

References

Anderson, T. and Lee, P. A. (1990). *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition.

Bartolomei, T. T. (2006). On modularity assessment of aspect-oriented software. Master's thesis, Kiel University of Applied Sciences, Kiel, Germany.

Castor Filho, F., Cacho, N., Figueiredo, E. M., Ferreira, R. M., Garcia, A., and Rubira, C. M. F. (2006). Exceptions and aspects: The devil is in the details. In *Proceedings of the 14th SIGSOFT FSE*, pages 152–162, Portland, USA.

Chavez, C. (2004). *A Model-Driven Approach for Aspect-Oriented Design*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil.

Colyer, A. and Clement, A. (2004). Large-scale AOSD for middleware. In *Proceedings of AOSD'04*, pages 56–65.

Cristian, F. (1989). Exception handling. In *Dependability of Resilient Computers*. BSP Professional Books.

Cui, Q. and Gannon, J. (1992). Data-oriented exception handling. *IEEE Transactions on Software Engineering*, 18(5):393–401.

Diotalevi, F. (2004). Contract enforcement with aop. IBM DeveloperWorks - http://www-128.ibm.com/developerworks/library/j-ceaop/.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Garcia, A., Rubira, C., Romanovsky, A., and Xu, J. (2001). A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222.

Garcia, A. and Rubira, C. M. F. (2000). An architectural-based reflective approach to incorporating exception handling into dependable software. In Romanovsky, A. et al., editors, *Advances in Exception Handling Techniques*, LNCS 2022. Springer-Verlag.

Goodenough, J. B. (1975). Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696.

Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. Addison-Wesley.

Haase, A. (2002). Java idioms: Exception handling. In *Proceedings of EuroPLoP'2002*, pages 41–70.

Harbulot, B. and Gurd, J. R. (2006). A join point for loops in aspectj. In *Proceedings of AOSD'06*, pages 63–74, Bonn, Germany.

Kersten, M. (2005). Aop tools comparison, part 1: Language mechanisms. AOPWork - http://www-128.ibm.com/developerworks/java/library/j-aopwork1/index.html.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of the 11th ECOOP*, pages 220–242.

Laddad, R. (2003). *AspectJ in Action*. Manning.

Lippert, M. and Lopes, C. V. (2000). A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd ICSE*, pages 418–427.

Mezini, M. and Ostermann, K. (2003). Conquering aspects with caesar. In *Proceedings of the 2nd AOSD*, pages 90–99.

Papurt, D. M. (1998). The use of exceptions. *Journal of Object-Oriented Programming*, 11(2):13–17, 32.

Parnas, D. L. and Würges, H. (1976). Response to undesired events in software systems. In *Proceedings of the 2nd ICSE*, pages 437–446, San Francisco, USA.

Randell, B. and Xu, J. (1995). The evolution of the recovery block concept. In *Software Fault Tolerance*, chapter 1, pages 1–21. John Wiley Sons Ltd.

Soares, S., Laureano, E., and Borba, P. (2002). Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th OOPSLA*, pages 174–190.

Suvee, D., Vanderperren, W., and Jonckers, V. (2003). Jasco: an aspect-oriented approach tailored for component-based software development. In *Proceedings of the AOSD'2003*, pages 21–29.

Tarr, P. L., Ossher, H., Harrison, W. H., and Sutton Jr, S. M. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st ICSE*, pages 107–119.

Applying Scrum and Organizational Patterns to Multi-site Software Development[1]

Lucas Cordeiro[2], Cassiano Becker[3], Raimundo Barreto[2]

[2]Departamento de Ciência da Computação - Universidade Federal do Amazonas (UFAM), Brazil

[3]BenQ Eletroeletrônica S.A, Manaus, Brazil

lcc@dcc.ufam.edu.br, cassiano.becker@benq.com, rbarreto@dcc.ufam.edu.br

Abstract. *This paper describes a pattern language for managing multi-site software projects which aims at minimizing the main problems present on the multi-site software development context. The practices and patterns of the proposed language were first identified from the literature and adapted according to the authors' experience after running some multi-site software projects. This exercise has led to the identification of two new patterns: "Stories Rework Subsystem", and "Plan Bugs On a Sustainable Pace", as well as to an alternative application of the existing "Inversion of Control" pattern to the organizational context.*

Keywords: *Multi-site Software Development, Scrum Agile Methodology, Lean Software Development, Organizational Patterns, Project Management.*

1. Introduction

Large software projects are usually split into components and developed by different teams, in some cases developed at different places. Software development projects, both large and small, have been consistently difficult to control and manage. Recent studies show that an average project take twice as long to do as its initial plans [Schwaber and Beedle 2002]. Communication overhead and effort to create and update documentation could be pointed as major sources of inefficiency behind project failures. Communication overhead is often introduced by a mismatch in the functionalities required by a given component and the way their development is assigned to separate development teams. In this case, a high rate of communication among teams is introduced, as components developed by one team depend on the services provided by components developed by teams located at different places.

Another problem in large software projects is the increased need for communicating requirements with a higher degree of formality. Requirements are essentially written to describe product characteristics that are proposed in response to a set of business needs. However, customers/users are often not completely sure of what they want, and their mind is likely to change during the time the product is being developed. Moreover, external forces such as competitor's products/services may also lead to changes or enhancements in requirements. Still, many details of what must be produced may be found out only during product development. Therefore, the fact that several development teams may be involved in a project with evolving user requirements calls for practices to efficiently manage the project *(team size and location)* and embrace changes *(scope flexibility),* even late in the development process.

Based on this context, we describe in this paper a pattern language composed of Scrum [Schwaber and Beedle 2002], Lean Software Development [Poppendieck and Poppendieck 2003] and Organizational patterns [Coplien and Harrison 2004] applied to the domain of multi-site software development. In our definition, multi-site software development can be described essentially by characteristics as follows: (i) the project is split into components and assigned to different development teams, (ii) teams are physically separated and may be part of different business organizations, (iii) there is a limited number of teams, such that a two-level hierarchy of coordination is sufficient (between two and five in our experience) (iv) teams are able to physically meet at non-prohibitive cost, if required.

The remainder of this paper is organized as follows: Section 2 provides an overview of the Scrum agile methodology and Organizational patterns. Section 3 introduces the structure of a pattern language in which the proposed patterns are included, and shows how these patterns relate to each other. Section 4 describes the proposed patterns and finally, section 5 summarizes this paper and provides goals of further research.

2. A Brief Look at the Agile Method and Patterns

This section looks briefly at the Scrum method and at the Organizational patterns that were used as basis for the pattern language four our multi-site software environment.

2.1. Scrum

Scrum is a simple and straightforward approach to manage the software development process based on the assumption that environmental (i.e. people) and technical (i.e. technologies) variables are likely to change during the process [Schwaber and Beedle 2002]. In order to manage these variables, Scrum employs the empirical process control model which strongly uses a feedback mechanism to monitor and adapt to the unexpected. Scrum is composed of 14 practices and some of its main practices include: **Sprint** practice which is the iteration work organized in 30-calendar-day. The **Sprint Planning** practice that consists of two meetings as follows: In the first meeting, the product backlog which contains a list of features, use cases, enhancements, and defects of the system is refined and re-prioritized by the product owner, stakeholders and goals for the next iteration are chosen. In the second meeting, the Scrum team figures out how to achieve the requests and creates the sprint backlog that contains detailed tasks to be

accomplished in the current iteration. In the **Sprint Review** practice, the Scrum team presents the results obtained at the end of each iteration by showing working software to the product owner, customers and other stakeholders. In the **Daily Scrum** practice, daily meetings are held at the same place and time with special questions to be answered by the Scrum team.

The Scrum process consists of three roles and the responsibility of each role is described as follows: **Scrum master** is the person responsible for ensuring that Scrum values, practices and rules are followed by the Scrum team. He/she is also responsible for mediating between management and Scrum team, as well as listening to progress and removes block points. **Product owner** is the person who is officially responsible for the project. This person creates and prioritizes the product backlog and ensures that it is visible to everyone. He/she is also responsible for choosing the goals for the next sprint and reviewing the system with other stakeholders at the end of every iteration.

Scrum team is responsible for working on the sprint backlog. The amount of work that will be addressed in the sprint is solely up to the team. They must assess what can be accomplished in the sprint during the sprint planning meeting. Therefore, the team has the authority to make most decisions, and ask for any block points to be removed.

2.2. Organizational Patterns

The organizational patterns described by [Coplien and Harrison 2004] can be combined with Scrum agile methods with the purpose of structuring the software development process of organizations. These patterns are split into four different pattern languages as follows: The **project management pattern language** provides a set of patterns that help the organization manage product development, clarify the product requirements, coordinate project's activities, generate system builds, and keep the team focus on the project's primary goals.

The **piecemeal growth pattern language** provides a set of patterns that help the organization define the overall management structure and amount of team members per project, ensure and maintain customer satisfaction, communicate system requirements, and ensure a common vision for all the people involved in the product development team. The **organizational style pattern language** provides a set of patterns that help the organization eliminate project's overhead and latency, ensure that the organization structure is compatible with the product architecture, organize work for developing products with geographically distributed teams, ensure that market needs will be met.

The **people and code pattern language** provides a set of patterns that help the organization define and keep the architecture style of the product, ensure that the architect is materially involved in implementation, and assign feature development to people in nontrivial projects. The **software configuration management pattern language** is not part of the organizational patterns, but was integrated into the proposed pattern language. These patterns were defined by [Berczuk 2002] and they offer patterns that help the development team define mechanisms for managing different versions of the work products, develop code in parallel, and identify what versions of code make up a particular component.

3. The Proposed Pattern Language

As previously said, the proposed pattern language is composed by patterns identified from languages with complementary concerns: the Scrum Methodology, Organizational Patterns, and Software Configuration Management pattern language. Besides these, the authors also identified from their experience the adoption of practices that pointed to two additional patterns. The resulting pattern language diagram is depicted in Figure 1.

From the resulting set of twenty-one patterns, only a subset was elected for a full description. These obeyed the following criteria:

- A pattern of fundamental importance to description of development process from the multi-site and agile aspect ("Surrogate Customer", "Code Line", and "Integration Build").
- A pattern that had not yet been applied to this context before (c.f. "Inversion of Control").
- A proposed new pattern identified by the authors ("Stories Rework Subsystem" and "Plan Bugs on a Sustainable Pace").

Although a greater number of patterns from the mentioned sources could be indeed mapped to the practices in our cases, we restricted the language to the ones which were more illustrative of the agile and multi-site aspects. It should be noted that these patterns are not intended to be exclusive to the multi-site development context, and will occur in many software development efforts.

The six patterns that will be described are depicted in gray in the pattern language diagram (see Figure 1). In the figure, the relationship PatternA→PatternB can be read as "PatternA can exist once PatternB is in place", that is, PatternA will find a proper context for its application once PatternB has been applied. As an example, the "Sprint Planning" pattern, (when the team sits to plan how to fulfill the goals selected for the next iteration), can be applied and really makes more sense once "Scenarios Define Problem" is in place (when the problem or product being targeted has been decomposed in prioritized stories to be worked). In other words, the resulting context once PatternA is applied can be understood as the initial context for PatternB as the arrows are followed. In addition, the connections simply suggest the probability of patterns occurring together.

Traversing the pattern language diagram vertically also provides a hint on the patterns positioning in the flow of development activities. On the top position, the first pattern is the "Work Queue", which describes the initial set of problems and requirements intended to be addressed by the iterative and incremental development effort. Following the arrows downward will present patterns moving into the solution domain, such as structures for the temporal organization in sprints, multi-site team distribution and the adoption of selected configuration management practices. The traversal concludes at the bottom with the "Integration Build" pattern, which will eventually materialize the results of all processes, practices and tools from each different development cycle into a concrete and valid functionality increment. The patterns are described in the next sections, following the sequence that they appear in the diagram.

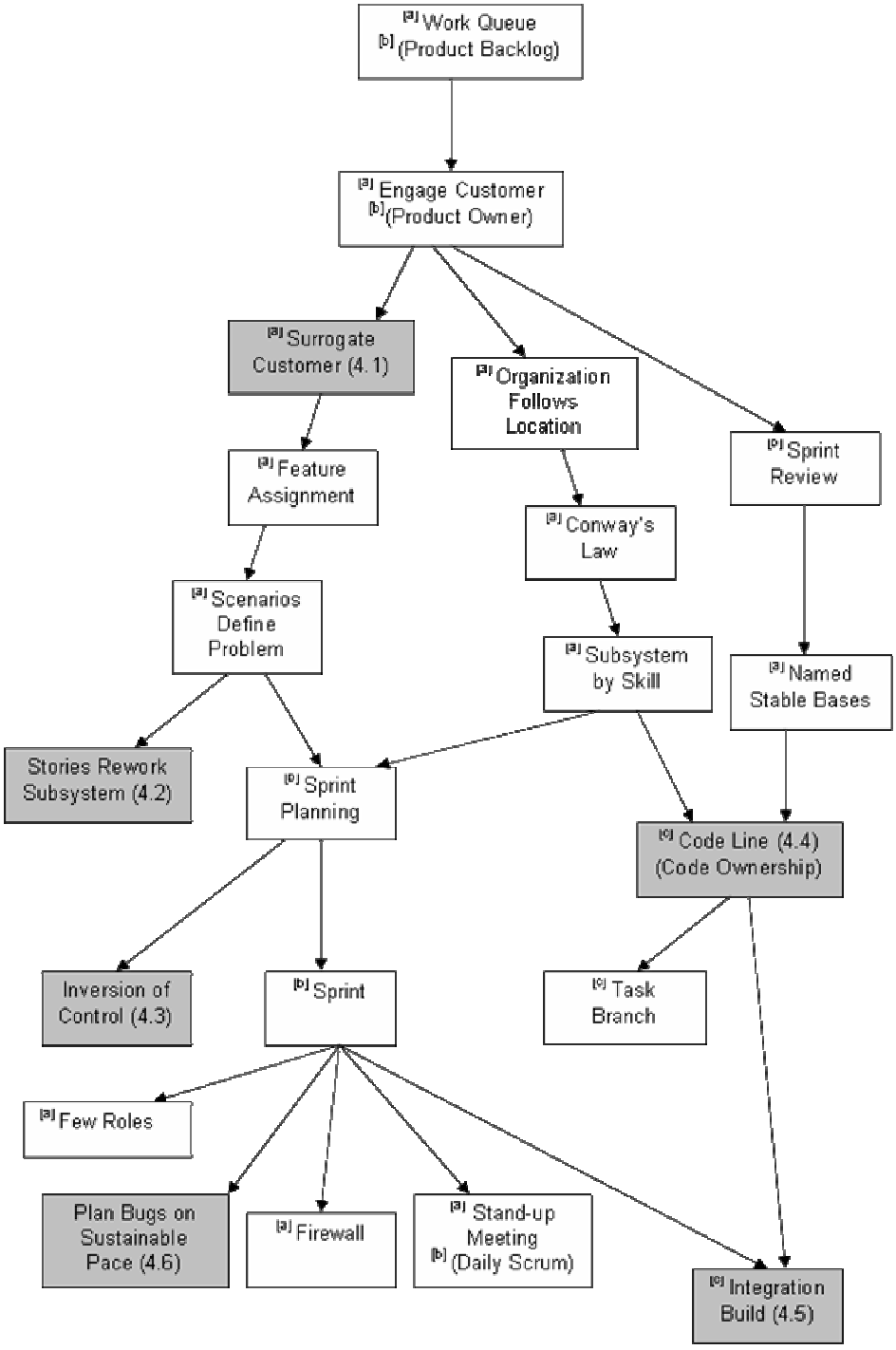


Figure 1. Proposed Pattern Language Structure. Patterns marked with [a] belong to Organization Patterns, [b] to Scrum and [c] to Software Configuration Management Patterns.

4. Patterns for Multi-site Software Development

This section is concerned with describing the patterns presented in section 3 in the following way: the context in which the pattern is applied, the problem that the pattern will solve, the forces that limit the pattern application, the solution of the problem, the related patterns, known uses and finally the resulting context that shows what happens if the solution is applied. The stars after the pattern name indicate the confidence level for the pattern in the multi-site environment. Moreover, we also indicate the pattern origin as follows: "O.P." (Organizational Patterns), "C.M." (Configuration Management Patterns), and "Authors" (the patterns proposed by the authors).

4.1. [**] Surrogate Customer [O.P.]

Alias: Surrogate Product Owner, Feature Leader

Context:

In a project adopting the Scrum methodology, the Product Owner is a central figure. He is the ultimate reference for product content, and his inputs are a major influence on the work performed at each sprint. For larger projects, however, when developed in a multi-site configuration, **a single central Product Owner is not likely to be able to respond to all the demand generated by the distributed development teams** to a satisfactory level of detail.

Problem:

Agile projects rely on close interaction with the customer. Feedback is required at least at each Sprint review and Release planning, but is encouraged to occur throughout the sprint course. With the communication boundaries introduced in multi-site projects, how to maximize information flow and feedback from customers to developers?

Forces:

- The development teams cannot take advantage of constant multi-mode communication channels due to their physical separation.
- Practical solutions usually involve round-trips from requirements to implementation in order to meet time and knowledge constraints.
- Domain knowledge cannot be expected to be fully available in the development team.
- Depending on the project nature (a new solution), a customer might not even exist yet.
- The product owner or customer might not have the necessary available time or detailed knowledge to interact with the development team.

Solution:

Software system functionalities should be split and grouped into features. A "Feature Leader" role is then defined, and will represent the product owner to all teams involved in the implementation of his/her feature set. The set of Feature Leaders can take advantage of closer interaction with the "master" product owner and at the same time will support the remote development teams in specification and decision making in the

sprint planning and throughout its development. The Feature Leader role will influence the development by:

- Defining stories and use case models: Stories and their prioritization are the customer's main contribution to the project in an agile environment. In a multi-site organization, the feature leader will provide more specialized support, in the subsystem or feature level than the product owner.
- Splitting stories: some stories, after their initial estimation, are found to exceed the capacity left for the iteration at hand. The Feature Leader will be able to help establish case by case criteria for decomposing the story (see description in Scenarios Refactor Subsystem).
- Establishing and deciding against trade-offs: when considering different design and implementation for fulfilling a given story, a set of solutions will present different balances on product quality. Although trade-offs might have been laid out clearly at the product-wide level, there might be specific local decisions to consider separately.
- Help establish a domain language: which represents the problem, concepts and solution at hand and which is understandable for both the developer and customer, enabling true two-way communication.
- Providing story acceptance criteria: Defining tests based on real examples for happy-path flows. Additionally, running and looking at partial software releases will usually provide valuable feedback.

Figure 2 describes how a solution for multi-site Scrum teams was proposed in projects the authors participated. In such a set-up, selected team members in the central Product Team were all co-located, and while engaging in ordinary team member activities at that level, acted as Product Owners for the separated subsystem teams.

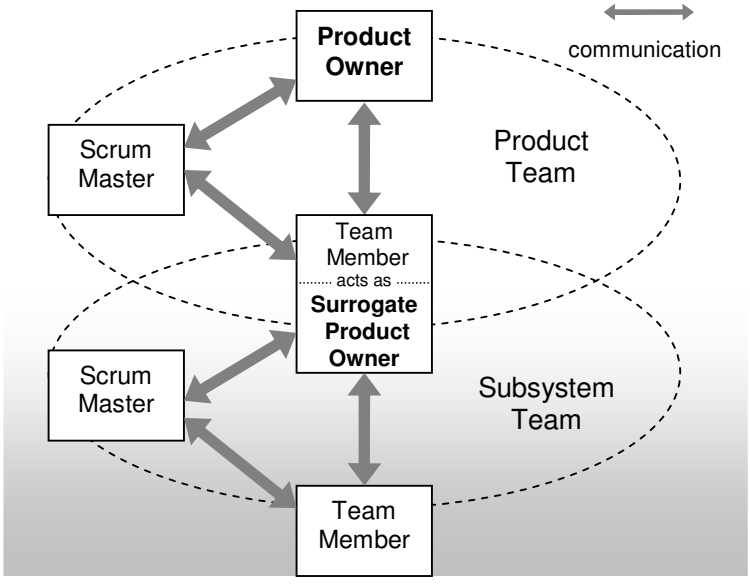


Figure 2. Surrogate Product Owner in a Multi-site Scrum setup

Related Patterns:

The "Product Owner" role, summarized before in this article, and the "Engage Customer" pattern are more general patterns which first described the need for closer interaction and feedback from the customer throughout the entire duration of the development cycle in agile environments.

Resulting Context:

Increased Feature Leader participation raises product perceived integrity, as the stories implemented benefit from a synthesis of the interaction and feedback between the feature leader and the developer.

Developers gain the possibility to discuss and clarify actual design and implementation alternatives in light of product-wide trade-offs. The creation of a common domain language representation is facilitated and is likely to emerge more naturally as a result of the discussions between the Feature Leader and the developer.

However, care must be taken not to over-interact with the development team and cause undesired congestion effects. These would result from an overflow or new requests or changes due to reconsideration, if within a given sprint. In that case, the "Firewall" pattern should be considered.

4.2. [**] Stories Rework Subsystems [Authors]

Context:

In multi-site project, different teams at separated locations will usually define and be assigned different subsystems (see patterns "Conway's Law" and "Organization Follows Location"). For a new story to be fulfilled, usually changes and additional functionalities must be implemented in more than one subsystem (see pattern:" Subsystem by Skill").

Furthermore, when an agile process is applied, stories or feature increments must be integrated and tested in the period **of one time limited iteration.** In the above configuration, *a tension* will generally appear **between the goals posed by a system-wide increment and the goals that each subsystem team** is likely to identify as most important when looking only to their restricted scope.

Problem:

How to coordinate goals and tasks as viewed from the subsystem team standpoint so that the system evolves as a whole and is integrated to fulfill product-wide stories within a given iteration?

Forces:

- An integrated version of working software is expected to be available at the end of each time-limited iteration. Within the course of the iteration, the teams have to make a decision on where to invest their effort at each moment, if on the evolution of the system, on or its stabilization for the integration.
- In a structure defined with "Subsystem per Skills", a separated team will tend to optimize the responsibilities assigned to their components. This will often conflict

with the goals of the whole system for that iteration, which depends on the integration of the functionalities of each subsystem for a given story.

- The problem of suboptimization [Principia] is present: "When you try to optimize the global outcome for a system consisting of distinct subsystems (…), you might try to do this by optimizing the result for each of the subsystems separately. This is called "suboptimization". The principle of suboptimization states that suboptimization in general does not lead to global optimization."
- The more separated or independent the teams working in the system for a given iteration are, more pronounced these forces will be.

Solution:

Introduce the notion to both subsystem and central teams that a **level of rework** should be expected on their subsystems because of the division of the project in sprints. A (perhaps too) simple analogy to this principle is the practice of fencing around a new construction building. The fence will be torn down before the building gets inaugurated, but it is the fencing that allows the construction work to proceed in a controlled way, better integrating the construction to the surrounding environment while work proceeds. Therefore, rework in this case should be understood as activities or code that is produced during the sprint, but which will not be present in the final releases of the product.

From the standpoint of subsystem teams, these activities will usually come in the form of local deviations from what the responsibilities of that subsystem would ideally imply if that subsystem would be the only one being developed. In practice, these local concessions are ultimately caused by the need to converge to integrated stories at the end of each iteration. Examples of activities that could be understood as dimensions of rework are next described:

- Splitting Stories: depending on the story estimates and on the load of each subsystem team in the iteration, a given story can be split to still fit the current iteration. It could be that the amount of work necessary for the split stories is greater than the work for the original [Cohn 2005] provides valuable advice for establishing splitting criteria.
- Splitting Across Data Boundaries: for example, selecting a subset of fields supported for a given form.
- Splitting On Operational Boundaries: for example, selecting a smaller number of operations (CRUD – create, update, delete) or more simple conditions.
- Postponing Cross-Cutting Concerns: for example, leaving out logging, error handling, or security treatment for the iteration being planned.
- Not meeting performance requirements: postponing non-functional requirement aspects.

Because each of these items will probably have to be revisited when the remaining scope is reconsidered, and because there is at least a small volume of code adaptation exclusive to the splitting, these practices might be interpreted as a source of rework. On the other hand, for many larger stories, splitting will be indeed the most efficient way to keep complexity and risk under control.

Coding stubs and mock objects in order to compensate for the absence of subsystem functionality might also be interpreted as unnecessary work for the goals of a given subsystem. Mock objects or stub interface implementations might be interpreted as "inventory" effort, as they will not eventually make it as functionality for that given subsystem. However, when seen from the whole, having such mock objects timely available to other subsystems might be essential for allowing the rest of the system to grow optimally.

Therefore, in order to **enable incremental integration to happen in a multi-site project environment**, the notion that subsystems should expect a level of rework between iterations should be introduced. Project management instruments and measurement tools should be adapted to accommodate for those aspects, for example, acknowledging each local concession causing local under-optimization to the affected team, and focusing measurement on overall progress and performance, rather than local. [Poppendieck 2003] provides good analysis and recommendation on contractual issues that arise in an agile environment.

Related Patterns:

- The "Work Split", "Named Stable Bases" and "Incremental Integration" patterns and the "Thin Slice Story Writing" approach, all describe situations and techniques applicable for incremental and iterative methods that focus on optimizing development output in an environment with complexity and uncertainty
- "Architect Controls Product" has been proposed as a promoter of consensus and conceptual integrity. It acts as a central role that looks at how the subsystems and teams involved in the current iteration can integrate for best fulfilling the goals selected. This integrating role takes the lead for facilitating each subsystem team to see, within their own subsystem, what compromises they can identify so that the stories as a whole are optimized, even if this means subsystem increments depart from ideal. "Surrogate Product Owner" might also fulfill this need, if discussions focus on the splitting of stories between iterations.
- The "Subsystem by Skill" pattern describes a common organizational pattern where "Stories Rework Subsystem" is likely to appear.

Resulting Context:

In a multi-site configuration, having this notion included in the planning and design of solutions at each iteration is a condition for achieving patterns "Named stable bases" and "Incremental integration". Blind denial or avoidance the notion of rework might lead to poor strategies for identifying goals that are manageable within an iteration, and can the prevent system from growing efficiently while maintaining close integration points.

The rework resulting from the compromises taken in each subsystem in a given iteration will have to be considered and re-estimated on the following iterations, reinforcing the need for adaptive planning. Within the limits of a single subsystem and a given iteration, such activities are not generally considered as rework, and are instead understood as regular refactoring.

Also important to take into account, the implications for the measures of performance and quality should be focused first on the feature as a whole, and only secondarily on the performance of each subsystem. Otherwise, subsystem teams will perceive a stronger incentive to optimize their characteristics, which will lead to sub-optimization.

Typical roles that should benefit from the awareness of this pattern are the ones involved in the planning of features at the beginning of each sprint (mostly Scrum Master, Architect and representatives of each distributed team in the planning session). By acknowledging that some level of sub-optimization (in this context that means rework between iterations) is natural and might even be required for the optimization of the system as a whole, conflicting situations might have their causes recognized and discussed more productively.

The more predictable the project is (especially in technology and requirements), the less intermediate integration points will it need, and more work will be able to be performed by teams in parallel, leading to ideally minimum rework. However, for less predicable projects, where a more iterative and adaptive approach is more appropriate, allowing and accounting for rework activities as described in this pattern is likely to lead to increased overall efficiency and lowered risk.

Known uses:

- The lean principle "See the Whole" from [Poppendieck 2003] emphasizes the importance of carefully choosing system-wide variables to measure and optimize, while stating that this will often be accompanied by a relaxation on performance at the local (subsystem) level.
- [Lehman 2000] in his multi-year studies on software evolution proposes eight laws for software evolution planning and management. His "Second Law: Growing Complexity" states that "As an E-type system is evolved, its complexity increases unless work is done to maintain or reduce it" and introduces the notions of Progressive and Anti-regressive work. The rational behind the need for anti-regressive work is closely related to the context and solution here presented.
- The practices of refactoring, as well as the use of stubs and mock objects, are well established in agile software development. They share the notion of work that is revisited or discarded as iterations evolve.

4.3. [*] Inversion of Control [Authors]

Aliases: Don't Call Us We Call You

Context:

In a multi-site organization, communicating and assuring understanding of desired product characteristics to development teams is further complicated by the added communication boundaries. The Product Owner is the ultimate responsible for deciding and prioritizing the stories which make up the solution to the problem. However, depending on the size of the project, a number of details that will eventually affect the perceived integrity of the product are likely to pop up during development, and cannot be expected to be foreseen or discussed with a central product owner timely enough.

If “Surrogate Customer” is applied, as described in this article, the overall team structure is scaled-up and a communication channel for product characteristics can be established between the central product team (see Figure 2) and the subsystem teams.

If a degree of detailed specifications **are expected for each selected feature during each sprint**, this can easily become a bottleneck in the timeframe of a given iteration. The separation of teams occurring in a multi-site environment makes this problem even more important.

Problem:

How to communicate desired product or feature functionality to distributed teams in an agile context, where the selection of stories to be worked is decided at each iteration?

Forces:

- Users and customers are not able to completely state exactly what they want.
- Even if the software developers know all the requirements, many of the details they need to develop the software become clear only as they develop the system.
- Even if all the details could be known up front, it is difficult for a developer to absorb in productive way that many details.
- Even if we could understand all the details, product and project changes occur.

While the software development literature has produced extensive recommendations on the characteristics of well written requirements (concrete, testable, realizable), achieving this in practice is usually easier said than done. Customer state that describing requirements takes too much of their time, and developers often find that they lack in detail or are ambiguous.

Solution:

The pattern “Inversion of Control” has been proposed by [Fowler 2004] as an object oriented design pattern for web application frameworks, in order to eliminate unwanted dependencies in the wiring between framework and application components. In our multi-site and organizational context, the “Inversion of Control” analogy is suggested to describe the way requirements activities can be alternatively handled between the product definition team (Product Owners and it surrogates) and the distributed subsystem development teams.

The solution consists of having the implementing team responsible to continuously refine and revise requirements and solution specification *in the format and level of detail of their preference* (story writing, acceptance tests, schema matrices, verbal and prose descriptions, diagrams). Documentation should only be produced to the level of detail and formality which helps in the communication of the problem and its proposed solution. More recently, developers and analysts have found a reason to move further into each other’s territory in order to cause their language to overlap on top of common domain knowledge representation.

Also, another contribution from agile methods is to promote acceptance tests as the preferred format for requirements. Acceptance test are usually easier to write than requirements because they are based on concrete cases and are written by example,

which also helps eliminate ambiguity. If tests are written in such a way that they allow for automatic execution, they will also provide for instant feedback and progress measurement.

In the "Inversion of Control" pattern, a typical flow of information between the customer and the development team could be described as follows:

1) The Product Owner and its surrogates are initially involved in laying out the initial story description, establishing the prioritization of the quality dimensions, providing examples of happy path tests, and occasionally pointing to existing external standards where applicable.

2) Based on the initial conversation and a subset of the information above, the development team can analyze the problem and write an initial proposal for the solution. In the process of analyzing and proposing a solution, the development team will be in a better position to provide estimates and propose simplifying or splitting criteria in case the estimates values or uncertainty level is too high. If a UI interface prototype has not been given, a sketch can be proposed.

3) The first requirements-analysis-design-validation micro-cycle can be closed a few days after the start of each iteration, when both the developers and product owner surrogates meet to review and discuss with the help of the support material produced.

4) During the course of the sprint, details, alternative flows and corner cases will be identified. The development team is encouraged to constantly feedback its findings and doubts to be revised by the product owners. Each doubt or limitation raised during the sprint refinement can be either accepted as part of the solution space provided or can be fed back to the product backlog in order to be addressed in a further sprint.

Related Patterns:

- "Surrogate Customer", in this article, established the organizational roles on top of which this solution can be applied.
- "Community of Trust" is a pre-condition for the shift in the division of labor in the requirements elicitation and solution creation between product owners and developers to be effective.

Resulting Context:

When "Inversion of Control" is applied to multi-site requirements communication:

- The proposed solution will naturally include the judgment and limitations seen by the implementing team for that iteration (could be reworked on a further it).
- Documentation effort will be prioritized only to the efficient and necessary level of detail and formality which is relevant for the development in the iteration.
- The process of refining the requirements will allow for better estimates and will increase the engagement from the implementing team.
- Early analysis will cause the development team to raise and communicate their external dependencies to other subsystems.

<u>Risks and downsides:</u>

Over reliance on the inversion proposed in this pattern has its danger. The Product Owner role has the ultimate knowledge and responsibility over the problem domain. That is, at least the problem description, major constraints and trade-off dimensions have to be clearly set out by the customer team at the beginning of each iteration, otherwise the expected bootstrapping for the solution might be at risk. As potential risks to the application of this pattern, the following items could be pointed:

1) Having the implementing team to deal with documentation requires analysis capability, which cannot be taken for granted in all teams. In larger projects, however, we felt that a higher number of individuals was willing to step in explore these skills. This was sometimes even felt as a factor of motivation for those individuals inclined.

2) The idea of writing documentation is likely to cause discomfort in an agile environment, and to accommodate for that, the notion of flexibility in both the format and level of detail in the artifacts was introduced. Content produced focused on detailing practical limits, exceptional cases, points of variance and screen refinements; all points that developers felt was key to their technical decisions.

3) The boundary between eliciting requirements and solution providing has to be agreed between product owners (and it surrogates) with developers so that decision making is balanced to the level of detail each side has condition to provide. To the extent of our experience, this balance point varies with team composition, the degree of novelty (uncertainty) of the requirement being worked, and the level of trust between teams. Therefore, for this shifting in balance to be effective, it is necessary that "Community of Trust" [Organizational Patterns] be assured, which is a risk to be analyzed and mitigated in a multi-site (or multi-company) environment.

4.4. [***] Codeline [C.M.]

Context:

Large software systems are usually split into components or subsystems and developed by development teams that may be located at different places. Each development team is responsible for a couple of components or subsystems. They have their own software processes and tools to deal with software configuration management [Louzado and Cordeiro 2005]. Each development team has to implement system tasks (e.g., implement or enhance a requirement and fix a bug) and should not disrupt the activities of other development teams.

Problem:

Components or subsystems making up the system have dependencies, i.e., component B needs the services provided by component A. Changes in the interface or semantics of a component may affect other components of the system. As the components are developed by different development teams, how to keep them synchronized?

Forces:

- Development teams involved in the system development process have different software processes and tools to deal with software configuration management.
- The partition of the system functionalities into components is likely to cause dependencies among components.

- The allocation of these components among different development teams is likely to require a high rate of communication among development teams.
- The work of different development teams must be integrated at least once a week in order to provide feedback on the system functionalities to the customer/user.

Solution:

Components that have dependencies should be allocated to the same development team or at least be allocated to the development teams that are at the same place and/or time zone. Different codelines should be created, one for each development team in order to isolate changes and do not disrupt the work of other development teams. Another development line, called here mainline, should also be created to allow the development teams to integrate their components and generate new system builds. Interface or semantics changes in components must be communicated in advance through the weekly meetings. If describing information is required, then the development team should create an artifact that helps other development teams adapt to the change.

Related Patterns:

- The "Mainline" pattern [Berczuk and Appleton 2002] is applied when there are many people to develop a product and merging must be kept as low as possible. Therefore, it describes a mechanism to keep the number of active development line to a manageable set.
- The "Active Development Line" [Berczuk and Appleton 2002] pattern is applied to developers that want to integrate and test their changes very often during the development process. Therefore, it describes a mechanism to create an active development line by keeping a rapidly evolving development line stable enough to developers.

Known Uses:

- The mainline pattern used by [Louzado and Cordeiro 2005] in a multi-site software development project creates different codelines (one for each partner) and assigns a codeline policy. Moreover, there is a mainline that allows the build manager to integrate the components and generate new system builds.
- An agile codeline management proposed by [Berczuk 2003] creates codeline structures that isolate the components that need to be kept stable from those that are in active development. He also associates policies (how the codeline should be used) for each codeline that is created during the project lifetime.
- The codeline practice proposed by [Wingerd and Seiwald 1998] instantiates this pattern by assigning to each codeline an owner and a policy. They also create a mainline which provides an ultimate destination for changes (e.g., bug fixing, new features) and represents the linear evolution of the software product.

Resulting Context:

Components are grouped into subsystems. Each subsystem is allocated to a development team. Still, there may remain dependencies among subsystems as a higher layer requires services provided by lower layers. Therefore, after creating the codelines, each development team is able to work on its own development line without disrupting the

work of other development teams. The weekly meetings make it possible to synchronize the teams and improve communication. Weekly meetings enables planning which system functionalities, enhancements and bug fixing will be part of the next delivery. On a weekly basis, each development team delivers code to a build manager who is responsible for generating new versions of the system. Each team delivery comes with release notes that states what artifacts have been developed.

4.5. [***] Integration Build [C.M.]

Context:

The software is split into components and developed by teams, at different rates. Each development team is composed by several developers that are responsible for a set of systems requirements. Each developer works on its own *private workspace* and is isolated from the work of other developers [Louzado and Cordeiro 2005]. On the other hand, working software is expected to be delivered on a frequent basis to customers/users. Therefore, a means for integrating code frequently is needed with the purpose of reducing integration problems and providing early feedback to customers.

Problem:

There are several developers working on the production of the software. One developer may depend on the work of another developer. If both developers take long without integrating their code (components) into the product codeline, the number of integration problems might increase substantially. These occur because the system code evolves during the time between the task creation and completion. In this scenario, several tasks are integrated into the main trunk and the code in which the team members started working is different from the code currently available in the main trunk. How to coordinate the contribution from subsystem teams so that changes in one subsystem are integrated in a controlled way, while keeping development pace?

Forces:

- Software integration should occur very often in order to reduce integration problems and provide frequent feedback to customers/users.
- If developers integrate code and generate product builds very often then there is the possibility to spend more time integrating than developing code.
- The most important software functionalities must be implemented and integrated as earlier as possible during the development process in order to provide feedback to customers/users.
- Software development takes months to be accomplished and if it is integrated very often, stable versions of the system should be uniquely identified.

Solution:

Each development team should have a unique window to deliver and integrate the code into the product codeline. For a large system, both daily builds may take place on the codeline of each development team, as well as should one product build per week. For each weekly delivery carried out by the development teams, they should assign a tag in

their codeline and provide the release notes. In addition, they should solve the integration problems that may take place during the integration process.

When different teams share a product codeline, "Integration Build" provides most benefits when performed in a strict sequential mode. That is, only one subsystem team integrates its changes into the main codeline at a time, even if their components logically/physically separated from the remaining subsystems. Only after code increments introduced by one subsystem team are integrated into the product codeline should the next subsystem team by allowed to integrate its contribution. Integration in this sense is typically composed by: (i) check-out (update) of latest version from product codeline (ii) merging it with local changes in the workspace (iii) building and sanity-testing of merged version in the workspace (iv) check-in of integrated version on the product codeline. For the last activity, each development team can appoint an integrator to be responsible for integrating the team's code into the project's mainline (see Codeline pattern). Figure 3 describes a typical workflow with sequential integration.

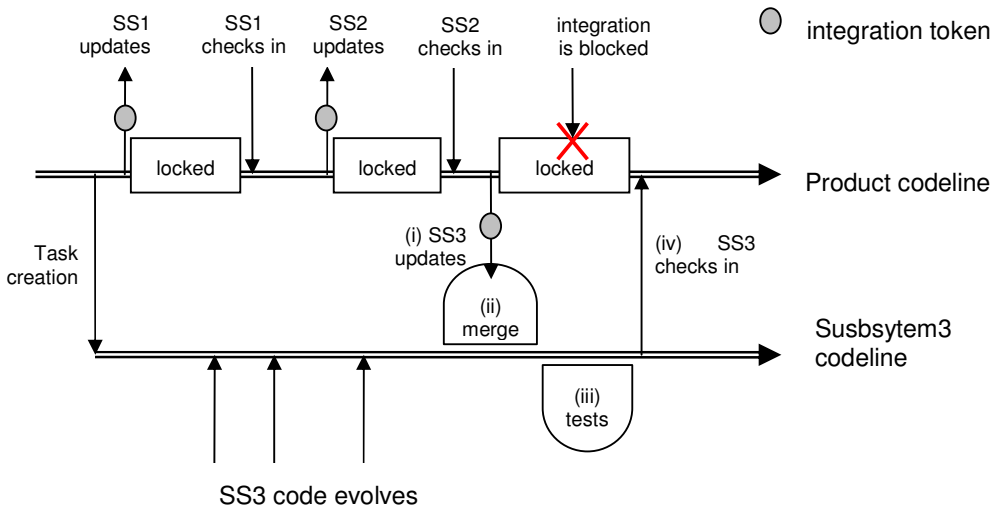


Figure 3. Sequential Integration

Moreover, specific dates/times can be assigned to each development team in order for the integration process to take place. Therefore, this sequential integration always allows a latest version of the system to be regularly identified. It is important to emphasize that the sequential integration does not imply that the development team cannot integrate the latest version of the code in its own codeline.

Related Patterns:

- The "Integration Build" pattern [Berczuk and Appleton 2002] is applied when it is necessary to make sure that components work together in an iterative and incremental approach. Therefore, it allows developers to frequently integrate their code by doing an integration build periodically.
- The "Named Stable Bases" pattern is needed when developers want to integrate software frequently with the purpose of keeping stability and progress. Therefore, it

describes a mechanism to give the stable system a name by which developers can work against.

- The "Build Prototypes" pattern is applied when requirements and design decision must be verified in order to reduce the risk of wasted cost and missed expectations. Therefore, it provides mechanisms to build prototype whose purpose is to help validate requirements and assess risks.

Known Uses:

- The integration build described by [Louzado and Cordeiro 2005] instantiates this pattern by adopting an "integration by stage" approach which provides a progressive integration of the product.
- The incremental integration proposed by [Berczuk 1996] provides a mechanism to allow developers to build the software periodically. This periodic build is also checked for interface compatibility and testing. Therefore, it encourages developers to build from the latest software release and provide time to fix incompatibilities.
- The continuous integration described by [Beck 1999] instantiate this pattern to allow developers to integrate and release code into the repository every few hours. One developer integrates at any time and it takes place only when all unit tests have passed or a smaller piece of the functionality is implemented.

Resulting Context:

If this sequential integration process is adopted in the project, i.e. if one development team has a specific date/time on the week to integrate the code that do not happen at the same date/time of another development team then integration problems may substantially be reduced. Another important benefit is that as the software is built on a weekly basis then it can provide great feedback to customer/users that need working software to clarify system requirements. The software that is produced on a weekly basis receives a unique identification that helps developers identify stable versions of the system. In addition, it allows customers/users to validate only stable versions of the system.

4.6. [***] Plan Bugs on a Sustainable Pace [Authors]

Context:

During the sprint planning, each team member decides which system's functionalities he/she will implement for the next sprint. The system's functionalities are decomposed into activities and are estimated by the team members. At the end of the sprint, the system's functionalities (product backlog items) that were committed to that sprint should be fulfilled by team members in order to be demonstrated to high-level management and customers. The builds generated during the sprint are tested during the same period in order to ensure the product's quality. Therefore, a number of bugs are likely to be found by the test team for the system's functionalities that were implemented in previous or in the current sprint.

Problem:

The test team is constantly testing and identifying bugs, which are added to an existing unsolved bugs list found in previous iterations. Depending on the bugs' criticality, the team members are expected to solve them as soon as possible in order to ensure the product quality. But as team members are committed to the activities of the current sprint, how will they manage to fix these bugs and at the same time ensure that the committed activities will be fulfilled at the end of the sprint?

Forces:

- The global software builds are generated and tested on a weekly basis. The bugs are created and assigned directly to the responsible person through a collaborative development environment tool (CDE).
- The team member responsible for the functionality in which the bug was found should not be interrupted so often because he/she has to complete the activities that were committed to the current sprint.
- The bug that was found at a given functionality might be so important to the customer that it acquires a higher priority than the other activities which are currently running. Therefore, this bug should be fixed as soon as possible by the responsible team member.
- The bug that was found at a given functionality might also impact other important functionalities or might affect the whole system. Therefore, this bug should acquire a higher priority than the other activities which are currently running.
- The development team implements new features in the current sprint and at the same time, it must keep the bug rate as low as possible.

Solution:

Introduce a bug planning process in order to control and manage the product's bugs and avoid project's interruptions. In this process, the test team provides the most critical bugs for each system's component. After that, each feature leader (see Surrogate Customer pattern) reviews the critical bugs, selects them based on the criticality, and informs the project leader. Then the project leader communicates the bugs to be fixed to the development teams. Each development team evaluates the list of bugs and informs to the project leader if the bugs will be fixed in the current sprint. This process is cyclical and its frequency can be higher than the sprint time, as effort for fixing a bug is typically lower than the effort for implementing a new feature. For sprints of one month, the recommended frequency is once a week. Also, as the software builds are generated and tested on a weekly basis (following "Integration Build"), it makes sense for the bug planning process to take place on a weekly basis (sustainable pace). When planning, the bugs, priorities, status, and deadlines should be defined by the project leader or by the person responsible for the feature in which that bug belongs.

The priority may be classified as **critical, high, medium, and low**. The priority level of the bug is according to the feature's importance and the amount of test cases that are blocked because of this bug. In addition, the status of the bug may be classified as **new, started, reopened, resolved, and closed**. After planning the bugs, the leader of each development team involved in the project should analyze if the bugs that are planned can be fulfilled given the workload of its team members. If the bugs can be

fixed without compromising the goals committed to the current sprint then the leader sends an e-mail informing that all the bugs are accepted. Otherwise, he/she commits **only the bugs that his/her team will be able to fix and deliver,** taking into account supporting information as priority, effort and risk. It is of utmost importance that planning and bug-fixing be kept to a sustainable pace during project's sprint. Frequent overtime is usually considered a symptom of serious problems in a team. Therefore, if bugs are planned frequently and according to the team' workload, overtime is substantially reduced. As a result, the correct application of this pattern may contribute to higher code quality as well as happier, more creative, and healthier team.

It is important to emphasize that in case the bug is committed during the bug planning but not delivered on the specified deadline, then the leader of the team should explain the reason why the bug was not fixed and delivered. This situation should not be common, but can take place if the subsystem team does not investigate enough in detail or if it is not able to easily reproduce the bug before it commits to it.

Related Patterns:

- The "Don't Interrupt an Interrupt" pattern can be used when someone is already working in "interrupt mode" on a critical issue of the project. Therefore, this pattern advises that the person who is working on this issue should continue handling it before moving on to the new one.

Known Uses:

- The bug planning described by [Churchville 2006] provides a mechanism to plan bugs in distributed software development projects by defining the risk, frequency, and severity. According to the [Churchville 2006], bugs with high-risk fix, low frequency and severity may not be fixed earlier in the project iterations. Nevertheless, bugs with high severity have always high priority to be fixed. Therefore, for each bug to be fixed, the person who plans the bug should evaluate if the bug fixing provides benefits. On the other hand, the bug fixing should be carried out later in the project.
- The test scripts technique used by [Fowler 2006] represent another approach to plan bugs during the project's iteration. In this scenario, the test scripts are written out before the start of the iteration by a system analyst/tester. These test scripts are written out based on the customer's requirements that should be implemented for a given iteration. During the iteration, regular builds are generated which allows the customer to correct misunderstandings as well as refine their own understandings. As the builds are generated, the customer runs the software and spot the bugs found in the system. After that, the bugs pointed out by the customer are fixed in the same iteration depending on the bug criticality.

Resulting Context:

If the "Plan Bugs on a Sustainable Pace" is adopted, then the goals committed to the sprint by the development teams have a higher probability of being fulfilled. In addition, this bug planning ensures that critical bugs are fixed during the sprint and consequently it keeps the product's quality as high as possible. Therefore, the zero-defect policy is usually not achieved during the sprints. The zero defect policy requires a high effort to fix the bugs which might directly impact the sprint goals. Nevertheless, the software's

bugs should be prioritized according to the features importance, and the decision to work on them should be evaluated in each project's sprint.

Another important result of the application of this pattern is that when the team leader commits the bug then he/she allocates developers to fix it and ensure that the bug will be fixed and delivered as promised at the beginning of the bug planning. Therefore, the development teams concentrate on fixing the bug while carrying out the sprint's activities. Another result is that when a critical bug is found by the test team but not planned, then the development team responsible for that bug is not interrupted to fix it.

5. Conclusions

This paper presented an application of the Scrum methodology, Lean software development, as well as Organizational patterns in the context of multi-site software development. This paper describes the application of six selected patterns, with two of them being proposed as new patterns ("Plan Bugs on a Sustainable Pace" and "Stories Rework Subsystem") and one as an alternative application of an existing pattern ("Inversion of Control"). The first proposed pattern **"Plan Bugs on a Sustainable Pace**" is applied when the project is composed of several project's issues and the level of interruption is very high. Therefore, this pattern describes mechanisms to plan bugs on a sustainable pace in order to control and manage the product's quality and avoid project's interruptions.

The second proposed pattern **"Stories Rework Subsystem"** is applied when development teams are separated by layer (as in pattern "Subsystem by Skill") and stories or feature increments must be integrated and tested within one time limited iteration. Therefore, this pattern provides means to decompose, refine, and prioritize a story in order to fit into one iteration. The pattern **"Inversion of Control"** can be used in a multi-site organization when the need to communicate and assure understanding of requirements is of primary concern. Therefore, this pattern describes a mechanism where the team who will implement the functionality, will be responsible for writing the detailed requirements of that functionality in their preferred format.

As most agile practitioners advocate, we also believe that **co-location is most effective for the majority of software development endeavors**. However, there are still a number of reasons that **require development to be performed in multi-site configuration**, some of them **external to the team's influence**. The main drawback that we found about this configuration is communication overhead. In this case, excessive effort is spent to keep the development teams synchronized and to create and update the documentation. With this paper, we proposed a set of good practices and Software Engineering patterns that we expect can help minimize the main drawbacks present on the multi-site context.

References

Beck, K. (1999). *Extreme Programming Explained – Embrace Change*. Addison-Wesley.

Beedle, M.; Devos, M.; Sharon Y.; Schwaber, K.; Sutherland, J.; (1999). *Scum: An extension pattern language for hyperproductive software development.* In: Harrison, N.; Foote, B.; Rohnert, H. Pattern Languages of Program Design 4. Addison-Wesley.

Berczuk, S. (1996). *Configuration Management Patterns*. In the proceedings of the 1996 Pattern Languages of Programming Conference, PloP'96. Available at http://www.berczuk.com/pubs/PLoP96/. Last visit [3rd June 2007].

Berczuk, S.; Appleton, B. (2002). *Software Configuration Management Patterns.* First Edition, Addison-Wesley.

Berczuk, S. (2003). *Agile Codeline Management*. This paper was published as a StickyMinds Original article.

Bret, T. (2004). *Parallel Development Strategies for Software Configuration Management*. Published at the Summer 2004 issue of Methods & Tools. Available at http://www.methodsandtools.com/mt/download.php?summer04. Last Visit [3rd June 2007].

Churchville, D. (2006). *ExtremePlanner: Agile Project Management for Distributed Software Teams*. http://www.extremeplanner.com/blog/2006/06/biggest-misconception-in-software.html. Last Visit [7th July 2007].

Cohn, Mike (2005). *Agile Estimating and Planning*. Robert Martin Series, Prentice Hall.

Coplien, J. O.; Harrison, N. B. (2004). *Organizational Patterns of Agile Software Development.* First Edition, Prentice Hall.

Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection pattern*. Available at http://www.martinfowler.com/articles/injection.html. Last visit [28th December 2006].

Fowler, M. (2006). *Using an Agile Software Process with Offshore Development.* Available at http://www.martinfowler.com/articles/agileOffshore.html. Last visit [7th July 2007].

Lehman, M. M. (2000) - Rules and Tools for Software Evolution Planning and Management http://www.doc.ic.ac.uk/~mml/feast2/papers/pdf/611_2.pdf

Louzado D. A.; Cordeiro, L. C. (2005). *Aplicando Padrões de Gerência de Configuração de Software em Projetos Geograficamente Distribuídos*. *Proceedings of the 5º Latin American Conference on Pattern Languages of Programming (SugarLoafPlop'2005).*

Poppendieck, Mary and Poppendieck, Tom (2003) *Lean Software Development: An Agile Toolkit*. First Edition, Addison Wesley.

Poppendieck, Tom (2003) *The Agile Customer's Toolkit.* Available at www.poppendieck.com/pdfs/Agile_Customers_Toolkit_Paper.pdf. Last visit [26th December 2006].

Principia Cybernetica. Available at http://pespmc1.vub.ac.be/SUBOPTIM.html. Last visit [26th December 2006].

Schwaber, K., and Beedle, M. (2002)**.** *Agile Software Development with Scrum*. First Edition, Series in Agile Software Development, Prentice Hall.

Wingerd L., Seiwald, C. (1998). *High-level Best Practices in Software Configuration Management*. Springer Berlin, Vol. 1439, pp. 57-66.

Um Padrão para Requisitos Duplicados

Ricardo Ramos[1], João Araújo[2], Ana Moreira[2], Jaelson Castro[1,],
Fernanda Alencar[1], e Rosangela Penteado[3]

[1] Universidade Federal de Pernambuco (UFPE) - Brasil
{rar2, jbc}@cin.ufpe.br, fmra@ufpe.br
[2] Universidade Nova de Lisboa (UNL) - Portugal
{ja, amm}@di.fct.unl.pt
[3] Universidade Federal de São Carlos (UFSCAR) - Brasil
rosangel@dc.ufscar.br

Abstract. With the insights gained with approaches that deal with the information duplication problem, this paper shows the pattern Encapsulated Requirements (*Requisitos Encapsulados)* that describes a solution to duplicated requirements. The pattern is independent of approaches and can be instantiated by any approach that is used to produce a requirements document.

Resumo. Com as lições aprendidas em abordagens que tratam do problema de duplicação de informações, este artigo apresenta o padrão *Requisitos Encapsulados* que descreve uma solução para duplicação de requisitos. O padrão é independente de abordagens, podendo ser instanciado para qualquer abordagem que seja utilizada para produzir um documento de requisitos.

1 Introdução

A duplicação de informações pode acontecer nas várias fases do desenvolvimento de um software. Além de dificultar a compreensibilidade pode existir um aumento no tamanho dos artefatos do sistema e, por conseqüência, do seu custo [Sommerville 2003 e Pressman 2002]. Isso poderia ser evitado caso os projetos fossem estruturados, especificados e modularizados de forma mais eficiente.

Fowler e outros (2000) propõem refatorações (*refactoring,* em inglês) para solucionar as duplicações que ocorrem no código orientado a objetos, enquanto Kiczales e outros (1997) propõem encapsular as informações duplicadas, espalhadas e entrelaçadas em aspectos. Apesar de ter tido no início maior enfoque na implementação, o desenvolvimento de software orientado a aspectos vem sendo utilizado em todos as fases de desenvolvimento [Rashid et al 2003].

Com base nos ensinamentos da programação orientada a aspectos e da refatoração, descrevemos aqui o padrão *Requisitos Encapsulados* para eliminar a duplicação de informações que podem ocorrer em requisitos. O padrão proposto é independente, podendo ser utilizado num documento de requisitos produzido por uma abordagem

qualquer. Neste artigo a estrutura da solução descrita pelo padrão é instanciada para casos de uso.

O padrão aqui apresentado faz parte de um projeto maior cujo objetivo é avaliar a qualidade de documentos de requisitos, encontrando trechos que podem ser melhorados com a aplicação de padrões de requisitos e a utilização de refatorações [Ramos et al 2006a, 2006b e 2006c].

Este artigo segue a seguinte estrutura: a Seção 2 trata da descrição do padrão Requisitos Encapsulados, seguindo o formato sugerido por Appleton (2006) e na Seção 3 são relatadas as conclusões.

2 O Padrão Requisitos Encapsulados

2.1 Propósito

Eliminar duplicações que podem ocorrer em requisitos.

2.2 Problema

Duplicação de informação é um risco ao custo de um sistema [Sommerville, 2003]. Sempre que o mesmo requisito estiver em diversos locais de um documento, criando múltiplas instâncias, a sua manutenção e a sua evolução tornam-se onerosas. Quando um engenheiro de software tiver a necessidade de modificar um requisito duplicado, terá de encontrar todas as suas instâncias e certificar que a mudança é consistente em todo o documento. Adicionalmente, a inserção de requisitos duplicados num documento pode aumentar o seu tamanho, dificultando o seu entendimento e desestimulando a sua leitura.

2.3 Contexto

A duplicação de informações é uma situação que ocorre quando (i) o mesmo requisito está duplicado em diferentes estruturas de um documento de requisitos ou (ii) o mesmo requisito está duplicado na mesma estrutura de um documento de requisitos.

Uma duplicação é contextualizada por ter duas descrições semanticamente idênticas de um mesmo requisito. Porém, as especificações podem estar diferentes sintaticamente, necessitando assim uma avaliação atenta do engenheiro de software para identificar o que é uma informação duplicada. Em alguns casos apenas parte de um requisito pode estar duplicada; nesses casos devemos re-escrever essa parte para melhor clarificar a duplicação.

O padrão se aplica ao contexto de documentos de requisitos estruturados que podem ter sido produzidos por qualquer abordagem de descrição de requisitos. Entretanto, por esta característica de ser independente, é necessário que se instancie este padrão a abordagem que se deseja utilizar.

Jacobson (2005) faz uma ressalva quanto à duplicação de informações no nível de requisitos. Segundo o autor, no contexto de casos de uso, em algumas situações a duplicação é uma forma necessária de reuso de um requisito.

2.4 Forças

As forças que influenciam a utilização da solução descrita pelo padrão são as seguintes:

1. A duplicação de informações aumenta os custos com a manutenção do documento de requisitos e o potencial para inserção de erros. Todas as vezes que uma mudança afetar um desses requisitos duplicados, é preciso modificar todos os locais onde eles aparecem.
2. A utilização de uma única estrutura para encapsular as informações duplicadas contribui para:
   2.1. aumentar a modularização,
   2.2. melhorar a localização,
   2.3. diminuir o tamanho do documento de requisitos.

2.5 Solução

A solução que propomos é independente e tem como intenção poder ser instanciada para qualquer abordagem que produza um documento de requisitos. A descrição do padrão contém as variáveis que devem ser instanciadas:

<requisito> Necessidades básicas do cliente: uma condição ou capacidade requisitada por um usuário, para resolver um problema ou alcançar um objetivo. Em algumas abordagens podem ser expressos por: passos, atividades, tarefa, uma descrição textual entre outros.

<estrutura> Módulos de decomposição utilizados no documento de requisitos. Em algumas abordagens podem ser expressos por: casos de uso, meta (*goal*), tema (*theme*), ponto de vista (*viewpoint*) entre outros.

A solução compreende nas seguintes etapas:

1 - Identificar[1] e analisar o **<requisito>** duplicado. Se o **<requisito>** for similar, mas não exatamente o mesmo, existe a necessidade de separar a parte duplicada. Em alguns casos, a melhor solução é reescrever o **<requisito>** para melhor evidenciar a parte duplicada.

2 - Criar uma nova **<estrutura>** que encapsule o **<requisito>** e nomeá-la.

3 - Selecionar o **<requisito>** identificado na etapa 1 como sendo duplicado.

4 - Adicionar os **<requisitos>** selecionados na nova **<estrutura>**.

5 - Remover os **<requisitos>** das **<estruturas>** originais. Atualizar a numeração, se houver. Criar os apontadores[2] das **<estruturas>** originais para as novas **<estruturas>**.

6 - Averiguar se as **<estruturas>** estão aceitáveis[3] sem os **<requisitos>** que foram removidos e se permanecem com as mesmas funcionalidades dos originais.

7 - Atualizar as referências das **<estruturas>** dependentes.

Se existir a necessidade de manter o relacionamento entre a **<estrutura>** original e a nova, o engenheiro de software deverá providenciar os pontos que façam esse relacionamento. Mecanismos de extensão e inclusão (por ex., para casos de uso), pontos de corte (em desenvolvimento orientado a aspectos) entre outros podem ser criados.

---

[1] Segundo as situações descritas no contexto deste padrão.

[2] Estes apontadores podem ser desde a adição de uma pré-condição ou mesmo uma relação de inclusão até a criação de uma estrutura que indique a composição.

[3] Se não existe uma quebra da seqüência das informações tornando impossível o entendimento de forma coerente.

2.6. Estrutura para Casos de Uso

As atividades descritas nesta Seção são instanciadas da solução independente com a intenção de eliminar a duplicação de requisitos em descrições de casos de uso. A instanciação das variáveis para casos de uso será: <requisito> = atividade e <estrutura> = caso de uso.

1 - Identificar, segundo o contexto, e analisar a **atividade** duplicada. Se a **atividade** for similar, mas não exatamente a mesma, existe a necessidade de separar a parte duplicada. Em alguns casos, a melhor solução é reescrever a **atividade** para melhor evidenciar a parte duplicada.

2 - Criar um novo **caso de uso** que encapsule a **atividade** e nomeá-lo.

3 - Selecionar a **atividade** identificada na etapa 1 como sendo duplicada.

4 - Adicionar as **atividades** selecionadas no novo **caso de uso**.

5 - Remover as **atividades** dos **casos de uso** originais. Atualizar numeração, se houver. Criar os apontadores dos **casos de uso** originais para os **novos casos de uso**.

6 - Averiguar se os **casos de uso** estão aceitáveis sem as **atividades** que foram removidas e se permanecem com as mesmas funcionalidades dos originais.

7 - Atualizar as referências dos **casos de uso** dependentes.

Se existir a necessidade de manter o relacionamento entre o **caso de uso** original e o novo, o engenheiro de software deverá especificar os pontos que façam esse relacionamento. Mecanismos de extensão e inclusão podem ser criados.

2.7. Exemplo

Esta seção apresentará dois exemplos em que ilustramos as duas situações descritas no contexto do padrão. O primeiro exemplo apresenta a duplicação de requisitos em duas estruturas distintas, no segundo a duplicação ocorre na mesma estrutura. Será utilizada, para cada exemplo, a mesma instância do padrão descrita na Seção anterior.

No primeiro exemplo, as figuras 1 e 2 mostram dois casos de uso, `Cadastrar Novo Empregado` e `Cadastrar Novo Produto`, em que as quatro primeiras atividades são semanticamente e sintaticamente idênticas, caracterizando assim os requisitos duplicados em dois casos de uso distintos. Nota-se que no fluxo secundário de atividades, as atividades (4, 4.1 e 4.2) também são idênticas em ambos os casos de uso. Nesta situação, não é necessário uma re-escrita dos requisitos para melhor clarificá-los.

As atividades 5, 6 e 7 do caso de uso da Figura 1 são semelhantes às da Figura 2, porém não caracterizam uma situação de duplicação. Apesar de tratarem do mesmo interesse, o cadastro de informações, cada caso de uso é especifico no cadastro de um conjunto de informações distintas.

Caso de uso – Cadastrar Novo Empregado
Pós-condições:
Novo empregado é registrado
Fluxo principal de atividades:
1. Uma janela de identificação é exibida.
2. O usuário digita o seu nome.
3. O usuário digita sua senha.
4. O usuário é validado.
5. O usuário preenche o formulário de cadastro de empregados com as seguintes informações: nome, login, senha, endereço.
6. O usuário confirma a operação
7. Os dados do novo empregado são guardados.

Fluxo secundário de atividades:
4. O usuário não é validado.
   4.1. Uma mensagem de aviso de senha ou nome de usuário inválido é mostrada.
   4.2. O fluxo recomeça da atividade 1 do fluxo principal.

Figura 1 – Caso de uso Cadastrar Novo Empregado.

Caso de uso – Cadastrar Novo Produto
Pós-condições:
Novo produto é registrado
Fluxo principal de atividades:
1. Uma janela de identificação é exibida.
2. O usuário digita o seu nome.
3. O usuário digita sua senha.
4. O usuário é validado.
5. O usuário preenche o formulário de cadastro de produto com as seguintes informações: descrição, tipo, valor, data de entrega, localização.
6. O usuário confirma a operação
7. Os dados do novo produto são guardados.

Fluxo secundário de atividades:
4. O usuário não é validado.
   4.1. Uma mensagem de aviso de senha ou nome de usuário inválido é mostrada.
   4.2. O fluxo recomeça da atividade 1 do fluxo principal.

Figura 2 – Caso de uso Cadastrar Novo Produto.

A solução para o problema de duplicação de requisitos desses casos de uso (Figuras 1 e 2) seguiu as seguintes etapas como descritas na estrutura do padrão:

1 - As atividades identificadas foram as realçadas nas Figuras 1 e 2. Nesse caso as atividades são semanticamente e sintaticamente idênticas não necessitando ser re escritas.

2 - O caso de uso `Login` (Figura 3) foi criado.

3 - As atividades identificadas na etapa 1 foram selecionadas.

4 - As atividades selecionadas foram adicionadas no caso de uso `Login`. As atividades de 1 a 4 do fluxo principal e as 4, 4.1 e 4.2 do fluxo secundário são basicamente copiadas para o caso de uso `Login`.

5 - As atividades que foram selecionadas nos casos de uso `Cadastrar Novo Empregado` e `Cadastrar Novo Produto` foram removidas. A numeração das atividades que restaram foi atualizada. Foi adicionado como pré-condição nos dois casos de uso (`Cadastrar novo Empregado` e `Cadastrar novo produto`) a descrição da necessidade do usuário estar validado pelo sistema. As Figuras 4 e 5 mostram os casos de uso após a utilização do padrão.

6 - Os casos de uso `Cadastrar Novo Empregado` e `Cadastrar Novo Produto` estão aceitáveis e permanecem com a mesma funcionalidade dos originais.

7 - Neste exemplo não há casos de uso dependentes.

Pós-condições:
Usuário validado.
Fluxo principal de atividades:
1. Uma janela de identificação é exibida.
2. O usuário digita o seu nome.
3. O usuário digita sua senha.
4. O usuário é validado.

Fluxo secundário de atividades:
4. O usuário não é validado.
4.1. Uma mensagem de aviso de senha ou nome de usuário inválido é mostrada.
4.2. O fluxo recomeça da atividade 1 do fluxo principal.

Figura 3 – Caso de uso Login.

Pré-condições:
O usuário deve estar validado. → Apontador ao Login
Pós-condições:
Novo Empregado é registrado
Fluxo principal de atividades:
1. O usuário preenche o formulário de cadastro de empregados com as seguintes informações: nome, login, senha, endereço.
2. O usuário confirma a operação.
3. Os dados do novo empregado são guardados.

Figura 4 – Caso de uso Cadastrar Novo Empregado (após a aplicação do padrão).

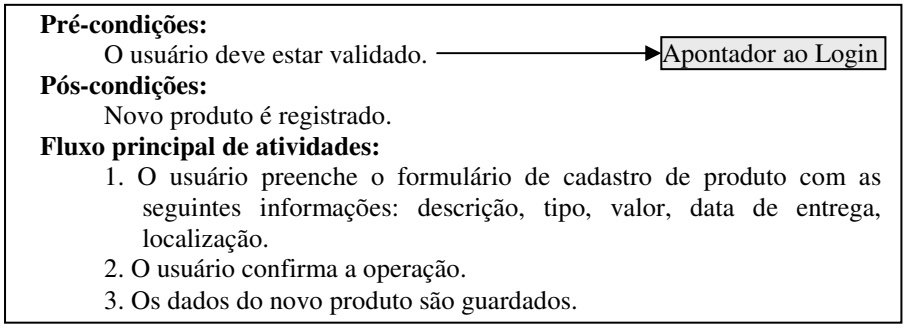


Figura 5 – Caso de uso Cadastrar Novo Produto (após a aplicação do padrão).

O segundo exemplo apresenta uma situação em que a duplicação de informações acontece no mesmo caso de uso. O caso de uso `Iniciar Aplicação`, Figuras 6a e 6b, é o primeiro a ser utilizado pelo usuário de um dado sistema. As primeiras atividades validam o usuário e o atribui um perfil que foi previamente cadastrado pelo administrador do sistema. Esse perfil dá ao usuário permissão e restrições às opções do menu do sistema.

A informação duplicada aparece toda vez que é necessário o sistema mostrar a mensagem de aviso "Esta função não pode ser realizada para esse perfil de usuário" (em destaque na figura 6b). A mensagem deve aparecer toda vez que o usuário escolher uma opção do menu que ele não tenha permissão de utilizá-la devido às restrições do seu perfil.

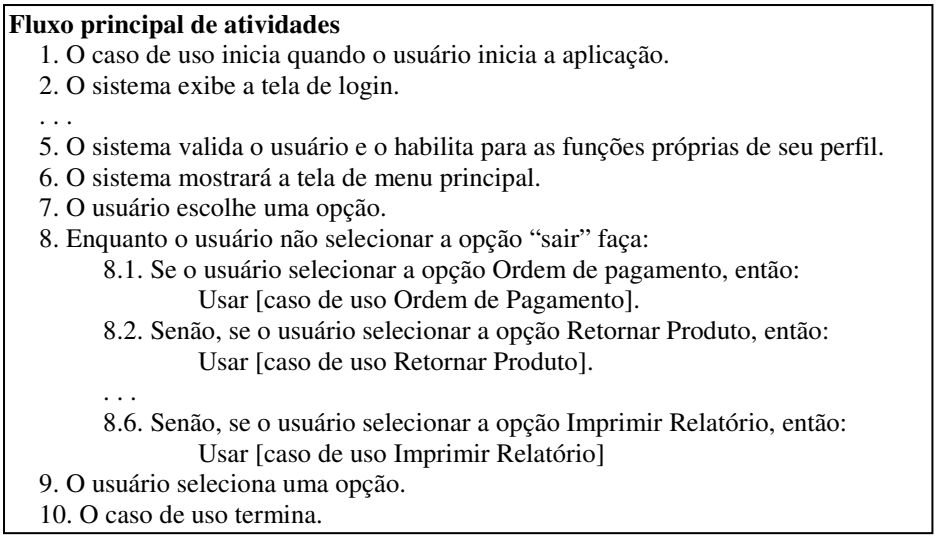


Figura 6a – Caso de uso Iniciar Aplicação (fluxo principal).

Fluxo secundário de atividades

8.1. Se o usuário selecionar a opção Ordem de pagamento, então:

Exibir a mensagem "Esta função não pode ser realizada para esse perfil de usuário"

8.2. Senão, se o usuário selecionar a opção Retornar Produto, então:

Exibir a mensagem "Esta função não pode ser realizada para esse perfil de usuário"

. . .

8.6. Senão, se o usuário selecionar a opção Imprimir Relatório, então:

Exibir a mensagem "Esta função não pode ser realizada para esse perfil de usuário"

Figura 6b – Caso de uso Iniciar Aplicação (fluxo secundário).

A aplicação do padrão para o problema de duplicação de requisitos apresentado na figura 6b seguiu a seguinte seqüência de etapas:

1 - As atividades duplicadas foram identificadas e estão destacadas na figura 6b. Nessa situação as atividades estão no mesmo fluxo e são semanticamente e sintaticamente idênticas não necessitando serem re-escritas.

2 - O caso de uso `Exibir Mensagem` (Figura 7) foi criado.

3 - As atividades identificadas na etapa 1 foram selecionadas.

4 - As atividades selecionadas foram adicionadas no caso de uso `Exibir Mensagem`. A atividade "Exibir a mensagem" que está repetida nas atividades de 8.1 à 8.6 do fluxo secundário é adicionada ao novo caso de uso `Exibir Mensagem`, como sendo a atividade 1 do fluxo principal de execução.

5 - As mensagens que foram selecionadas no caso de uso `Iniciar Aplicação` foram removidas. Foram adicionados, no mesmo lugar onde estavam às mensagens, referências ao caso de uso Exibir Mensagem. A Figura 8 mostra como ficou o caso de uso `Iniciar Aplicação` após a utilização do padrão.

6 - O caso de uso `Iniciar Aplicação` (Figura 8) está aceitável e permanece com a mesma funcionalidade.

7 - Neste exemplo não há casos de uso dependentes, porém caso haja mais algum caso de uso que necessite utilizar a mensagem deve-se fazer uma referência ao caso de uso `Exibir Mensagem`.

Caso de Uso - Exibir Mensagem

Fluxo principal de atividades

1. Exibir a mensagem "Esta função não pode ser realizada para esse perfil de usuário".

Figura 7 – Caso de uso Exibir Mensagem.

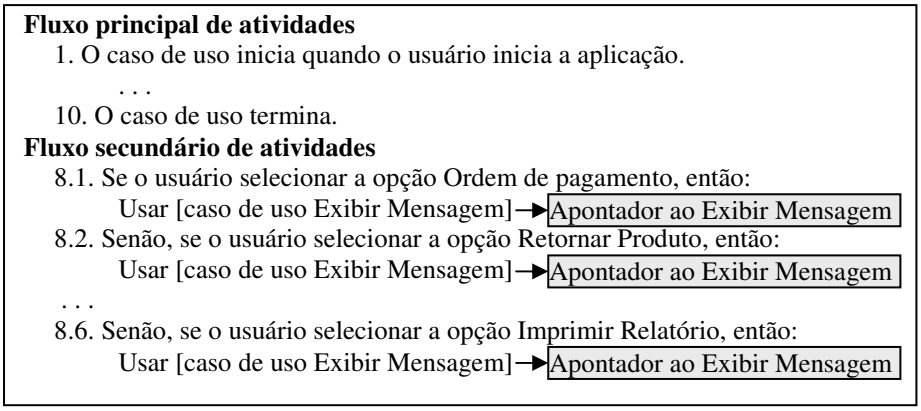


Figura 8 – Caso de uso Iniciar Aplicação *parcial* (após a aplicação do padrão).

2.8. Contexto Resultante

Após a utilização do padrão os requisitos que são identificados como duplicados são encapsulados em apenas uma estrutura. Essa solução tem os seguintes resultados positivos:

1. melhor localização do requisito.
2. diminuição do tamanho do documento de requisitos.
3. facilidade de reuso por outras estruturas do sistema, ou mesmo o reuso do requisito em outros sistemas.

Os Resultados que podem ser considerados negativos na aplicação deste padrão são:

1. o aumento do número de estruturas do sistema, uma vez que se criará uma nova estrutura para cada requisito duplicado.
2. a criação de pequenas estruturas que encapsulam requisitos, com poucas atividades. No caso do documento de requisitos estruturado com casos de uso pode-se utilizar a solução descrita pelo padrão *Large Use Case*, proposta por Gunnar, O. e Karin (2004). Este padrão, quando possível, agrupa casos de uso que são pequenos e\ou tem poucas funções juntamente com outros que estão no mesmo contexto.

2.9. Usos Conhecidos

A solução descrita pelo padrão Requisitos Encapsulados para solucionar o problema da duplicação de informações é conhecida em vários contextos.

Fowler e outros (2000) descrevem, em um catalogo, diversos problemas que podem ocorrer em nível de código orientado a objetos, esses problemas são chamados pelos autores de "*badsmells*". Para cada problema também são descritas possíveis soluções com a utilização de refatoração (*refactoring,* em inglês). Uma das soluções dada para o *badsmell* nomeado "Código Duplicado" (*Duplicated Code*, em inglês) é a utilização da refatoração "Extrair Classe" (*Extract Class*, em inglês). Esta solução tem como conceito central encapsular os trechos de código que estão duplicados em apenas uma classe.

Gunnar e Karin (2004) além de descreverem um conjunto de padrões para casos de uso também descrevem boas praticas para evitar a duplicação de informações em casos de uso. Em uma dessas é descrita a pratica de utilizar a generalização entre casos de uso, sempre que possível, para evitar a duplicação de informações. Assim, esta pratica consiste em mover as informações duplicadas para um único caso de uso. A Figura 9 ilustra duas situações: (a) informações duplicadas em 2 casos de uso, `Cadastrar Cliente Físico` e `Cadastrar Cliente Jurídico`, (b) uma possível solução para a situação anterior, em que a generalização da informação duplicada em um único caso de uso (`Cadastrar Cliente`) permite que os dois casos de uso compartilhem desta informação e tratem somente de suas características especificas.

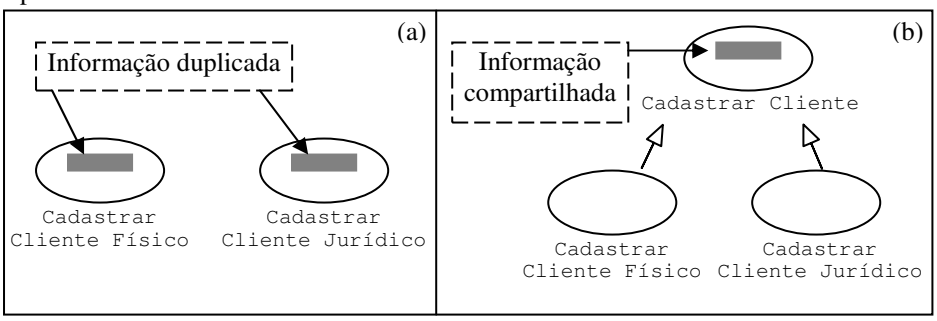


Figura 9 - Informações duplicadas solucionadas com a utilização da generalização.

Rashid, Moreira e Araújo (2004) descrevem um modelo para utilizar orientação a aspectos [Kiczales e outros, 1997] em documentos de requisitos. A eliminação de informações duplicadas é uma entre outras das vantagens promovidas pela utilização deste paradigma. Esta eliminação consiste em identificar a informação que esta duplicada e adicioná-la em um aspecto.

Alencar e outros (2006 e 2007) utilizam os recursos da orientação a aspectos para eliminar informações duplicadas em modelos gerados pelo framework i* [Yu, E., 1995]. Nesta abordagem o objetivo é melhorar a facilidade de entendimento dos modelos gerados pelo i*. Diretrizes são elaboradas para ajudar a utilização da abordagem. Assim como na abordagem de Rashid, Moreira e Araújo (2004) as informações duplicadas são identificadas e adicionadas em um aspecto.

3 Conclusões

Este artigo apresentou um padrão para requisitos chamado *Requisitos Encapsulados* que descreve uma solução para o problema de duplicação de informações no nível de requisitos. O padrão tem uma solução independente que pode ser instanciada para qualquer abordagem que seja utilizada para a produção de um documento de requisitos.

Soluções como a proposta pelo padrão *Requisitos Encapsulados* são recorrentes na abordagem orientada a aspectos, cujo objetivo é a modularização dos interesses (do inglês *concerns*) que podem estar duplicados, espalhados e/ou entrelaçados com outros interesses.

Poucos padrões descrevem soluções para problemas encontrados em descrições de requisitos. A maioria dos padrões encontrados, para requisitos, descreve soluções para modelos, como por exemplo, os padrões descritos por Gunnar e Karin (2004).

Como a descrição de requisitos é usualmente informal, muitas falhas, tais como duplicações de informações, inconsistência de requisitos, pouco reuso e falta de clareza prejudicam o entendimento do documento de requisitos. Isto gera aumento do custo no desenvolvimento de um sistema. Se os erros puderem ser corrigidos na fase de requisitos não serão levados para as fases seguintes do desenvolvimento, diminuindo assim o custo na manutenção destes erros.

Agradecimentos

Este trabalho foi financiado por vários órgãos de incentivo à pesquisa (CNPq Proc. 304982/2002-4 & Proc. 142248/2004-5; CAPES Proc. BEX 3478/05-0; & CAPES/ GRICES Proc. 129/05).

Referências Bibliográficas

Alencar, F., Moreira, A., Araújo, J, Castro, J., Silva, C., Mylopoulos, J.: Towards an Approach to Integrate i* with Aspects. In: Proc. of 8th International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2006), in conj. with CAiSE'06. Luxembourg, June (2006).

Alencar, F., Moreira, A., Araújo, J., Castro, J., Ramos, R., and Silva, C.: Proposal to deal with the Complexity of i* Models with Aspects. In: the First International Conference on Research Challenges on Information Science – RCIS'07. Ouarzazate, Morocco, April, (2007) (to appear).

Appleton, Brad. Patterns and Software: Essential Concepts and Terminology, disponível na WWW na URL: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html - última visita em 29/12. (2006)

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Object Technology Series. Addison-Wesley (2000)

Gunnar, O. e Karin, P. "Use Cases Patterns and Blueprints". In: Addison Wesley Professional - November (2004)

Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley (2005)

Kiczales, G.; Lamping, J.; Mendhekar, A. "RG: A Case-Study for Aspect-Oriented Programming." In: SPL97. Palo Alto Research Center, Technical Report (1997)

Pressman, R. "Engenharia de Software". Makron Books, 5ª edição (2002)

Ramos, R. A., Carvalho, A., Monteiro, C., Silva, C., Castro, J. F. B., Alencar, F., Afonso, R. "Avaliação da Qualidade de um Documento de Requisitos Orientado a Aspectos". In: IX Ibero-American Workshop on Requirements Engineering and Software Environments - IDEAS'06. La Plata, Argentina (2006a)

Ramos, R. A., Araújo, J., Castro, J. F. B., Moreira, A., Alencar, F., Silva, C. "Uma Abordagem de Instanciação de Métricas para Medir Documentos de Requisitos Orientados a Aspectos". In: III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos WASP´2006. Florianópolis, Santa Catarina - Brasil (2006b)

Ramos, R. A., Araújo, J., Castro, J. F. B., Moreira, A., Alencar, F., Silva, C. "Um Modelo de Qualidade para Avaliar Documentos de Requisitos Orientados a Aspectos". In: Desarrollo de Software Orientado a Aspectos, DSOA 2006, Asociado a XV Jornadas de Ingeniería del Software y Bases de Datos. Sitges - Barcelona (2006c)

Rashid, A., Moreira, A. e Araújo, J. "Modularization and Composition of Aspectual Concerns". In: International Conference on Aspect-Oriented Software Development, ACM, Boston, USA (2003)

Sommerville, I. "Engenharia de Software". Addison- Wesley (2003)

Yu, E.: Modeling Strategic Relationships for Process Reengineering. Ph.D. thesis, Department of Computer Science, University of Toronto, Canada (1995).

Analysis patterns for Customer Relationship Management (CRM)

Mei Fullerton and Eduardo B. Fernandez
Dept. of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
meifullerton@yahoo.com, ed@cse.fau.edu

1. Introduction

In today's global trading environment, from the traditional way of selling products or services, to auctioning anything on-line, companies big or small typically have customers or partners from all over the world. Companies need to keep track of their customers, interact with them, prospect potential customers, and try to forecast what their customers will be buying in the future. In fact, not just commercial companies need these functions, but any institution that interacts with individuals or other institutions, such as universities, clubs, or social associations. We refer to all these as *organizations* because they all need similar structure and some related functions to deal with their customers or members. While universities and clubs for example, do not really trade, they need to attract students or members, keep track of their information, and interact with them in many ways. A flexible and robust customer/member data model is needed to capture all this information and accommodate different cultures, organizational structures, and backgrounds.

We describe here some aspects of recording information about customers for an organization in a trading community that sells products or services to its customers, which can be other organizations or individuals (*parties*). A trading community is defined as a group of entities taking part in some type of commerce or exchange. It includes persons and organizations. Entities in a trading community may play roles other than Seller and Buyer, such as Partner, Contact, Distributor, Dealer, Agent, Influencer, etc. Customer relationship has a broader context than classical customers, not only it represents the customer model, it also represents multiple organizations and multiple relationships that exist in a complex matrix-like environment.

There has been much work on related domain-specific areas, such as analysis patterns for Accounting [Fer02], Course Management for educational settings [Yua03], and Reservations [Fer99], but they do not capture a generic model that can be specific to the trading community. There have been also some patterns about specific aspects of CRM, e.g. [Fow97, Hay96, Sil01]. We introduce here the Party Relationship analysis pattern, which captures relationships of parties with other parties, where the party is an organization or an individual. Location aspects of these parties are described in the Party Locations and Contacts pattern. The Customer Relationship Management pattern combines these two patterns and adds account aspects. These patterns are intended for application or database designers.

Section 2 introduces an example which is used for all the patterns discussed here. Section 3 presents the Party Relationship pattern, while Section 4 discusses the Party Locations and Contacts pattern. Section 5 presents the CRM pattern. We end with some conclusions. For conciseness, the first two patterns are presented using simplified templates, while the last pattern uses a complete POSA-like template.

2. Example

Office Enterprise sells office products and services to its customers. It has traditional brick and mortar

retail stores, but it also sells products on line and via mail catalog. Figure 1 shows some of the typical parties and relationships that exist in the Office Enterprise's business domain (the lines with arrows represent possible associations). It has employees. It has customers who can be individuals (B2C customers) or organizations (B2B customers). It has suppliers who manufacture or distribute products and who can also sell directly to the company's customers. It also has partners who may sell products or provide services to Office Enterprise's customers directly. Office Enterprise communicates with its customers and suppliers through contacts and addresses. Each customer or supplier may have more than one contact. They may also have more than one address, such as mailing address, billing address, or shipping address. Last but not least, Office Enterprise has competitors who compete with it for suppliers and customers, and it needs to know about them.

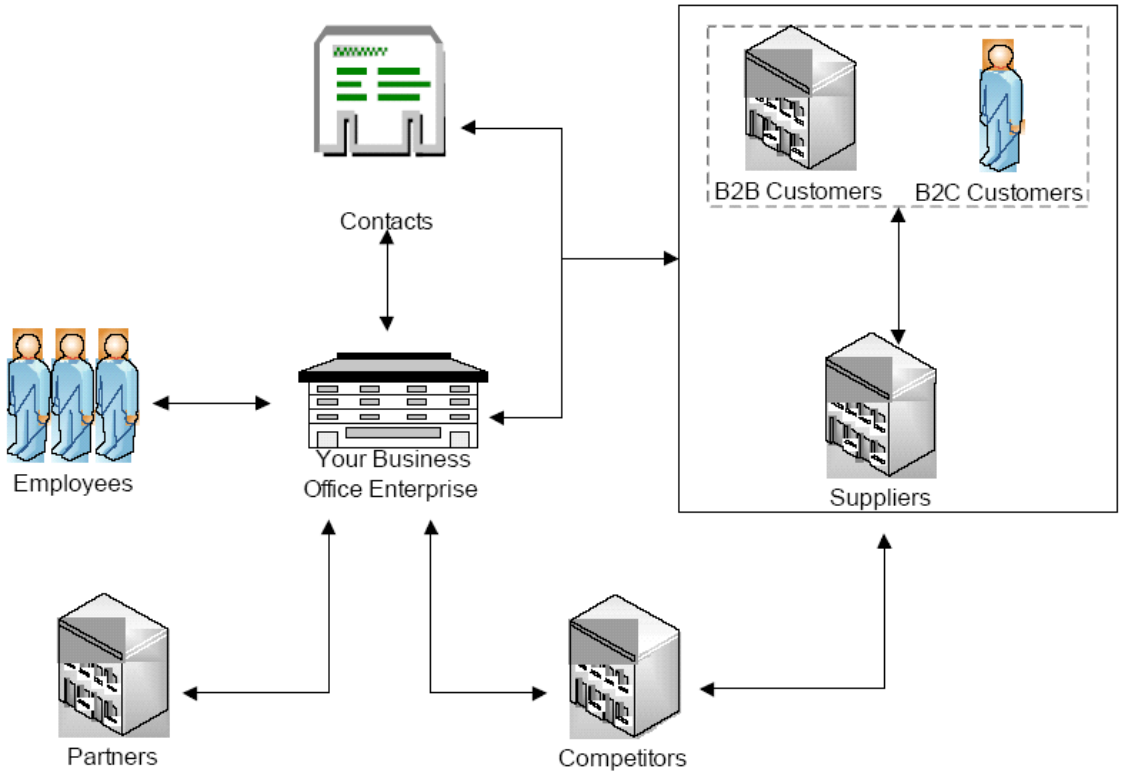


Figure 1 – A company and its relationships

3. Party Relationship Pattern

This pattern describes the parties and the relationships between parties in a trading community or institutions with members, customers, or users

Context

Organizations in a trading community or institutions which have members or customers need to interact in many ways.

Problem

Companies or organizations need to interact with many other organizations or individuals to conduct their business. Those organizations may have complex relationships with the organization and with each other. How do we model the complex relationship between parties so that the company knows the answers to the following key questions at all times: Who are my customers? How are they related to each other? What are their characteristics? Who are my competitors? Who are my partners? Who are my suppliers?

The solution is affected by the following **forces**:

- We need to know how other parties are related to our organization so our interactions with them are appropriate and effective.
- Parties can be individuals or organizations, and we want to consider both types. Otherwise we would exclude potential customers or partners for example.
- An organization is itself a party and can have relationships to itself as well as to other parties.
- Parties can be related to each other in more than one way, maybe in a peer or hierarchical fashion.
- A party can have many relationships with another party, and furthermore, the relationships are dynamic, they can change at any given time.
- Relationships are reciprocal, they can be organization-to-organization, person-to-person, or organization-to-person.
- We need to model inter- and intra- organization relationships, and non-business relationships (Spouse Of or Child Of are examples of non-business relationships). Non-business relationships may be useful for special promotions or advertisement.
- We need to describe any type of relationship, including the ability to capture company branches, competitors, resellers, business partners, etc.

Solution

Define a **Party** as a **Person** or an **Organization** that is of interest in a business context (Figure 2). **Person** is a unique individual, while **Organization** is a legal entity recognized by some government authority, i.e. a branch, a subsidiary, a legal entity, a holding company, etc. **Party relationship** links two Parties to indicate the nature of the relationship between them. This association may also indicate the direction of the relationship, superior or subordinate, as well as their roles in the relationship. For example, in an employee/employer relationship, employee is a role while employer is another role. Some example relationships are: Client of/Contractor to, Supplier to/Distributor for, Seller to/Customer of, Reports to/Manager of, Employer of/Employee of, and Partner of.

Known Uses

SAP's mySAP Business Suite includes a CRM package that handles customers and partners [sap07].

Consequences

This pattern provides the following benefits:

- We can indicate how a party (including our own organization) is related to us and to other parties and describe the type of relationship it has with them.
- Parties can be individuals or organizations.
- By the use of role names in associations we can indicate how parties interact with each other.
- Parties can be related in any way, there is no restriction in the type of relationship.
- We can model business and non-business relationships, as well as inter- and intra-organization relationships (as far as they are named entities).

A possible liability is unnecessary complexity for institutions that have few and simple relationships with other parties.

Related Patterns

This pattern is an extension of the Party Pattern [Fow97]. Fowler describes a party as a person or organization but does not consider how parties are related to each other. However, he uses this concept in several specific relationships, e.g. accountability. An earlier version of this pattern comes from [Hay96], who uses a similar definition and considers reporting relationships between parties. Silverston [Sil01] considers also party relationships of a more general type. All these books use ad-hoc notation, not UML, and don't consider dynamic aspects (we show dynamic aspects in Section 5). A Person pattern [Rod03] emphasizes the roles played by a person in organizations. [Yod02] describes a Party Type pattern that represents types of parties.

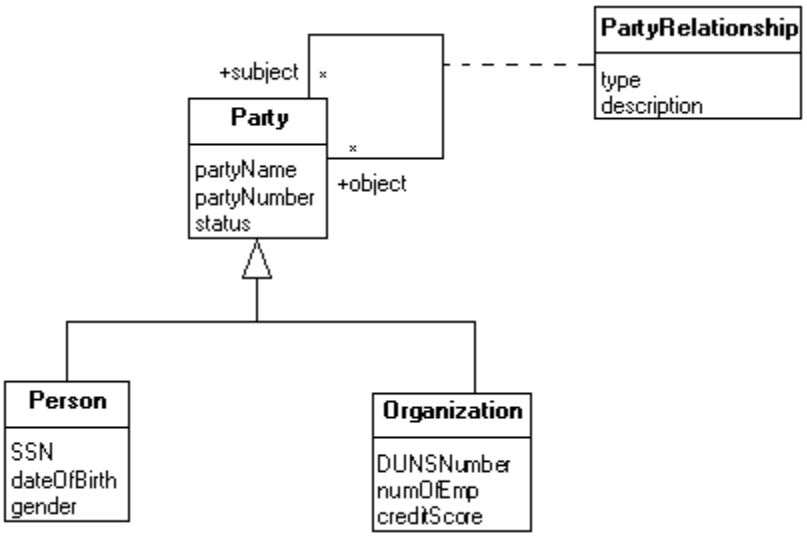


Figure 2 – Class diagram for the Party Relationship pattern.

4. Party Locations and Contacts Pattern

The Party Locations and Contacts pattern describes the places, the contacts, and the associated communication channels in a trading community.

Context

Organizations in a trading community or institutions which have members or customers need to interact in many ways.

Problem

The company needs to know where its parties such as customers, suppliers, partners, competitors are located, and for a specific purpose, who the contacts are, and how to contact them. How do we model the multiple locations (including their purposes), and the multiple contacts for a given party? And how to model multiple communication points for a party or a location for different purposes?

The solution is affected by the following forces:

- Companies or institutions usually have many locations, which are used for different purposes, e.g. sales outlet, customer information, research group.
- Companies or institutions usually have many contacts, intended for different purposes.
- Communication points can be different, based on the purpose of the communication. For example, some points are for email contacts, some are for on-site visiting, some are for EDI (Electronic Data Interchange) communications.

Solution

In the class diagram of Figure 3, every party has many locations, where a **Location** is essentially an address of a physical location. A party can have many locations for different purposes, and a location can be used by many parties. A **Party Site** describes how a location used for that party. **Party Site Use** is the use of a party site (billing, shipping, training) and describes the purpose of that location; for example, mailing address, home address, billing address, or shipping address.

Contact is a person with whom we can communicate for some purpose, whether in-person, over-the-phone, or through other electronic means. A party can have many contacts, and a contact can be used by many parties. A **Party Contact** links the party and contact and indicates that the contact is used for the particular party, as well as the role or function of this contact. A **Communication Point** is an identifier for a typically electronic point of contact, for example a telephone number, an email address, a web URL, an EDI, etc. A Party, a Party Contact, or a Party Site can have one or more communication points for different purposes.

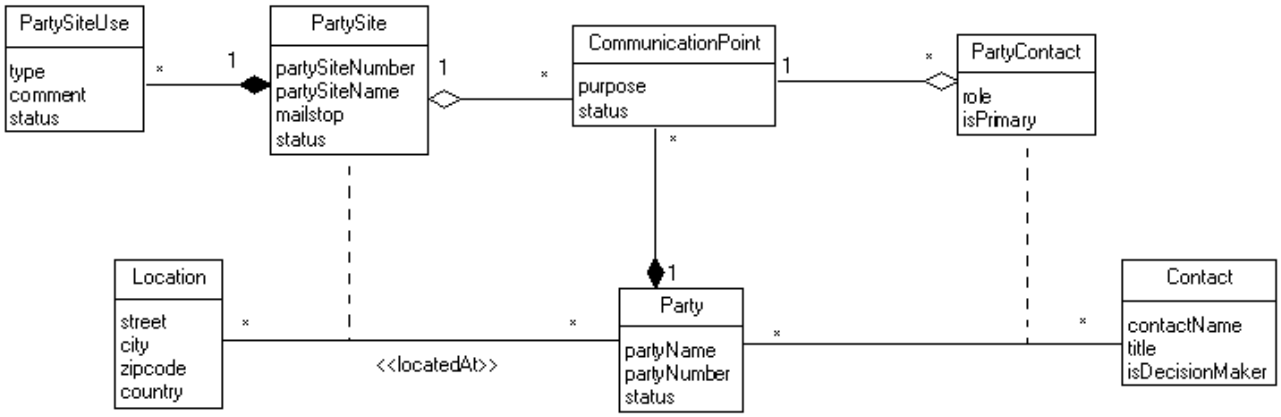


Figure 3 – Party Locations and Contacts Pattern

Known Uses

There are several commercial CRM products that have implemented versions of the Party Locations and Contacts Pattern. For example:

- RightNow [rig07] has a Locator web package to help their customers find products, store locations, and contacts.
- Oracle Customer Data Hub [Ora06].
- Siebel Customer Relationship Management Application [Sie06].

Consequences

This pattern provides the following benefits:

- It indicates the locations of a party and what purpose they serve.
- It indicates the contacts of a party with respect to an institution and their purpose.
- It indicates the communication points needed to reach a party.

Again the solution is rather complex for many applications which don't require so much flexibility.

Related Patterns

This pattern usually complements the Party pattern. For example, the Party pattern in [Hay96] also has the concept of geographic location. Silverston's sales force model has the concepts of contact and contact method [Sil01]. A Contact pattern is described in [Rod03], which describes possible attributes and collaborations of contacts; that is, it can expand some of the details of our pattern. [Yod02] apply the Observation pattern of Fowler [Fow97] to describe aspects of a Party, one of which could be Location.

5. Customer Relationship Management Pattern

This pattern describes the business relationships of an enterprise, considering its interaction with customers, partners, suppliers, and similar entities. It also describes the locations and contacts to apply those relationships and some aspects of their accounts. This pattern is a composite pattern made of the Party Relationship and Party Locations and Contacts patterns.

Context
Organizations in a trading community or institutions which have members or customers need to interact in many ways.

Problem
An enterprise needs to manage all its related parties, such as customers, prospects, suppliers, employees, distributors, and their relationships, so the enterprise can gain valuable insight into their prospect base, understand their needs, increase sales, foster tighter and more profitable relationships with the customers, and make better business decisions. It also needs to keep track of their locations and their accounts with the enterprise. For successful business actions, a company needs to know the answers to questions such as: Who are my customers? What are their preferences? What is the status of their accounts? Where are they located? How to contact them? To be able to answer the above questions at all times, the system needs to understand the organization's customer and other parties' relationships. It also needs to know about their locations and contacts as well as the status of their accounts.

The solution is affected by the following forces:

- We need to know how other parties are related to our organization so our interactions with them are appropriate and effective.
- A party can have many relationships with the organization, and furthermore, the relationships can change at any given time.
- We need to model inter- and intra- company relationships, non-business relationships and user-defined relationships.
- We need to model the capability to offer personalized services or products, each customer has his/her own preferences, and the preferences can change dynamically.
- We need to keep track of the status of their accounts and the type of their accounts.
- Companies or organizations usually have many locations, which are used for different purposes, e.g. sales outlet, customer information, research group.
- Companies or organizations usually have many contacts, intended for different purposes.
- Communication points can be different, based on the purpose of the communication. For example, some points are for email contacts, some are for visiting, some are for EDI communications.
- The complete model should be easy to understand and to implement.

Solution
We combine the two patterns shown earlier. These patterns represent the complex relationships among those entities and model inter and intra company relationships, non-business relationships, and user-defined relationships. We describe the creation and maintenance of the customer information, including organizations, locations, and the network of hierarchical relationship among them. We also keep information about the status of their accounts and their preferences.

Structure
The Customer Relationship Management pattern contains classes from the two previous patterns. Parties represent persons or organizations which have some business relationship with the organization. An **Account** is created once a party makes a purchase or establishes a financial agreement. Accounts also have locations (are assigned to a **site,** where each site can have several uses) and can be related to other accounts.

Figure 4 shows the classes involved. **Party** represents an entity, either a **person** or an **organization**. **Party Relationship** links two Parties to indicate the nature of the relationship between them, regardless of their type. **Location** is essentially an address of a physical location. A **Party Site** uniquely identifies the association between the party and the location and indicates that the particular location is used for that party. **Party Site Use** indicates the use of a party site (billing, shipping, training). **Contact** is a person with whom we can communicate, whether in-person, over-the-phone, or through other electronic means. A **Party Contact** links the party and contact and indicates that the contact is used for the particular party. **Communication Point** indicates an electronic point of contact.

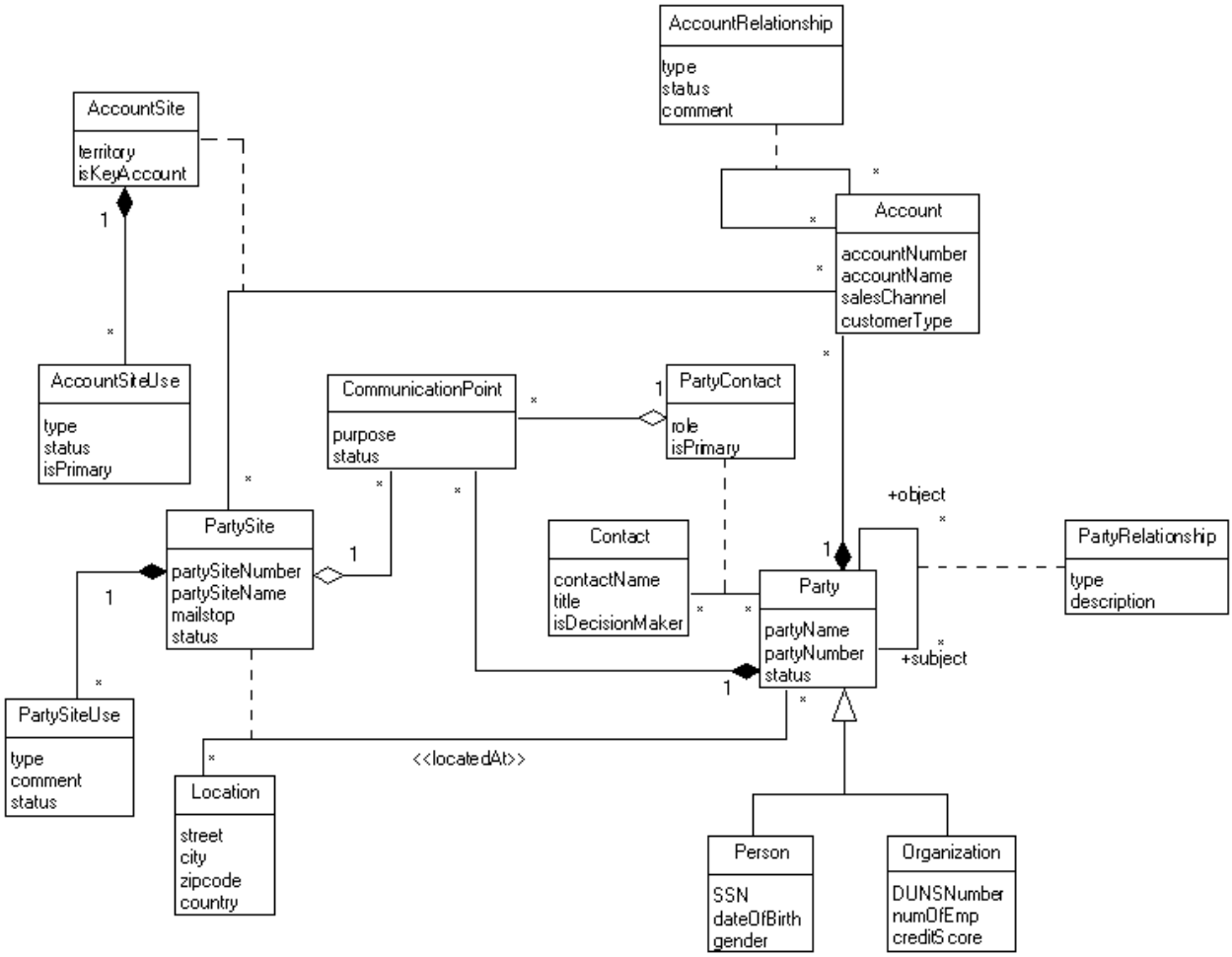


Figure 4 – Class Diagram for the Customer Relationship Pattern

Dynamics

The sequence diagram in Figure 5 shows the use case for opening an account. A person or institution opening an account becomes a party, other entities related to the party such as a location, a contact, and a communication point are created. Finally, the actual account is created. Other use cases include adding a contact to a party, adding a communications point to a party, etc.

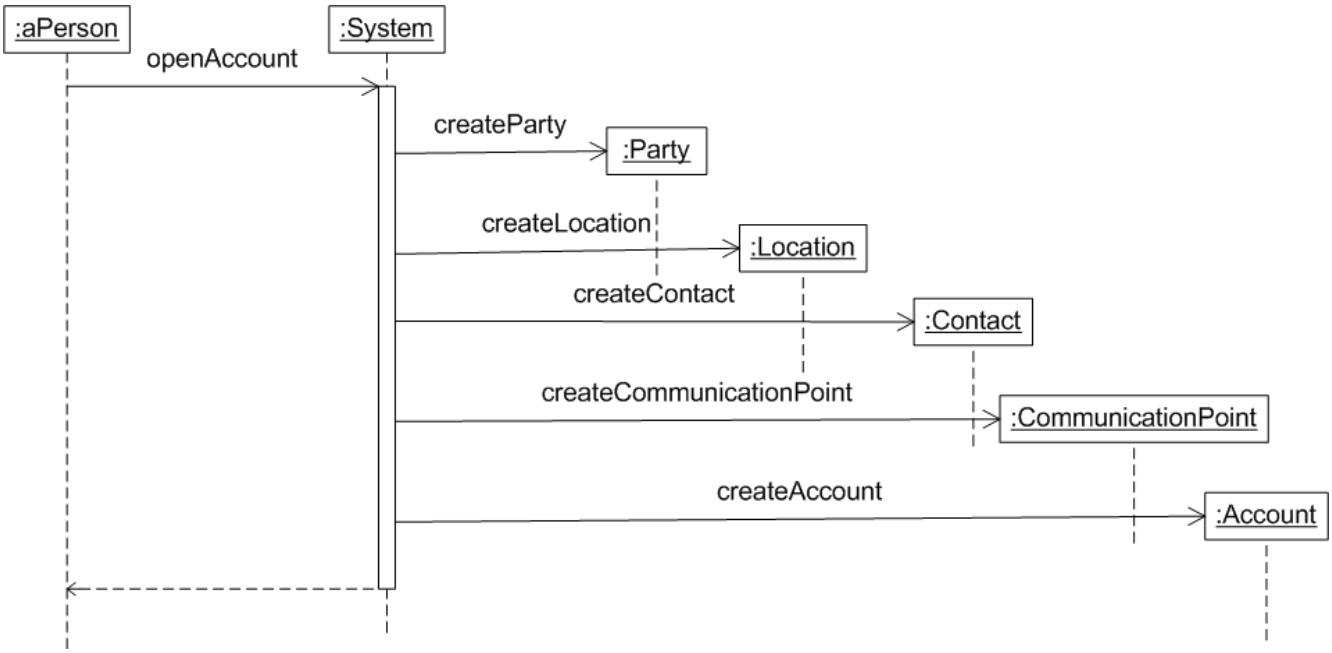


Figure 5 – Sequence Diagram for opening an Account

Known Uses

There are several commercial CRM products that have implemented customer relationship models. For example:

- Oracle Customer Data Hub [Ora06] uses the Trading Community Architecture (TCA). TCA is a common repository for name and address information.
- Siebel Customer Relationship Management Application [Sie06]
- Microsoft CRM software [Mic06]
- SAP's mySAP Business Suite includes a CRM package [sap07].
- SalesForce CRM Unlimited Edition includes all these functions [sal07].
- NetSuite CRM+ includes all these functions [net07].

Consequences

This pattern has the following benefits:

- A Customer Relationship analysis pattern promotes broader reuse; it can be reused in many different domains, such as retail companies, financial institutions, educational, public or government sectors, etc.
- It addresses business, non-business and user-defined relationships.
- It includes the basic information needed for efficient customer relationship management.
- It provides a complete view of a party and all of its relationships with the company, and its relationships with other members of the trading community.
- It keeps the status of accounts.

The pattern has the following liabilities:

- It may be too complex for small businesses.

Related Patterns

- [Cyp05, Fow97, Hay96, Sil01] present several variations on these patterns as well as some complementary patterns and related models.
- The Account Analysis Pattern [Fer02] adds more functions to the accounts shown here, including keeping track of transactions.
- CRM functions interact and may overlap with other business functions such as Order Management, Sales Force Automation, and Marketing, for which there are some patterns or standard models (see [Fow97, Hay96, Sil01]).

6. Conclusions and Future Work

Customer/member information is vital to any organization. We have presented two patterns that handle specific aspects of customer relationships and a composite pattern that combines their functions and adds a few other functions. These patterns can be used to build conceptual application models for this domain.

Business information may be highly sensitive; for example, it contains financial information that is subject to regulations, such as credit card information or purchasing records. Proper security is needed to handle this information. We are working on an extension of this pattern where role rights and other security constraints are superimposed on the functional aspects, according to our secure development methodology [Fer06].

Acknowledgements

We thank our shepherd Tiago L. Massoni for his perceptive and knowledgeable comments that significantly helped improve the quality of the paper. FAU's Secure Systems Research Group (www.cse.fau.edu/~ed) also made valuable comments. The participants in the writers' workshop at SugarLoafPLoP 2007 (Richard Gabriel, Joe Yoder, Ademar Aguiar, Maria Lencastre, Rosana Braga, Jorge Forneron, Jorge Ortega-Arjona, Mark Perry) provided very useful comments.

References

[Cyp05] P. Cyphers, "Trading Community Architecture", http://repo.solutionbeacon.net/SBStandardTCAPresentation2005.pdf

[Fer00] E.B. Fernandez and X. Yuan, "Semantic analysis patterns", *Procs. of 19th Int. Conf. on Conceptual Modeling*, ER2000, 183-195. Also available from: http://www.cse.fau.edu/~ed/SAPpaper2.pdf

[Fer02] E.B.Fernandez and Y.Liu, "The Account Analysis Pattern", *Procs. of EuroPLoP (Pattern Languages of Programs) 2002*. http://www.hillside.net/patterns/EuroPLoP/submissions-2002.html

[Fer06] E.B. Fernandez, M.M. Larrondo-Petrie, T. Sorgente, and M. VanHilst, "A methodology to develop secure systems using patterns", Chapter 5 in "*Integrating security and software engineering: Advances and future vision"*, H. Mouratidis and P. Giorgini (Eds.), IDEA Press, 2006, 107-126.

[Fow97] M. Fowler, *Analysis Patterns-Reusable Object Models*, Addison-Wesley, 1997

[Hay96] D.Hay, *Data model patterns-- Conventions of thought*, Dorset House Publ., 1996.

[Mic06] Extending Microsoft CRM with Reusable Patterns, http://msdn2.microsoft.com/en-us/library/ms913853.aspx

[net07] http://www.netsuite.com/portal/products/main.shtml Accessed February, 2007.

[Ora06] Oracle, The Oracle Trading Community Architecture, http://www.oracle.com/data_hub/cdh.html

[rig07] RightNow Technologies, http://www.rightnow.com/ Accessed February 2007.

[Rod03] A. Rodrigues Silva, "Resources and roles based patterns: The Contact, Person, Organizational Unit and Organization patterns", *Procs. of EuroPLoP 2003*.

[sal07] http://www.salesforce.com/company/ Accessed February 2007.

[sap07] SAP United States, http://www.sap.com/usa/solutions/business-suite/crm/index.epx

[Sie06] Siebel, Customer Relationship Management Applications http://www.oracle.com/applications/siebel.html

[Sil01] L. Silverston, *The data model resource book (revised edition)*, Vol. 1, Wiley 2001,

[Yod02] J. Yoder and R. Johnson. "The Adaptive Object Model Architectural Style", Procs. of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02) at the World Computer Congress in Montreal 2002, August 2002. http://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.htm

[Yua03] X. Yuan and E. B. Fernandez, "An analysis pattern for course management", *Procs. of EuroPLoP'03, 899-907.*

The Parallel Layers Pattern

A Functional Parallelism Architectural Pattern for Parallel Programming

Jorge L. Ortega-Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM
jloa@fciencias.unam.mx

Abstract. *The Parallel Layers pattern is an architectural pattern for parallel programming used when the problem is understood in terms of functional parallelism. This pattern describes a solution in a layered form, in which each layer is composed of two or more components that are able to simultaneously exist and perform the same operation.*

1. Introduction

Parallel processing is the division of a problem, presented as a data structure and/or a set of actions, among multiple processing components that operate simultaneously. The expected result is a more efficient completion of the solution to the problem. The main advantage of parallel processing is its ability to handle tasks of a scale that would be unrealistic or not cost-effective for other systems [CG88, Fos94, ST96, Pan96]. The power of parallelism centres on partitioning a big problem in order to deal with complexity. Partitioning is necessary to divide such a big problem into smaller sub-problems that are more easily understood, and may be worked on separately, on a more "comfortable" level. Partitioning is especially important for parallel processing, because it enables software components to be not only created separately but also executed simultaneously.

Requirements of order of data and operations dictate the way in which a parallel computation has to be performed, and therefore, impact on the software design [OR98]. Depending on how the order of data and operations are present in the problem description, it is possible to consider that most parallel applications fall into one of three forms of parallelism: *functional parallelism*, *domain parallelism*, and *activity parallelism* [OR98]. Examples of each form of parallelism are the Pipes and Filters pattern [OR05], representing functional parallelism; the Communicating Sequential Elements pattern [OR00], as an example of domain parallelism; and Shared Resource [OR03], as an instance of activity parallelism.

2. The Parallel Layers Pattern

The Parallel Layers pattern is an extension of the Layers pattern [POSA96, Shaw95, SG96] *with elements of functional parallelism. Parallelism is introduced when two or more components of a layer are able to simultaneously exist, normally performing the same operation. Components can be created statically, waiting for calls from higher layers, or dynamically, when a call triggers their creation.*

Functional parallelism is the form of parallelism described in terms of a series of simultaneous step ordered operations, applied on ordered data with predictable organization and interdependencies. As each step represents a change of the input for value or effect over time, an amount of communication between components in the solution should be considered. Conceptually, data is repeatedly divided and transformed [CG88, Fos94, Pan96].

Example: Single-Source Shortest Path Algorithm

Search is defined as a systematic examination of a problem space, starting from an initial state and terminating at some final state or states. Each of the intermediate states, between the initial and the final states, can be reached by applying an operation on a given state. This operation is determined by an objective function that assures heading to the final state.

Any search problem can be conveniently represented using a graph. Given a graph is a set of vertices and edges. Each edge has a positive integer weight representing the distance between the vertices it connects (Figure 1). The objective, hence, is to search for the shortest path between the source vertex and the rest of the vertices.

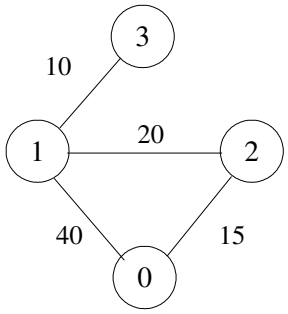


Figure 1. A typical graph

The Single-Source Shortest Path (SSSP) algorithm was originally proposed by Dijkstra, and described later by Chandy and Misra [CM88]. It is an efficient algorithm for exhaustively searching into this kind of graph representation. The SSSP algorithm is applied in cycles. In a cycle, the algorithm selects the vertex with the minimum distance, marking it as having its minimum distance determined. On the next cycle, all unknown vertices (those vertices whose minimum distance to the others has not been determined) are examined to see if there is a shorter path to them via the most recently marked vertex. Algorithmically, the SSSP algorithm reduces the search time to $O(N^2)$ because *N-1* vertices are examined on each cycle. Hence, *N1* cycles are still required to determine the minimum distances.

A sequential approach considers that the graph can be represented by an adjacency matrix G, whose elements represent the weight of the edges between vertices. In this approach two additional data structures are used: a boolean array `Known`, to determine which vertices have had their distance established, and an array `D` to record the most recently established distance between the source and vertices. A function `MinV` returns the vertex with the shortest

unknown distance of the two vertices passed as its arguments. If one vertex is known, the other vertex is returned. It is assumed that `MinV` is not called with two known vertices. The sequential pseudocode is shown in Figure 2.

```
Begin
   For i:=0 to N-1
     Known[i]:=(i=0) // only source vertex is known
   For i:=0 to N-1
     D[i]:= G[0,i] // initial distance of source to vertex
   LastKnown := 0 // only source is known
   KnownCount := 1

   While KnownCount < N
     MinVertex : = 0
     For i:= 1 to N-1 // check the shorter route via last marked vertex
        if Not Known[i]
           D[i] := Min(D[i], D[LastKnown] + G[LastKnown, i])
        MinVertex := MinV(MinVertex, i)
     End For
     // select next vertex to mark known
     LastKnown := MinVertex
     Known [LastKnown] := TRUE
     KnownCount ++
   End While
End
```

Figure 2. Pseudocode for the sequential SSSP algorithm.

However, this algorithm can potentially be carried out more efficiently by:

1. Using a group of parallel components that exploit the tree structure representing the search, and
2. Simultaneously calculating the value minimum distance for each vertex, and only then, computing and marking the overall minimum distance vertex.

Context

Starting the design of a software program for a parallel system, using a particular programming language for certain parallel hardware. Consider the following contextual assumptions:

- The problem to solve, expressed as an algorithm and data, is found to be an open ended one, that is, involving tasks of a scale that would be unrealistic or not cost-effective for other systems to handle. Consider the SSSP algorithm example: since its execution time is $O(N^2)$, if the number of vertices is large enough, the whole computation grows up to an enormous extent.
- The parallel platform and programming environment to be used are known, offering a reasonably level of parallelism in terms of number of processors or parallel cycles available.
- The programming language to be used, based on a certain paradigm, is determined, and a compiler is commonly available for the parallel platform. Many programming languages

have parallel extensions for many parallel platforms [Pan96], as it is the case of C, which can be extended for a particular parallel computer or use libraries to achieve process communication [ST96].

- The main objective is to execute the tasks in the most time-efficient way.

Problem

An algorithm is composed of two or more simpler sub-algorithms, which can be divided into further sub-algorithms, and so on, recursively growing as an ordered tree-like structure until a level in which the sub-parts of the algorithm are the simplest possible. The order of the tree structure (algorithm, sub-algorithms, sub-sub-algorithms, etc.) is a strict one. Nevertheless, data can be divided into data pieces which are not strictly dependent, and thus, can be operated on the same level in a more relaxed order. If the whole algorithm is performed serially, it could be viewed as a chain of calls to the sub-algorithms, evaluated one level after another. Generally, performance as execution time is the feature of interest. Thus, how do we solve the problem (expressed as algorithm and data) in a cost-effective and realistic manner?

Forces

Considering the problem description and granularity and load balance as other elements of parallel design [Fos94, CT92] the following forces should be considered:

- Perform a computation as a tree structure of ordered sub-computations. For example, in the SSSP, each minimum distance for each vertex is calculated using the same operation several times, but using different information per layer.
- Data can be only vertically shared among layers. In the SSSP example, data is distributed through the tree structure, where autonomous operations are carried out.
- The same group of operations can be independently performed on different pieces of data. In the SSSP example, the same operation is performed on each subgroup of data to obtain its minimum distance from the lower layers. So, several distances can be obtained simultaneously.
- Operations may be different in size and level of complexity. In the SSSP example, operations are similar from one layer to the next, but the amount of data processed tends to diminish.
- Dynamic creation and destruction of components is preferred over static, to achieve load balance. For example, in the SSSP example, the creation of new components in lower layers can be used to extend the solution to larger problems.
- Improvement in performance is achieved when execution time decreases. Our main objective is to carry out the computation in the most time-efficient way. The question is: how can the problem be broken down to optimise performance?

Solution

Use functional parallelism to execute the sub-algorithms, allowing the simultaneous existence and execution of more than one instance of a layer component through time. Each one of these instances can be composed of the simplest sub-algorithms. In a layered system, an

operation involves the execution of operations in several layers. These operations are usually triggered by a call, and data is vertically shared among layers in the form of arguments for these function calls. During the execution of operations in each layer, usually the higher layers have to wait for a result from lower layers. However, if each layer is represented by more than one component, they can be executed in parallel and service new requests. Therefore, at the same time, several ordered sets of operations can be carried out by the same system. Several computations can be overlapped in time [POSA96, Shaw95].

Structure

In this architectural pattern, different operations are carried out by conceptually-independent entities, ordered in the shape of layers. Each layer, as an implicit different level of abstraction, is composed of several components that perform the same operation. To communicate, layers use calls, referring to each other as components of some composed structure. The same computation is performed by different groups of functionally related components. Components simultaneously exist and process during the execution time. An Object Diagram, representing the network of components that follows the parallel layers structure is shown in Figure 3.

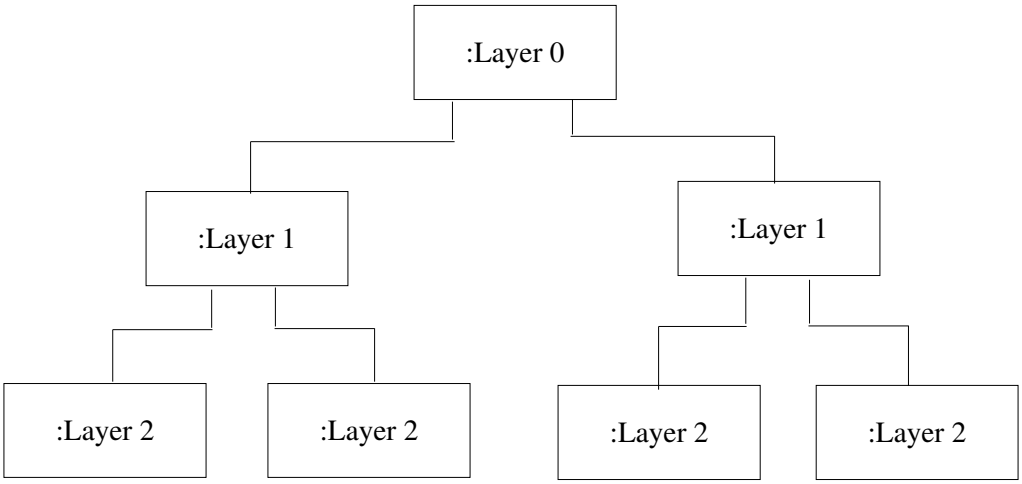


Figure 3. Object Diagram of the Parallel Layers pattern.

Participants

- **Layer component**. The responsibilities of a layer component are to allow the creation of an algorithmic tree structure. Hence, it has to provide a level of operation or functionality to the layer component above, while delegating operations or functionalities to the two or more layer components below. It also has to allow the flow of data and results, by receiving data from the layer component above, distributing it to the layers components below, receiving partial results from these components, and making a result available to the layer

component above. Each component is independent from the activity of other components. This makes it easy to execute them in parallel.

Dynamics

As the parallel execution of layer components is allowed, a typical scenario is proposed to describe its basic run-time behaviour. All layer components are active at the same time, accepting function calls, operating, and returning or sending another function call to other components in lower level layers. If a new function call arrives from the client, a free element of the first layer takes it and starts a new computation.

As stated in the problem description, this pattern is used when it is necessary to perform repeatedly a computation, as series of ordered operations. The scenario presented here takes the simple case when two computations, namely **Computation 1** and **Computation 2**, have to be performed. **Computation 1** requires the operations *Op.A*, which requires the evaluation of *Op.B*, which needs the evaluation of *Op.C*. **Computation 2** is less complex than **Computation 1**, but requires to perform the same operations *Op.A* and *Op.B*. The parallel execution is as follows (Figure 4):

- The **Client** calls a component **Layer A1** to perform **Computation 1**. This component calls to a component **Layer B1**, which similarly calls a component **Layer C1**. Both components **Layer A1** and **Layer B1** remain blocked waiting to receive a return message from their respective sub-layers. This is the same behaviour than the sequential version of the *Layers* pattern [POSA96].

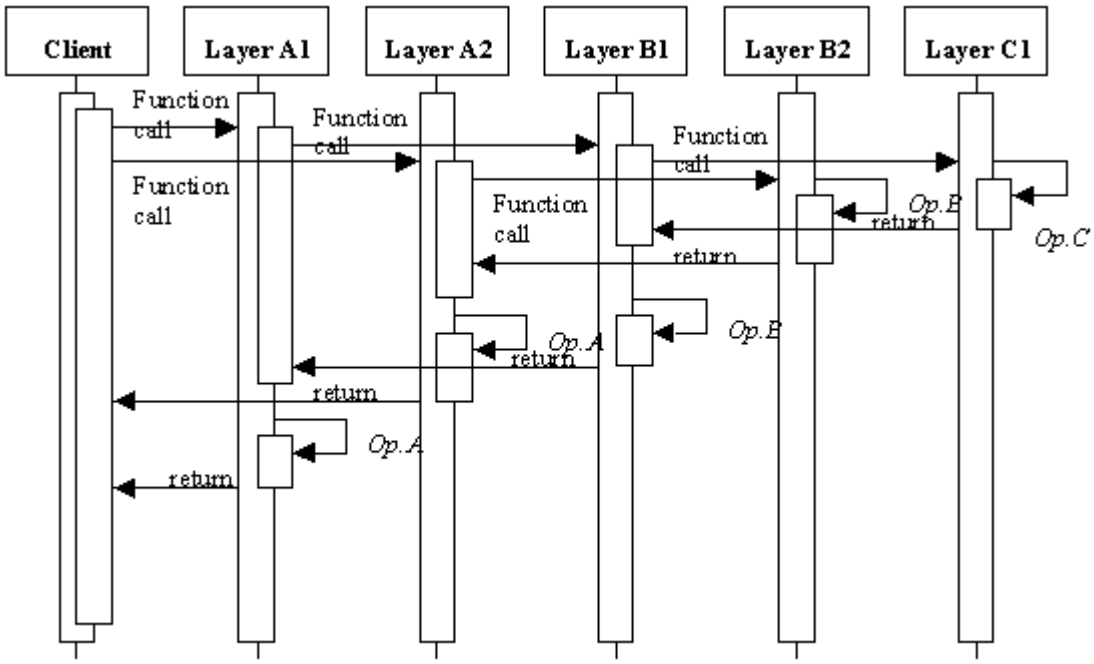


Figure 4. Interaction Diagram of the Parallel Layers pattern.

- Parallelism is introduced when the **Client** issues another call for **Computation 2**. This cannot be serviced **by Layer A1**, **Layer B1** and **Layer C1**. Another instance of the component in Layer A, called **Layer A2** - that either can be created dynamically or be waiting for requests statically - receives it and calls another instance of Layer B, **Layer B2**, to service this call. Due to the homogeneous nature of the components of each layer, every component in a layer can perform exactly the same operation. That is precisely the advantage of allowing them to operate in parallel. Therefore, any component in Layer B is capable to serve calls from components in Layer A. As the components of a layer are not exclusive resources, it is in general possible to have more than one instance to serve calls. Coordination between components of different layers is based on a kind of client/server schema. Finally, each component operates with the result of the return message. The main idea is that all computations are performed in a shorter time.

Implementation

An architectural exploratory approach to design is described below, in which hardware-independent features are considered early, and hardware-specific issues are delayed in the implementation process. This method structures the implementation process of parallel software based on four stages [OR98]. During the first two stages, attention is focused on concurrency and scalability characteristics. In the last two stages, attention is aimed to shift locality and other performance-related issues. Nevertheless, it is preferred to present each stage as general considerations for design instead of providing details about precise implementation. These implementation details are pointed more precisely in the form of references to design patterns for concurrent, parallel, and distributed systems of several other authors [Sch95, Sch98a, Sch98b, POSA00].

1. *Partitioning.* Initially, it is necessary to define the basic Layer pattern system which will be used with parallel instances: the computation to be performed is decomposed into a set of ordered operations, hierarchically defined and related, determining the number of layers. Following this decomposition, the component representative of each layer can be defined. For a concurrent execution, the number of components per-layer depends on the number of requests. Several design patterns have been proposed to deal with layered systems. Advice and guidelines to recognise and implement these systems can be found in [POSA96, PLoP94]. Also, consider the patterns used to generate layers, like *A Hierarchy of Control Layers* [AEM95] and the *Layered Agent Pattern* [KMJ96].

2. *Communication.* The communication required to coordinate the parallel execution of layer components is determined by the services that each layer provides. Characteristics that should be carefully considered are the type and size of the shared data to be passed as arguments and return values, the interface for layer components, and the synchronous or asynchronous coordination schema. The implementation of communication structures between components depends on the features of the programming language used. Usually, if the programming language has defined the communication structures (for instance, function calls or remote procedure calls), the implementation is very simple. However, if the language does not support communication between remote components, it is proposed

the construction of an extension in the form of a communication subsystem. Design patterns can be used for this. Particularly, patterns like the *Broker* pattern [POSA96], the *Composite Messages* pattern [SC95], the *Service Configurator* pattern [JS96, POSA00] and the *Visibility and Communication between Control Modules* and *Actions Triggered by Events* [AEM95] can help to define and implement the required communication structures.

3. *Agglomeration.* The hierarchical structure is evaluated with respect to the expected performance. Usually, systems based on identical layer components present a good load-balance. However, if necessary, using the conjecture-test approach, layer components can be refined by combination or decomposition of operations, modifying their granularity to improve performance or to reduce development costs.

4. *Mapping*. In the best case, each layer component executes simultaneously on a different processor, if enough processors are available. Usually this is not the case. An approach proposes to execute each hierarchy of layers on a processor, but if the number of requests is large, some layers would have to block, keeping the client(s) waiting. Another mapping proposal attempts to place every layer on a processor. This simplifies the restriction about the number of requests, but if not all operations require all layers, this may overcharge some processors, introducing load-balance problems. The most realistic approach seems to be a combination of both, trying to maximise processor utilisation and minimise communication costs. In general, mapping of layers to processors is specified static, allowing an internal dynamic creation of new components to serve new requests. As a "rule of thumb", a *Parallel Layers* pattern system will perform best on a shared-memory machine, but a good performance can be achieved if it can be adapted to a distributed-memory system with a fast communication network [Pan96, Pfis95].

Example Resolved

The potential parallelism for the SSSP is explained as follows. On each cycle, the current distance to a given vertex must be compared to the distance to the vertex via the last known vertex and the minimum recorded as the new distance. This calculation depends only on the graph array G. Thus, the minimum distance for each vertex can be computed and marked. If there are N processes, the algorithm would have a running time $O(N\log_2 N)$. N-1 cycles are still required to compute the minimum of all vertices. However, each cycle will require one time step to update the minimum for each vertex and $O(\log_2 N)$ time steps to compute the overall minimum vertex.

To move to a parallel solution, we must determine two things:
1. the communications network topology that will be used, and
2. what information will be stored on the processors and what will be passed as messages.

Partitioning

Both communication and computation of a minimum can be done in $O(\log_2 N)$ time by using a cubic array of processes. In such an arrangement, each process would compute its minimum

distance; then half of the processes would pick the minimum between its distance and that of a neighbour in one dimension (Figure 5). Half of these processes would in turn select a minimum, until the root process selects the global minimum distance vertex. Communication and selecting the minimum can be done in $O(log_2N)$ time, assuring an overall $O(Nlog_2N)$ performance.

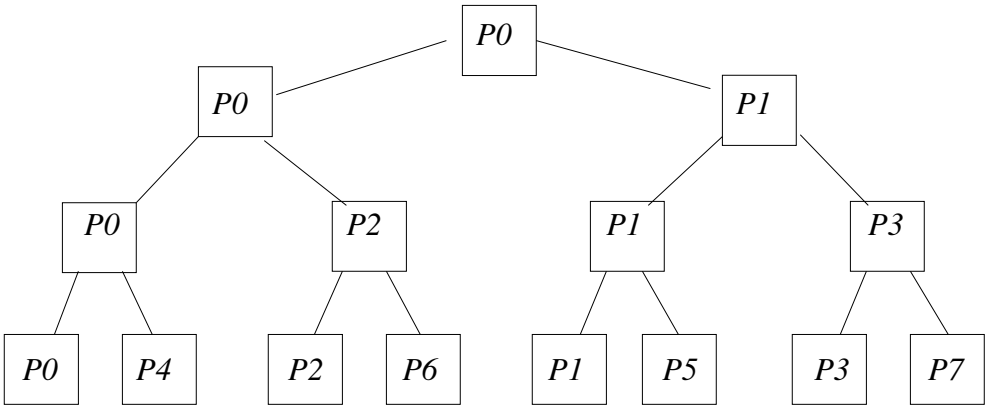


Figure 5. Tree representation for the SSSP algorithm.

Communication

The communication for *N* processes has to consider how to distribute data over the network of processes. This is done by reviewing the computations of a root and children processes, and determining what data must be available for the computations.

The root process *P0* calculates which of the two vertices has the shorter unknown distance. To do so, it must have available which vertices have already had their distances marked (the array `Known`), and the distance and id of the vertices being compared.

The children processes, on the other hand, must compare their current vertex distance to the distance between the last known vertex and themselves. Thus, they must have available the original graph G and the distance and id of the last known vertex. In addition, some children processes will be calculating the minimum between two vertices, so they will also need to know which of the vertices are known.

The basic data that needs to be communicated between processes is the id of the vertex and its most recent distance. This data will be used to calculate the minimum distance vertex and to announce which vertex has been marked as known. Thus, a message is a two-element array, one being a vertex id, the second a distance.

Since the message marking a vertex is distributed to all vertices, each process can keep track of which vertices are known. Thus each should locally store and update the array `Known`. Likewise, the graph G, which is not changed during the computation, must be distributed to all processes and stored locally before computation begins.

Finally, the function `MinV` would no longer have access to the array `D` to look up the distances of the vertices being compared. The parameters must be changed so that the distances of the vertices being compared are passed as well as the vertex identifiers.

Agglomeration and Mapping

If a 3D-cube is used for the computations (Figure 6), the code for synchronising and communicating between the root process and the remaining processes would be as the one shown in Figures 7 and 8, respectively.

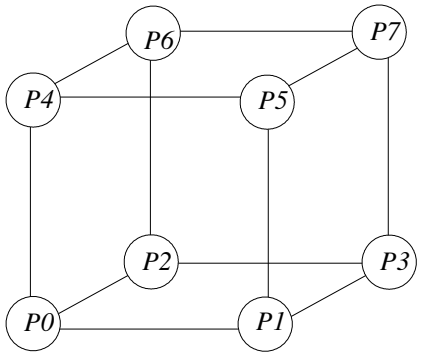


Figure 6. A 3D-cube.

```
Process 0 (the root process)
    i := 1
    While i < N
       // receive distances from 3 neighbours
       MinVertex := 0
       receive vertex id from z dimension
       MinVertex := MinV(MinVertex,Zvertex)
       receive vertex id from y dimension
       MinVertex := MinV(MinVertex,Yvertex)
       receive vertex id from x dimension
       MinVertex := MinV(MinVertex,Xvertex)
       Known[MinVertex] := TRUE // Update Known array
       LastKnown := MinVertex
       distribute LastKnown out x, y and z // Inform neighbours of the result
       i++
     End While
End Process 0
```

Figure 7. root process (Process 0).

```
Process k, 1<=k<N
// The remaining processes
     i := 1
     While i < N
        // find overall unknown minimum distance vertex
        LocalMinVertex := k
        if k < 4 then begin
           // processes 1, 2, and 3 receive and compute min
           receive Zvertex from z dimension
           LocalMinVertex := MinV(LocalMinVertex,Zvertex)
        else
           // processes 4, 5, 6, and 7 send out their vertices
           send LocalMinVertex out z dimension
        if k < 4 then
           // processes 4, 5, 6, and 7 do nothing
           if k = 1 then begin
               // process 1 receives and computes minimum
               receive Yvertex from y dimension
               LocalMinVertex := MinV(LocalMinVertex,Yvertex)
           else
               // processes 2 and 3 send out their vertices
               send LocalMinVertex out y
        if k = 1 then
           // process 1 sends its local min to process 0
           send LocalMinVertex out x
        // now receive overall minimum vertex LastKnown from Process 0
        if k = 1 then begin
           // process 1 receives from 0, distributes to 3
           receive LastKnown in x dimension
           send LastKnown in y dimension
        else
           if k < 4 then
               // processes 2 and 3 receive from 0 and 1, distribute to 4 and 5
               receive LastKnown in y dimension
               send LastKnown in z dimension
           else
                // processes 4, 5, 6, and 7 receive from 0, 1, 2, and 3
               receive LastKnown in z dimension
        D[k] := Min(D[k],D[LastKnown]+G[LastKnown,k]")"
        // now update Distances
        i++
     End While
End Process k
```

Figure 8. The children processes (Process *k*).

Synchronisation is achieved by the links between processes. Thus process 3 cannot compute the minimum distance vertex between itself and process 7 until process 7 sends its distance. Once computed, it sends the distance to process 1, which in turn waits until this message is received to compute the minimum between processes 3 and 1.

Known uses

- The homomorphic skeletons approach, developed from the Bird-Meertens formalism and based on data types, can be considered as an example of the *Parallel Layers* pattern: individual computations and communications are executed by replacing functions at different levels of abstraction [ST96].
- Tree structure operations like search trees, where a search process is created for each node. Starting from the root node of the tree, each process evaluates its associated node, and if it

does not represent a solution, recursively creates a new search layer, composed of processes that evaluate each node of the tree. Processes are active simultaneously, expanding the search until they find a solution in a node, report it and terminate [Fos94, NHST94].

- The Gaussian elimination method, used to solve systems of linear equations, is a numerical problem that is solved using a Parallel Layers structure. The original system of equations, expressed as a matrix, is reduced to a triangular form by performing linear operations on the elements of each row as a layer. Once the triangular equivalent of the matrix is available, other arithmetic operations must be performed by each layer to obtain the solution of each linear equation [Fos94].

Consequences

Benefits

- The *Parallel Layers* pattern, as the original *Layers* pattern, is based on increasing levels of complexity. This allows the partitioning of the computation of a complex problem into a sequence of incremental, simple operations [SG96]. Allowing each layer to be presented as multiple components executing in parallel allows to perform the computation several times, enhancing performance.
- Changes in one layer do not propagate across the whole system, as each layer interacts at most with only the layers above and below, that can be affected. Furthermore, standardising the interfaces between layers usually confines the effect of changes exclusively to the layer that is changed. [POSA96, SG96].
- Layers support reuse. If a layer represents a well-defined operation, and communicates via a standardised interface, it can be used interchangeably in multiple contexts. A layer can be replaced by a semantically equivalent layer without great programming effort [POSA96, SG96].
- Granularity depends on the level of complexity of the operation that the layer performs. As the level of complexity decreases, the size of the components diminishes as well.
- Due to several instances of the same computation are executed independently on different data, synchronisation issues are restricted to the communications within just one computation.
- Relative performance depends only on the level of complexity of the operations to be computed, since all components are active [Pan96].

Liabilities

- Not every system computation can be efficiently structured as layers. Considerations of performance may require a strong coupling between high-level functions and their lower-level implementations. Load balance among layers is also a difficult issue for performance [SG96, Pan96].
- Many times, a layered system is not as efficient as a structure of communicating components. If services in upper layers rely heavily on the lowest layers, all data must be transferred through the system. Also, if lower layers perform excessive or duplicate work,

there is a negative influence on the performance. In certain cases, it is possible to consider a Pipe and Filter architecture instead [POSA96].

- If an application is developed as layers, a lot of effort must be expended in trying to establish the right levels of complexity, and thus, the correct granularity of different layers. Too few layers do not exploit the potential parallelism, but too many introduce unnecessary communications. The granularity and operation of layers is difficult, but related with the performance quality of the system [POSA96, SG96, NHST94].
- If the level of complexity of the layers is not correct, problems can arise when the behaviour of a layer is modified. If substantial work is required on many layers to incorporate an apparently local modification, the use of Layers can be a disadvantage [POSA96].

Related patterns

The *Parallel Layers* pattern extends the *Layers* pattern [POSA96] and the Layers style [Shaw95, SG96] for parallel systems. Several other related patterns are found in [PLoP94]; more precisely, *A Hierarchy of Control Layers* pattern, *Actions Triggered by Events* pattern, and those under the generic name of *Layered Service Composition pattern*. The *Divide and Conquer* pattern [MSM05] describes a very similar structural solution to the *Parallel Layers* pattern. However, its context and problem descriptions do not cope with the basic idea that, in order to guide the use of parallel programming, it is necessary to analyse how to divide the algorithm and/or the data to find a suitable partition, and hence, link it with a programming structure that allows for such a division.

3. Summary

The goal of the present work is to provide software designers and engineers with an overview of the Parallel Layers pattern as a description of a common structure used for parallel software systems. Its application depends on the feasibility of the algorithm to be expressed in the form of a tree, which maps into the layers structure. Also, such an application is based on allowing data to be divided into pieces which are operated without a dependence among themselves. The architectural pattern described here is directly related with several developments in the field of algorithmic analysis, where it is proven its efficiency when dealing with fixed size problems. This pattern can be also linked with other current pattern developments for concurrent, parallel and distributed systems. Work on patterns that support the design and implementation of such systems has been addressed previously by several authors [Sch95, Sch98a, Sch98b, POSA00].

4. Acknowledgements

The author wishes to thank Joseph W. Yoder, my shepherd, for his important suggestions and advises for the improvement of this paper. This paper has been developed as part of the Subproject EN101603 of the Support Program to Institutional Projects for Teaching Improvement (PAPIME), supported by DGAPA-UNAM.

5. References

[AEM95]Aarsten, A., Gabriele Elia, G., and Giuseppe Menga, G. *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing.* Department of Automatica e Informatica, Politecnico de Torino. In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

[CG88] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs. A Guide to the Perplexed.* Yale University, Department of Computer Science, New Heaven, Connecticut. May 1988.

[CM88] K. Mani Chandy and J. Misra. *Parallel Programming Design.* Addison-Wesley, New York, 1988.

[CT92] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Inc., Boston, 1992.

[Fos94] Ian Foster. *Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publishing Company, 1994.

[JS96] Prashant Jain and Douglas C. Schmidt. *Service Configurator. A Pattern for DynamicConfiguration and Reconfiguration of Communication Services.* Third Annual Pattern Languages of Programming Conference, Allerton Park, Illinois. September 1996.

[MSM05] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. A Pattern Language for Parallel Programming. Addison Wesley Software Patterns Series, 2005.

[NHST94] Christopher H. Nevison, Daniel C. Hyde, G. Michael Schneider, Paul T. Tymann. *Laboratories for Parallel Computing.* Jones and Bartlett Publishers, 1994.

[OR98] Jorge L. Ortega-Arjona and Graham Roberts. *Architectural Patterns for Parallel Programming.* Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing, EuroPloP'98. Universitätsverlag Konstantz GmbH, 1999.

[OR00] Jorge L. Ortega-Arjona. *The Communicating Sequential Elements Pattern.* Proceedings of the 7th Annual Conference on Pattern Languages of Programming, PloP'98. Washigton University Technical Report wucs-00 29, 2000.

[OR03] Jorge L. Ortega-Arjona. *The Shared Resource Pattern.* Proceedings of the 10th Annual Conference on Pattern Languages of Programming, PloP 2003. Washigton University Technical Report wucs-00 29, 2000.

[OR05] Jorge L. Ortega-Arjona. *The Pipes and Filters Pattern.* Proceedings of the 10th European Conference on Pattern Languages of Programming, EuroPloP 2005. Universitätsverlag Konstantz GmbH, 2005.

[Pan96] Cherri M. Pancake. *Is Parallelism for You?* Oregon State University. Originally published in Computational Science and Engineering, Vol. 3, No. 2. Summer, 1996.

[Pfis95] Gregory F. Pfister. *In Search of Clusters. The Coming Battle in Lowly Parallel Computing.* Prentice Hall Inc. 1995.

[PLoP94] James O. Coplien and Douglas C. Schmidt (editors). *Patterns Languages of Programming.* Addison-Wesley, 1995.

[POSA96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, Ltd., 1996.

[POSA00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2. Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Ltd., 2000.

[SC95] Aamond Sane and Roy Campbell. *Composite Messages: A Structural Pattern for Communication Between Components.* OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. October 1995.

[Sch95] Douglas Schmidt. *Accepted Patterns Papers.* OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers.html. October, 1995.

[Sch98a] Douglas Schmidt. *Design Patterns for Concurrent, Parallel and Distributed Systems.* http://www.cs.wustl.edu/~schmidt/patterns-ace.html. January, 1998.

[Sch98b] Douglas Schmidt. *Other Pattern URL's. Information on Concurrent, Parallel and Distributed Patterns.* http://www.cs.wustl.edu/~schmidt/patterns-info.html. January, 1998.

[Shaw95] Mary Shaw. *Patterns for Software Architectures.* Carnegie Mellon University. In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall Publishing, 1996.

[ST96] David B. Skillicorn and Domenico Talia. *Models and Languages for Parallel Computation.* Computing and Information Science, Queen's University and Universita della Calabria. October 1996.

Paginador de Objetos

Wellington Pinheiro, Paulo Fernando, Fabio Kon

Departamento de Ciência da Computação
Instituto de Matemática e Estatística – Universidade São Paulo

{wrp,pfgom,kon}@ime.usp.br

***Abstract**: A wide variety of applications have to manipulate large quantities of objects in memory. However, often the available memory to store these objects is not enough to hold the entire set of objects simultaneously. The Object Paginator pattern presents a solution to the problem of manipulating large quantities of objects applying a paging mechanism.*

***Resumo:** Várias aplicações necessitam manipular grandes quantidades de objetos na memória, porém, a memória disponível, normalmente, não é suficiente para armazenar todo esse conjunto de objetos simultaneamente. O Padrão Paginador de Objetos apresenta uma solução para o problema da manipulação de grandes quantidades de objetos, através de um mecanismo de paginação.*

Objetivo

O objetivo do `Paginador de Objetos` é fornecer um mecanismo que permita o acesso a um conjunto de objetos por partes, definidas como páginas, mantendo o controle da navegação nesses objetos da página corrente. O acesso por partes torna-se necessário, uma vez que todo o conjunto de objetos não pode ser armazenado simultaneamente no meio de acesso rápido (e.g., memória), sendo que a maioria dos objetos permanece em um meio de acesso lento (e.g., disco rígido).

Motivação

Suponha uma aplicação que tenha como finalidade gerenciar um grande hospital público. Uma das características dessa aplicação é armazenar em um meio persistente a informação de todos os medicamentos que foram consumidos em um determinado mês (movimentação de medicamentos). Ao final de cada mês, um funcionário do departamento de suprimentos executa no sistema uma operação de consolidação de movimentação e a geração de uma listagem apresentando as informações dessa movimentação consolidada. Essa listagem apresenta basicamente o nome do medicamento e a quantidade total movimentada.

O sistema em questão utiliza objetos para fazer a representação desses medicamentos que serão manipulados. Cada objeto contém o nome do medicamento e a quantidade movimentada. Podem existir várias movimentações de um determinado medicamento em um só dia.

No cenário apresentado, as operações de consolidação e geração da listagem necessitam que os medicamentos sejam agrupados e cada grupo processado. Para executar tais operações é necessário que os objetos sejam todos manipulados na memória principal. O problema que surge é que o sistema pode manter uma quantidade

muito grande de medicamentos armazenados no meio persistente, de forma que não é possível carregar todos os objetos na memória para fazer a consolidação e a geração da listagem.

Uma forma de resolver o problema é utilizar um mecanismo que permita a recuperação e a manipulação do conjunto de objetos em partes (páginas na memória de acesso rápido) garantindo que a navegação no conjunto ocorra por demanda, de uma forma mais transparente possível.

A Figura 1 apresenta um possível modelo de relacionamento entre classes para solucionar o problema.

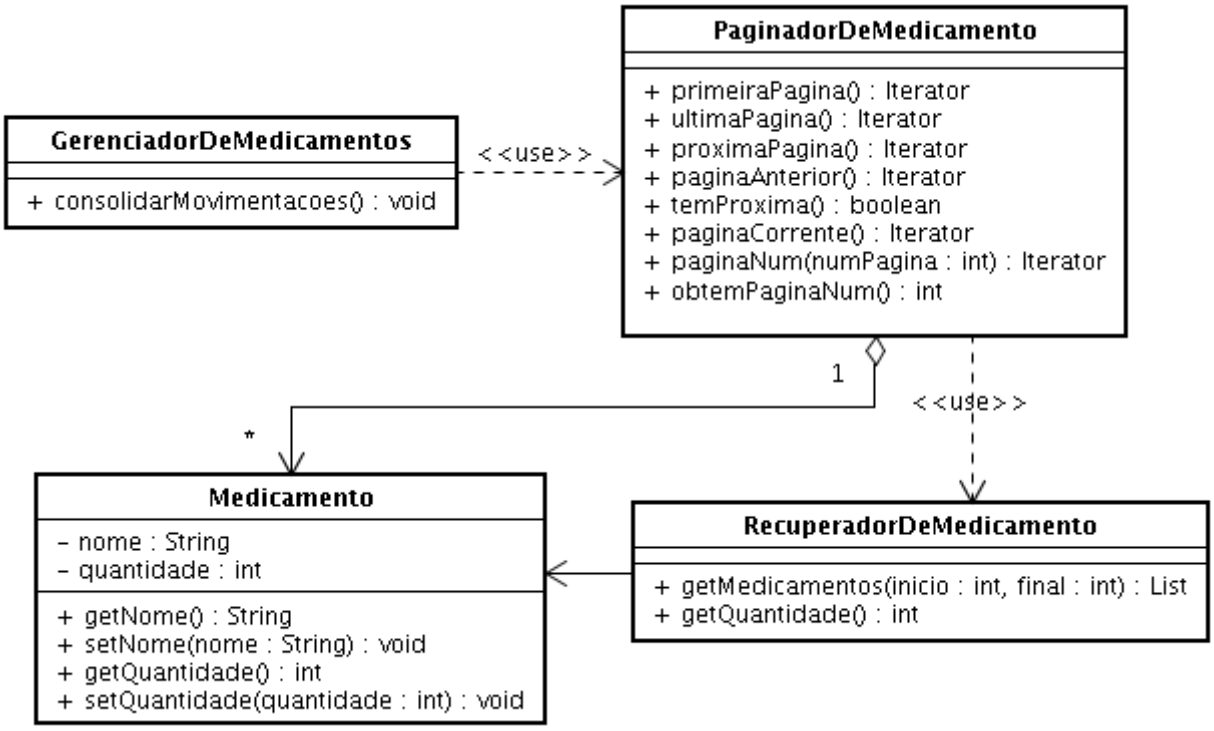


Figura 1: Solução Concreta para o Problema de Paginação

A classe `PaginadorDeMedicamento` é responsável por controlar a paginação dos objetos do tipo `Medicamento` (representações de medicamentos com suas quantidades movimentadas) em páginas de tamanho e ordenação pré-definidas, mantendo também informações a respeito da página atual, do primeiro e último objetos que pertencem à essa página, pois essas informações serão necessárias quando houver necessidade de acessar outras páginas (anterior, posterior ou uma página específica). `PaginadorDeMedicamento` utiliza a classe `RecuperadorDeMedicamento` para acessar os objetos que são mantidos em persistência. A classe `GerenciadorDeMedicamentos` é cliente de `PaginadorDeMedicamento`, definindo o método `consolidarMovimentacoes`, que utiliza o mecanismo de paginação para fazer a consolidação das movimentações de medicamentos. `consolidarMovimentacoes` faz as requisições das páginas contendo os medicamentos para `PaginadorDeMedicamento`, que acessará o mecanismo de persistência e retornará a nova página. Após o processamento intermediário da página

de medicamentos, `consolidarMovimentacoes` poderá pedir ao `PaginadorDeMedicamento` a próxima página e assim sucessivamente até que todos os objetos requisitados tenham sido processados.

Através do `PaginadorDeMedicamento` o cliente poderá acessar todo o conjunto de objetos do tipo `Medicamento`, disponibilizados por páginas, não sendo necessário manter o conjunto inteiro na memória ao mesmo tempo.

Para facilitar a navegação nos medicamentos da página corrente, `PaginadorDeMedicamento` poderá retornar uma implementação de um Iterador (padrão Iterator [Gamma et al. 1995]) específico para `Medicamento`.

Aplicabilidade

- Um sistema necessita acessar uma quantidade muito grande de objetos, mas não pode carregá-los todos de uma vez no meio de acesso mais rápido (memória principal);
- Permitir a navegação em um conjunto muito grande de objetos, escondendo os mecanismos de recuperação e acesso, mantendo o estado atual dessa navegação.

Estrutura

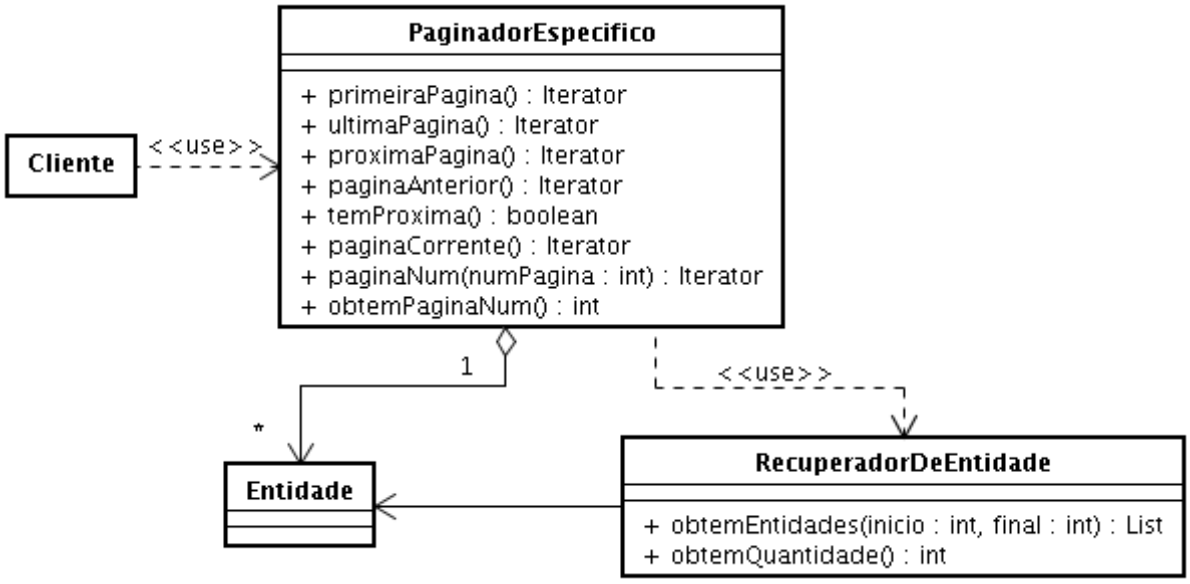


Figura 2: Solução Abstrata para o Problema de Paginação

Participantes

- `Cliente`: Qualquer classe que utilize os serviços do `PaginadorEspecifico`.
- `PaginadorEspecifico`: Classe de controle de paginação, com algum conhecimento a respeito da área de negócio. Deve conhecer as classes que recuperam dados do meio persistente.
- `Entidade`: Entidade de negócio que o `PaginadorEspecifico` armazenará em páginas e disponibilizará para os clientes. Na prática pode ser

qualquer tipo de classe.

- `RecuperadorDeEntidade`: Classe responsável pela recuperação das entidades do meio persistente.

Colaborações

- O cliente envia uma mensagem para o `PaginadorEspecifico` pedindo que ele carregue e disponibilize alguma página (primeira, última ou página específica);
- O `PaginadorEspecifico` acessa um objeto do tipo `RecuperadorDeEntidade` para recuperar a lista de objetos armazenados no meio persistente, obedecendo às informações referentes à página atual e o pedido de página;
- `PaginadorEspecifico` recupera uma lista de instâncias de `Entidade` e a devolve ao cliente quando este solicitar, através de um mecanismo de iteração.

A Figura 3 apresenta a criação do `PaginadorEspecifico` e a recuperação da primeira página de dados, retornando-a ao cliente. Observe o uso do parâmetro `tamPagina` na criação do paginador para definir o tamanho da página de objetos que o paginador armazenará.

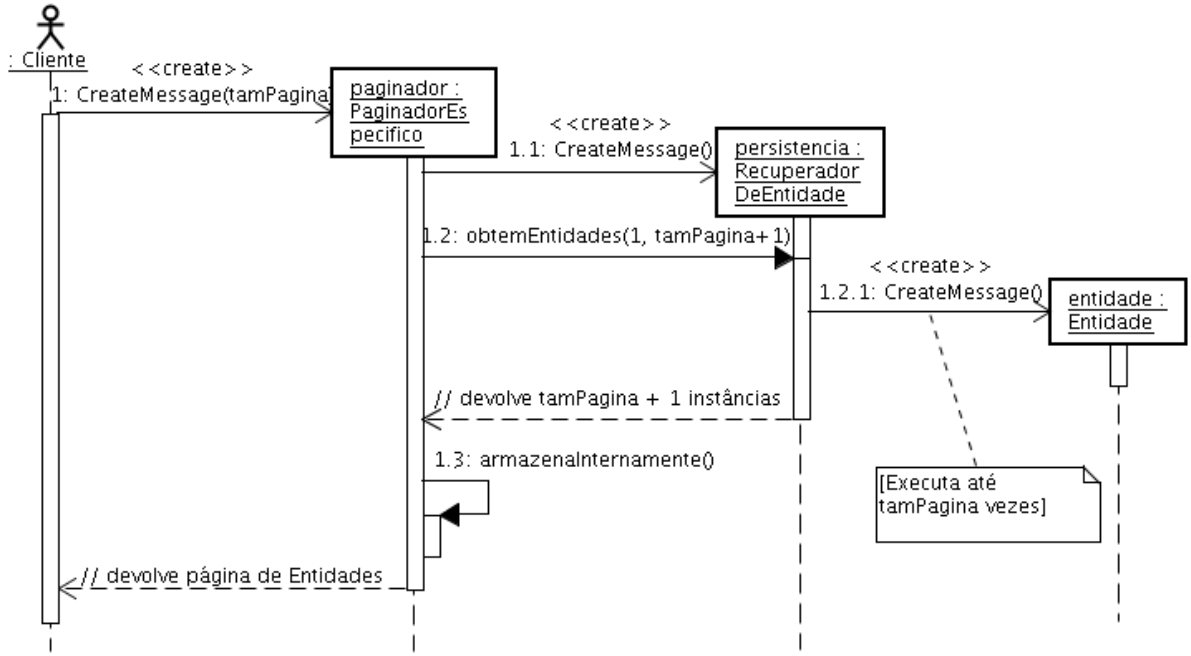


Figura 3: Criação do Paginador de Objetos

A Figura 4 é um exemplo de navegação entre páginas. O método `getEntidades` do `RecuperadorDeEntidade` recebe o intervalo de objetos que será recuperado da persistência.

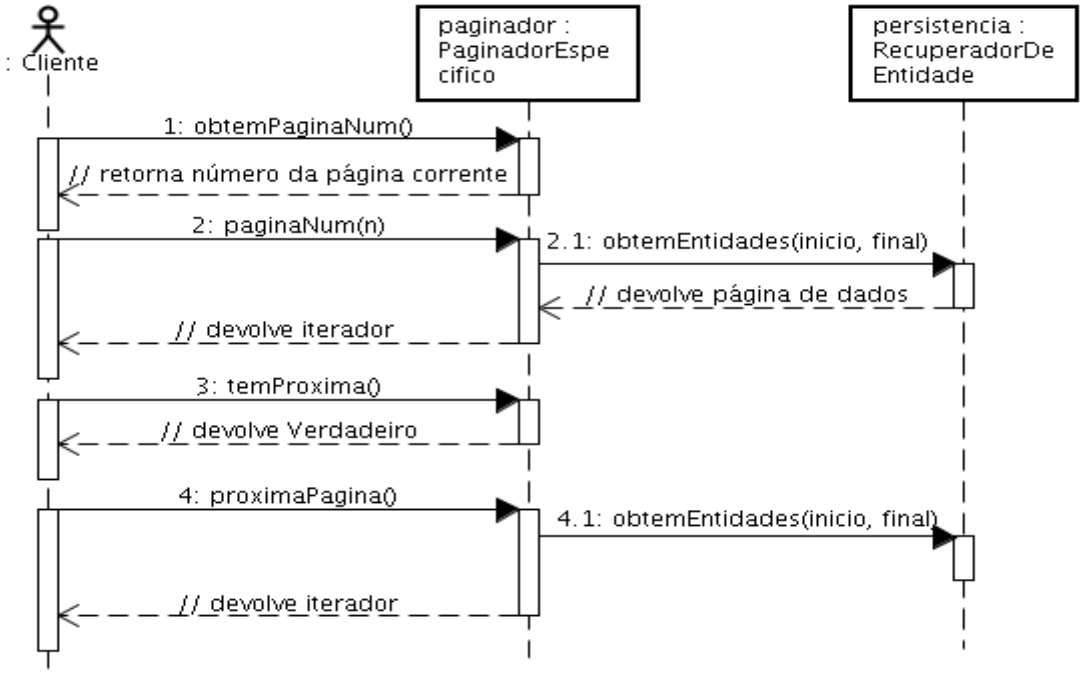


Figura 4: Exemplo de Navegação entre Páginas

A Figura 5 mostra como pode ser feita a navegação nos objetos da página corrente. Para fazer essa navegação pode-se utilizar um iterador específico para o tipo de entidade utilizada.

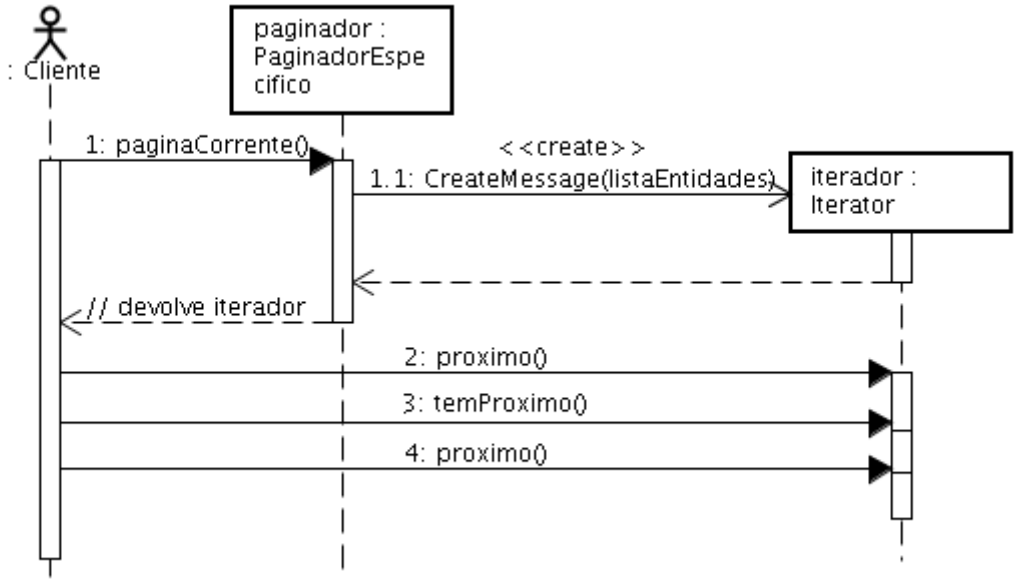


Figura 5: Exemplo de Navegação nos Elementos da Página Corrente

Conseqüências

+ O padrão Paginador de Objetos permite que grandes quantidades de dados, representados como objetos, possam ser acessados e manipulados sem que estes estejam todos carregados ao mesmo tempo na memória;

+ Provê o acesso por páginas a um conjunto de objetos, controlando a navegação sobre esse conjunto;

+ Serve como um controlador de navegação em um conjunto de objetos. Quando utilizado nesse sentido, a diferença entre o Paginador de Objetos e o Iterator [Gamma et al. 1995] é o fato do Paginador manter informações a respeito das páginas na qual está sendo feita a navegação, bem como informações que permitam a recuperação dessas páginas;

– O uso do Paginador de Objetos pode dificultar tarefas como a mudança na forma de ordenação dos dados em tempo de execução;

– Devido ao fato do Paginador de Objetos acessar o mecanismo de persistência para cada página solicitada, pode haver uma degradação de desempenho se comparado ao método onde todos os objetos são carregados de uma só vez. Em situações onde alto desempenho é crítico, e a questão memória não é problema, talvez seja mais interessante considerar o acesso a todos os dados de uma só vez sem utilizar o Paginador de Objetos.

Implementação

O padrão Paginador de Objetos pode ser implementado utilizando várias estratégias. Esta seção apresenta algumas dessas estratégias para implementação.

1. A forma mais simples de implementação do Paginador de Objetos é definir a classe `PaginadorEspecifico` como apresentada na Figura 1 e criar mecanismos internos para o controle de navegação entre as páginas;

2. Outra abordagem é fazer com que `PaginadorEspecifico` seja uma implementação do padrão Iterator [Gamma et al. 1995], permitindo assim um nível ainda mais alto de abstração. A desvantagem dessa implementação é o fato de tornar as semânticas de uso dos iteradores no sistema mais complexo, pois haveria iteradores de páginas e iteradores de objetos mantidos pela página;

3. A navegação entre páginas pode ser feita de uma forma transparente para o usuário, fazendo com que `PaginadorEspecifico` detecte quando o usuário tenta acessar um objeto que não está na página corrente e fazer a carga automática dessa página. Dessa forma, o Paginador de Objetos toma uma característica de memória temporária de objetos (“*pool”);*

4. Uma implementação mais sofisticada do Paginador de Objetos permite que objetos possam ser alterados ou removidos enquanto estão sendo iterados. Caberia ao `PaginadorEspecifico` notificar a persistência dessas alterações;

5. Quando uma aplicação utiliza paginadores para diversos tipos de entidades, uma classe abstrata de paginação pode ser usada com a função de controle de navegação, e para cada tipo de entidade poderá ser criada uma classe concreta

que herda desta classe abstrata e implementa operações específicas para um determinado tipo de entidade a ser tratado;

6. Pode-se ainda definir a classe de paginação como parametrizada (Generics em JAVA ou Templates em C++) permitindo que o cliente defina qual tipo de entidade as classes de paginação manipularão.

Exemplo de Código da Solução

Como implementação do Paginador de Objetos, será utilizado o primeiro exemplo apresentado, o `PaginadorDeMedicamento`, implementado na linguagem Java.

A classe `PaginadorDeMedicamento` deve ser responsável pelo controle da navegação entre páginas e o momento no qual uma nova página será carregada e disponibilizada (vale observar que o `PaginadorDeMedicamento` deve solicitar à alguma classe utilitária que recupere uma determinada página do meio persistente, garantindo assim um baixo acoplamento com as classes responsáveis pelos mecanismo de persistência em questão).

A classe `PaginadorDeMedicamento` contém o atributo `tamPagina` que armazenará o tamanho das páginas que serão mantidas pela instância do paginador.

```
public class PaginadorDeMedicamento {
     private int tamPagina;

     public int obtemTamanhoDaPagina() {
          return this.tamPagina;
     }
          .
          .
          .
}
```

Além de `tamPagina`, `PaginadorDeMedicamento` também define um conjunto de atributos que possuem a finalidade de manter a página atual (`numeroPagina`), o número total de páginas armazenado no meio persistente (`totalPaginas`), a lista com os objetos da página corrente (`medicamentos`) e uma referência para o objeto responsável por recuperar os dados do meio persistente (`rm`).

```
private int numeroPagina = -1;
private int totalPaginas = 0;
private List<Medicamento> medicamentos =
     new ArrayList<Medicamento>();
private RecuperadorDeMedicamento rm =
     new RecuperadorDeMedicamento();
```

As funcionalidades de navegação nas páginas normalmente resultam no acesso à persistência para recuperação dos dados, dessa forma, será definido um método no objeto de acesso a persistência para o qual será passado o intervalo de objetos que serão recuperados. Existem várias formas de implementar esse mecanismo, e para esse caso, optamos por um mecanismo simples, baseado em um número seqüencial que é dado a cada objeto na persistência. `PaginadorDeMedicamento` define um método que receberá o número da página desejada e ele se encarregará da carga da página:

```
private void carregaPagina(int numeroPagina) {
     int tamanhoDaPagina = obtemTamanhoDaPagina();
     int i = tamanhoDaPagina * this.numeroPagina;
     int f = i + tamanhoDaPagina;
     this.numeroPagina = numeroPagina;
     this.medicamentos.clear();
     this.medicamentos.addAll(
          this.rm.obtemMedicamentos(i, f));
}
```

O método `carregaPagina` calcula o intervalo de objetos que devem ser recuperados, atualiza a página corrente, remove da memória principal os objetos da página anterior e finalmente recupera o conjunto de medicamentos através de uma solicitação para `rm`. Os objetos recém recuperados são armazenados nessa nova página atual.

Para que possa ser feito o cálculo em `carregaPagina`, é necessário conhecer o tamanho da página (quantidade de objetos por página). Essa informação é passada na criação do `PaginadorDeMedicamento`, como um parâmetro para o construtor:

```
public PaginadorDeMedicamento(int tamPagina) {
     this.tamPagina = tamPagina;
     this.totalPaginas = (int) Math.ceil((double)
          rm.getQuantidade() / (double) tamPagina);
}
```

Observe que no construtor é feito o cálculo do número total de páginas.

Os métodos de navegação entre páginas são semelhantes, logo, serão apresentados somente aqueles que solicitam ao `PaginadorDeMedicamento` a última página e uma página específica:

```
public Iterator ultimaPagina() {
     carregaPagina(totalPaginas);
     return paginaCorrente();
}
public Iterator paginaNum(int numPagina) {
     carregaPagina(numPagina);
     return paginaCorrente();
}
```

Os métodos que recuperam páginas, na verdade, delegam esse trabalho ao método `carregaPagina`. Essa implementação é bem simples, mas no caso de implementações mais robustas, deve haver verificações de erros na carga ou eventuais tentativas de navegações em páginas inválidas.

Uma vez que a página foi carregada, o acesso aos objetos dessa página pode ser disponibilizado através de um `Iterator`, como é mostrado logo abaixo:

```
public Iterator paginaCorrente() {
     return new MedicamentoIterator(
          this.medicamentos);
}
```

O método `paginaCorrente` cria um objeto personalizado do `Iterator` para `Medicamento` e o devolve ao cliente. Outra forma de implementar esse mecanismo é fazer com que logo após a carga da página seja criado um objeto do tipo `MedicamentoIterator` que é mantido válido enquanto a página não é alterada. Caso a página seja alterada, os iteradores anteriores devem ser invalidados, não permitindo que os clientes continuem fazendo uso.

As classes `Iterator` e `IteratorMedicamento` são implementações simples do padrão Iterator [Gamma et al. 1995].

A classe de acesso à persistência pode variar de acordo com as necessidades do sistema, mas nesse exemplo, `RecuperadorDeMedicamento` conterá dois métodos importantes:

```
public List<Medicamento> obtemMedicamentos(
     int inicio, int fim)
public int obtemQuantidade()
```

`obtemMedicamentos` retorna do meio persistente uma lista de medicamentos onde as suas chaves estejam no intervalo começando em `inicio` (inclusivo) e `fim` (exclusivo). `obtemQuantidade` retorna a quantidade total de objetos armazenados no meio persistente e é utilizado pelo paginador no cálculo da quantidade de páginas disponíveis.

```
public class Cliente {
     public static void main(String[] args) {
          PaginadorDeMedicamento paginador =
               new PaginadorDeMedicamento(10);
          while (paginador.temProxima()) {
               paginador.proximaPagina();
               Iterator it =
                    paginador.paginaCorrente();
               while (it.temProximo()) {
                    System.out.println(it.proximo());
               }
          }
     }
}
```

Por fim, a classe `Cliente` é um exemplo de cliente que utiliza a estrutura de paginação.

No do método `main` é definida uma variável local, `paginador`, como sendo um paginador. São utilizados dois laços `while`, um para fazer a iteração das páginas e outro para a iteração sobre a coleção de medicamentos mantida em cada página, apresentando cada um desses medicamentos na saída padrão.

Usos Conhecidos

- Vários sítios de compras pela Internet apresentam o comportamento de paginação. Por exemplo, os sítios das Lojas Americanas, Submarino e Livraria Saraiva permitem que o usuário faça pesquisas de seus produtos, obtendo como retorno uma coleção muito grande desses produtos, eventualmente. O sítio

permite que o usuário navegue por esse conjunto de produtos através páginas, oferecendo opções de navegação para próxima página, página anterior além de outras possibilidades;

- Java Server Faces (JSF) [Burns e Kitain 2006] é um conjunto de especificações e APIs voltadas para o desenvolvimento de aplicações WEB utilizando o padrão Model-View-Controller [Krasner e Pope 1988]. O Apache MyFaces [MYF] é uma implementação do JSF feita pelo Apache Group, que disponibiliza também uma extensão de componentes chamado de Tomahawk [TOM]. Entre os componentes do Tomahawk existe o HtmlDataScroller que é responsável por fazer a paginação de objetos em aplicações WEB. Este componente recebe uma lista de objetos e faz a paginação de acordo com parâmetros pré-definidos. Entre outras funcionalidades, esse componente de paginação permite a navegação entre páginas e a ordenação da coleção de elementos;
- SCORM (*Sharable Content Object Reference Model*) [SCORM 2006] é um conjunto de padrões técnicos, desenvolvido pelo Departamento de Defesa Americano que permite que sistemas de aprendizado baseados na Web encontrem, importem, compartilhem, reutilizem e exportem conteúdos de aprendizado de uma forma padrão. SCORM define o uso de objetos de aprendizado que podem conter vários recursos como textos, imagens e sons, além de uma regra de navegação e uso desses recursos da maneira a propiciar o aprendizado. Esses objetos de aprendizado são executados em Sistemas de Gerenciamento de Aprendizado ou Sistema de Gerenciamento de Conteúdo de Aprendizado (em inglês referem-se às siglas: LMS – *Learning Management Systems* e, LCMS – *Learning Content Management Systems*) sendo que esses sistemas devem obedecer às regras de navegação definidas nos objetos de aprendizagem. Para executar as tarefas de controle do fluxo de navegação, esses sistemas utilizam um mecanismo de paginação que carrega os recursos na memória, de acordo com a necessidade. Além das características básicas, esse mecanismo de paginação também deve ser dotado de uma inteligência adicional para permitir que seja feita uma análise a respeito da evolução no aprendizado do usuário, mudando o caminho de aprendizagem e conseqüentemente o fluxo de paginações.

Padrões Relacionados

- Iterator [Gamma et al. 1995]: O padrão Paginador de Objetos é normalmente utilizado junto do padrão Iterator para permitir uma navegação através dos objetos disponibilizados na página corrente. Outra vantagem no uso do Iterator é permitir que detalhes de navegação (como ordem, por exemplo) fiquem implementados de forma transparente para o paginador;
- Memento [Gamma et al. 1995]: Para fazer o controle da navegação, os paginadores podem utilizar o padrão Memento para obter o estado corrente da navegação e utilizá-lo posteriormente quando necessitar;
- Template Method [Gamma et al. 1995]: Em uma aplicação real pode ser necessário que vários paginadores específicos para determinados tipos de objetos sejam criados. Nesse contexto, poderíamos fornecer uma classe abstrata para os paginadores que definem o comportamento básico de todos os paginadores (como controle de navegação) e os detalhes necessários para uma implementação completa seriam delegados para as classes concretas que herdam dessa classe abstrata utilizando template methods;
- Data Access Object (DAO) [Alur et al., 2001]: Normalmente estamos interessados em fazer a paginação de um conjunto de dados que estão armazenados em um mecanismo persistente, assim, seria interessante que o paginador pudesse acessar os dados abstraindo a forma como estes são recuperados. O padrão DAO serve como essa camada de abstração que conhece os detalhes da persistência e fornece uma interface bem definida para a recuperação dos dados que o paginador necessita. Uma vantagem de utilizar o padrão DAO é permitir que as classes de controle do paginador fiquem independentes dos mecanismos de acesso a dados e da persistência;
- Value List Handler [Alur et al., 2001]: O Value List Handler é um paginador para aplicações distribuídas que permite a implementação de políticas de *cache* e controle de navegação em ambientes WEB ou em aplicações multi-camadas;
- Record Set [Fowler 2002]: O padrão paginador pode utilizar um Record Set para manter os objetos da página corrente. Para que isso seja possível, todos os métodos das classes responsáveis pela persistência, que retornam dados, devem retornam um Record Set contendo as informações referentes as entidades que o cliente está esperando. O paginador pode optar ainda por devolver o próprio Record Set para que o cliente faça sua manipulação;
- Paging [Noble e Weir 2001]: Paging é um paginador mais específico para ambientes de pouca memória primária, que permite a execução de programas "diretamente da memória secundária". Essa sensação de executar as aplicações na memória secundária é dada através de um mecanismo de paginação da memória, onde o ambiente (e,g., sistema operacional) carrega ou descarrega essas páginas de acordo com a demanda da aplicação, de uma forma transparente, dando a sensação de que sempre há memória disponível para alocação.

Referências Bibliográficas

Alur, D., Malks D. e Crupi, J. (2001), Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Burns, E. e Kitain R. (2006) "JavaServer Faces Specification Version 1. 2 - Rev A", http://jcp.org/aboutJava/communityprocess/mrel/jsr252/index.html, Acessado em: 29 de Junho de 2007.

Fowler M. (2002), Patterns of Enterprise Application Architecture. Addison-Wesley, Longman Publishing Co., Inc., Boston, MA, USA.

Gamma E., Helm R., Johnson R. e Vlissides J (1995), Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley.

Krasner G. E. e Pope S. T. (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80. J. Object Oriented Program, vol. 1, 3ª. edição, páginas 26–49.

MYF. Apache Myfaces Project. http://myfaces.apache.org/. Acessado em: 29 de Junho de 2007.

Noble J. e Weir C. (2001) Small memory software: patterns for systems with limited memory. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Shareable Content Object Reference Model SCORM (2006). http://www.adlnet.org, Acessado em: 29 de Junho de 2007.

TOM. Myfaces Tomahawk. http://myfaces.apache.org/tomahawk/index.html. Acessado em: 29 de Junho de 2007.

Padrão AutenticaConexão

Marcelo Antônio Albuquerque e Souza[1], Jerffeson Teixeira de Souza[2]

[1]Têxtil União S/A, Rodovia CE021 km 08, Distrito Industrial
61.939-906, Maracanaú – CE

[2]Universidade Estadual do Ceará (UECE), Av. Paranjana, 1700, Campus do Itaperi
60.740-903, Fortaleza – CE

marcelo2306@gmail.com, jeff@larces.uece.br

Resumo. *Definição de um mecanismo de autenticação para bancos de dados de forma segura e flexível utilizando schemas públicos independente de código fonte e transparente para o desenvolvedor.*

Palavras-chave : Padrões de Projeto, Autenticação em banco de dados, Encriptação de dados, Segurança de dados.

Abstract. *Define a secure and flexible authentication mechanism for databases using public schemas, independent of the source code and transparent to the developer.*

Keywords : Design Patterns, Database authentication, Data Encryption, Data security.

Nome

AutenticaConexão

Intenção

Prover um mecanismo de autenticação para diversos bancos de dados (BD) combinando a persistência de senhas em um schema público e o uso de função encapsulada para conexão, garantindo a inviolabilidade dessas senhas de acesso e a flexibilidade na manutenção das mesmas, tornando-as indisponíveis para a aplicação.

Contexto

Ambientes que utilizam aplicações onde a conexão com o BD seja realizada através de strings literais para *usuário* e *senha* (compreendendo *usuário* como usuário da instância do BD) e onde ocorram alterações no usuário/senha do Banco de Dados; aplicativos implantados em organizações diferentes e que por isso sejam necessários diversos códigos fonte para cada usuário/senha a ser autenticado pelos diferentes BD.

Problema

Em aplicações que utilizem um SGBD (Sistema Gerenciador de Banco de Dados) existe a necessidade de se efetuar uma conexão com o banco de dados através de um *usuário* e respectiva *senha* e freqüentemente essa informação está na forma de strings escritas literalmente dentro do código fonte em alguma classe ou rotina do sistema. Quando ocorre a mudança do nome do usuário ou da senha, é necessário reescrever esse código fonte. Além disso, a equipe de desenvolvimento tem acesso a um usuário/senha do BD de produção, o que pode não ser interessante sob o ponto de vista da segurança das informações.

Forças

- Senhas literais no código fonte são uma potencial falha de segurança, pois são visíveis a qualquer desenvolvedor que tenha acesso a esse código;
- A simples mudança de senha em um BD pode se tornar uma operação complexa caso existam várias aplicações diferentes acessando esse BD;
- A demissão de um desenvolvedor pode obrigar a reescrita dessa senha na aplicação;
- Uma coleção de senhas persistentes facilitaria o trabalho de administração do BD;
- A persistência de senhas em um schema torna-o um alvo em potencial para tentativas de quebra da criptografia.

Solução

Substituir usuário/senha de conexão do BD de produção por parâmetros não acessíveis ao usuário ou desenvolvedor – serão adotadas medidas do lado do BD e do lado da aplicação.

No lado do BD criar um schema (por exemplo, `PublicDB`) com uma única tabela (por exemplo `PublicUser`):

```
CREATE TABLE [PublicUser] (
       [Aplicacao] [Char] (10) NOT NULL,
       [Usuario] [Char] (10) NOT NULL,
       [Senha] [Char] (10) NOT NULL
)
```

Essas três colunas serão obrigatoriamente encriptadas – a segurança do BD estará totalmente dependente de quão segura será essa função de encriptação. As permissões de acesso à essa tabela serão concedidas a apenas um usuário e somente com poder de leitura (sugestão : `PublicDBO`). A senha de `PublicDBO` poderá ser de conhecimento público, visto que esse usuário tem acesso somente à dados encriptados.

Do lado da aplicação criar uma nova conexão à `PublicDB` através de `PublicDBO`. Em seguida encriptamos o nome da aplicação para criar uma string de busca no formato:

```
Select Usuário, Senha From PublicUser where Sistema = XXXXX
```

`XXXXX` será a string resultante da encriptação do nome da aplicação. Obtendo o usuário/senha vinculado à aplicação passamos os mesmos como parâmetros de um método/função que irá por fim autenticar o BD de produção. Dessa maneira conseguimos estabelecer a conexão com o BD sem que fosse necessário escrever o usuário/senha do BD de produção.

Para o DBA existirá aplicativo que permita ao mesmo alterar os registros de `PublicUser` quando ocorrer a mudança de usuário/senha no BD. Essa operação será sincronizada para não ocorrer erros de conexão.

A função de encriptação poderá utilizar o conceito de chave pública e privada e deverá estar encapsulada em uma DLL ou qualquer outro meio que impeça a visualização do código. Observa-se que neste caso não poderá ser usada encriptação de mão única, pois o DBA precisa desses código encriptados para manter os dados da tabela `PublicUser`. A chamada à função de encriptação também estará encapsulada para que não seja possível a visualização do retorno da função – que é exatamente o que queremos esconder: o usuário/senha do BD.

Exemplo

Iremos exemplificar usando a função `GetPublicAcess` cujo parâmetro será o nome do banco a ser conectado (`sDataBasetoFind`). A conexão será realizada no banco público e caso a busca com o valor encriptado de `sDataBasetoFind` seja bem sucedida, iremos obter os usuários e senhas válidos desse banco (`sUsertoFind` e `sPasswordtoFind`). Em seguida é realizada a conexão com o banco definido em `sDataBasetoFind`.

```
//GetPublicAcess é uma função encapsulada (o desenvolvedor não pode ter
//acesso ao seu código fonte) que obtém o usuário e senha do banco de
//trabalho definido no parâmetro sDataBasetoFind

Function GetPublicAcess(sDataBasetoFind : String):Boolean;
  //Definição das variáveis privadas que receberão usuário e senha de
  //autenticação
  Var sUsertoFind, sPasswordtoFind : String

begin
  //Bloco try..except para tratamento no caso de insucesso na conexão
  //efetuar encerramento do programa
  try

    //Comandos para inicializar uma conexão em delphi
    dbPublicDB.Params.Clear;
    dbPublicDB.LoginPrompt:=False;


    //Passando os dados para conexão com o schema público
    dbPublicDB.Params.Add('DATABASE NAME=PUBLICDB');
    dbPublicDB.Params.Add('USER NAME=PublicDBO');
    dbPublicDB.Params.Add('PASSWORD=faith');
    dbPublicDB.Connected:=True; //efetua a conexão
```

```
    //definindo componente tquery que irá retornar o usuário e senha
    //encriptados definidos para o banco de trabalho definido pelo parâmetro
    //SdataBasetoFind
    quBusca.database := dbPublicDB; //aponta para PublicDB

    //busca em PublicUser
    quBusca.SQL.Add('select Usuário, Senha from PublicUser');
    quBusca.SQL.Add('where Sistema=:pSis');

    //Encriptar parâmetro pSIs pois pois PublicUser é uma tabela com
    //conteúdo encriptada
    quBusca.ParamByName('pSIS').AsString := Cript(sDataBasetoFind);
    quBusca.Open; //Executa a query

    //Testa se a busca foi bem-sucedida
    if quBusca.Eof then
       Result := false // retorna false para não efetuar a conexão
    else begin
       {obtidos usuário e senha do banco de trabalho}
       sUsertoFind := DeCript(quBusca.fieldbyname('Usuario').AsString);
       sPasswordtoFind := eCript(quBusca.fieldbyname('Senha').AsString);
       quBusca.Close;

       //Prepara a conexão com o banco de trabalho
       dmTable.dbGTF.Connected:=False;
       dmTable.dbApplication.Params.Clear;
       dmTable.dbApplication.aliasname := 'MyAlias';

       //Nesse ponto pode-se estabelecer a conexão pois obteve-se o
       // usuário e senha do banco de trabalho
       dmTable.dbApplication.Params.Add('DATABASE ME='+sDataBasetoFind);
       dmTable.dbApplication.Params.Add('USER NAME='+sUsertoFind);
       dmTable.dbApplication.Params.Add('PASSWORD='+sPasswordtoFind);

       //Se a conexão não for bem-sucedida será processado o bloco except
       dmTable.dbApplication.Connected:=True;
    end;

    //fecha a conexão com PublicDB
    dbPublicDB.Connected := false;

  except
    //Insucesso na conexão
    Result := false;
  end;
end;

//método público de inicialização da aplicação onde ocorre a obtenção das
//senhas e conexão com o BD de trabalho
procedure TfmMain.FormCreate(Sender: TObject);

begin
    //tenta estabelecer a conexão com o banco definido como parâmetro de
    // GetPublicAcess
    if not GetPublicAcess('MyDataBase') thenbegin
         //Único acesso do desenvolvedor
         Application.MessageBox('Não foi possível conectar ao Banco de
                                 Dados','Atenção',mb_IconStop);
         Halt; //sai da aplicação
      end else
            begin
                //Conexão bem-sucedida, execução normal do programa a
                //partir desseponto a conexão com o banco de trabalho está
                //efetuada
            end;
end;
```

Contexto Resultante

A informação de usuário e senha do banco de produção passa a ser parâmetro da aplicação. Deixa de existir na aplicação as strings de usuário e senha para conexão com o SGBD. Quando houver mudança de senha no banco de produção não será necessário alterar essas strings no código fonte da aplicação. A equipe de desenvolvimento não terá acesso ao banco de produção porque não terá conhecimento do usuário/senha respectivos. No entanto, será necessário gerenciar outro schema no SGBD bem como desenvolver um algoritmo seguro para encriptação das strings de usuário.

Conseqüências

Positivas

- Aumento da segurança: as senhas do banco de dados deixam de ser visíveis a qualquer um que tenha acesso ao código fonte. Em vários casos práticos não é desejável que a equipe de desenvolvimento tenha acesso a essas senhas;
- Diminuição de reescrita de código fonte : usuário/senha como parâmetros da aplicação, deixa de existir a necessidade de alterar o código quando ocorrer mudança dos mesmos no banco de dados.

Negativas

- Aumento da complexidade da aplicação: Será necessária uma conexão extra com o banco de dados; a criação de um schema de autenticação no banco de dados; a criação de algoritmo de encriptação para manipulação de usuário e senha;
- Não se pode garantir a inviolabilidade de um BD caso um desenvolvedor tenha acesso ao nome de um banco de produção. Dentro da aplicação ele passa a ter acesso aos dados desse banco.

Racional

A adoção desse padrão implica na criação de um schema extra no BD e irá demandar um certo tempo na implementação da classe ou rotina de conexão e da classe ou rotina responsável pela encriptação dos dados. Esse tempo é largamente compensado pela produtividade adquirida na eliminação da necessidade de reescrever o código fonte, além do aumento da segurança da administração de dados, pois a equipe de desenvolvimento não terá acesso ao usuário/senha dos bancos de produção.

Usos Conhecidos

Têxtil União S/A – Fiação em Maracanaú – Ceará Valença Industrial – Tecelagem/tinturaria em Valença – Bahia Os sistemas internos das empresas utilizam esse padrão para conexão e autenticação das máquinas cliente ao servidor de banco de dados das mesmas.

Padrões Relacionados

Podemos encontrar documentação correlata em Hays, Loutrel e Fernandez [1] cujo framework aglutina as tarefas de autenticação, controle de acesso e filtragem de dados em ambientes distribuídos e também em Lehman [2] que dedica um capítulo à autenticação em banco de dados através da persistência dos parâmetros de conexão em tabelas de banco de dados. Em Fernandez [3] podemos analisar uma coleção de padrões para controle e acesso a nível de sistema operacional. Um padrão para autenticação em ambientes distribuídos pode ser encontrado em Fernandez [4].

Agradecimentos

Para a conclusão desse trabalho foi de fundamental importância a colaboração do Dr. Eduardo B. Fernandez graças à sua larga experiência em padrões de autenticação pôde fornecer preciosos conselhos para a melhoria desse documento. Agradecemos o apoio da Universidade Federal do Ceará, na figura da Dr. Vânia Vidal, coordenadora do curso de Especialização em Tecnologias da Informação e à Têxtil União S/A, local onde nasceu a idéia dessa implementação. Agradecemos também aos colegas Anderson Brando, Ellen Polliana, Kleber Rocha, Rafael Braga, Tiago Barros e todos os outros participantes do workshop de escritores, grupo B, do SugarLoafPLoP'2007 pela motivação e comentários essenciais ao aperfeiçoamento do trabalho.

Referências

[1] Viviane Hays, Marc Loutrel, Eduardo B. Fernandez, "The Object Filter and Access Control Framework", PloP 2000 Conference, http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/Fernandez3/Fernandez3.pdf.

[2] Clay Lehman , "Secure Authentication and Session State Management for Web Services", CSC 499 Honors Thesis, http://www.csc.ncsu.edu/academics/undergrad/honors/lehman/clehman.pdf.

[3] E.B.Fernandez and J.C.Sinibaldi, "More patterns for operating systems access control", Procs. EuroPLoP 2003, http://hillside.net/europlop

[4] E. B. Fernandez and R. Warrier, "Remote Authenticator/Authorizer", Procs. of the Pattern Languages of Programs Conference (PLoP 2003).

Linguagem de Padrões para Avaliação de Conhecimento em Objetos de Aprendizagem – Parte I

Ingrid T. Monteiro[1], Clayson Sandro[1], Cidcley T. de Souza[1]

NASH (Núcleo Avançado em Engenharia de Software Distribuído e Sistemas Hipermídia) /ITTI (Instituto de Telemática) – Centro Federal de Educação Tecnologia do Ceará – Fortaleza, CE - Brasil

{ingridtm,claysonsandro}@gmail.com, cidcley@cefetce.br

Abstract. *Learning Objects (LO) are resources which have been frequently applied to support computer-aided learning. In order to assess the assimilation effectiveness of the knowledge provided by such resources, they must supply mechanisms through which the learning of the presented contents can be measured. However, the notion of knowledge evaluation in LOs varies in several aspects according to the learning goals. Therefore, we present in the paper the first part of a pattern language regarding the knowledge evaluation process using LOs. The main goal of the paper is to provide, though patterns methodology, a set of recommendations to the knowledge of evaluation possibilities, by using LOs. The main goal of the paper is to provide a set of recommendations to allow for the better exploitation of LO contents, related to evaluation, through the successful experience's documentation.*

Resumo. *Objetos de Aprendizagem (OA) são recursos que vêm sendo utilizados largamente para dar suporte ao aprendizado apoiado por computador. Para que possamos aferir a eficácia na assimilação dos conhecimentos fornecidos por esses recursos, os mesmos devem oferecer mecanismos através dos quais se possa avaliar a aprendizagem dos conteúdos apresentados. Entretanto, a implementação da noção de avaliação de conhecimentos em OA difere em diversos aspectos de acordo com a intenção sobre o aprendizado. Nesse sentido, apresentamos nesse artigo a primeira parte de uma linguagem de padrões relacionados ao processo de avaliação de conhecimento com a utilização de OA. Esse trabalho tem a intenção de fornecer diretrizes para um melhor aproveitamento dos conteúdos dos OA, no que diz respeito à avaliação, a partir da catalogação de experiências bem sucedidas.*

1. Introdução

A necessidade do uso de novas tecnologias no processo de ensino e aprendizagem vem sendo cada vez mais presente no cotidiano de alunos e professores. Contudo, é preciso ampliar esta discussão com o objetivo de contextualizar as novas tecnologias da informação e da comunicação e suas relações com o ensino e aprendizagem na Educação [1]. Esta discussão surge com o anseio de modificar a forma como a Educação propõe o ensino e como os materiais educacionais são projetados, desenvolvidos e entregues àqueles que desejam aprender.

Atualmente, um dos materiais educacionais que procuram atender a esses objetivos são os OA – Objeto(s) de Aprendizagem, que são definidos como qualquer entidade, digital ou não digital, que pode ser utilizada, reutilizada ou referenciada durante o aprendizado apoiado sobre a tecnologia [2, 3]. Não há definição clara de limite de tamanho para um OA, existe, porém, o consenso de que ele deve ter um propósito educacional definido, um elemento que estimule a reflexão do estudante e de que sua aplicação não se restrinja a um único contexto [4].

Há diversos fatores que favorecem o uso de OA na área educacional, como por exemplo: a flexibilidade, a facilidade para atualização, a customização, a interoperabilidade e, por fim, o aumento do valor de um conhecimento. Os OA tratam de assuntos específicos utilizando as metodologias adequadas para aprendizagem de seu conteúdo, aplicando exemplos, testes, entre outras formas, para que o aluno tenha uma total compreensão do tópico ora apresentado. Todas estas vantagens são mais que suficientes para justificar a utilização dos OA nas diferentes modalidades de ensino.

É próprio dos objetos de aprendizagem, principalmente os difundidos na Web, entre outros fatores, em razão de sua natureza digital, o caráter heterogêneo de conteúdo, formas de elaboração, recursos utilizados e linguagem adotada. Nesse sentido, levando em conta esta heterogeneidade, discutir OA, do ponto de vista das possibilidades de avaliação, acarreta em um desdobramento de outras questões a ela inerentes e que correspondem às características dos OA. Existem diversas formas de avaliar um aluno utilizando este recurso: é possível seguir o método mais convencional de avaliação de perguntas e respostas; expor situações em que o aluno fornece o valor de variáveis, contribuindo para a construção ou desenrolar destas situações; apresentar uma explicação preliminar sobre o conteúdo e após isso lançar os questionamentos a respeito; e outras possibilidades que serão apresentadas no decorrer do artigo.

Entretanto, mesmo imerso em tamanha diversidade, é possível identificar padrões relacionados à avaliação do conhecimento assimilado utilizando OA. Para realizarmos a análise de forma organizada, identificamos alguns aspectos a considerar, quando se fala em avaliação dentro de OA. Esses aspectos, que são apresentados a seguir, formam a base para a nossa linguagem de padrões. Um detalhe importante é que, no escopo da linguagem e do artigo, tratamos apenas dos OA digitais, aplicações multimídia e interativas. Todos os OA apresentados neste artigo foram coletados em repositórios disponíveis na internet.

Antes de ingressarmos na descrição da linguagem de padrões desenvolvida nesta pesquisa, discutiremos um pouco questões relevantes no contexto do artigo, como *avaliação*, *padrões de software* e sua terminologia, incluindo o conceito de *linguagem*.

A seção seguinte corresponde à descrição da nossa linguagem, apresentando resumidamente todos os seus padrões e o relacionamento entre eles. Após esta fase, ingressamos na descrição formal dos padrões que fazem parte da primeira parte da linguagem. A última parte do artigo dedica-se a enumerar alguns trabalhos relacionados a nosso objeto de estudo, indicando aqueles que trazem padrões úteis ao nosso contexto.

2. Conceitos e Terminologias

Dentro do contexto educacional, a avaliação da aprendizagem é muito mais que uma disciplina de Pedagogia. Ela corresponde a toda uma área de conhecimento da educação, com muita pesquisa desenvolvida e, ao mesmo tempo, ainda em evolução.

A idéia de avaliação passou por muitas mudanças, partindo da concepção quantitativa de medição, até as visões qualitativas mais progressistas. Todas essas questões podem ser compreendidas tanto em trabalhos que discutem a primeira abordagem [5] [6], como nos que defendem a segunda [7] [8] [9]. São relevantes ainda as implicações da avaliação no contexto das novas tecnologias e da educação à distância [10] [11] [12].

Para este artigo, entende-se "avaliação" dentro de um OA como todas as formas pelas quais é necessária a intervenção do aluno no cenário do objeto, no que diz respeito à expressão do seu entendimento sobre determinado assunto e que, por ventura, possa ser utilizado como recurso para avaliar a aprendizagem deste conteúdo. As avaliações podem ser problemas, questões, situações cotidianas, entre outras formas. É desta avaliação que nos referimos no decorrer do artigo.

A respeito da teoria dos padrões de software, que teve sua origem nos estudos do arquiteto Christopher Alexander [13] [14], é importante saber que um padrão descreve um problema de projeto e uma solução geral para o problema em um contexto particular [15]. Desta forma, utilizando a definição alexandrina, cada padrão é uma regra de três partes, que expressa uma relação entre um certo contexto, um problema e uma solução.

Como forma de orientação para os leitores, os padrões apresentados nesse artigo são descritos utilizando-se oito elementos. O primeiro elemento, *Nome,* é a sua forma de identificação, representado, neste artigo pelo próprio título do tópico. O *Contexto* indica a situação em que o padrão deve ser aplicado. O *Problema* apresenta a questão que expressa o problema que o padrão resolve. O elemento *Forças* descreve as forças que direcionam o padrão para suas possíveis soluções. A *Solução* apresenta uma resposta à questão relacionada no elemento *Problema* e que resolve as forças da melhor forma possível. O *Racional* mostra porque a solução resolve o problema, como as forças foram tratadas e o que há por trás da solução. O *Contexto Resultante* indica o estado do sistema após a aplicação do padrão, apresentando freqüentemente suas conseqüências. Os *Usos Conhecidos* descrevem alguns dos lugares onde o padrão é utilizado e, por fim, os *Padrões Relacionados* identificam, quando existem, outros padrões que são importantes para o padrão descrito. É importante ressaltar que para este artigo esta seção traz apenas o relacionamento entre os padrões da linguagem, pois os padrões relacionados ao desenvolvimento de OA, de uma maneira geral, sem considerar os

padrões em particular, estão descritos na seção *Trabalhos Relacionados*. Exemplos sobre sua aplicação podem ser encontrados em [16].

Para o caso da nossa linguagem de padrões, os elementos descritos anteriormente não são todos obrigatórios. Além disso, os elementos *Problema* e *Solução* são suficientes para se ter uma visão geral do padrão, enquanto que os outros elementos explicam o raciocínio utilizado para a construção do mesmo, permitindo que o leitor tenha uma visão aprofundada do padrão.

Algo que ainda precisa ser definido é a expressão *linguagem de padrões* que representa o núcleo deste artigo. Conforme definida por Coplien [15], uma linguagem de padrões é uma coleção de padrões que necessitam de um ao outro para gerar um sistema. Um padrão isolado resolve um problema isolado; uma linguagem de padrões constrói um sistema. O autor acredita ainda que é através das linguagens de padrões que a abordagem de padrões mostra todo seu potencial.

É importante destacar que a linguagem de padrões deve ser completa, todos os aspectos do domínio abordado são importantes na definição dos padrões. Entretanto, cada padrão pode ser usado separadamente ou e conjunto com alguns padrões da linguagem. Desta forma, um padrão individualmente é considerado útil mesmo se a linguagem não for usada em sua plenitude [17].

A seção a seguir apresenta a linguagem de padrões para avaliação do conhecimento em OA desenvolvida neste artigo. Serão apresentados o escopo da linguagem, os aspectos considerados em seu desenvolvimento, a descrição resumida de todos os padrões e o esclarecimento de como eles se relacionam.

3. A Linguagem de Padrões

A linguagem de padrões apresentada nesse trabalho abrange uma quantidade considerável de fatores relacionados à avaliação do conhecimento através de OA. Assim, para melhor contextualizar este escopo, definimos um conjunto de aspectos importantes relacionados à noção de avaliação. É a partir destes aspectos que organizamos de forma sistemática e didática os padrões apresentados[1].

Tomando o OA do ponto de vista da avaliação, seis *Aspectos* foram então considerados para nossa análise: *Tipo de Avaliação*, *Propósito do Objeto de Aprendizagem*, *Seqüência das Questões*, *Relação entre Conteúdo e Avaliação*, *Recursos Utilizados* e *Comportamento Diante das Respostas*, todos tendo sempre em mente o contexto da avaliação. Desta forma, cada aspecto define um grupo/conjunto de padrões. O número de padrões para cada um destes aspectos varia entre dois e quatro, totalizando dezesseis padrões para a linguagem aqui apresentada. A Tabela 1 relaciona os grupos que ordenam esta linguagem de padrões, destacando o problema básico que cada conjunto procura solucionar.

[1] A própria questão da avaliação é demasiadamente abrangente, por isso, não entram no escopo da linguagem fatores como: eficácia da avaliação, avaliação para determinada disciplina, conteúdo visto antes da avaliação, percepção do usuário da avaliação por OA, entre outros.

Tabela 1 - Definição dos aspectos/conjuntos da linguagem

ASPECTO	DESCRIÇÃO
Tipo de Avaliação	Determina quem avalia o aluno.
Propósito do Objeto de Aprendizagem	Estabelece o principal intuito do OA: conteúdo ou avaliação.
Seqüência das Questões	Trata da forma que se dá a seqüência das questões?
Relação entre Conteúdo e Avaliação	Determina quem vem primeiro: o conteúdo ou a avaliação.
Recursos Utilizados	Corresponde à maneira de apresentar as questões e problemas.
Comportamento Diante das Respostas	Estabelece o que acontece depois que o aluno responde a uma questão.

Por questões de espaço, o escopo deste artigo comporta apenas os padrões relacionados aos dois primeiros aspectos: *Tipo de Avaliação* e *Propósito do Objeto de Aprendizagem*, somando-se entre eles quatro padrões. A seguir será apresentada uma descrição de todos os seis aspectos considerados no agrupamento dos padrões para esta linguagem. Apesar de não tratarmos aqui dos padrões pertencentes aos quatro últimos conjuntos, a sua descrição é importante para que se compreenda a linguagem por inteiro.

O primeiro aspecto, *Tipo de Avaliação*, como o nome indica, diz respeito ao tipo de avaliação presente no OA. A quem se direciona o resultado da avaliação? Quem deve tomar conhecimento da quantidade de acertos, do desempenho do aluno na resolução dos problemas? Dessa forma, definimos os seguintes padrões: AUTO-AVALIAÇÃO, obviamente, em que os alunos são auto-avaliados e AVALIAÇÃO SUPERVISIONADA, em que os professores têm acesso aos resultados, acompanham o processo de avaliação.

O aspecto *Propósito do Objeto de Aprendizagem* relaciona-se com a natureza do OA: existem alguns estritamente teóricos, que concentram esforços em apenas expor o conteúdo, sem preocupação com a fixação ou avaliação desse conteúdo. Esta modalidade de OA não será considerada em nossa análise exatamente por não fornecer recursos diretos de avaliação[2]. Um segundo tipo é aquele que ainda possui uma ênfase teórica, oferecendo, entretanto, recursos para fixação e avaliação do conteúdo (OBJETO DE APRENDIZAGEM TEÓRICO). A última possibilidade nesse grupo são os OA que não possuem conteúdo explicativo explícito: eles são os próprios exercícios, são a própria avaliação (OBJETO DE APRENDIZAGEM PRÁTICO). Como não há matéria formal dentro do OA, para utilizá-lo, ou seja, resolver os problemas existentes, o aluno deve conhecer o assunto previamente, é preciso que ele tenha passado por uma aula a respeito ou tenha estudado o conteúdo.

Seguimos então com a explicação dos aspectos analisados, agora tratando daqueles cujos padrões não serão descritos formalmente neste artigo, para que, como dito anteriormente, compreenda-se a relação dos conjuntos entre si e se perceba a linguagem de padrões em sua essência.

[2] É preciso deixar claro que a não disponibilidade de exercícios ou questionamentos deste tipo de objeto não compromete sua qualidade pedagógica. O professor pode realizar avaliações externamente ao OA, utilizando o método que considerar mais conveniente.

A respeito da seqüência das questões (aspecto *Seqüência das Questões*), observou-se que há duas possibilidades: a) As questões são dependentes umas da outras e dadas de forma seqüencial (SEQÜÊNCIA LINEAR). Para passar à questão seguinte (ou até para continuar no OA), é preciso resolver o problema, solucionar a pergunta anterior. b) Os problemas podem ser resolvidos em qualquer ordem (SEQÜÊNCIA NÃO LINEAR). Caso não saiba a resposta de uma questão, o aluno pode passar para a próxima ou prosseguir com a "leitura" do OA.

O aspecto seguinte, *Relação entre Conteúdo e Avaliação*, considera a relação entre o conteúdo apresentado e os exercícios, no que diz respeito à ordem de apresentação das matérias e das avaliações, dentro do OA. Alguns OA expõem todo o conteúdo primeiro e depois disponibilizam uma ou mais questões para o aluno (CONTEÚDO ANTES DA AVALIAÇÃO). Outros intercalam conteúdo com exercícios: na medida em que vão sendo adicionados conceitos, são apresentados exemplos para o aluno aplicá-los (CONTEÚDO E AVALIAÇÃO INTERCALADOS). Um terceiro tipo (AVALIAÇÃO ANTES DO CONTEÚDO) é aquele que não apresenta conteúdo inicialmente. Há, antes dele, uma situação-problema que o aluno deve solucionar. Depois de resolvida, vem a explicação do assunto envolvido no contexto apresentado.

O próximo aspecto estabelecido, *Recursos Utilizados,* está relacionado aos recursos gráficos e de interface utilizados nas questões e exercícios. Há uma infinidade deles: de questões de múltipla escolha a simulações, as mais diversas, passando ainda por preenchimento de lacunas e outras respostas em aberto. Definimos para nossa linguagem os seguintes padrões: QUESTÕES DE MÚLTIPLA ESCOLHA, QUESTÕES ABERTAS, SIMULAÇÕES e IMAGENS E GRÁFICOS.

Comportamento Diante das Respostas, o último aspecto identificado, refere-se ao tratamento dado pelo objeto às repostas dos alunos. Diz respeito ao que acontece dentro do OA após um erro ou um acerto. Algumas vezes, é possível retornar e tentar outra resposta (TENTATIVA E ERRO), outras vezes, é apresentada a conseqüência para a resposta escolhida (APRESENTAÇÃO DAS CONSEQÜÊNCIAS) e outras vezes, não é possível conhecer a resposta certa (AUSÊNCIA DE GABARITO).

Como síntese do que foi dito até aqui, a Tabela 2 apresenta todos os aspectos de avaliação considerados na pesquisa e os respectivos padrões identificados para cada um deles. Destacamos em cinza os padrões detalhados neste artigo.

Tabela 2 - Conjuntos da linguagem e seus padrões

ASPECTO	PADRÕES
Tipo de Avaliação	Auto-Avaliação Avaliação Supervisionada
Propósito do Objeto de Aprendizagem	Objeto de Aprendizagem Teórico Objeto de Aprendizagem Prático
Seqüência das Questões	Seqüência Linear Seqüência Não Linear
Relação Entre Conteúdo e Avaliação	Conteúdo Antes da Avaliação Conteúdo e Avaliação Intercalados Avaliação Antes do Conteúdo
Recursos Utilizados	Questões de Múltipla Escolha Questões Abertas Simulações

	Imagens e Gráficos
Comportamento Diante das Respostas	Tentativa e Erro Apresentação das Conseqüências Ausência de Gabarito

O É importante observar que os padrões da linguagem relacionam-se entre si de duas maneiras, conforme apresentado na Figura 1: a) os padrões de um grupo podem relacionar-se com os padrões de outro grupo; b) os padrões de um mesmo grupo podem ou não se relacionar entre si. Em outras palavras, é possível que um OA adote vários padrões da linguagem ao mesmo tempo. De fato, um OA deve utilizar pelo menos um padrão (mas não necessariamente apenas um) de cada aspecto considerado, dependendo da intenção do desenvolvedor. Por exemplo, o primeiro objeto de aprendizagem mostrado neste artigo (item *a* do tópico Usos Conhecidos na seção 4.1), usa os seguintes padrões: AUTO-AVALIAÇÃO, OBJETO DE APRENDIZAGEM PRÁTICO, SEQÜÊNCIA NÃO-LINEAR, CONTEÚDO E AVALIAÇÃO INTERCALADOS, QUESTÕES DE MÚLTIPLA ESCOLHA, SIMULAÇÕES, TENTATIVA E ERRO e APRESENTAÇÃO DAS CONSEQÜÊNCIAS. Desta forma, é possível desenvolver uma variedade imensa de OA, combinando-se os padrões entre si de cada conjunto.

Na descrição dos padrões, isto ficará claro com alguns exemplos de OA que aparecem em mais de um padrão, demonstrando assim o intercâmbio existente entre os padrões de cada aspecto.

A Figura 1 a seguir apresenta a relação entre os grupos e padrões da linguagem. Destacamos em cinza, os padrões descritos neste artigo. Conforme pode ser visto, existe relacionamento entre todos os grupos (indicado pela seta). Pela imagem, entende-se que todos os padrões de um grupo devem usar pelo menos um padrão de todos os outros grupos, com apenas uma exceção: vê se que no grupo *Propósito do Objeto de Aprendizagem*, apenas o OBJETO DE APRENDIZAGEM TEÓRICO relaciona-se com os padrões do grupo *Relação entre Conteúdo e Avaliação*, pois para utilizar os padrões deste último grupo, é necessário que exista conteúdo formal no OA, o que não é o caso do OBJETO DE APRENDIZAGEM PRÁTICO.

A respeito do relacionamento entre padrões do mesmo grupo, é possível perceber que três grupos apresentam padrões exclusivos entre si (caixa tracejada). Por exemplo, se um OA adota o SEQÜÊNCIA LINEAR, ele não pode utilizar o SEQÜÊNCIA NÃO-LINEAR. Os outros três grupos possuem padrões que podem usar um outro padrão do mesmo grupo. Por exemplo, nada impede que um OA que use o SIMULAÇÕES, adote também o QUESTÕES ABERTAS e o QUESTÕES DE MÚLTIPLA ESCOLHA.

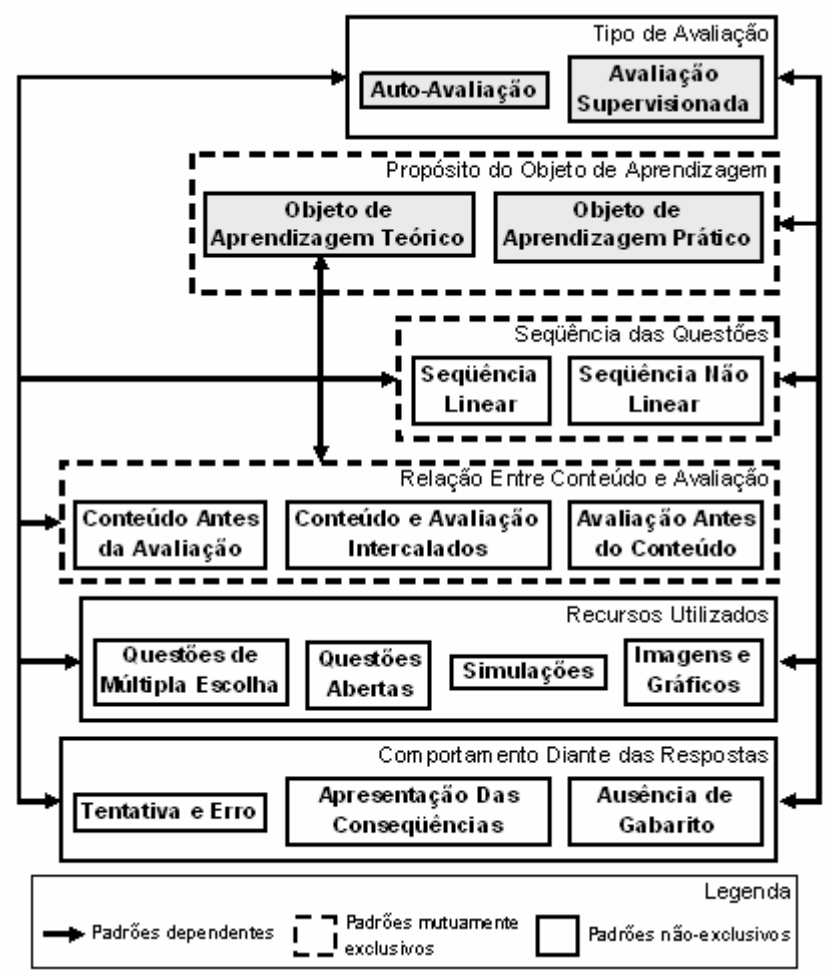


Figura 1 – Relação entre os padrões da linguagem

4. Aspecto *Tipo de Avaliação*

4.1. Padrão AUTO-AVALIAÇÃO

Contexto

Ao desenvolver um OA que ofereça possibilidades de fixação e avaliação do conteúdo, deseja-se que o aluno seja auto-avaliado, que ele mesmo solucione os problemas apresentados e tome consciência de seu desempenho.

Problema

Como permitir que um aluno que utilize um OA possa aferir, por si só, o seu aprendizado?

Forças

OA desenvolvidos para a Web oferecem diversas possibilidades de recursos de layout e interface, permitindo a criação de ferramentas interativas. Recursos comuns à Internet

são interessantes para os alunos, pois diferem dos métodos convencionais de sala de aula.

Aspectos como idade, conhecimento de tecnologia, disciplina abordada podem levar os alunos a não gostarem de ser auto-avaliados por meio de OA. Eles podem ter dificuldades de uso e compreensão da intenção da auto-avaliação.

A forma como os resultados dos problemas são apresentados aos alunos pode comprometer a interpretação de sua avaliação.

A relação entre as questões apresentadas e o conteúdo abordado no OA determina a qualidade da avaliação.

Solução

Ofereça recursos, dentro dos OA, que possibilitem aos alunos realizarem, de forma atrativa, sua auto-avaliação. Os alunos devem conhecer seu desempenho nas questões e problemas. A maneira mais simples é fornecer a resposta certa aos questionamentos, ou informar se o aluno errou ou acertou. Métodos de contagem de pontos, associados ao número de acertos são outras formas válidas do aluno perceber como se dá quantitativamente seu aprendizado. Os questionamentos devem corresponder ao conteúdo aprendido, para que a análise do aluno se faça de maneira direta. Toda a construção da avaliação dentro do OA deve ser feita de forma a convidar o aluno a conhecer seus resultados, a ter consciência do que foi aprendido.

Racional

A adoção do AUTO-AVALIAÇÃO permite que o aluno possa decidir sobre o seu próprio aprendizado. O padrão estabelece que a avaliação seja feita de forma interessante, levando o aluno a ter responsabilidade sobre seu conhecimento. É necessário ainda que esse tipo de avaliação seja aplicado para alunos que tenham maturidade para realizar uma melhor avaliação dos resultados obtidos. Além disso, deve ser levado em consideração o teor do conteúdo na elaboração da avaliação.

Contexto Resultante

Objetos de aprendizagem que utilizem o AUTO-AVALIAÇÃO permitem que o aluno tire conclusões sobre o seu desempenho, através de sua própria análise. Utilizando o padrão, o método auto-avaliativo contribui na formação do aluno, no que diz respeito à tomada de consciência, por si só, da evolução de seu aprendizado e de seu desempenho na fixação do conteúdo estudado. A partir da avaliação feita, o aluno tem condições de reforçar seu estudo exatamente nos aspectos em que percebeu maior deficiência.

Uma conseqüência da auto-avaliação enquanto método pedagógico é o desinteresse do aluno. Ela pode não levar a nada, pois o aluno toma conhecimento de seu desempenho, mas pode não tomar providências para melhorá-lo. Além disso, o aluno pode não ter consciência do objetivo da auto-avaliação, pode considerar as questões nos OA apenas como “exercícios de fixação” ou “joguinhos”, não levando a sério seus resultados.

Usos Conhecidos

Diversos OA observados utilizam AUTO-AVALIAÇÃO, pois a disponibilidade dos objetos pela Web favorece o estudo individual e solitário, exigindo do aluno a responsabilidade sobre sua própria avaliação. Observou-se assim a tendência de se deixar para o aluno a tarefa de perceber os erros e acertos nas questões resolvidas. Seguem abaixo alguns OA que utilizam esse padrão, destacando os recursos utilizados para essa auto-avaliação.

a. O salto dos recordes [18]: A idéia principal do objeto é permitir que o aluno compreenda algumas noções de Mecânica. O aluno deve fornecer as informações de ângulo e velocidade para um motoqueiro saltar com sucesso de uma rampa. Na tela do OA, há a indicação dos valores fornecidos pelo aluno (Figura 2). A todo momento, o aluno vai percebendo seu desempenho e avaliando quais valores deve usar, pois há a simulação do que ocorre com a moto, a cada valor informado. Pela imagem: o aluno inicialmente (parte superior da Figura 2) não obteve sucesso com o salto, há então a representação da explosão da moto (mancha clara indicada pela seta); depois (parte inferior da Figura 2), ele fornece valores corretos e consegue atravessar a rampa (moto em destaque). Prosseguindo no OA, há ainda questões relacionadas ao que ocorreu durante a simulação. Caso o aluno tenha compreendido bem a dinâmica nos valores das variáveis, vai ter consciência de seu aprendizado na resolução das questões.

Figura 2 - Auto-avaliação no OA “Salto dos Recordes”

b. A química das cores nos fogos de artifício [19]: Neste OA, o caráter auto-avaliativo está presente em alguns momentos: depois de toda a explicação do conteúdo, é fornecida uma tabela com elementos químicos e as cores que eles provocam nos fogos de artifício. Há um cronômetro (na Figura 3, indicado com uma seta) ao lado e o aluno deve memorizar a correspondência das cores em até trinta segundos. Este recurso desafia o aluno a memorizá-las mais rapidamente, além de informar a ele quanto tempo levou, já que pode prosseguir no OA assim que decorar as cores e isso pode levar menos tempo que o máximo permitido. Em seguida, é proposto um exercício que exige do aluno a correspondência das cores com os materiais apresentados. Além de acertar os elementos, o aluno deve fazê-lo em um tempo limitado (ilustrado pela queima de uma vela, como se vê na Figura 3). A cada erro ou acerto o aluno compreende a qualidade da memorização feita momentos antes. Na imagem, o brilho no céu representa a queima correta do rojão constituído de sódio, que emite uma cor amarela.

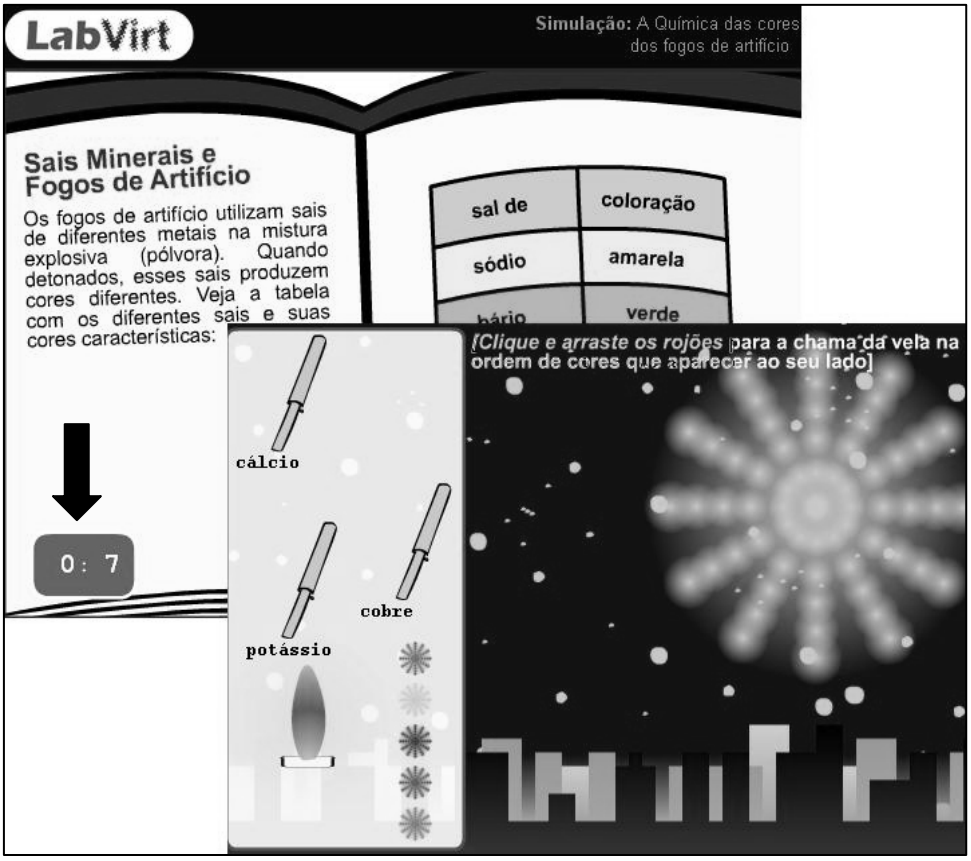


Figura 3 - Auto-avaliação no OA “Química das cores nos fogos de artifício”

c. Calculadora quebrada [20]: Este é um OA que se caracteriza muito mais como um jogo de raciocínio. A auto-avaliação existe no desafio de conseguir todos os números solicitados no tempo determinado (cronômetro indicado na Figura 4 pela seta escura). O aluno pode avaliar-se também através dos níveis existentes. Pode concluir como está seu desempenho matemático, a partir de que nível máximo conseguiu alcançar.

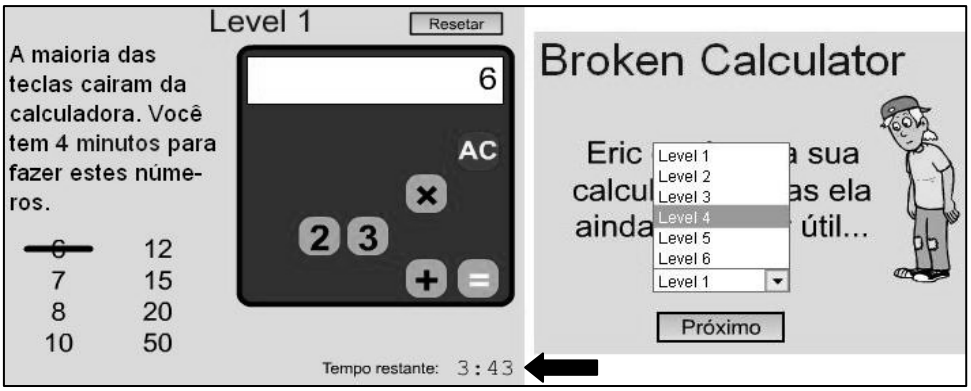


Figura 4 - Auto-avaliação no OA "Calculadora quebrada"

d. Os Concelhos[3][21]: É um exemplo da utilização de contagem de pontos para a auto-avaliação do aluno. É um OA desenvolvido para a disciplina de História e caracteriza-se por um conjunto de questões de múltipla escolha a respeito dos Concelhos e documentos históricos de Portugal. Cada pergunta vem acompanhada de um texto e de algumas imagens relacionados. Ao final do questionário, são passados para o aluno seus resultados em forma de porcentagem, além de apresentar as questões que ele acertou, as que ele errou e as respostas corretas. A Figura 5 mostra a primeira questão apresentada e os resultados obtidos, destacando-se a exibição de uma das questões erradas.

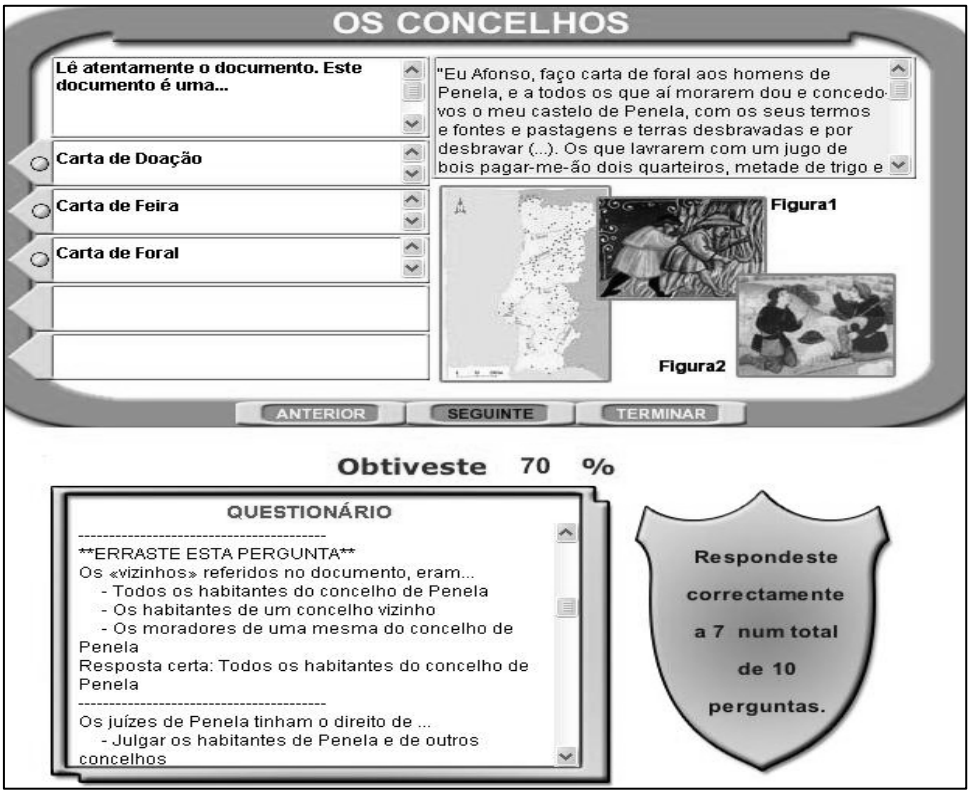


Figura 5 - Auto-avaliação no OA "Os Concelhos"

[3] Grafia original do português de Portugal. Refere-se a um acontecimento histórico e tem significado diferente da palavra "conselho".

Padrões Relacionados

O AUTO-AVALIAÇÃO relaciona-se com o AVALIAÇÃO SUPERVISIONADA de forma não excludente, pois um mesmo OA pode adotar os dois padrões.

Os padrões do grupo *Propósito do Objeto de Aprendizagem* são utilizados para determinar o foco do OA, que pode ter uma ênfase teórica ou prática.

A continuidade do OA, ou seja, a "liberdade" de leitura que o aluno possui é determinada pelos padrões do grupo *Seqüência das Questões*. São os padrões deste grupo que determinam se o aluno pode, por exemplo, passar para uma questão seguinte sem acertar a resposta da anterior.

Em um OA que adote o AUTO-AVALIAÇÃO, desde que possua conteúdo formal, é preciso estabelecer a ordem com que os conteúdos e as avaliações são apresentadas. Isto é feito com os padrões do grupo *Relação entre Conteúdo e Avaliação.*

Toda a avaliação dentro do OA é feita através de recursos pedagógicos e de interface. Os padrões do grupo *Recursos utilizados* determinam as formas de apresentação dos problemas dentro do OA.

Em uma auto-avaliação, a forma de se apresentar a resposta das questões ao aluno contribui para que ele tome conhecimento da qualidade de sua aprendizagem. Dessa forma, para decidir a maneira com que as respostas serão apresentadas, devem ser usados os padrões do grupo *Comportamento Diante das Respostas*.

4.2. Padrão AVALIAÇÃO SUPERVISIONADA

Contexto

Ao desenvolver um OA que oferece possibilidades de fixação e avaliação do conteúdo, deseja-se que o professor tome conhecimento do desempenho do aluno, através das questões propostas no próprio OA.

Problema

Como construir OA em que o professor tome conhecimento dos resultados da avaliação aplicada ao aluno?

Forças

A utilização de OA muitas vezes se dá sem a presença do professor. O aluno tende a estudar sozinho ou com algum outro aluno. Essa característica cria uma barreira "física" para o avaliador.

Neste método, a avaliação é feita por meio digital, com um recurso diverso daqueles aos quais os alunos estão habituados, o que pode gerar uma falta de comprometimento dos alunos.

Os resultados devem ser apresentados de modo a permitir a compreensão do professor sobre o desempenho do aluno na assimilação do conteúdo.

Como os OA possuem vários recursos e características que despertam a atenção dos alunos, é necessário que a avaliação seja mais agradável e não seja tão "traumática" como as provas comuns.

Algumas questões de segurança relacionadas à autoria das respostas devem ser consideradas quanto à utilização desses resultados na avaliação formal dos alunos.

Solução

Elabore questões e exercícios com algum recurso que permita ao professor ter conhecimento do resultado do desempenho do aluno. Há diversas maneiras de o professor conhecer o desempenho do aluno: ele pode acompanhá-lo no uso do OA e ver como ele se sai nos problemas; pode receber as respostas do aluno ou o resultado das questões por endereço eletrônico ou por SMS para o celular ou outros dispositivos móveis; entre outras maneiras. O envio pode ser feito de forma automática pelo OA ou manualmente pelo próprio aluno, dependendo do formato das questões. No AVALIAÇÃO SUPERVISIONADA, o importante é fornecer ao professor meios diretos para que ele possa acompanhar a aprendizagem do aluno. A maneira pela qual os seus resultados são repassados para o professor é uma escolha do desenvolvedor: eles podem estar agrupados por blocos de questões, conforme for organizado o OA; podem vir em forma de porcentagem ou número absoluto de acertos; podem ser ordenadas por questões certas ou erradas; entre diversas outras formas.

Racional

Com a utilização do AVALIAÇÃO SUPERVISIONADA, o aluno será avaliado pelo professor, a ele cabe considerar o desempenho do aluno, contabilizando ou não o resultado da avaliação no esquema de pontuação e notas convencional. Isso fornecerá uma motivação para que o aluno empenhe-se na utilização do OA. Caso o professor deseje utilizar o OA para dar notas aos alunos, devem ser consideradas as questões de segurança em relação ao acesso aos resultados.

Usos Conhecidos

Nos OA listados a seguir, pode-se perceber um certo esforço em levar ao professor o conhecimento da avaliação do aluno. Nota-se que o resultado da avaliação é muito mais voltado para o professor, embora também seja importante para o aluno. Alguns OA oferecem aos usuários a possibilidade de entrar em contato com o professor, por meio de formulário próprio do objeto, encaminhando a mensagem diretamente para o e-mail do professor, ou usando outras estratégias, como nos seguintes OA:

a. *Geography Quiz* [22]. Este objeto consiste em um jogo de perguntas e respostas de geografia envolvendo os continentes, com variação de pontos entre as questões. Ele oferece duas maneiras (seta escura na Figura 6) de estabelecer comunicação com o professor: a) utilizando o comando "*Submit Results*", que enviará o resultado do *quiz* (quantidade de questões acertadas) diretamente para o e-mail do professor, através de uma ferramenta de gerenciamento de e-mail (Outlook, Eudora etc); e b) utilizando o comando "*Task Manager*", através do qual o aluno pode comunicar-se com o

professor, enviando-lhe uma mensagem, preenchida em um formulário (em destaque na Figura 6).[4]

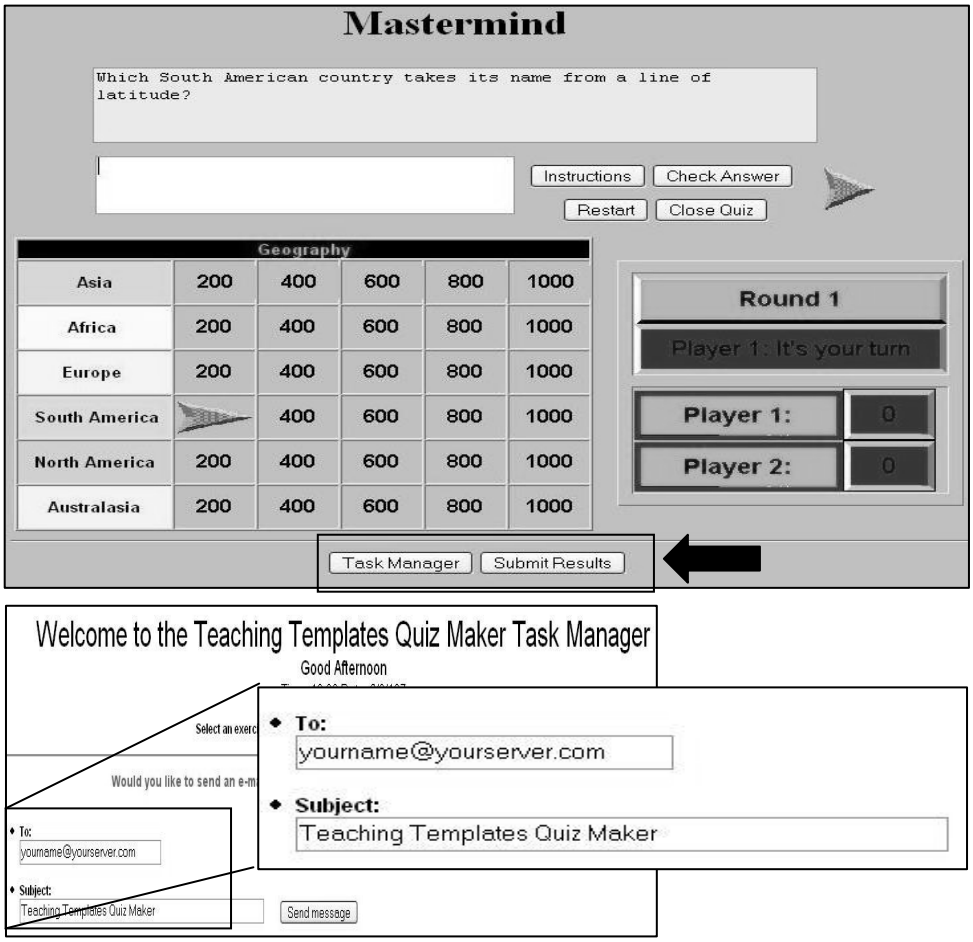


Figura 6 - AVALIAÇÃO SUPERVISIONADA no OA "Geography Quiz"

b. É hora de colocar as coisas no lugar! [23]: A idéia principal é que o aluno faça conexões com elementos ligados à genética e monte essas relações em um quadro, utilizando imagens e conectores (Figura 7). Como são muitas possibilidades, o próprio aluno não tem como ter certeza de que seu trabalho está correto. Então, é recomendado que o professor observe a montagem feita por ele e avalie se está coerente.

[4] Estes são alguns dos recursos disponibilizados pela ferramenta de autoria Tac-soft [14], a partir da qual foi feito o objeto de aprendizagem citado.

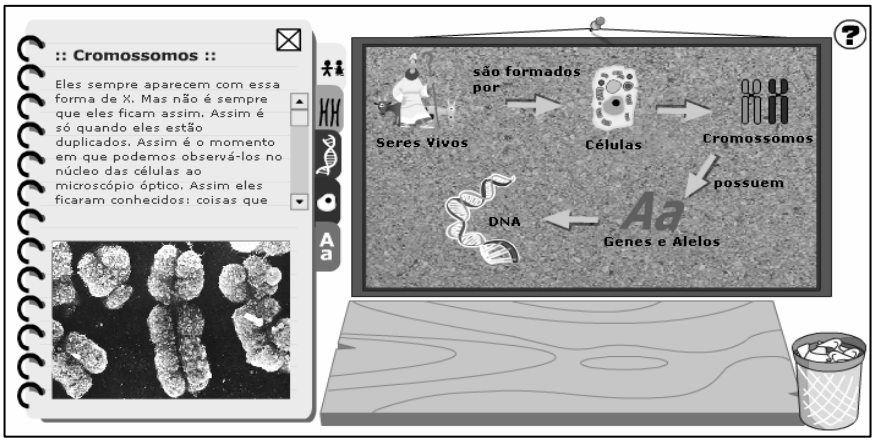


Figura 7 - AVALIAÇÃO SUPERVISIONADA no OA "É hora de colocar as coisas no lugar!"

c. Uma aventura na União Européia [13]: O objeto é semelhante a um jogo que envolve conhecimentos gerais (cultura, arte, geografia) dos países da Europa. Logo ao entrar no site, o usuário é convidado a conhecer a colocação dos participantes (Figura 8) dispostos em forma de ranking[5] para então iniciar a "aventura" proposta pelo OA. O professor pode conhecer o desempenho do avaliado por meio de sua colocação no ranking e utilizar essa informação para adicionar pontos ou bônus nas notas dos alunos. Na imagem há um exemplo de desafio com o qual o aluno se depara durante a utilização do OA.

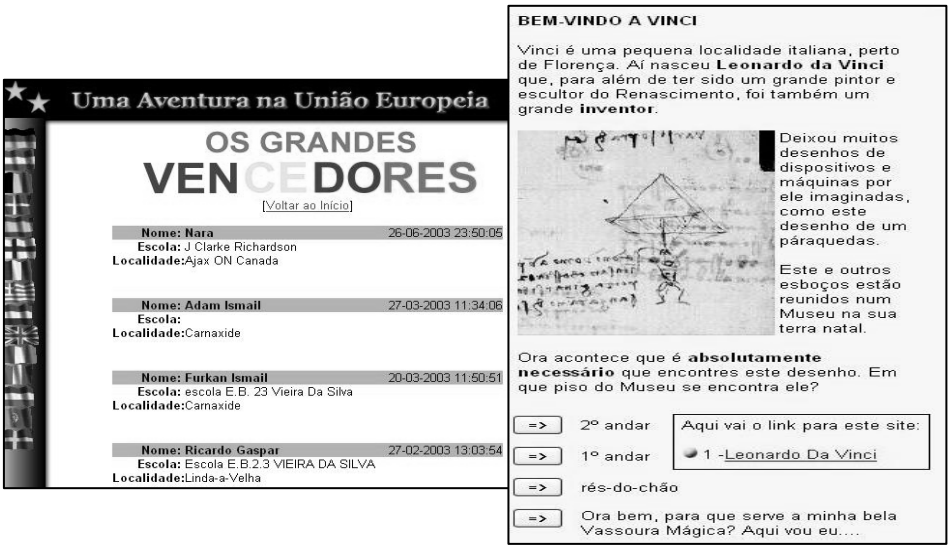


Figura 8 - AVALIAÇÃO SUPERVISIONADA no OA "Geography Quiz"

[5] O recurso de ranking também é um recurso oferecido por uma ferramenta de autoria da qual originou-se o OA em questão. Dessa vez trata-se do Quandary [15].

Padrões Relacionados

Caso se deseje que, além do professor, o aluno também tenha conhecimento dos resultados de sua avaliação, deve ser utilizado AUTO-AVALIAÇÃO.

Com os padrões do grupo *Propósito do Objeto de Aprendizagem*, o professor determina se a avaliação do aluno será feita em um OA que apresenta a explicação do conteúdo ou em um OA estritamente prático.

Os padrões do grupo *Seqüência das Questões* são utilizados para determinar a maneira com que os alunos têm acesso às questões (de forma linear ou não), gerando conseqüências na supervisão do professor. Por exemplo, é mais difícil para ele acompanhar o andamento na solução de questões que podem ser resolvidas fora de ordem.

Através dos padrões do grupo *Relação entre Conteúdo e Avaliação*, o professor determina em que momento dentro do OA ocorre a avaliação supervisionada.

O formato dos problemas também determina a complexidade da avaliação supervisionada pelo professor. Questões de múltipla escolha, por exemplo, são mais simples de serem corrigidas. Os padrões do grupo *Recursos utilizados* são necessários então para definir estes detalhes.

O que acontece no OA depois que o aluno lança uma resposta é relevante para o professor avaliar aspectos como quantidade de tentativas e exatidão das respostas dadas pelos alunos.

5. Aspecto Propósito do Objeto de Aprendizagem

5.1. Padrão OBJETO DE APRENDIZAGEM TEÓRICO

Contexto

Ao se desenvolver um OA, o professor/desenvolvedor pode aprofundar, nos mais diversos níveis, o conteúdo apresentado. O OA pode ser a primeira ferramenta pela qual o aluno está tendo contato com o assunto, ou pode apresentar-se como um reforço, uma revisão de um conteúdo anteriormente abordado. Neste contexto, considera-se importante a presença de explicação, de detalhamento do conteúdo estudado, para junto dele serem apresentados problemas e questionamentos a respeito.

Problema

Como disponibilizar material para os alunos, através do qual eles possam aprender o assunto de interesse e posteriormente serem avaliados?

Forças

O conteúdo de um OA depende muito da natureza teórica da disciplina e do assunto abordado em particular. Alguns deles exigem explicações mais detalhadas para que a avaliação seja feita satisfatoriamente.

O momento de apresentação do conteúdo para o aluno é um fator relevante. Caso seja a primeira vez que ele entre em contato com o assunto apresentado, é essencial que haja a sua descrição, sua fundamentação teórica e não apenas exercícios e avaliações.

OA que associam conteúdo à avaliação têm uma implementação mais complexa, no sentido de necessitar uma estruturação pedagógica coerente entre conteúdos e questões avaliativas.

Solução

Desenvolva OA que agreguem ao mesmo tempo conteúdo e exercícios de fixação e/ou avaliação. Devem ser objetos que possuam claramente a explicação sobre um determinado assunto e em outro momento apresente problemas relacionados, não obrigatoriamente nesta ordem. O importante é reconhecer que teoria e prática aparecem em blocos distintos. Para o OBJETO DE APRENDIZAGEM TEÓRICO, o objetivo principal é apresentar o conteúdo e a partir dele lançar problemas para o usuário.

Racional

A utilização de OA teóricos é importante, principalmente, para a introdução de novos conceitos. Esses OA devem ser acompanhados de exercícios para facilitar a assimilação do conhecimento. Por conta disso, mesmo sendo complexa a implementação desse tipo de OA, a sua utilização justifica-se pela eficiência para o aprendizado do aluno, que é alcançada pela existência de conteúdo formal e avaliação integrados.

Contexto Resultante

A adoção do OBJETO DE APRENDIZAGEM TEÓRICO implica em uma utilização mais completa do objeto, de forma integrada, em que o aluno não precisa recorrer a fontes externas. Dependendo do aprofundamento dado, o usuário é capaz de absorver toda a matéria estudando apenas pelo OA.

Usos Conhecidos

É possível encontrar vários objetos que seguem o OBJETO DE APRENDIZAGEM TEÓRICO, todos com as seções teóricas bem explícitas dentro do corpo do OA. Listamos aqui apenas alguns exemplos:

a. A química das cores nos fogos de artifício [19]: Conforme apresentado em sua descrição inicial, e AUTO-AVALIAÇÃO, é apresentada inicialmente toda a base teórica a respeito da química relacionada às cores, apresentando sua correspondência com os elementos químicos (ver Figura 3). Após isso, o aluno pode ser avaliado (AUTO-AVALIAÇÃO, conforme dito anteriormente), testando seus conhecimentos através dos problemas apresentados.

b. É hora de colocar as coisas no lugar! [23]: Antes de iniciar a montagem do quadro com as relações dos elementos de genética, há uma exposição dos conceitos relacionados, explicando cada um dos elementos exibidos (ver Figura 7). O aluno só consegue realizar a atividade com sucesso se estudar o conteúdo disponível.

c. A química dentro de um bolo [16]. Este OA também é um bom exemplo de mistura entre conteúdo e avaliação. O objetivo é abordar a questão do balanceamento de

substâncias na Química e o OA inicia com uma analogia aos ingredientes de um bolo. Antes de ser exibido o conteúdo propriamente dito, o aluno é questionado sobre as proporções necessárias de cada ingrediente para fazer mais de um bolo (Figura 9). Em seguida, é apresentada a explicação a respeito do balanceamento de equações.

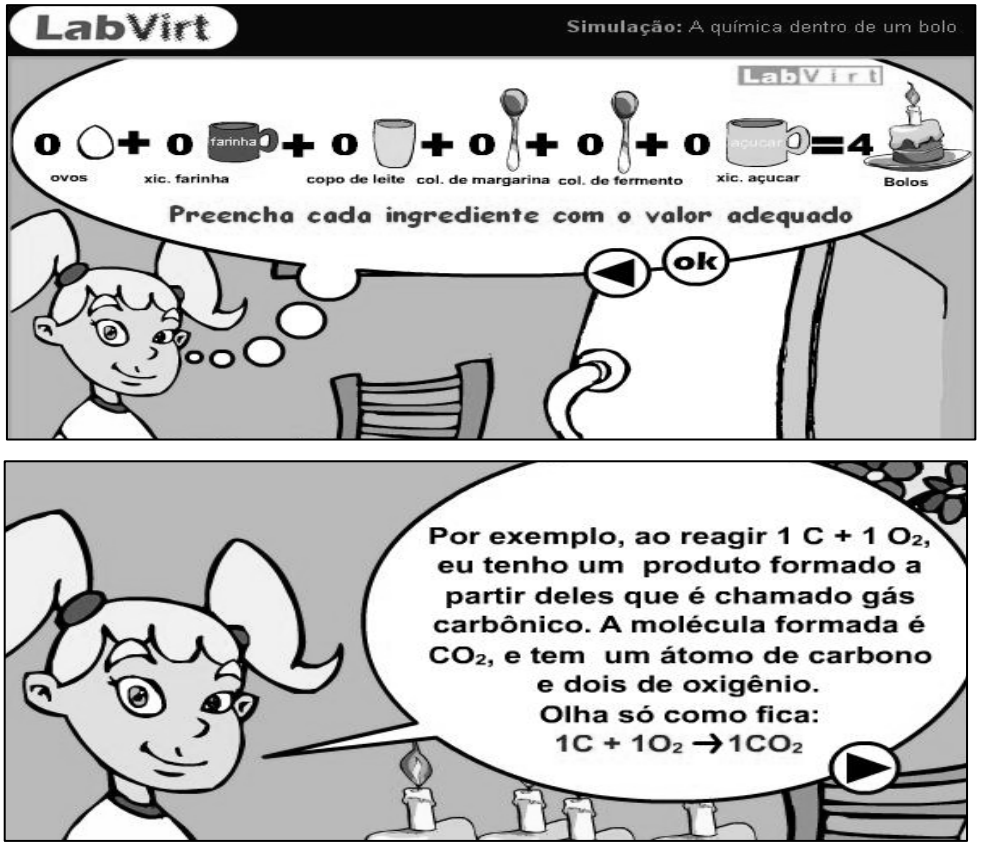


Figura 9 - OA Teórico "A química dentro de um bolo"

Padrões Relacionados

Um OA teórico é indicado para uso em cojunto tanto com o AUTO-AVALIAÇÃO quanto com o AVALIAÇÃO SUPERVISIONADA, dependendo de quem vai receber os resultados da avaliação.

Os padrões dos grupos *Seqüência das Questões* e *Relação entre Conteúdo e Avaliação* são necessários para determinar, por exemplo, se o aluno tem acesso ao conteúdo apenas quando acerta uma questão inicial ou se pode "navegar" entre as questões apresentadas, solucionando-as na ordem que desejar.

O grupo *Recursos Utilizados* provê padrões para auxiliar no desenvolvimento de questões relacionadas ao conteúdo presente no OA.

Com os padrões do grupo *Comportamento Diante das Respostas*, é possível fornecer um retorno ao aluno sobre suas respostas e relacioná-las com o conteúdo apresentado.

5.2. Padrão OBJETO DE APRENDIZAGEM PRÁTICO

Contexto

Após o professor explicar um determinado conteúdo, é recomendável que ele ofereça possibilidades para os alunos testarem seus conhecimentos, fixarem a matéria apresentada e, em última instância, serem avaliados, por ele ou pelo professor. Outro caso comum é quando o aluno já sabe um determinado conteúdo, já viu isso em sala ou estudou por conta própria, e deseja exercitar ou testar seus conhecimentos.

Problema

Como fornecer aos alunos uma forma direta de praticar um determinado conhecimento assimilado?

Forças

Para que o aluno encaminhe-se diretamente para exercícios e avaliações, é necessário que ele possua os pré-requisitos teóricos necessários.

A utilização do objeto de forma indiscriminada (sem a exposição da matéria) pode dificultar o aprendizado do aluno.

Os questionamentos e problemas apresentados devem estar de acordo com a intenção do uso do OA. Assim, o aluno poderá associar a explicação vista anteriormente (visto em sala de aula ou por conta própria) com o exercício/avaliação proposto nele.

O desenvolvimento dos problemas e questões depende apenas da criatividade do desenvolvedor/professor, visto que não há conteúdo formal no OA, aos quais os problemas devam estar vinculados, como ocorre em OBJETO DE APRENDIZAGEM TEÓRICO.

Solução

Crie OA que sejam essencialmente práticos, que contenham apenas questões ou problemas a serem solucionados pelos alunos. Para OBJETO DE APRENDIZAGEM PRÁTICO não é necessário apresentar qualquer conteúdo[6], pois a premissa é que ele já seja de conhecimento do aluno. A característica chave deste padrão é que ele não traz a teoria dentro do OA, não há explicação da matéria (ela já foi vista), ao aluno cabe apenas a resolução dos problemas.

Racional

A utilização de OBJETO DE APRENDIZAGEM PRÁTICO é recomendada para a assimilação de conceitos vistos em sala de aula. É importante que os exercícios sejam realizados no mesmo nível em que os conteúdos foram apresentados.

[6] Isso não significa que o objeto de aprendizagem com esse caráter prático seja desprovido de conteúdo. Referimo-nos apenas ao conteúdo formal, descritivo, explicativo, que, no caso deste padrão, realmente não existe.

Contexto Resultante

O resultado da utilização do OBJETO DE APRENDIZAGEM PRÁTICO é a possibilidade de uma forma mais direta de avaliação por parte do professor e do aluno. Ambos podem conhecer mais rapidamente o nível do aprendizado do aluno.

É possível que a utilização de um OA desenvolvido através desse padrão resulte numa deficiência do aprendizado. Isso ocorre devido à falta de mecanismos para auxiliá-lo quando houver dúvidas ou necessidade de maiores esclarecimentos a respeito da matéria de que trata os problemas.

Usos Conhecidos

Os exemplos de OA mais simples para este padrão são aqueles que se caracterizam como "jogos" ou desafios (*puzzles*, *quizzes*, palavras cruzadas, entre outros), tanto de habilidade de raciocínio, quanto de nível de conhecimento sobre um determinado assunto. Como usos conhecidos desse padrão temos:

a. *Base Blocks Addition* [28]: A utilização do objeto consiste em resolver os problemas de aritmética propostos, deslocando-se os blocos e seus agrupamentos dentro das áreas correspondentes de unidades, dezenas, centenas e milhar (Figura 10). O aluno que utilizar o OA necessariamente terá que possuir noções de soma e multiplicação. Não há conteúdo explicativo, o usuário deve apenas solucionar os problemas apresentados.

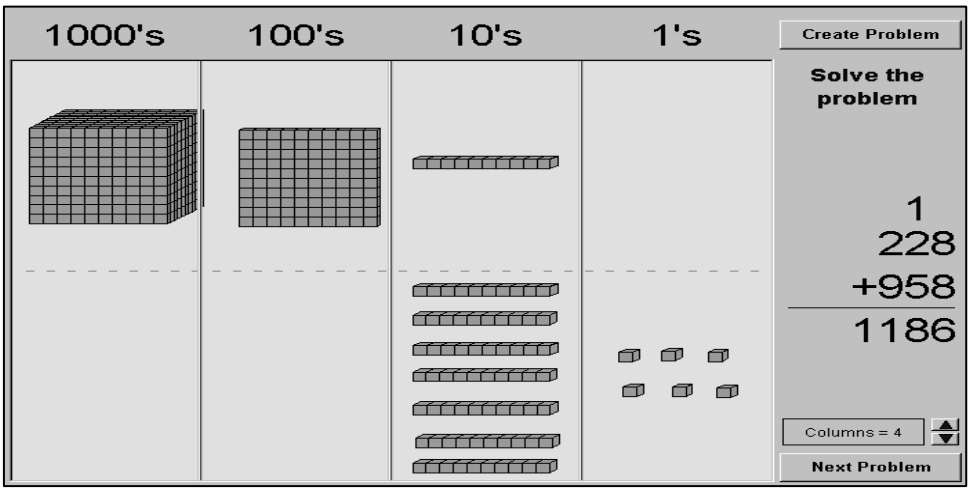


Figura 10 - OA Prático "*Base Blocks Addition*"

b. *Algebra Balance Scale* [18]: Da mesma forma que o OA anterior, este não apresenta nenhum conteúdo. A idéia principal é relacionar os dois lados de uma equação com os dois pratos de uma balança e assim descobrir o valor da incógnita da equação (Figura 11). O objeto apresenta uma seqüência dessas equações que devem ser solucionadas pelo aluno.

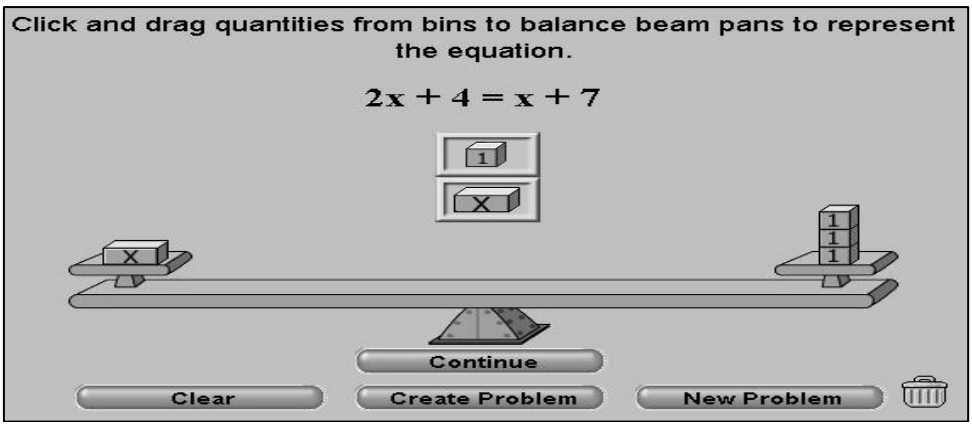


Figura 11 - OA Prático "*Algebra Balance Scale*"

c. Um dublê em apuros [19]: Ao utilizar o OA, o usuário deve fornecer um valor para a distância que um dublê deve saltar de um avião para conseguir cair dentro de um lago. Não há explicações ou qualquer outro conteúdo, apenas o ambiente para a simulação de cada valor fornecido pelo aluno. Na Figura 12, nota-se que o dublê alcançará o lago, pois o valor (2000m) da posição do eixo x informada pelo aluno é satisfatório.

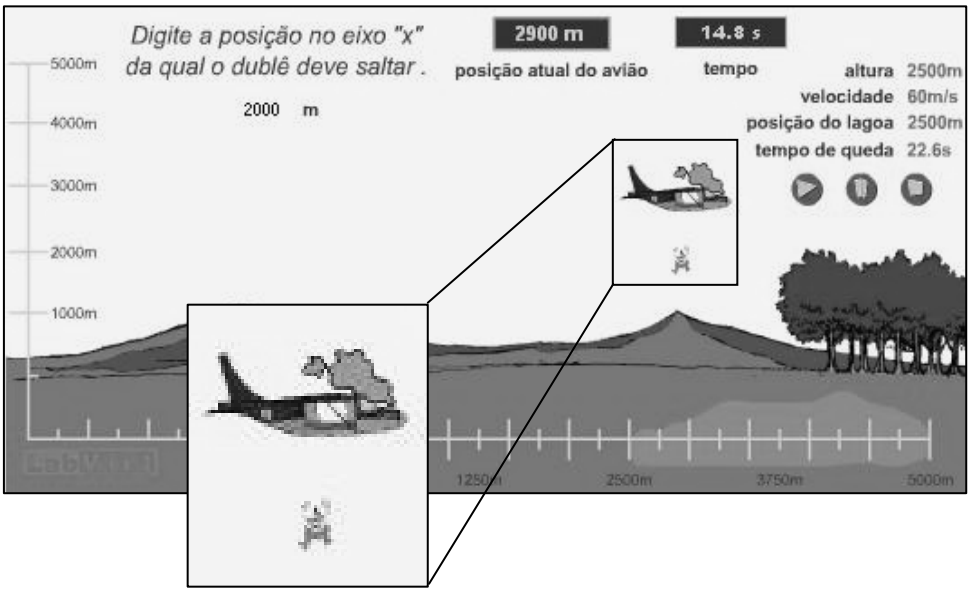


Figura 12 - OA Prático "Um dublê em apuros"

Padrões Relacionados

Em um OA que utilize o OBJETO DE APRENDIZAGEM PRÁTICO, é preciso definir se apenas o aluno terá acesso aos resultados de sua avaliação ou se eles serão consultados pelo professor. Conforme o objetivo, será usado então o AUTO-AVALIAÇÃO ou o AVALIAÇÃO SUPERVISIONADA.

As questões apresentadas no objeto prático devem seguir um dos tipos de seqüência presentes nos padrões do grupo *Seqüência das Questões.*

As maneiras pelas quais o aluno vai praticar seu conhecimento por meio das questões são estabelecidas pelos padrões do grupo *Recursos Utilizados.*

O tratamento dado pelo OA às respostas dos alunos é determinado pelos padrões do grupo *Comportamento Diante das Respostas.*

6. Trabalhos Relacionados

A utilização de padrões para a catalogação de práticas relacionadas ao desenvolvimento de material instrucional no suporte educacional pode ser encontrado em trabalhos como [31], onde foi apresentada a Linguagem de Padrões Cog-Learn. Essa linguagem relaciona um conjunto de padrões pedagógicos que abordam questões de planejamento e seqüência de cursos baseados em práticas de aulas presenciais e padrões de IHC, obtidos de projetos Web e que abordam questões de interação, layout, planejamento e estruturação de material instrucional. Dentre os padrões apresentados na linguagem Cog-Learn, podemos identificar alguns padrões cujas abordagens podem ser mapeadas na elaboração de OA e, portanto, podem ser aplicados junto com os padrões que apresentamos nesse trabalho, sendo particularmente importantes os padrões *ESTRUTURAÇÃO DO CONHECIMENTO* e *CONTEXTUALIZAÇÃO*, que se preocupam na organização dos conteúdos de modo a facilitar a apresentação de novos conceitos aos alunos.

Já em [32], o foco principal é a definição de padrões para tratar aspectos de adaptação de materiais instrucionais para novos contextos educacionais. Para esse fim são descritos padrões relacionados a diferentes áreas de adaptação, como layout e conteúdos. Também esses padrões podem ser aplicados à criação de OA. De fato, a adaptação de conteúdos, e particularmente de material utilizado para avaliação, pode ser necessária de forma a se conseguir uma melhor aplicação dos OA para públicos diferentes. Nesse contexto, padrões como *CORRECT ARRANGEMENT OF ELEMENTS* e *TRANSLATION* podem ser importantes para facilitar a adaptação de conteúdos para se atingir um determinado objetivo avaliativo.

Em [33] diversos padrões são identificados com a finalidade de se identificar mecanismos de registro de utilização de um recurso instrucional digital. Também aqui conseguimos visualizar uma importante relação entre esses padrões com os que apresentamos nesse artigo. De fato, quando falamos de verificar a assimilação de um determinado conteúdo através de um OA, que é um dos objetivos dos nossos padrões, também devemos considerar a forma de como se gerar e armazenar informações que possam fornecer subsídios para se aferir o grau de assimilação desses conteúdos. Assim, a aplicação do padrão *AUTOMATIC GRADING OF STUDENTS' ANSWERS* pode fornecer mecanismos que permitam a geração automática de resultados de avaliações em um OA. Já a aplicação do padrão *CLASSIFICATION OF STUDENTS* pode permitir a identificação de um *ranking* relacionado às avaliações fornecidas por um determinado OA.

7. Agradecimentos

Os autores agredecem o apoio financeiro concedido pelo CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) e pelo CPQT (Centro de Pesquisa e Qualificação Tecnológica). Gostaríamos também de fazer um agradecimento especial ao nosso *shepherd* Rohit Gheyi, pelas valiosas sugestões e comentários, que nos fizeram

refletir em muitos pontos e ajudaram a melhorar bastante o conteúdo e a forma deste artigo.

Referências Bibliográficas

[1] Fernandes, N.L.R. (2004) *Professores e computadores: navegar é preciso*, Porto Alegre: Mediação, pp. 36-41.

[2] Wiley, D.A. (2000), *Connecting learning objects to instructional design theory*: A definition, a metaphor, and a taxonomy in D. A. Wiley (Ed.), The Instructional Use of Learning Objects. Documento on-line, disponível em: <http://reusability.org/read/chapters/wiley.doc>. Acessado em: 17 de novembro de 2006.

[3] IEEE. Learning Technology Standardization Committee (LTSC). Disponível em: < http://ieeeltsc.org/>. Acessado em: 19 de novembro de 2006. [4] BETTIO, R.W. de & Martins, A. *Objetos de aprendizado: um novo modelo direcionado ao ensino a distância*. In: 9o. Congresso Internacional de Educação a Distância, 2002, São Paulo - SP. Acessado em 24 de novembro de 2006. Documento on-line, disponível em <http://www.universiabrasil.net/materia/materia.jsp?id=5938>.

[5] Bonniol, J.J. & Vial, M. (2001). *Modelos de avaliação: textos fundamentais*, Porto Alegre: ARTMED.

[6] Luckesi, C.C., (1996). *Avaliação da aprendizagem escolar*, São Paulo: Cortez.

[7] Franco, M.L.P.B. (1995). *Pressupostos epistemológicos da avaliação educacional*. In SOUZA, C. P. de. Avaliação do rendimento escolar, Campinas: Papirus.

[8] Hadji, C. (2001). *A avaliação desmistificada*, Porto Alegre: ARTMED.

[9] Hoffmann, J.M.L. (1995). *Avaliação mediadora: Uma prática em construção da pré-escola à universidade*, Porto Alegre: Educação e Realidade.

[10] OLIVEIRA, E.S.G. & Costa, M.A. *A avaliação na educação a distância: desafios e progressos*, Rio de Janeiro: UFRJ. Documento on-line, disponível em http://www.universia.pr/congreso/41/41.rtf. Acessado em: 17 de abril de 2007.

[11] OLIVEIRA, E. S. G. et al (2006). *A avaliação da aprendizagem na educação a distância: o diálogo entre avaliação somativa e formativa*. In: 1ª Reunião Anual da ABAVE, 2006, Belo Horizonte. Anais da 1ª Reunião Anual da ABAVE, 2006. Acesso em: 28 de dezembro de 2006. Disponível em: http://www.abave.org.br/publicacao.do?acao=buscar&codpublicacao=129&dest=mostra.

[12] NEDER, M. L. C. (1996). *Avaliação na Educação a Distância: significações para definição de percursos*. In: Educação à Distância, organizado por Oreste Preti. Cuiabá: UFMT/NEAD. Acesso em 28 de dezembro de dezembro de 2006. Disponível em: http://www.nead.ufmt.br/documentos/AVALIArtf.rtf.

[13] ALEXANDER, C. et al. (1977). *A Pattern Language*, New York: Oxford University Press.

[14] ALEXANDER, C. (1979). *The Timeless Way of Building*, New York: Oxford University Press.

[15] COPLIEN, J. O. (1996). *Software Patterns*, USA: SIGS Books & Multimedia.

[16] BUSCHMANN, F., et al., *Pattern-Oriented Software Architecture*, John Wiley and Sons, New York, NY., 1996.

[17] BRAGA, R.T.V. (2001). *Introdução aos padrões de software*, São Paulo: ICMC – Universidade de São Paulo. Documento online, disponível em: http://sugarloafplop2005.icmc.usp.br/NotasDidaticasPadroes.pdf. Acesso em: 17 de abril de 2007.

[18] Laboratório Didático Virtual – USP. *O salto dos recordes*. Disponível em: <http://www.labvirt.futuro.usp.br/applet.asp?time=10:08:28&lom=10707>. Acessado em 03 de janeiro de 2007.

[19] Laboratório Didático Virtual – USP. *A química das cores nos fogos de artifício*. Acessado em 03 de janeiro de 2007. Disponível em: <http://www.labvirtq.futuro.usp.br/applet.asp?time=10:05:44&lom=10819>.

[20] Racha a Cuca. *Calculadora quebrada*. Documento on-line, disponível em <http://rachacuca.com.br/calculadora-quebrada/>. Acessado em 03 de janeiro de 2007.

[21] Centros de Competências Nónio. *Os Concelhos*. Disponível em: <http://nonio.eses.pt/asp/nonio2/soft/wpquest/quest/quest.htm>. Acessado em 03 de janeiro de 2007.

[22] Tac-Software. *Geography Quiz*. Documento on-line, disponível em: <http://www.tac-soft.com/Demoquizzes/GeographyMM.html>. Acessado em 03 de janeiro de 2007.

[23] Rede Interativa Virtual de Educação - Rived. *É hora de colocar as coisas no lugar!*. Acessado em 03 de janeiro de 2007. Documento on-line, disponível em: <http://rived.proinfo.mec.gov.br/curso/objetos/bio/index.htm>.

[24] Centros de Competências Nónio. *Uma aventura na União Européia*. Documento on-line, disponível em: <http://nonio.eses.pt/asp/europa/index.htm>. Acessado em 03 de janeiro de 2007.

[25] Tac-Software. *Teaching Templates*. Documento on-line, disponível em: <http://www.tac-soft.com/>. Acessado em 03 de janeiro de 2007.

[26] Half-baked Software. *Quandary*. Documento on-line, disponível em: <http://www.halfbakedsoftware.com/quandary.php>. Acessado em 03 de janeiro de 2007.

[27] Laboratório Didático Virtual – USP. *A química dentro de um bolo*. Acessado em 03 de janeiro de 2007. Documento on-line, disponível em: <http://www.labvirtq.futuro.usp.br/applet.asp?time=17:04:48&lom=10623>.

[28] National Library of Virtual Manipulatives. *Base Blocks Addition*. Documento on-line, disponível em: <http://nlvm.usu.edu/en/nav/frames_asid_154_g_2_t_1.html>. Acessado em 03 de janeiro de 2007.

[29] National Library of Virtual Manipulatives. *Algebra Balance Scale*.Acessado em 03 de janeiro de 2007. Documento on-line, disponível em: <http://nlvm.usu.edu/en/nav/frames_asid_201_g_4_t_2.html?open=instructions>.

[30] Laboratório Didático Virtual - USP. *Um dublê em apuros*. Disponível em: <http://www.labvirt.futuro.usp.br/applet.asp?time=13:15:26&lom=10536>. Acessado em 03 de janeiro de 2007.

[31] TALARICO NETO, A; et al. *Cog-Learn: uma Linguagem de Padrões para e-Learning*. Revista Brasileira de Informática na Educação, Rio de Janeiro, 13(3), p. 33-50, 2006.

[32] ZIMMERMANN, B., et al. *Patterns for Tailoring E-Learning Materials to Make them Suited for Changed Requirements*. VikingPLoP 2006, Helsingör, Dänemark. 2006.

[33] GIBERT-DARRAS, F.; et al. *Towards a Design Pattern Language to Track Students' Problem-Solving Abilities*. Artificial Intelligence in Education Conference: Workshop on Usage Analysis in Learning Systems, Amsterdam, The Netherlands. 2005.

Patterns for Documenting Frameworks – Process

Authors **Ademar Aguiar, Gabriel David**
INESC Porto, Faculdade de Engenharia, Universidade do Porto
Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal
E-mail: ademar.aguiar@fe.up.pt, gtd@fe.up.pt

Abstract Good design and implementation are necessary but not sufficient pre-requisites for the successful reuse of object-oriented frameworks. Although not always recognized, good documentation is crucial for effective framework reuse, but it is often hard, costly, and tiresome to produce it, especially when not aware of its key problems and the best ways to address them. The patterns here presented are from a set of related patterns that describe proven solutions to recurrent problems of documenting object-oriented frameworks. In particular, this document presents *process patterns*, addressing problems and solutions related with the process of writing documentation (e.g. *which activities, roles and tools are needed?*), which complement the set of *artefact patterns* previously published by the authors addressing problems closely related with the documentation itself.

Introduction

Object-oriented frameworks are a powerful technique for large-scale reuse capable of delivering high levels of design and code reuse. As software systems evolve in complexity, object-oriented frameworks are increasingly becoming more important in many kinds of applications, new domains, and different contexts: industry, academia, and single organizations.

Although frameworks promise higher development productivity, shorter time-to-market, and higher quality, these benefits are only gained over time and require up-front investments. Before being able to use a framework successfully, users usually need to spend a lot of effort on understanding its underlying architecture and design principles, and on learning how to customize it, which all together implies a steep learning curve that can be significantly reduced with good documentation and training material.

This paper contributes with two additional patterns to the work in progress of writing a pattern language to help on documenting frameworks 0[2][3][4], and therefore to help developers on employing frameworks more effectively.

Pattern Language

The pattern language comprises a set of interdependent patterns that aim at helping developers on becoming aware of the typical problems they will face when documenting object-oriented frameworks. The patterns were mined from existing literature, lessons learned, and expertise on documenting frameworks, based on a previous compilation about framework documentation [5].

The pattern language describes a path commonly followed when documenting a framework, not necessarily from start to end to achieve effective results. In fact, many frameworks are not documented as completely as suggested by the patterns, due to different kinds of usage (white-box or black-box) and different balancing of tradeoffs between cost, quality, detail, and complexity. One of the goals of these patterns is precisely to expose such tradeoffs in each pattern, and to provide practical guidelines on how to balance them to find the best combination of documents, activities and tools to the specific context at hands.

According to the nature of the problems addressed, the patterns are organized in *artefact patterns*, which address questions such as *which kinds of documents to produce? what should they include? how to relate them?* 0[2][3], and are overviewed in the appendix, and *process patterns*, which address questions such as *how to do it? which activities, roles and tools are needed?*, are strictly related with the process of cost-effectively documenting frameworks, and to which belong the patterns here documented.

Process Patterns

As the name suggests, this category of patterns are primarily concerned with the *process* of documenting object-oriented frameworks, and not so much with the artefacts themselves, as those are the major concern of the *artefacts patterns*.

Framework documentation is produced mainly during framework development, resulting in tutorials and user guides teaching how to use the framework, and design documents to explain how it works and describe its underlying design principles and mechanisms, among other documents.

Once produced, framework documentation is then used and reviewed during all phases of framework development. It is probably at framework instantiation, during application development, that documentation is used in a more intensive way. It acts as a means of communicating important information from the original framework designers, primarily to framework users, but also to other framework designers and framework maintainers.

The incorporation of comments and feedback from readers is very important for improving the quality of future revisions of the documentation, so it is important to establish an effective bidirectional communication mechanism between documentation authors and readers.

As a framework evolves during the expected long life of the respective framework, the accompanying documentation must evolve as well, and therefore the maintenance of documentation is an activity to be taken in consideration during all framework's life.

The conceptual life cycle of framework documentation is similar to the typical life cycle of technical documentation, which can be seen as organized in five basic activities: *configuration*, *production*, *organization*, *usage*, and *maintenance* (see Figure 1). Although these activities are not all mandatory, neither not necessarily needing to follow the exact order shown, they reflect very well all what is involved (roles, activities, and information flows).

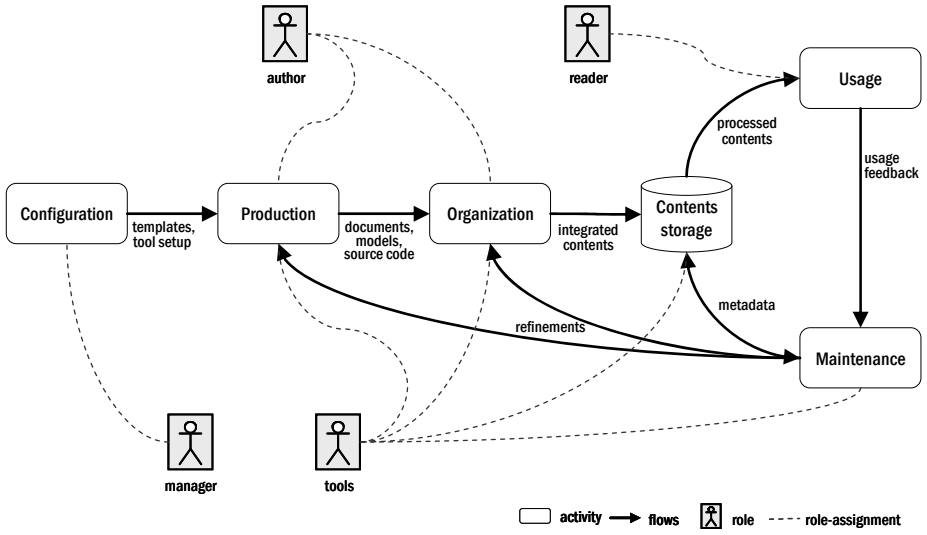


Figure 1 - Typical activities of documenting frameworks.

This document presents patterns addressing recurrent problems somehow related with some of these activities.

To describe all the patterns in a concise way, we have adopted a pattern form similar to Christopher Alexander's, including the most essential sections, concretely: *Name-Context-Problem-Solution-Consequences* [7]. References to other patterns of this pattern language are formatted as following: ANOTHER PATTERN. Thumbnails for the artefact patterns are included in the appendix, and thumbnails for the process patterns are below.

Before going to the detail of each pattern, we will briefly overview all the process patterns by summarizing each pattern's intent, which are also depicted in Figure 2.

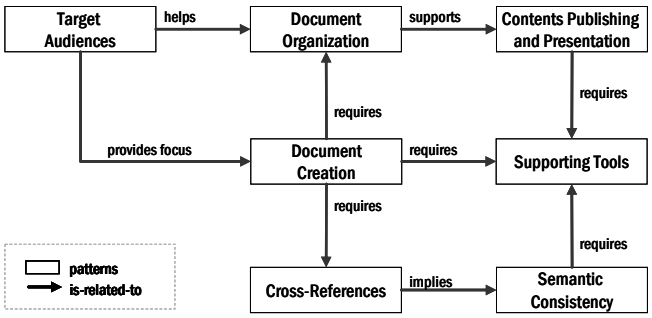


Figure 2 - Documentation process patterns and their relationships.

Target Audiences pattern describes one of the first activities in the overall process of documenting (a framework), which is to define and prioritize the audiences intended to be addressed by the documentation. Once having defined the audiences to target, the contents can then be more effectively created and organized to be presented through the most appropriate views and formats for those audiences.

Document Creation pattern provides hints on the main activity of documentation. It explains how to streamline the creation of documentation artefacts (documents, models, source code fragments, etc.) both by developers and technical writers, in order to yield good quality and cost-effective documentation.

Cross-References pattern addresses the problem of linking and relating different documentation artefacts (e.g. explanations, models and source code of examples), to provide good navigability between all the contents involved, and therefore to minimize the obstacles to the strategies that readers spontaneously adopt when trying to understand and learn something new.

Semantic Consistency pattern tells you how to cope with the difficulties of maintaining the semantic consistency between related software artefacts during development (source code, models, and documents) to enable their continual review and modification throughout the lifecycle and thus continuously preserve its accuracy and value for the readers.

Document Organization pattern provides hints of several kinds (e.g. storage, metadata, guidelines, conventions, templates) that help to achieve a good organization of all the possible documentation involved, to keep all the contents consistent, well structured, well integrated, easy to browse, easy to find, and easy to maintain.

Content Publishing and Presentation pattern describes the ultimate activity of documentation, the reason why it is produced and organized. The pattern addresses issues on using documentation, not only to read contents in a presentation format, but also to browse, search, select, and navigate through the contents, what sometimes requires processing of contents (transforming, filtering, composing, etc.), to present them in the most convenient format to the reader.

Tool Support pattern addresses the problem of ensuring quality and reducing the typical high costs associated with the production and maintenance of framework documentation. The pattern teaches you how to automate the documentation process the best as possible, while retaining the flexibility and adaptability to different developers and environments.

Pattern **Target Audiences**

You are about to start documenting a framework to preserve and communicate the knowledge you have that might be helpful to others willing to understand it.

Problem It is commonly accepted that good technical documentation can significantly improve the process of learning, understanding and reusing frameworks. However, it is often hard, costly and tiresome to define and write good documentation for a framework because we need to satisfy different audiences, to encompass several purposes, and to support different kinds of reuse.

How to drive the documentation activities in order to effectively produce the most valuable documents for its readers?

Forces **Completeness.** To be complete, the overall documentation of a framework usually combines a lot of information that must be produced, organized and maintained consistent. In concrete, it must include framework information about the application domain covered, the specific purpose, how-to-use, how it works, and also internal design details. As a result, this often requires combining a large diversity of contents, gathered from different types of documents, possibly represented in different notations, and with different presentation requirements.

Usefulness. To be useful, the contents must be properly tailored to meet the needs of each category of software engineers involved in framework-based application development, which may play different roles, may have varying levels of experience, and therefore look for different kinds of information.

Time and costs. To control documentation costs under affordable values and to ensure production and maintenance times acceptable, it is wise and mandatory to restrict or prioritize the set of documents and contents to produce.

Solution **Start the process of documenting a framework by clearly defining and prioritizing the audiences to be addressed.**

Typically, there are five main kinds of framework users, with different documentation requirements, to consider or not as target audience: *framework selectors*, *application developers*, *framework developers*, *framework maintainers*, and *developers of other frameworks* [5].

Framework selector is someone (manager, project leader, developer) responsible for deciding which frameworks to use in an application development project.

Framework selectors will look for a short description of the framework's purpose, the domain covered (FRAMEWORK OVERVIEW) and an explanation of the most important features of the framework, possibly illustrated with a set of examples (SPIRAL COOKBOOK, GRADED EXAMPLES).

Application developer is a software engineer that wants to customize a framework to the needs of the application at hand. In a first place, they want to identify which points must be customized (CUSTOMIZATION POINTS), and to know how to

implement such customizations, rather than to understand why it must be exactly done that way.

The application developer needs prescriptive documentation capable to guide her find out which hot spots must be used, which set of classes to subclass, which methods to override, and which objects to interconnect (SPIRAL COOKBOOK). It must be expected that the application developer is not knowledgeable on the application domain and she is not an experienced software developer.

Framework developer is a software engineer involved in the design and implementation of a framework. Framework developers must have a good understanding of the overall architecture and its rationale.

They need also the most detailed view over the framework design internals (DESIGN INTERNALS), the application domain (FRAMEWORK OVERVIEW), and the hot spots that support its flexibility (CUSTOMIZATION POINTS). The information needed must be described at several levels of abstraction, from a high level of abstraction to a concrete level of detail. It usually contains several kinds of artifacts ranging from architectural models and design patterns to abstract algorithms and concrete source code (TRAVERSABLE CODE).

Framework maintainer is a software engineer responsible for the maintenance and evolution of a framework. Usually, framework maintainers are the original framework developers, but this is not always the case.

Their needs in terms of documentation are very similar to those of framework developers, but the documentation has to be more descriptive, instead of prescriptive, because original framework designers can't predict how the framework might be extended in the future through additional flexibility on existing hot spots, or in additional hot spots. It is expected that the framework maintainers are both domain experts and software experts.

Developers of other frameworks usually study existing frameworks, even frameworks for other domains, to find ways of providing flexibility at the hot spots of the framework they are developing.

They have special interest on information at a high level of abstraction, such as abstract solutions and design patterns (DESIGN INTERNALS). The documentation requirements are similar to those of framework maintainers, except that they don't need the concrete details about the framework, but rather the abstract ideas. It is expected that framework developers are expert software designers but not necessarily domain experts for the framework they are mining for ideas.

From all these audiences, application developers often represent the majority. Framework developers are also a very important audience because they are authors and intensive users of framework documentation simultaneously.

Consequences Once defined the concrete audiences to target, it becomes easier to decide which types of documents are more important to write, which have higher priority, and what to include in them, considering its usefulness to the audiences on target (DOCUMENT CREATION). It becomes also easier to decide how to organize the documents (DOCUMENT ORGANIZATION), and which are the most appropriate

views and formats to present them to target audiences (CONTENT PUBLISHING AND PRESENTATION).

One possible drawback of defining target audiences is the risk of being more pragmatic than it should be, and therefore to not consider other audiences not considered of high-priority but that could also benefit from the documentation.

As a result, having in mind a specific audience, documentation usefulness can be guaranteed, completeness relaxed, and this will help to reduce documentation effort therefore being more effective.

Known Uses To have well-defined target audiences for a document is a good practice of technical documentation, and although it can't be proved that this pattern was followed, for example, JUnit and HotDraw frameworks include different documents having different audiences in mind, suggesting that existed some kind of concern about defining audiences to target when writing documentation.

HotDraw. In a paper about the framework authored by Ralph Johnson [8], it is presented a pattern language to document the HotDraw framework, comprising a set of patterns, one for each recurrent problem of using the framework. In that work, the goal is to document the design of the framework, possibly having in mind, primarily, advanced framework users.

JUnit. The document of JUnit [9], named "A Cook's Tour", is devoted to explain how JUnit was designed, possibly targeted for advanced users, or other framework developers. Another document, named "JUnit: Test infected: Programmers love writing tests", is clearly targeted for typical framework users.

Pattern **Document Creation**

You have decided the TARGET AUDIENCES and the types of documents more valuable to them that must be created.

Problem Contents production typically includes the elaboration of technical documents and models, and also the formatting and integration of documents and source code. In addition, to support good contents navigability, it is also very important to cross-reference all kinds of contents exhaustively.

To be useful and complete, framework documentation must include a lot of contents, gathered from different types of documents, at different moments of the development life cycle, and produced by different kinds of people. Besides knowing *what* to document and to *whom* (TARGET AUDIENCES) it is also very important to know *who*, *how*, and *when* to document.

How to produce the documents required in a cost-effective way, orchestrating all participants involved, and promoting their cooperation?

Forces **Quality.** Good technical documentation is the ultimate goal of documentation writers, and what readers definitely look for. But producing good quality documentation comes with many issues, of which the difficulty of defining and assessing its quality is perhaps the first one.

Discipline. A well-defined process identifying roles, techniques, activities, and guidelines is very important to effectively producing documentation. Although discipline by itself does not ensure the quality of an activity or final product, there is however a direct relation between process maturity and product quality. Discipline helps on improving productivity, reducing costs, and is fundamental to enable cooperation in large teams. Very prescriptive guidelines and increased formality can be used to improve discipline, but at the cost of higher inefficiency, less pleasant activities, and constraining creativity.

Agility. As technical writing requires creativity, it is important to reduce formalities to the minimum, if we want to face documentation as a set of activities that are *simple*, *flexible*, almost *neutral*, and *easy to adapt*. Iteration and feedback are also very important to evolve quality of documents smoothly and naturally.

Cost. As documentation effort must not outweigh its benefits, it is important to ensure appropriate mechanization of human activities, and automation of repetitive tasks.

Value. Documents are written to satisfy the reader, so it is important to assess its value to the reader.

Solution **Create documents by following the most agile process allowed by your project, guaranteeing that the final resulting artifacts have the required level of quality and are the most valuable for the readers.**

Below you can find a set of roles and practices to adopt to help you on improving documentation agility.

Roles The documentation involves the collaboration of different kinds of people in different phases of the process.

Developers, such as framework developers, and framework maintainers, are responsible for content creation mostly during the development phase.

Technical writers are responsible to structure, guide, review and conclude the documentation;

Documentation managers are responsible for configuring and maintaining the documentation base, namely the template documents, template instances, and filtering, transforming and formatting documents according to the needs of each audience.

Core practices Depending on the writers' discipline, documentation managers can enforce or flexibilize the documentation rules with the goal of achieving good quality. The more flexible and informal the process, the more attractive it will be for the writers, because formality often compromises creativity. However, too much flexibility may result in inconsistent writing styles and presentation, if the writers are not well disciplined.

Collective ownership. By default, all documents must be readable and editable by anyone involved in the project. Collective ownership of documents usually leads to better documents, because everyone can contribute, resulting in richer and more complete documents. The documents can be reviewed later by a technical writer to improve its homogeneity, consistency of terms, writing style and formatting.

Collaborative writing. Write in collaboration with other people, to assess the understandability, completeness, and accuracy of the document.

Create simple documents, but just simple enough. A document easy to read must be succinct. It shouldn't contain everything, but only the enough information that fulfills its purpose and the intended audience. The simplicity and understandability of contents must be evaluated by the readers.

Create several documents at once. To represent all the aspects of a framework, and to serve all the audiences and purposes, it is necessary to use different documents (e.g. recipe, example, hook description, and pattern). Editing them in parallel can help writers on "dumping" their knowledge more effectively, as writers can document almost every aspect they have in mind without switching contexts. Cross-references must be used to link the separated but related documents (CROSS-REFERENCES).

Publish documents publicly. Publicly available documents, published for everyone to see, support knowledge transfer and improves communication and understanding. The feedback from readers is improved and the overall quality of documents is quickly improved.

Document and update only when needed. To be cost-effective, documents should be created and iteratively refined only when needed, not when desired.

Reuse documentation. Reuse contents and structure of existing documentation in order to improve the productivity and quality of the documentation. Reusable contents must be modular, closed, and readable in any order.

Supplementary practices

Use simple tools. Simple tools can help readers focus more on the contents, rather than on the presentation (SUPPORTING TOOLS). Good examples of simple but effective documentation tools are wikis.

Define and follow documentation standards. Writers must agree and follow a common set of documentation conventions and standards on a project.

Document it, to understand it. To document helps on formalizing ideas, by focusing on single aspects, in isolation from others less relevant, and this helps on the understanding process.

Consequences

Once defined the documents to write, and in order to produce the most valuable documents to the readers on target, it is important to adopt a documentation process that satisfies the project needs and perfectly balances cost, quality, discipline and agility.

The adoption of agile documentation practices helps to reduce costs and maximize the value to the reader, while promoting collaboration between team elements.

Know-Uses

In order to achieve good quality documentation with small effort, several well-documented open-source frameworks follow documentation processes encompassing some of the agile practices above defined. Very good examples are Apache and Eclipse frameworks.

Credits

In first place, the authors would like to thank Linda Rising, our shepherd, for the valuable comments and feedback provided during shepherding. We also want to thank Neil Harrison, Uwe Zdun, Rosana Teresinha Vaccare Braga, and Ralph Johnson, for the comments and feedback provided during the shepherding of other patterns from this pattern language for documenting frameworks, and Eduardo Fernandez, Kevlin Henney, Klaus Marquardt, Sergiy Alpaev, Sami Lehtonen, Allan Kelly, Ian Graham, Alexander Füllebornand, Martin Schmettow, Michalis Hadjisimouand, Richard Gabriel, Joseph Yoder, Mark Perry, Maria, and all the other participants of the writer's workshops at VikingPLoP'2005, EuroPLoP'2006, PLoP'2006, and SugarLoafPLoP'2007, for the motivation, comments and suggestions they provided.

References

[1] Aguiar, A., and David, G. (2005). Patterns for Documenting Frameworks – Part I. In Proceedings of VikingPLoP'2005, Helsinki, Finland (to be published).

[2] Aguiar, A., and David, G. (2006). Patterns for Documenting Frameworks – Part II. In Proceedings of EuroPLoP'2006, Irsee, Germany (workshopped).

[3] Aguiar, A., and David, G. (2006). Patterns for Documenting Frameworks – Part III. In Proceedings of PLoP'2006, Portland, Oregon, USA (workshopped).

[4] FEUP, doc-it project web site, http://doc-it.fe.up.pt/.

[5] Aguiar, A. (2003). A minimalist approach to framework documentation. PhD thesis, Faculdade de Engenharia da Universidade do Porto.

[6] Hargis, G. (2004). Developing quality technical information. Prentice-Hall, 2nd edition.

[7] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). A Pattern Language. Oxford University Press.

[8] Johnson, R. (1992). Documenting frameworks using patterns. In Paepcke, A., editor, OOPSLA'92 Conference Proceedings, pages 63–76. ACM Press.

[9] Beck, K. and Gamma, E. (1997). JUnit homepage. Available from http://www.junit.org.

Appendix

This appendix briefly presents the *artefact patterns* that complement the *process patterns* previously described. They address problems and solutions related with the documents to produce (*which kinds of documents to produce? what should they include? how to relate them?*).

Artefact Patterns

Artefact patterns address problems related with the documentation itself, here seen as an autonomous and tangible product independent of the process used to create it. They provide guidance on choosing the kinds of documents to produce, how to relate them, and what to include there.

Similarly to other technical documentation, the overall quality of framework documentation is complex to determine and assess, and this is perhaps the first issue. Documentation must have quality, that is, it must be easy to find, easy to understand, and easy to use [6]. Task-orientation, organization, accuracy, and visual effectiveness are among all documentation quality attributes, the most difficult ones to achieve on framework documentation [5].

From the reader's point of view, the most important issues are on providing accurate task-oriented information, well-organized, understandable, and easy to retrieve with search and query facilities. From the writer's point of view, the key issues are on selecting the contents to include, on choosing the best representation for the contents, and on organizing the contents adequately, so that the documentation results of good quality, while easy to produce and maintain.

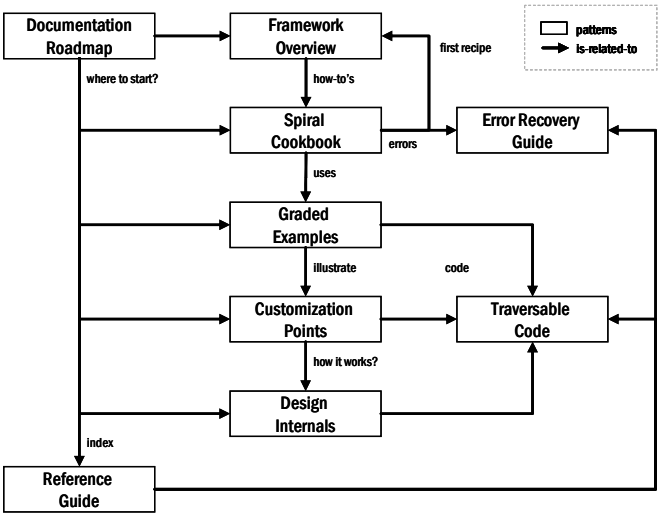


Figure 3 - Documentation artefact patterns and their relationships.

Patterns overview

To describe the patterns, we have adopted the Christopher Alexander's pattern form: *Name-Context-Problem-Solution-Example* [7]. Before going to the detail of each pattern, we will overview the pattern language with a brief summary of each pattern's intent. For contextual purposes, all the artefact patterns are overviewed below and depicted in Figure 3 highlighting the two patterns described in this paper.

Documentation Roadmap helps on deciding what to include in a first global view of the documentation that can provide readers of different audiences with useful and effective hints on what to read to acquire the knowledge they are looking for 0.

Framework Overview tells you to provide introductory information, in the form of a framework overview, briefly describing the domain, the scope of the framework, and the flexibility offered, because contextual information about the framework is the first kind of information that a framework user looks 0.

Cookbook & Recipes describes how to provide readers with information that explains how-to-use the framework to solve specific problems of application development, and how to combine this prescriptive information with small amounts of descriptive information to help users on minimally understanding what they are doing [2].

Graded Examples describes how to provide and organize example applications constructed with the framework and how to cross-reference them with the other kinds of artefacts (cookbooks, patterns, and source code) [2].

Customization Points describes how to provide readers with task-oriented information with more design detail than cookbooks and recipes so that readers can quickly identify the points of the framework (hot-spots) they need to customize and thus get a quick understanding about how they are supported (hooks) [3].

Design Internals explains how to provide detailed design information about what can be adapted and how the adaptation is supported, by referring the patterns that are used in its implementation and where they are instantiated [3].

Reference Guide tells you what to include as reference information and how to structure the documentation to make it the most complete and detailed as possible to assist advanced users when looking for descriptive information about the artefacts and constructs of the framework.

Traversable Code provides hints on how to organize and present source code, both of the examples and the framework itself, when desired, to make it easy to browse and navigate, from, and to, other software artefacts included in the overall documentation, namely models and documents.

Error Recovery Guide explains how to help users on understanding and solving the errors they encountered when using the framework by guiding users on the customization process and revealing the most important design principles and details.

Modelo de Melhoria do Processo de Software para Micro e Pequenas Empresas baseado em Padrões – Discussão e Levantamento Preliminar

Tarciane C. Andrade, Fabrício G. Freitas[1], Jerffeson Teixeira de Souza

Universidade Estadual do Ceará (UECE)
Av. Paranjana, 1700, Campus do Itaperi
60.740-903, Fortaleza – CE

tarciane@gmail.com, fabriciogf@uece.br, jeff@larces.uece.br

Resumo. *Este artigo inicia o levantamento e a discussão de um conjunto de Padrões de Processos de Software, aqui documentados sob a forma de Patlets, desenvolvidos para lidar com as dificuldades encontradas na implantação de Modelos de Melhoria do Processo de Software em Micro e Pequenas Empresas – MPEs de Software. O objetivo é a extração das características comuns dos modelos de qualidade a fim de auxiliar na implantação da qualidade de software nas MPEs. O conjunto de Padrões de Processos de Software é utilizado como um modelo base que permite a introdução de conceitos fundamentais de qualidade no processo de desenvolvimento de software nessas empresas a baixo custo.*

Palavras-chave: *Melhoria do Processo de Software, Qualidade de Software, Micro e Pequenas Empresas de Software, Padrões de Processos, Patlets.*

Abstract. *This paper initiates a description and a discussion of a set of Process Software Patterns, here documented as Patlets, developed to deal with the difficulties found in the adoption of Software Process Improvement Models on Small Software Organizations. The goal is the extraction of the common characteristics of the quality models in order to facilitate the adoption of software quality in Small Organizations. The set of Process Software Patterns is to be used as an intermediate quality model, allowing the introduction of basic concepts of quality in the software development process in these organizations with low cost.*

Keywords: *Software Process Improvement, Software Quality, Small Software Organizations, Process Patterns, Patlets.*

1. Introdução

As empresas que trabalham com desenvolvimento de software possuem um desafio: produção de software de qualidade. Esta qualidade está relacionada principalmente ao desenvolvimento de sistemas menos propensos a falhas e mais eficientes, ao cumprimento de prazos e ao menor custo de desenvolvimento possível.

Essa necessidade de produzir software de qualidade tem exigido cada vez mais ferramentas e técnicas da Engenharia de Software. Neste sentido, organismos

[1] Apoio financeiro da Fundação Cearense de Pesquisa – FUNCAP.

empresariais, universidades e entidades de pesquisa têm proposto metodologias desenvolvimento de sistemas, bem como padrões de software e técnicas para melhoria da qualidade do processo e do produto de software com o intuito de possibilitar a garantia da qualidade do software.

Em Engenharia de Software, um Padrão de Software é uma descrição de uma solução geral para um problema recorrente em qualquer das etapas de desenvolvimento de um software [Coplien 1996]. Esses padrões podem se apresentar de várias formas, incluindo padrões de análise, padrões de projeto, padrões organizacionais e padrões de processo. Um Padrão de Processo, segundo [Coplien 1996], descreve os processos básicos, que associados a recomendações, definem práticas bem sucedidas e relacionadas aos processos. O estabelecimento de padrões de processo depende do conhecimento vasto do domínio da aplicação daquele processo. No caso do domínio de desenvolvimento de software vários padrões são estabelecidos. Padrões em geral, e padrões de processo em particular, ajudam na documentação de boas soluções de forma que estas possam ser reutilizadas com maior facilidade. Os *Patlets*, [Coplien et. al. 2004], [Grone 2006], [Harrison 1999], são padrões resumidos que contém somente os elementos essenciais (nome, contexto, problema, solução e usos conhecidos) e servem para referenciar um padrão completo ou como passo intermediário na documentação de um padrão completo.

Existem, atualmente, diversos modelos de qualidade focados na melhoria do processo de desenvolvimento como um todo, entre eles: ISO 9000:2000 [ISO 2000], ISO/IEC 15504 [ISO 2003], ISO/IEC 12207 [ISO 2002], CMMI [SEI 2005], PSP [SEI 1997], TSP [SEI 2000], MR-MPS [SOFTEX 2006], PMBOK [PMI 2000] e ISO 10006:2000 [ISO 2000] os quais funcionam como guia de boas práticas durante o processo de produção de software. As grandes empresas (segundo [MCT 2005], empresas com mais de 100 empregados) que atuam na área de desenvolvimento de software têm obtido êxito na implantação de tais modelos, principalmente em virtude da disponibilidade de recursos humanos e financeiros. As Micro e Pequenas Empresas de Software – MPEs (segundo [MCT 2005], empresas com até 10 e 50 empregados, respectivamente), em contrapartida, podem encontrar dificuldade na tentativa de implantação de tais modelos devido à grande quantidade de processos exigida por eles, do elevado custo financeiro e da necessidade de envolvimento de vários recursos em diferentes papéis.

Nesse contexto, este artigo relata, inicialmente, os problemas enfrentados por MPEs no ramo de software na tentativa de implantar os modelos de qualidade existentes no mercado. Em seguida, é apresentado o resultado da extração de características comuns entre os principais modelos de qualidade de software existentes através da documentação dessas características na forma de Padrões de Processo, documentados no formato de *patlets*. O foco do conjunto de Padrões de Processos aqui apresentados está na garantia da qualidade de software em todo o processo de desenvolvimento. Além de contribuir para superar as dificuldades encontradas na implantação dos modelos de qualidade de software e adequar a sua aplicação à realidade de poucos recursos das MPEs de software, o conjunto de padrões contribuirá na construção de um modelo de qualidade simplificado. Dessa forma, o cenário alcançado pela aplicação de tal modelo simplificado introduz conceitos fundamentais da garantia da qualidade a baixo custo, apresentando-se como um passo intermediário entre um ambiente de desenvolvimento

sem garantia da qualidade e a situação encontrada após a implantação de um modelo de qualidade completo.

2. Dificuldades Encontradas na Melhoria do Processo de Software nas Micro e Pequenas Empresas

Nas últimas duas décadas o SPI (*Software Process Improvement)* tem se tornado um fator chave no aumento da produtividade e qualidade no desenvolvimento do software, interferindo na competitividade das empresas de software e até mesmo na sobrevivência no mercado. Os objetivos do SPI são produzir e garantir softwares de qualidade no tempo estimado, dentro do orçamento previsto e com as funcionalidades desejadas.

Vários modelos [ISO9000:2000 2000], [ISO12207 2002], [ISO15504 2003], [CMMI 2005] e metodologias de SPI possuem abordagem direcionada para grandes empresas de desenvolvimento de software. Entretanto, as MPEs se esforçam para tentar implantar ou ainda adaptar estes modelos.

Serrano et. al. [Serrano et. al. 2006] apresentam que as dificuldades encontradas pelas MPEs em implantar um modelo de qualidade de software se deve à falta de um guia de implantação direcionado a estas empresas, tão bem como o tempo e o custo para tal implantação.

Em [Oktaba 2006], Oktaba utiliza critérios para avaliar se os principais modelos de SPI atendem às MPEs de software. Os critérios utilizados foram: adequação para pequenas e médias empresas com baixos níveis de maturidade, baixo custo de implantação e avaliação, específico para desenvolvimento de software, definido como um conjunto de processos baseados em práticas reconhecidas internacionalmente. Nenhum dos modelos avaliados atendeu a todos estes requisitos.

Em [Laryd et. al. 2000], Laryd et. al. mostram a necessidade de iniciar o processo de melhoria de qualidade do software o quanto antes nas MPEs. Eles retratam, em primeiro lugar, a importância de iniciar antes do caos se instalar. Se a primeira solução não for possível, o melhor é iniciar o programa de melhoria de software enquanto a empresa ainda é pequena, com poucos analistas e com um simples gerente, por exemplo. Entretanto, se uma empresa está tendo sucesso, ela tende a crescer. Tende a aumentar a quantidade de analistas, de desenvolvedores, de gerentes por projetos, de produtos e serviços. É neste ponto que se observa um aumento das dificuldades do processo de desenvolvimento, pois se torna difícil de gerenciar, de agrupar as inúmeras versões dos produtos entre outros. Para evitar esse tipo de problema, uma empresa deve ter o foco no processo de melhoria do software antes mesmo que se torne necessário, ou seja, mais ou menos no início da sua estruturação.

Para [Kelly et. al. 1999], existem diferentes culturas entre pequenas e grandes empresas. Em pequenas empresas os empregados esperam estar envolvidos em todos os aspectos do processo de engenharia de software. Em tal situação, o processo de melhoria de software é visto como introdução de burocracias que restringem a liberdade individual.

Brodman e Habra et. al. em [Brodman et. al. 1997], [Habra et. al. 1999] fazem uma adaptação no CMM para atender as MPEs. Entre os principais problemas abordados quanto ao CMM que levaram a esta decisão estão: sobrecarga de documentação, necessidade de gerentes em muitas camadas, excesso de revisões,

recursos limitados, altos custos com treinamentos, práticas irrelevantes e inadequadas às micro e pequenas empresas.

Outras dificuldades encontradas para implantação de modelos de qualidade nas MPEs são descritas em [Souza et. al. 2002], [Revankar et. al. 2006], [Herndon et. al. 2006]. Pequenas empresas possuem orçamentos reduzidos para melhoria de processos, o que restringe até mesmo o investimento em treinamentos para os seus membros. Além disso, a visão do cliente deve acompanhar as mudanças na melhoria do processo de software da empresa, pois é comum o cliente ter contato direto com a equipe de desenvolvimento o que compromete o gerenciamento do projeto. Na ausência de um membro da equipe não há substituto, podendo uma atividade ser cancelada. Um ponto importante é o apoio da alta gerência na mudança de visão da estrutura da empresa com a implantação do modelo de qualidade de software.

Por outro lado, as MPEs mesmo enfrentando todos os problemas, principalmente com recursos limitados podem disseminar rapidamente os processos implantados, uma vez que o número de empregados é comparativamente pequeno. Baseado na análise dos problemas acima, MPEs de software necessitam de uma abordagem diferenciada para melhoria de processo em comparação com grandes empresas.

3. Conjunto de *Patlets* de Processos de Software para Micro e Pequenas Empresas

Diante das dificuldades enfrentadas por MPEs, observaram-se dois tipos de abordagens para a implantação do processo de melhoria de software em tais empresas. A primeira delas é a adequação dos modelos existentes com a escolha de apenas um subconjunto de processos [Kelly et. al. 1999], [Habra et.al. 1999], [Serrano et. al. 2006], [Bezerra et. al. 2005], [Carmody 2006]. A outra abordagem é a criação de um modelo próprio tendo como base os modelos existentes [Laryd et. al. 2000], [Silva et. al. 2003], [SOFTEX 2006], [Oktaba 2006], [Revankar et. al. 2006]. A escolha de uma destas abordagens depende, entre outros fatores, do intuito de cada empresa, do orçamento reservado, e da existência ou não de algum processo de melhoria no desenvolvimento de software na mesma. Contudo, não existe nenhuma abordagem satisfatória e simples que permita às empresas que não possuam um processo de garantia da qualidade bem definido, implantá-la de tal forma que permita, em seguida, a aplicabilidade de qualquer um dos modelos existentes de forma natural e com o mínimo de custo.

Portanto, surge a oportunidade da elaboração de um conjunto de Padrões de Processos, baseados na obtenção de características comuns entre os modelos atuais, que servirá como guia de implementação de qualquer um dos modelos de qualidade. Assim, o presente artigo propõe uma nova abordagem para garantir a qualidade no processo de desenvolvimento de software nas MPEs e facilitar, se for o caso, o emprego dos modelos de qualidade existentes.

Como forma de iniciar o trabalho de levantamento dos Padrões de Processo de Software nos modelos de qualidade e adaptações existentes para MPEs de software foi documentado um conjunto de *patlets*. Os *patlets* foram extraídos a partir do cruzamento inicial de informações das duas abordagens de melhoria da qualidade de software citadas anteriormente: processos dos modelos existentes atualmente e processos das adaptações realizadas pelas MPEs.

No total foram obtidos seis *patlets*. Os *patlets* foram distribuídos em todo o processo de desenvolvimento de software com levantamento inicial dos papéis de cada membro da empresa. Neste momento, não foram citados os artefatos, também chamados de produtos de trabalho, de entrada e saída de cada padrão de processo. Vale ressaltar também que alguns *patlets*, futuramente, podem ser desmembrados em outros.

Os *patlets* estão documentados e organizados de acordo com o seguinte formato:

- **Nome do Padrão:** descreve o nome do padrão, e referencia o contexto e o problema. É através dele que o padrão se torna conhecido;
- **Contexto:** descreve em quais circunstâncias o problema surge;
- **Problema:** descreve o problema a ser resolvido;
- **Solução:** descreve o que é necessário ser feito para resolver o problema;
- **Usos Conhecidos:** descreve aplicações do padrão em modelos existentes.

Os *patlets* encontrados estão descritos de forma sucinta na Tabela 1:

Tabela 1. Resumo dos *Patlets* Encontrados

Nome do Padrão	Problema	Solução
Garantia da Qualidade dos Processos e dos Produtos	Como assegurar que os processos e produtos de trabalho estão de acordo com a metodologia adotada?	Estabeleça critérios de avaliação, como o quê, quando e como serão avaliados;
Revisão por Pares	Como detectar os defeitos no produto de trabalho?	Realize encontros formais ou informais de revisão;
Gerência de Configuração	Como controlar os produtos de trabalho e manter a integridade e rastreabilidade das suas versões?	Cada membro do projeto deve armazenar, atualizar e recuperar os seus respectivos produtos de trabalho através de sistema específico;
Medição	Como medir o software de forma quantitativa?	Estabeleça os objetivos das medições, as perguntas para cada objetivo, e as métricas que respondam às perguntas;
Verificação	Como assegurar que os produtos de trabalho refletem apropriadamente os requisitos especificados por eles?	Realize testes funcionais, Revisão por Pares, testes de integração, por exemplo;
Validação	Como assegurar que os requisitos do cliente foram atendidos?	Execute as validações através de testes de aceitação, testes alfa e beta, teste de desempenho, por exemplo;

A seguir, descrição detalhada dos *patlets* encontrados:

Nome do Padrão: *Garantia da Qualidade dos Processos e dos Produtos*

Contexto: Durante todo o ciclo de desenvolvimento do software é necessário assegurar que os processos estão sendo seguidos e que os produtos de trabalho produzidos estão de acordo com a metodologia, procedimentos e padrões previamente definidos.

Problema: Como assegurar que os processos e produtos de trabalho estão de acordo com a metodologia, procedimentos e padrões adotados e prover a melhoria contínua?

Solução:

Crie o papel do SQE (*Software Quality Engineering*) como responsável por garantir que os processos e produtos de trabalho estão sendo seguidos dentro do projeto. O SQE não deve acumular papéis em virtude da necessidade de imparcialidade para garantir a qualidade.

Realize as atividades abaixo, com o papel do SQE:

Para avaliar os processos:

1. Estabeleça critérios de avaliação, como o quê, quando e como serão avaliados;
2. Crie marcos (*milestones*[2]) no projeto para efetuar a avaliação dos produtos;
3. Utilize os critérios de avaliação, definidos anteriormente, para garantir a aderência aos processos;
4. Identifique e registre as não conformidades encontradas;
5. Realize ações corretivas quando necessário;
6. Identifique e registre as lições aprendidas que podem melhorar o processo de desenvolvimento no futuro.

Para avaliar os produtos de trabalho:

1. Defina os produtos de trabalho que devem ser avaliados;
2. Crie marcos no projeto para efetuar a avaliação;
3. Utilize o padrão de *Revisão por Pares* para revisar os produtos de trabalho de forma a garantir a aderência à metodologia, padrões e procedimentos;
4. Identifique e registre as não conformidades encontradas;
5. Realize ações corretivas quando necessário;
6. Identifique e registre as lições aprendidas que podem melhorar o processo no futuro.

Usos Conhecidos:

Herndon et. al. em [Herndon et. al. 2006] escolheram o processo de *Garantia da Qualidade dos Processos e dos Produtos* do CMMI [SEI 2005], representação contínua, para implantar a melhoria da qualidade do software em duas MPEs.

Habra et. al. em [Habra et. al. 1999] construíram um micro modelo de avaliação baseado nos modelos CMMI e ISO/IEC 15504, chamado de OWPL, onde foram

[2] *Milestone* ou marco é um evento programando que signifique a conclusão de um trabalho principal ou de um grupo de trabalhos relacionados. Um marco, pela definição, não tem esforço ou uma duração associada. Um marco é apenas uma bandeira no plano de desenvolvimento para significar que algum trabalho terminou e é usado como um ponto de verificação do projeto para validar como o projeto está progredindo e revalidar o trabalho restante.

escolhidos oito processos, entre eles, o processo de Garantia da Qualidade dos Processos.

Laryd et. al. em [Laryd et. al. 2000] fizeram uma adaptação do CMM para MPEs e escolheram seis áreas de processos do CMM, entre elas, o processo de *Garantia da Qualidade dos Processos e dos Produtos*. Aqui, eles criaram o papel do SQE para assegurar a imparcialidade da garantia da qualidade. O SQE exerce um único papel no projeto.

O ISO/IEC 15504 [ISO 2003] contém na categoria de processos de Suporte – SUP, o processo de Garantia da Qualidade para aderência aos produtos de trabalho e aos processos.

O CMMI [SEI 2005] utiliza este processo ao garantir que os processos planejados estão sendo implementados e garantir a entrega de produtos e serviços de alta qualidade.

O MPS-BR [SOFTEX 2006] utiliza este processo para garantir que os produtos de trabalho e a execução dos processos estão em conformidade com os planos e recursos pré-definidos.

Nome do Padrão: *Revisão por Pares*

Contexto: Em várias fases do desenvolvimento de software é necessário descobrir os defeitos que possam ser eliminados, incluindo: implementação incompleta dos requisitos, problemas na integração com outros sistemas, interfaces de projeto inadequadas, e erros de codificação.

Problema: Como detectar os defeitos no produto de trabalho?

Solução:

1. Membro responsável pelo produto de trabalho a ser revisado: realize encontros formais ou informais de revisão;
2. Se o produto de trabalho não possuir *checklist* dos atributos que precisa atender, faço-o.
3. Entregue, na reunião, o produto de trabalho com respectivo *checklist* para outro membro do projeto revisar;
4. Outro membro do projeto: preencha o *checklist* para verificar a aderência do produto de trabalho;
5. Devolva o produto de trabalho revisado ao membro responsável com respectivo *checklist* para ser analisado.

Usos Conhecidos:

O ISO/IEC 15504 [ISO 2003] utiliza o processo de *Revisão por Pares* para realizar a verificação dos produtos de trabalho no processo de *Verificação*.

No caso do CMMI [SEI 2005] este processo é uma atividade do processo de *Verificação* como forma de garantir que os produtos de trabalho estão em conformidade com seus requisitos.

O TSP [SEI 2000] utiliza o processo de Revisão para efetuar revisões de código e de projeto.

Nome do Padrão: *Gerência de Configuração*

Contexto: Em virtude das constantes mudanças durante todas as fases da construção do software, ocorre a necessidade de controlar e manter a integridade e rastreabilidade sistemática das versões dos produtos de trabalho.

Problema: Como controlar os produtos de trabalho e manter a integridade e rastreabilidade das suas versões?

Solução:

1. Identifique os produtos de trabalho, itens de configuração, que necessitam ser controlados, por exemplo, produtos que são entregues aos clientes e produtos internos, como documentos, diagramas e códigos-fonte;
2. Estabeleça um mecanismo para gerenciar o controle de versão, através de algum sistema computacional de controle de versão;
3. Cada membro do projeto deve armazenar, atualizar e recuperar os seus respectivos produtos de trabalho no sistema;
4. Escolha um analista no projeto (papel de integrador) para ficar responsável pela definição e criação das linhas de base (*baselines*[3]) para uso interno ou para entrega de produto ao cliente;
5. Disponibilize, através do sistema, árvore com histórico das versões e com as diferenças entre sucessivas linhas de base.

[3] Uma *baseline* ou linha de base reflete um instante específico de um projeto e a situação exata de determinados itens de configuração no momento da criação da mesma. Pode-se dizer que é uma "fotografia" dos itens de configuração previamente especificados. Desta forma, a cada nova etapa executada (requisitos, desenvolvimento, testes, homologação, implantação,) a Gerência de Configuração deverá criar uma linha de base que servirá como referência para o processo de desenvolvimento.

Usos Conhecidos:

Layrd e Orci em [Laryd et. al. 2000] fizeram uma adaptação do CMM para micro e pequenas e escolheram seis áreas de processos do CMM, entre elas, o processo de *Gerência de Configuração*.

Habra et. al. em [Habra et. al. 1999] construíram um micro modelo de avaliação, chamado de OWPL, baseado nos modelos CMMI e ISO/IEC 15504, onde foram escolhidos oito processos, entre eles, o processo de *Gerência de Configuração*.

Revankar et. al. [Revankar et. al. 2006] desenvolveram um framework de processos chamado de Rapid-Q com as melhores práticas do CMMI, ISO 9001 e padrões do IEEE. O Rapid-Q possibilita a implantação e melhoria dos processos de micro e pequenas empresas de forma modular, flexível e com baixo custo. Entre os processos escolhidos para compor o Rapid-Q está o processo de *Gerência de Configuração*.

Carmody [Carmody 2006] implantou a melhoria dos processos na Universidade de Medicina de Pittsburgh baseado no nível 2 do CMMI e do ITIL [OGC 2001]. Entre os processos escolhidos, está o processo de *Gerência de Configuração*.

O ISO/IEC 15504 [ISO 2002] contém na categoria de processos de Suporte – SUP, o processo de *Gerência de Configuração* para garantir o controle das versões dos produtos de trabalho.

O CMMI [SEI 2005] utiliza este processo ao garantir o controle dos itens de configuração e manter as linhas de base.

O MPS-BR [SOFTEX 2006] utiliza este processo para estabelecer e manter a integridade de todos os produtos de trabalho de um processo ou projeto e disponibilizá-los a todos os envolvidos.

Nome do Padrão: *Medição*

Contexto: Durante todas as fases ciclo de desenvolvimento de software surge a necessidade de medir as características do software quantitativamente quanto, por exemplo, a aspectos gerenciais e técnicos, para auxiliar no apoio a decisões.

Problema: Como medir o software de forma quantitativa?

Solução:

1. Estabeleça os objetivos da medição, que podem ser gerenciais e/ou técnicas, por exemplo, “controlar as mudanças nos requisitos em determinado período”, “reduzir defeitos” e “aumentar produtividade”;

2. Derive de cada objetivo as perguntas cujas respostas determinam se os objetivos foram ou não alcançados, por exemplo, "qual o percentual de requisitos alterados?";
3. A partir das perguntas, decida o que deve ser medido para ser capaz de responder as perguntas adequadamente (definição das métricas), por exemplo, "nº de requisitos alterados/nº de requisitos alocados", "número de defeitos";
4. Cada membro do projeto deve ficar responsável pela coleta dos dados referentes ao seu escopo no projeto, por exemplo, o gerente de projeto deve ficar responsável pela coleta da métrica "nº de requisitos alterados/nº de requisitos alocados". A coleta deve ser realizada através de sistema computacional específico para coleta de métricas;
5. Disponibilize os resultados a todos os membros envolvidos no projeto.

Usos Conhecidos:

Em Franca et. al. [Franca et. al. 1998] define uma ferramenta para controle de medições para MPEs baseada no *Goal Question Metric* [Basili et. al. 1994], onde são estabelecidos os objetivos a serem medidos, dos objetivos são geradas questões e as métricas surgem para responder as questões propostas a fim de atender os objetivos traçados.

Laryd e Orci em [Laryd et. al. 2000] fizeram uma adaptação do CMM para MPEs e escolheram seis áreas de processos do CMM, entre elas, o processo de *Medição*.

A norma ISO 9000:2000 [ISO 2000] está organizada em cinco seções de requisitos, entre elas a seção de Medição, Análise e Melhorias que tem como foco a medição, análise dos dados e aperfeiçoamento dos processos e produtos.

O ISO/IEC 15504 [ISO 2003] contém na categoria de processos, de Gestão – MAN, o processo de *Medição* para coletar e analisar dados de produtos e processos, para apoiar nas decisões. As medições são realizadas através da identificação das necessidades do projeto.

O CMMI [SEI 2005] utiliza este processo ao estabelecer os objetivos das medições, ao definir os critérios e métricas. Além de analisar os resultados das medições e registrá-las.

O PSP [ISO 1997] utiliza o processo de Medição Pessoal para registrar individualmente o tempo gasto em cada etapa do ciclo de desenvolvimento, por exemplo.

O MPS-BR [SOFTEX 2006] utiliza este processo para coletar e analisar os dados relativos aos produtos desenvolvidos e aos processos implementados na empresa e em seus projetos.

Nome do Padrão: *Verificação*

Contexto: Na tentativa de minimizar os defeitos e riscos associados ao longo de todo o desenvolvimento do software, surge a necessidade de avaliar se os produtos de trabalho atendem completamente aos requisitos para eles especificados ou condições impostas a eles nas atividades anteriores.

Problema: Como assegurar que os produtos de trabalho estão sendo desenvolvidos adequadamente, ou seja, que refletem apropriadamente os requisitos especificados por eles?

Solução:

1. Identifique os produtos de trabalho a serem verificados;
2. Defina os métodos de verificação para cada produto de trabalho, por exemplo, testes funcionais, *Revisão por Pares*, testes de integração;
3. Execute a verificação dos produtos de trabalho selecionados contra os seus requisitos através dos métodos definidos anteriormente;
4. Analise e registre os resultados das verificações;
5. Realize ações corretivas quando necessário.

Usos Conhecidos:

Herndon e Salars em [Herndon et. al. 2006] escolheram o processo de *Verificação* do CMMI [SEI 2005], representação contínua, para garantir que os produtos de trabalho estão em conformidade com seus requisitos.

O ISO/IEC 15504 [ISO 2003] contém na categoria de processos de Suporte – SUP, o processo de *Verificação* para garantir que os produtos de trabalho estão de acordo com os requisitos.

O CMMI [SEI 2005] utiliza este processo ao definir os produtos de trabalhos a serem verificados e utiliza o processo de *Revisão por Pares* para verificar sua conformidade.

O TSP [SEI 2000] utiliza o processo de *Verificação* para verificar os produtos de trabalho quanto a projeto, lógica, reutilização e registra os defeitos encontrados e os corrige.

O MPS-BR [SOFTEX 2006] utiliza este processo para confirmar que cada serviço e/ou produto de trabalho do processo ou do projeto reflete apropriadamente os requisitos especificados.

Nome do Padrão: *Validação*

Contexto: Durante a fase de construção do software ocorre a necessidade de garantir que o software desenvolvido e/ou componente do software atende completamente aos requisitos definidos pelo cliente.

Problema: Como assegurar que os requisitos para o software do cliente foram atendidos?

Solução:

1. Identifique os produtos de trabalho a serem validados, por exemplo, protótipos, versão alfa e beta;
2. Estabeleça o ambiente necessário pra executar a validação como equipamentos e ferramentas de testes;
3. Execute as validações através de, por exemplo, testes de aceitação, testes alfa e beta, teste de desempenho;
4. Analise e registre os resultados das atividades de validação;
5. Realize ações corretivas quando necessário.

Usos Conhecidos:

Herndon e Salars em [Herndon et. al. 2006] escolheu, entre outros, o processo de *Validação* do CMMI para assegurar que o produto desenvolvido e/ou componentes estão de acordo com os requisitos impostos pelo cliente.

O ISO/IEC 15504 [ISO 2003] contém na categoria de processos de Suporte – SUP o processo de *Validação* para garantir que o produto produzido está de acordo com acordado com o cliente.

O CMMI [SEI 2005] utiliza este processo para identificar os produtos de trabalhos a serem validados, executar a validação através de testes, analisar e registrar os resultados.

O MPS-BR [SOFTEX 2006] utiliza este processo para confirmar que o produto ou componente do produto atenderá ao seu uso pretendido quando colocado no ambiente para qual foi desenvolvido.

4. Conclusão e Trabalhos Futuros

Diante do exposto, o presente trabalho está voltado na documentação de um conjunto de "boas práticas" para obtenção de requisitos mínimos da qualidade de desenvolvimento de software a fim de reduzir as mudanças e custos ora sofridos com a implantação dos modelos atuais. O levantamento inicial do conjunto de *patlets* aqui proposto visa a unificação das características comuns dos modelos de qualidade com o objetivo de auxiliar a implantação da qualidade de software em micro e pequenas empresas reduzindo a necessidade de explorar e pesquisar qual dos modelos existentes se adequaria melhor às suas realidades.

A relevância deste trabalho de pesquisa está contida no problema relacionado à quantidade de modelos de melhoria de qualidade existentes hoje no mercado e na dificuldade que micro e pequenas empresas de software enfrentam para iniciar o processo de qualidade no desenvolvimento de software.

Os próximos passos do projeto incluem a captura, documentação e refinamento dos padrões de processos extraídos dos modelos de melhoria de qualidade de software existentes, bem como a criação dos artefatos de entrada e saída dos padrões e a definição dos papéis de cada um no processo de desenvolvimento de software. Posteriormente, será realizada a validação do conjunto de padrões de processos através de um estudo de caso com aplicabilidade em uma pequena empresa de software.

Espera-se com a conclusão do projeto obter um conjunto de padrões de processo de qualidade no desenvolvimento de software de tal forma que auxilie as empresas interessadas em iniciar a implantação de um modelo de melhoria da qualidade. Com a implantação desses futuros padrões as empresas estarão aptas a, posteriormente, seguir qualquer um dos modelos e obter a certificação desejada.

5. Agradecimentos

Especiais agradecimentos ao Prof. Sérgio Soares, nosso *shepherd*, pelos comentários e sugestões importantes que ajudaram a melhorar o conteúdo deste artigo. Agradecemos também aos colegas Anderson Brando, Ellen Polliana, Kleber Rocha, Rafael Braga, Tiago Barros e todos os outros participantes do workshop de escritores, grupo B, do SugarLoafPLoP'2007 pela motivação e comentários essenciais ao aperfeiçoamento do trabalho.

Referências

Ahern, D., Armstrong, J., Clouse, A., Ferguson, J., Hayes, W. and Nidiffer, K. (2005) "CMMI SCAMPI Distilled: Appraisals for Process Improvement", http://www.sei.cmu.edu/cmmi/adoption/books.html.

Anacleto, A., Wangenheim, C., Salviano, C. and Savi, R. (2003) "15504MPE – Desenvolvendo um método para Avaliação de Processo de Software em MPEs Utilizando a ISO/IEC 15504", SIMPROS – Simpósio Brasileiro de Melhoria de Processos de Software, Recife.

Appleton, B. "Patterns and Software: Essential Concepts and Terminology", http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

Basili, V., Caldiera, G., Rombach, H. (1994) “The Goal Question Metric Approach”, Encyclopedia of Software Engineering.

Bezerra, I., Carneiro, D., Nibon, R., Carneiro, R. and Araujo, S. (2005) “Capacitação em Melhoria de Processo de Software: Uma Experiência da Implantação do SW-CMM em um Grupo de Pequenas Empresas”, 2005.

Carmody, C. (2006) “A Giant Taking Small Steps”, Proceedings of the First International Research Workshop for Process Improvement in Small Settings – Selected Case Studies, Janeiro.

Coplien, J. and Schmidt, D. (1995) “Pattern Languages of Program Design”, Addison-Wesley.

Coplien, J. O. (1996) “Software Patterns”, SIGS Books & Multimedia, USA.

Coplien, J. and Harrison, N. (2004) “Organizational Patterns of Agile Software Development”, Prentice Hall.

Dangle, K., Larsen, P. and Shaw, M. (2005) “Software Process Improvement in Small Organizations: A Case Study”, IEEE Computer Society, Dezembro.

Franca, L., Staa, A. and Lucena, C. (1998) “Medição de Software para Pequenas Empresas: Uma Solução Baseada na Web”, PUC – Rio.

Grone, B. (2006) “Conceptual Patterns”, 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems.

Habra, N., Niyitugabira, E., Lamblin, A.C. and Renault, A., (1999) “Software Process Improvement for Small Structures: First Results of a Micro-Evaluation Framework”, in Proceedings of the European Conference on Software Process Improvement SPI'99, Barcelona, Spain.

Habra, N., Niyitugabira, E., Lamblin, A.C. and Renault, A., (1999) “Software Process Improvement in Small Organizations Using Gradual Evaluation Schema”, in Proceedings of the International Conference on Product Focused Software Process Improvement, Oulu, Finland, 381-396.

Harrison, N. (1999) “A Pattern Language for Shepherds: A Pattern Language for Shepherding”, Proceedings of the 6th Annual Conference on the Pattern Languages of Programs, p. 15-18, Agosto.

Herndon, M., Salars, S. (2006) “Two Case Studies in Implementing Model Based Process Improvement in Small Organizations”, Proceedings of the First International Research Workshop for Process Improvement in Small Settings – Process Improvement Approaches and Models, Janeiro.

International Organization for Standardization. (2000) “ISO 9000:2000 Quality Management System”, http://www.iso.org, Janeiro.

International Organization for Standardization. (2000) “ISO 10006:2000. Quality management - Guidelines to quality in project management”, http://www.iso.org.

International Organization for Standardization. (2002) “ISO/IEC 12207 Information Technology – Software Life Cycle Processes”, Amd, 1.

International Organization for Standardization. (2003) “ISO/IEC 15504: Information Technology – Process Assessment”, http://www.isospice.com.

Johnson, D., Brodman, J. (1997) “Tailoring the CMM for Small Businesses, Small Organizations, and Small Projects”, IEEE Computer Society, nº 8.

Kelly, D. and Culleton , B. (1999) “Process Improvement for Small Organizations. Silicon & Software Systems”, IEEE, Outubro.

Laryd, A. and Orci, T. (2000) “Dynamic CMM for Small Organizations”, Proceedings of the 1st Argentine Symposium on software Engineering (ASSE 2000), p. 133-149.

MCT – Ministério de Ciência e Tecnologia. (2005) “Pesquisa Nacional de Qualidade e Produtividade no Setor de Software Brasileiro”, Brasil.

OGC-Office of Government Comerce. (2001) “ITIL: The Key to Managing IT Services Best Practice for Service Support”. United Kingdom Stationery Office.

Oktaba, H. (2006) “MoProSoft: A Software Process Model for Small Enterprises”, Proceedings of the First International Research Workshop for Process Improvement in Small Settings – Process Improvement Approaches and Models, Janeiro.

Paulk, M. (1998) “Using the Software CMM in Small Organizations”, The Joint 1998 Proceedings of the Pacific Northwest Software Quality Conference and the Eighth International Conference on Software Quality, p. 350-361.

Project Management Institute. (2000) “A Guide to the Project Management Body of Knowledge - PMBOK Guide”.

Revankar, A., Mithare, R. and Nallagonda, V. (2006) “Accelerated Process Improvements for Small Settings”, Proceedings of the First International Research Workshop for Process Improvement in Small Settings – Selected Case Studies, Janeiro.

Serrabo, M., Oca, C. and Cedilho, K. (2006) “An Experience on Implementing the CMMI in a Small Organization Using the Team Software Process”, Proceedings of the First International Research Workshop for Process Improvement in Small Settings – Process Improvement Approaches and Models, Janeiro.

Silva, O., Borges, C., Salviano, C., Crespo, A., Sampaio, A. and Roullier, A. (2003) “Aplicação da ISO/IEC TR 15504 na Melhoria do Processo de Desenvolvimento de Software de uma Pequena Empresa”, Simpros.

Softex. (2006) “MPS.BR – Melhoria de Processo do Software Brasileiro,Guia Geral”, Maio.

Software Engineering Institute. (1997) “The Personal Software Process – PSP”, http://www.sei.cmu.edu/tsp/psp.html.

Software Engineering Institute. (2000) “TSP – The Team Software Process”, Technical Report, http://www.sei.cmu.edu/tsp.html.

Software Engineering Institute. (2005) “Capability Maturity Model Integration - CMMI”, http://www.sei.cmu.edu/cmmi.

Souza, A., Oliveira J. and Jino, M. (2002) “Riscos de Implantação de Processo de Software em Empresas do Centro-Oeste Brasileiro”, Universidade Católica de Goiás.

A secure analysis pattern for handling legal cases

Eduardo B. Fernandez (*), David L. la Red M. (**), Jorge Forneron (***), Valeria E. Uribe (**), and Gisela Rodriguez G. (**)

(*) Dept. of Computer Science and Eng., Florida Atlantic University, Boca Raton, FL, USA
(**) Dpto. de Informática, Facultad de Cs. Exactas, Universidad Nacional del Nordeste, Corrientes, Argentina
(***) Dpto. de Informática, Facultad de Ciencias Aplicadas, Universidad Nacional de Pilar, Pilar, Paraguay

Abstract

We present here a S*ecure Semantic Analysis Pattern (SSAP).* This is an analysis pattern that combines functional and security aspects. In particular, this SSAP is intended to describe the handling of legal cases, where a client is either suing another party (a plaintiff) or is being defended from a suit (a defendant). To describe SSAPs we have extended a common template with sections on possible attacks (the possible attacks in each action of a use case), needed policies (to prevent or mitigate the attacks), and secure structure (the class model of the solution with security constraints).

1. Introduction

We have proposed the use of Semantic Analysis Patterns (SAPs) to build conceptual models of applications [Fer00]. A SAP is a composite pattern that corresponds to a few fundamental use cases. Using SAPs is possible to build conceptual models in a simpler and more reliable way. We have also developed a methodology to build secure systems [Fer06a]. In this methodology we add instances of security patterns to the functional parts of the conceptual model to define security constraints at the application level. These constraints are then enforced by the lower architectural levels.

We can use SAPs as part of our secure system development methodology. We extend the SAPs to consider possible attacks to the fundamental use cases that define it, and we define policies to prevent the attacks. This is the application of an idea proposed in [Fer06b] which emphasizes that the secure design of a system should be based on its expected types of attacks. Since the SAPs are used to build the conceptual model of an application, we have now a portion of a conceptual model where functional and security aspects are integrated from the start. We call this a S*ecure Semantic Analysis Pattern (SSAP).* In particular, we present here a SSAP to handle legal cases. To describe SSAPs we have extended the template of [Bus96] with sections on possible attacks (the possible attacks in each activity of a use case), needed policies (to prevent or mitigate the attacks), and secure structure (the class model of the solution with security constraints). SSAPs follow the current tendency in security research of integrating business functions with security aspects from the beginning of the development life cycle [Nag05, Sch06a].

Section 2 describes a specific SSAP, a pattern for the Secure Handling of Legal Cases. As indicated, this pattern is intended for system developers trying to incorporate security in their designs. We do not assume legal expertise and a glossary at the end of the paper defines basic law terms.

2. Secure Handling of Legal Cases

This pattern describes the handling of legal cases where a client is either suing another party (a plaintiff) or is being defended from a suit (a defendant). The pattern includes the necessary policies (in the form of security patterns) to stop or mitigate the expected attacks.

2.1 Example

The SueThem law firm is having trouble staying in business. It keeps some documents in electronic form and others in paper. Documents are hard to find and get easily accessed by unauthorized persons. It is hard for the company to keep track of their customers and to know how much it should charge them. The conduction of cases is disorganized, which leads to losing cases because of lack of preparation.

2.2 Context

A legal firm sues parties (persons, organizations, or groups) on behalf of their clients; it can also defend their clients when they are sued. We call a *legal case* the sequence of actions (process) needed to pursue a suit until its completion. The standard legal system of most countries allows parties to sue other parties. There are different types of lawsuits but they are not of interest here. Interactions between the people involved can be in person, by telephone, by regular mail, or by email. Law firms are commercial entities and must compete with other law firms for clients.

2.3 Problem

A law suit or defense implies a sequence of actions and generates many documents of several types. If the firm doesn't organize properly these actions and the corresponding documents, it will have problems in conducting the suit or defense, which will result in unnecessary expenses and in a higher possibility of losing the case. Because the information handed in a case is very sensitive, there is motivation to misuse it. We need to consider possible attacks and take measures to avoid them. We consider here the main use cases in this process: Handle Legal Case (for a plaintiff), Handle Legal Case (for a defendant), and its auxiliary use cases Keep Track of Costs, Research Case, and Billing. Figure 1 shows the actors involved in these use cases. 'Other' represents here people involved in the case such as witnesses or experts. There are other related use cases such as writing of wills or divorce cases, which are left out for simplicity and to make the pattern more reusable. How do we model this system to consider all these factors in a balanced way?

The solution to this problem is affected by the following **forces**:

- *Unpredictability of activities.* The sequence of activities in a case is usually unpredictable. Depositions, witness court appearances, lawyer briefs to the court might be required in any sequence depending on the course of the case.
- *Unpredictablity of people*. Complex cases may require several lawyers with the assistance of some secretaries. The actual number of these people might be hard to predict. In addition to the defendant and the plaintiff (and their respective opponents) we may need witnesses, experts, and other people. Who they are and when they are needed depends on the case.

- *Logistics.* The total effort and duration of a case is variable and we need to keep track of expenses, time used, supplies, etc., so we can bill our clients.
- *Precedent searching.* Handling cases require searching for precedents (similar cases). To do research for cases, lawyers and secretaries make use of libraries and the Internet and may download many documents.
- *Access control to information*. The information about customers, billing, assignment of lawyers, and other aspects related to a current case must be accessible only to authorized persons.
- *Control of documents.*Legal documents can only be created by authorized persons and their use (reading or modification) should also be controlled.
- *Confidentality*. Communications between lawyers and clients must be confidential.
- *Auditability.* Government regulations apply to law firms and their information must be easily auditable.

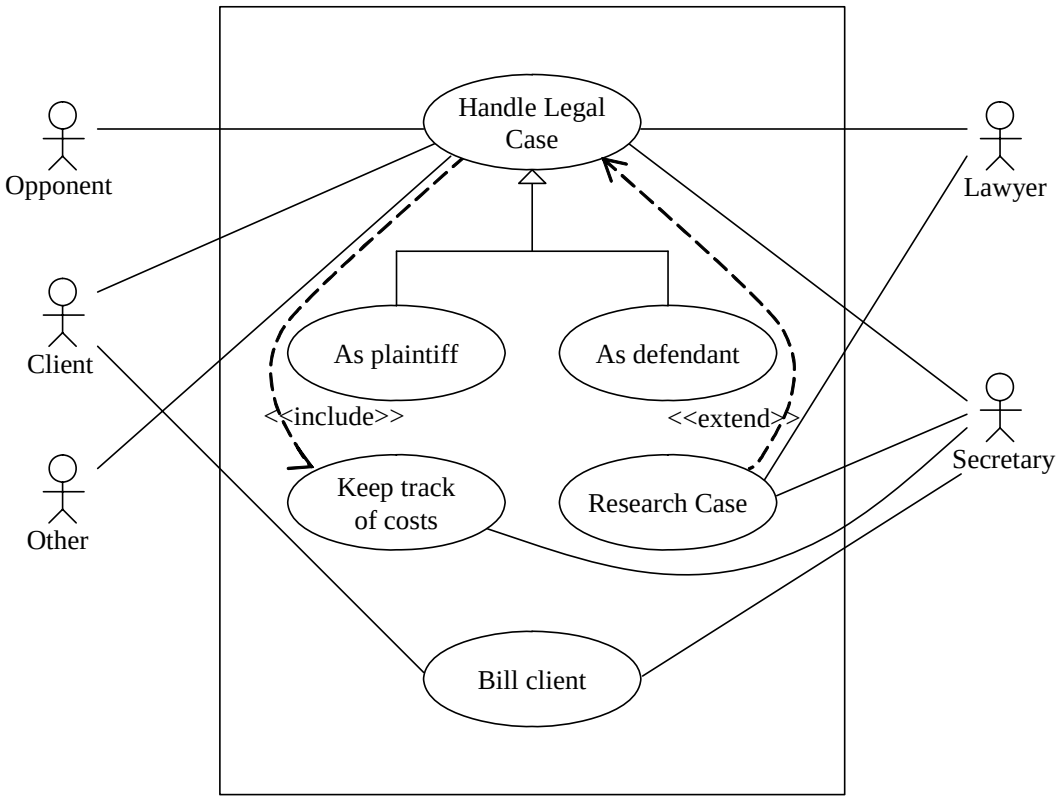


Figure 1. Use cases for handling legal cases

2.4 Possible attacks

Figure 2 shows an activity diagram for the sequence for handling a case followed by billing, tracking of costs, and related case research. Following the approach of [Fer06b], in order to analyze the possible attacks (threats) we consider each activity in the activity diagram of Figure 2 and see how it can be subverted by the attacker. In this diagram External People indicates either the opponent or other people involved in the case. The possible attacks are then:

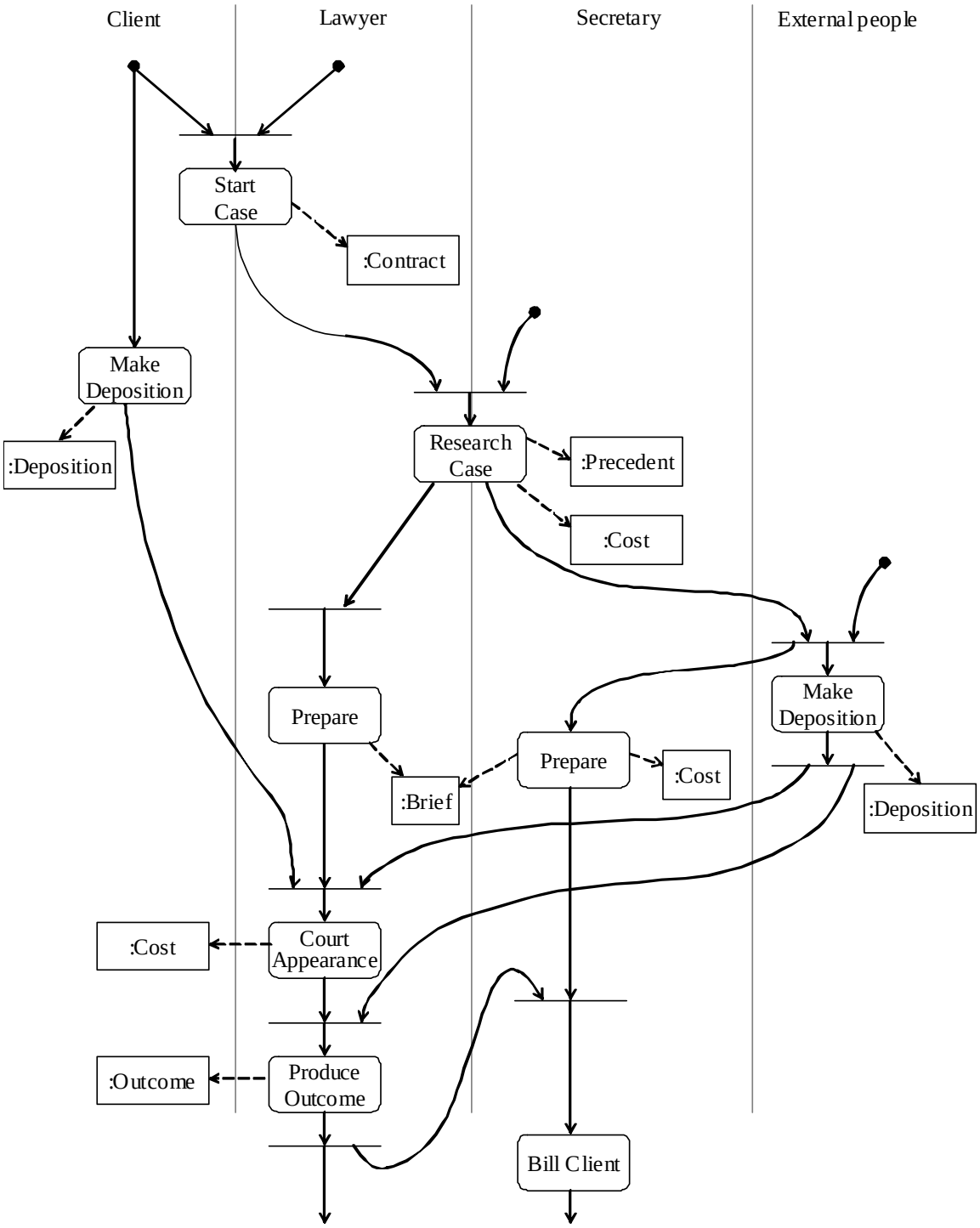


Figure 2. Activity diagram of a case handling

A1 In the ‘start case’ activity, the client or the responsible lawyer might be impostors.
A2 A lawyer might create a false contract.
A3 The client or the external people might give a false deposition.
A4 A lawyer may change a deposition.
A5 A lawyer or a secretary may produce intentionally incorrect precedents, briefs, or costs.
A6 A secretary may produce an increased or decreased bill.
A7 A lawyer may change some aspects of the outcome to collect a higher fee.
A8 A lawyer can disseminate client or case information for monetary gain.
A9 An external attacker may read/change case information or access client/lawyer communications.

2.5 Solution

Because the handling of cases is rather unpredictable and we use a variety of knowledge experts in its handling, this problem can be conveniently modeled as a Blackboard pattern [Bus96]. The case itself becomes a blackboard and the experts providing knowledge to the case are the lawyers, witnesses, or experts. The control is based on the status of the case and is embodied in the scheduling of activities.

Structure

Figure 3 shows a class diagram of the conceptual model for the functional aspects of this pattern. Class **Case** represent the case itself (in the role of Blackboard), and it includes as components classes **Cost** (describes accrued costs), **CaseDocument, Outcome** (the result of the case), and **Scheduling** (the control role of the Blackboard). A **Client** is responsible for a case, and with each case there are some associated **ExternalPeople** (opponents, witnesses, experts). A CaseDocument can be a **Contract**, a **Precedent**, a **Brief**, or a **Deposition**. **Lawyers** and **Secretaries** are **Employees** of the **Law Firm** and can be assigned to cases (we assume this assignment has been done beforehand). A Secretary in the case keeps track of Costs. A Lawyer in the case is responsible for the general conduction of the case, including scheduling.

Dynamics

Figure 4 shows a sequence diagram describing some typical steps for the use case Handle Legal Case as Plaintiff or Defendant. The Client starts the case with the responsible lawyer. This lawyer creates an instance of a case and later does some research for it. He assigns an assistant lawyer to prepare a brief for the court and schedules the client to make a deposition. The other use cases are simpler and not shown for conciseness.

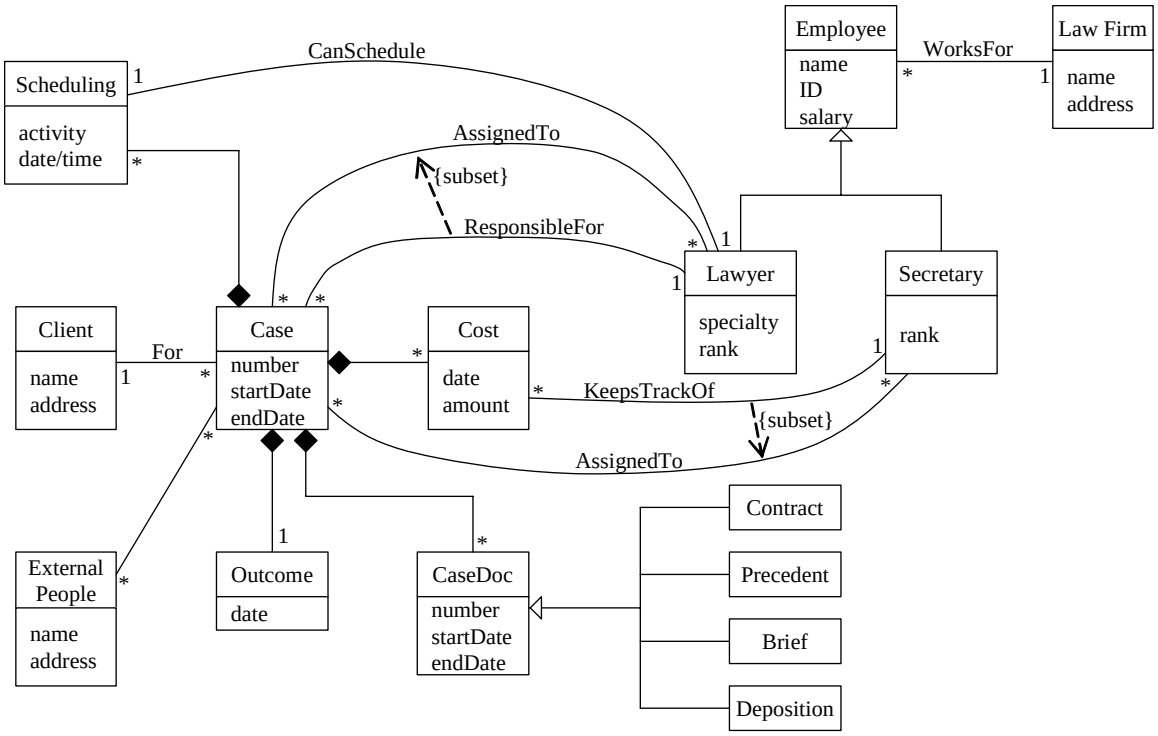


Figure 3. Class diagram for the Handle Legal Cases pattern.

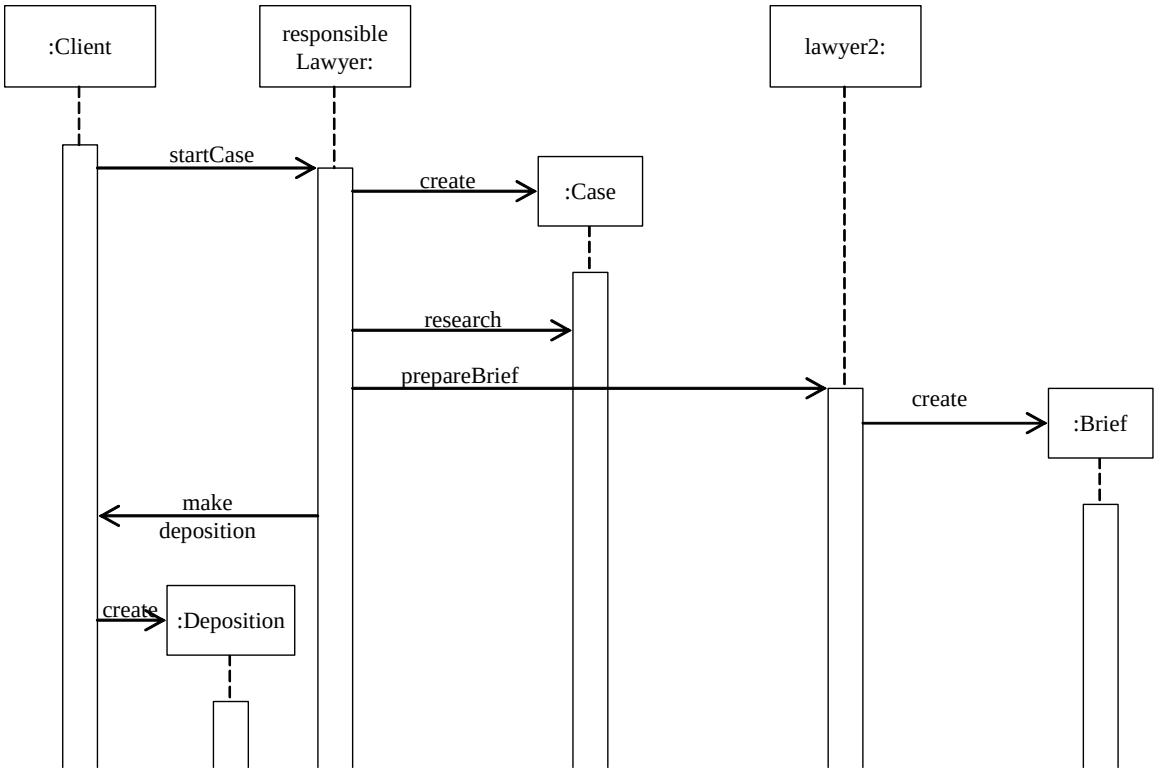


Figure 4 Sequence diagram for use case ‘Handle Case’.

Secure structure

The attacks identified earlier mean that we need the following policies to avoid or mitigate them:

A1 Mutual authentication, to avoid impostors.
A2 Authorization to restrict only lawyers to create contracts, and logging to record possible illegal actions from a lawyer.
A3 Logging, to keep records for future auditing that could detect false depositions.
A4 Authorization and document protection against change.
A5 Authorization and logging, to restrict who can perform these actions and to keep records for future auditing.
A6 Logging, to record suspicious actions of a secretary.
A7 Separation of duty. Two lawyers must concur on the fees to be charged.
A8 Logging, to record possible illegal actions of lawyers.
A9 Authorization and access control to stop external attacks and cryptography to protect communications

From these policies we can define abstract security mechanisms to stop or mitigate the identified threats. Figure 5 shows the relevant part of the conceptual model of Figure 3 with the addition of instances of Authentication, Authorization, and Logging patterns to realize the identified policies. We assume that the authorization policies follow a Role-Based Access Control (RBAC) model and the diagram defines the rights for each role. Both the responsible lawyer (who interacts with the client), and the client must have information to authenticate each other (requiring two instances of the Authenticator pattern). The CaseLog (an instance of the Log pattern) records accesses to the case data. We also need an instance of the Reference Monitor, not show here for simplicity (see [Fer06b].

Example resolved

The SueThem law firm has now a systematic structure to conduct its cases. All its documents are reflected in the conceptual model and can be easily retrieved and audited. The company can now keep track of the costs associated with a case. Documents and other case information can be protected from illegal access.

Consequences

This pattern has the following advantages:

- The Blackboard structure accommodates well unpredictable sequences of activities.
- We can assign lawyers and secretaries dynamically depending on the course of the case.
- The model includes knowledge sources that can be the client, the opponent, witnesses, expert witnesses, and other people.
- It is possible to track the current costs of the case.
- Applying legal regulations to the company is easy because all documents are described by classes with controlled access and we keep a log of accesses.

- Searching for precedents (similar cases) can be done as part of the case handling, we can store this information for future use, and we can associate it to the different stages of the case.

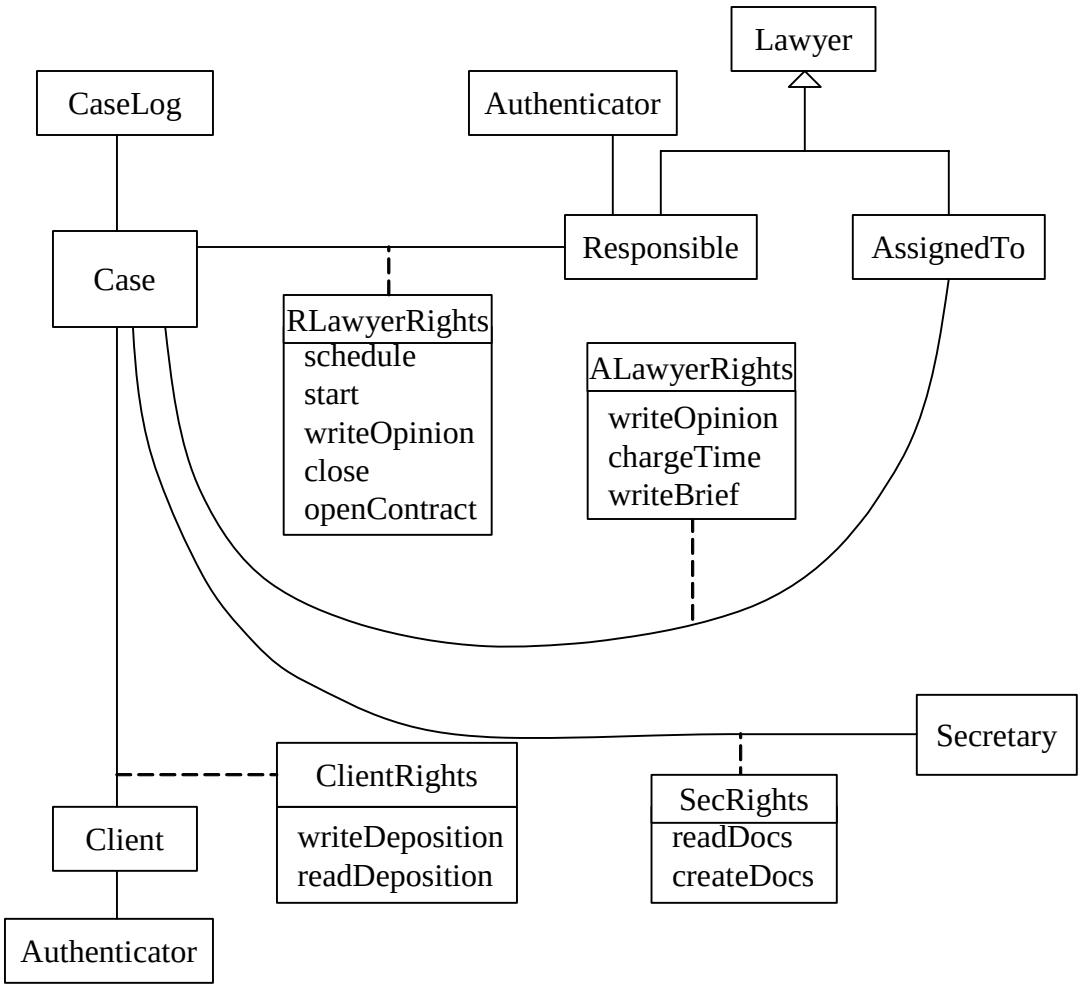


Figure 5. Security additions to the class diagram.

Liabilities include:

- The order in which some activities are performed has an effect in the outcome but the lawyers must decide on the scheduling and the pattern does not help here.
- We might not be able to find all possible attacks, which could allow some attacks to still happen.
- The actual implementation may allow new types of attacks. For example, code flaws may allow an attacker to get control of the operating system and thus to the case data.

Effect on security:

- We can define precise role rights, e.g. an expert can only add to the information, not change it, a lawyer can decide on the next step, bring new witnesses, but cannot change depositions.
- A designer building a system of this type can produce software that performs its functions and is at the same time reasonably secure.
- The RBAC structure enforces authorized access to the information and employees can make sure that they are talking to the person they intend.
- Cryptographic methods can be added to prevent document modification, e.g. hashing [Gol06].

Known uses
Many large law firms follow a similar structure.

See also
- The *Blackboard* pattern [Bus96] is the basis for the central function of the case.
- The client and the external people can be described by a *Party* pattern to indicate that they can be individuals or organizations [Fow97].
- Assignment of lawyers and secretaries uses the *Resource Assignment* pattern [Fer05].
- The rights structure follows an RBAC pattern [Sch06b].
- Authentication is performed by means of instances of the Authenticator pattern [Sch06b].

Acknowledgements

We thank our shepherd, Jorge Ortega Arjona, who provided valuable suggestions that have clearly improved this paper. The Secure Systems Research Group at FAU (www.cse.fau.edu/~security), and the participants in the writers' workshop at SugarLoafPLoP 2007 (Richard Gabriel, Joe Yoder, Ademar Aguiar, Maria Lencastre, Paulo Borba, Rosana Braga, and Mark Perry) provided very useful comments.

References

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern Oriented Software Architecture: A System of Patterns, Volume 1*, Wiley, 1996.

[Fer00] E.B. Fernandez and X. Yuan, "Semantic analysis patterns", *Procs. of 19th Int. Conf. on Conceptual Modeling*, ER2000, 183-195. Also available from: http://www.cse.fau.edu/~ed/SAPpaper2.pdf

[Fer05] E.B.Fernandez, T. Sorgente, and M. VanHilst, "Constrained Resource Assignment Description Pattern". *Proceedings of the Nordic Conference on Pattern Languages of Programs, Viking PLoP 2005*, Otaniemi, Finland, 23-25 September 2005.

[Fer06a] E. B. Fernandez, M.M. Larrondo-Petrie, T. Sorgente, and M. VanHilst, "A methodology to develop secure systems using patterns", Chapter 5 in "*Integrating*

security and software engineering: Advances and future vision", H. Mouratidis and P. Giorgini (Eds.), IDEA Press, 2006, 107-126.

[Fer06b] E. B. Fernandez, M. VanHilst, M. M. Larrondo Petrie, S. Huang, "Defining Security Requirements through Misuse Actions", in *Advanced Software Engineering: Expanding the Frontiers of Software Technology*, S. F. Ochoa and G.-C. Roman (Eds.), International Federation for Information Processing, Springer, 2006, 123-137.

[Fow97] M. Fowler, *Analysis Patterns-Reusable Object Models*, Addison-Wesley, 1997.

[Gol06] D. Gollmann, *Computer security (2nd Ed.)*, Wiley, 2006.

[Nag05] N. Nagaratnam, A. Nadalin, M. Hondo, M. McIntosh, and P. Austel, "Business-driven application security: From modeling to managing secure applications", *IBM Systems Journal*, Vol. 44, No 4, 2005, 847-867.

[Sch06a] A. Schaad, "Security in Enterprise Resource Planning systems and Service-Oriented architectures", *Procs. of SACMAT'06*, ACM, June 2006, 69-70.

[Sch06b] M. Schumacher, E.B.Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering",* Wiley 2006.

Appendix. Glossary of legal terms

Brief--a formal document that sets forth the main contentions with supporting statements or evidence.

Contract--a binding, legally enforceable agreement between two or more parties.

Defendant--a person required to make answer in a legal action or suit.

Deposition—a testimony taken down in writing under oath.

Expert—a person having or displaying special skill or knowledge derived from training or experience.

Opponent--one that takes an opposite position (as in a debate, contest, or conflict).

Plaintiff--a person who brings a legal action.

Precedent--something done or said that may serve as an example or rule to authorize or justify a subsequent act of the same or an analogous kind.

Suit--an action or process in a court for the recovery of a right or claim.

Witness--one who testifies in a cause or before a judicial tribunal.

State MVC: Estendendo o padrão MVC para uso no desenvolvimento de aplicações para dispositivos móveis

Tiago Barros, Mauro Silva e Emerson Espínola

C.E.S.A.R – Centro de Estudos e Sistemas Avançados do Recife

{tgfb, mjcs, ele}@cesar.org.br

Resumo. *Aplicações para dispositivos móveis podem ser implementadas para várias plataformas diferentes, como J2ME, BREW, Symbian, Windows Mobile e Embedded Linux. No entanto, apesar de diferentes, estas plataformas possuem certa semelhança em sua arquitetura, pois todas são dirigidas a eventos. O Padrão SMVC tem por objetivo capturar estas semelhanças ao propor uma arquitetura em que possamos utilizar uma máquina de estados dentro do padrão MVC.*

Abstract. *There are many different platforms for mobile application development such as J2ME, BREW, Symbian, Windows Mobile and Embedded Linux. Although different, they have some common architecture, because each platform is event-driven. The SMVC pattern has the intent to catch these common elements of all platforms by include a state machine inside MVC.*

1. Introdução

Em desenvolvimento para dispositivos móveis as funcionalidades são muito centradas em cenários de uso baseados na interação com o usuário. Tais questões estão facilmente relacionadas com aspectos de manipulação de eventos originados pelos mesmos. Esses eventos, no contexto da codificação, são observados sob as seguintes vertentes: apresentação do modelo de dados; gerenciamento e controle dos eventos; e manipulação da interface com o usuário.

O padrão de arquitetura MVC *(Model-View-Controller)* [Krasner and Pope 1998] é bastante utilizado no desenvolvimento de aplicações para dispositivos móveis pois determina a separação de uma aplicação em três elementos. O *Model* é formado por entidades que representam os dados da aplicação. A *View* tem por objetivo realizar a apresentação destes dados e capturar os eventos do usuário; sendo representada pelas telas. O *Controller* faz a ligação entre o *Model* e a *View*, realizando o tratamento dos eventos, atuando sobre o *Model* e alterando os elementos da *View* para representar a nova forma dos dados.

Neste artigo, será apresentada uma extensão do padrão MVC para o desenvolvimento de aplicações para dispositivos móveis chamado *State MVC (SMVC)*. O padrão MVC será instanciado para o contexto de aplicações para dispositivos móveis e dois níveis

a mais serão sugeridos para que a manipulação de eventos seja realizada de maneira mais eficiente e escalável.

O SMVC é aplicado em cenários de desenvolvimento onde a mudança de interfaces e camada de controle necessitem de rapidez e eficiência, sem que o modelo arquitetural adotado seja um entrave à mudança. O público-alvo principal deste artigo são desenvolvedores de aplicações para dispositivos móveis.

2. SMVC

2.1. Objetivo

Fornecer uma arquitetura para desenvolvimento de aplicações para dispositivos móveis baseada numa extensão do MVC para uma maior eficiência, escalabilidade e melhor escrita de código.

2.2. Contexto

Ainda que o desenvolvimento para dispositivos móveis necessite de ambientes que são orientados a eventos, eles não dispõem de uma estrutura adequada para uma programação eficiente. É comum no desenvolvimento de aplicações em plataformas desta natureza [Forman and Zahorjan 1994] - BREW, J2ME, Symbian, Embedded Linux e Windows Mobile - que o código seja confuso, mesclando em um único lugar o tratamento de todos os eventos da aplicação. Neste código então, torna-se necessário adicionar diversas *flags* de controle potencializando o número de erros.

É importante ressaltar que as aplicações de interação com o usuário - especialmente para celulares - rodam em um único processo, não permitindo assim o bloqueio de uma aplicação em detrimento de outra. Para solucionar este conflito, as plataformas utilizam uma função de *callback*[1] que faz o tratamento dos eventos enviados para a aplicação.

Muito embora esta solução seja de grande valia do ponto de vista de programação, uma única função para tratar todos eventos torna a aplicação cada vez mais complexa. Neste momento, então, é natural que apareçam mecanismos de controle para representar os diferentes estados da aplicação, por exemplo: ao receber o evento `eventX` com o `flag1` ligado (`TRUE`), uma determinada ação deve ser tomada; recebendo o mesmo evento `eventX` com `flag1` igual a `FALSE` outra ação deve ser iniciada. Percebe-se que os valores das *flags* são de fato os estados existentes na aplicação e são eles que devem ser tratados.

Desta forma, um tratamento adequado para os estados e eventos, além de tornar a programação mais simples e intuitiva, proporciona um maior desacoplamento e garante uma melhor extensibilidade e manutenabilidade da aplicação.

2.3. Problema

Como então manipular *eventos* e *estados* garantindo maior fator de produtividade (em linhas de código), facilidade na manutenção e, sobretudo, agregando escalabilidade a novas funcionalidades? Imprimir simplesmente o padrão MVC nas soluções para dispositivos móveis não garante o sucesso na implementação.

[1]Mecanismo utilizado para a realização de operações assíncronas. Uma função é passada como parâmetro e chamada quando a operação termina.

A adoção de padrões arquiteturais que garantem a separação eficiente entre interface e controle facilitam o tratamento dos desafios inerentes à computação móvel.

2.4. Forças

- A adição de novas funcionalidades deve ser facilitada através do desacoplamento do *Controller* e da *View*.
- Projetar a aplicação onde os estados sejam organizados como classes, com métodos para tratar cada evento.
- Deve minimizar a utilização de recursos (memória) pela aplicação, proporcionando um mecanismo para carregá-los quando necessário e liberá-los quando não estiverem mais em uso.
- A descentralização do tratamento de eventos deve ser atingida através da distribuição deste tratamento para os estados.
- A adição, remoção e modificação de estados da aplicação devem ser feitas de maneira a se evitar grande impacto na arquitetura.
- A reutilização de telas comuns deve ser garantida, através de um mecanismo que proporcione o gerenciamento destas telas.

2.5. Solução

Para resolver o problema apresentado, um padrão de projeto composto [Riehle 1997] chamado *State MVC* é sugerido. Este padrão consiste na extensão do padrão MVC, baseada na composição entre os padrões *State* [Gamma et al. 1994] e *Manager* [Sommerlad 1997] para representar o *Controller* do MVC, a fim de fornecer uma melhor manipulação e tratamento de eventos e estados. Além disto, um mecanismo para o controle de telas também baseado no padrão *Manager* implementa a *View*, proporcionando reutilização de telas e facilidade de manutenção.

2.6. Estrutura

A estrutura do SMVC é representada através do digrama de classes UML [Booch et al. 1998] da Figura 1.

Abaixo segue a descrição de cada classe participante do padrão:

- *Application*
  A classe `Application` representa o *Model* do padrão MVC. Além deste papel, esta classe também faz a interface com a plataforma alvo, representando o ponto de entrada da aplicação e possuíndo métodos de inicialização (`StartApp`) e finalização da aplicação (`StopApp`), bem como os métodos para pausar (`PauseApp`) e continuar (`ResumeApp`) a mesma.

- *StateManager*
  A classe `StateManager` representa uma máquina de estados, sendo responsável pela transição dos estados da aplicação e por chamar o método do estado que trata cada evento recebido pela aplicação. Dentro do padrão MVC, esta classe, junto com os estados propriamente ditos, representa o *Controller*.

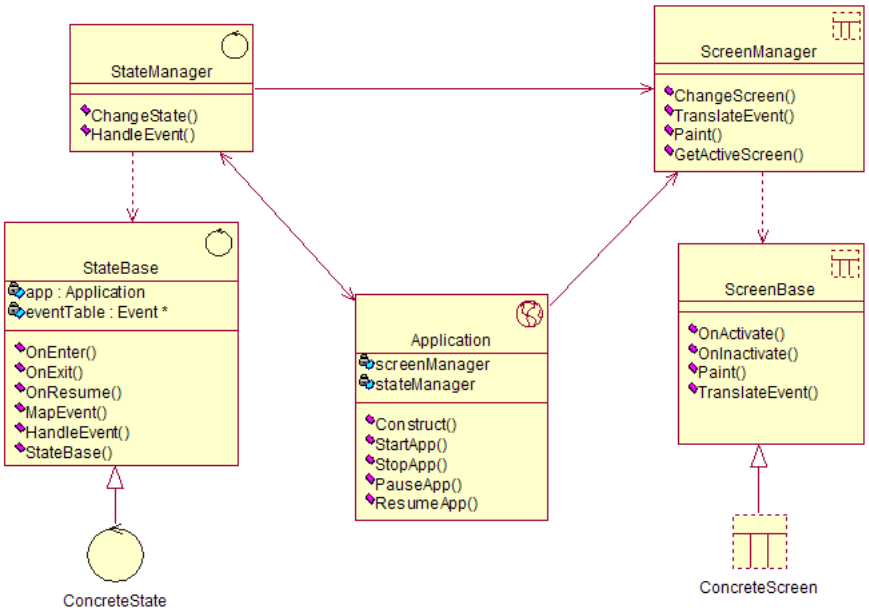


Figura 1. Diagrama de classes do padrão SMVC

- *StateBase*
  `StateBase` representa um estado da aplicação. Esta é uma classe abstrata, da qual todos os estados concretos devem herdar.

- *ConcreteState*
  Deve ser criada uma classe `ConcreteState` para cada estado da aplicação. Cada `ConcreteState` deve implementar um método para cada evento a ser tratado pelo estado.

- *ScreenManager*
  `ScreenManager` é responsável por gerenciar as telas da aplicação. Esta classe, junto com as telas propriamente ditas, representam a *View* do padrão MVC.

- *ScreenBase*
  A classe `ScreenBase` é uma classe abstrata da qual todas as telas da aplicação devem herdar. Ela possui métodos para exibição dos dados da aplicação na tela do dispositivo.

- *ConcreteScreen*
  Cada tela da aplicação é uma `ConcreteScreen`. Esta classe herda de `ScreenBase` e deve implementar seus métodos abstratos.

2.7. Dinâmica

A Figura 2 mostra o diagrama de seqüência da construção de uma aplicação que utilize o padrão SMVC. A aplicação tomada por exemplo implementa dois estados (`StateA` e `StateB`) e uma tela (`Screen1`), que são classes concretas (implementações de `ConcreteState` e `ConcreteScreen`) que herdam de `StateBase` e `ScreenBase`, respectivamente.

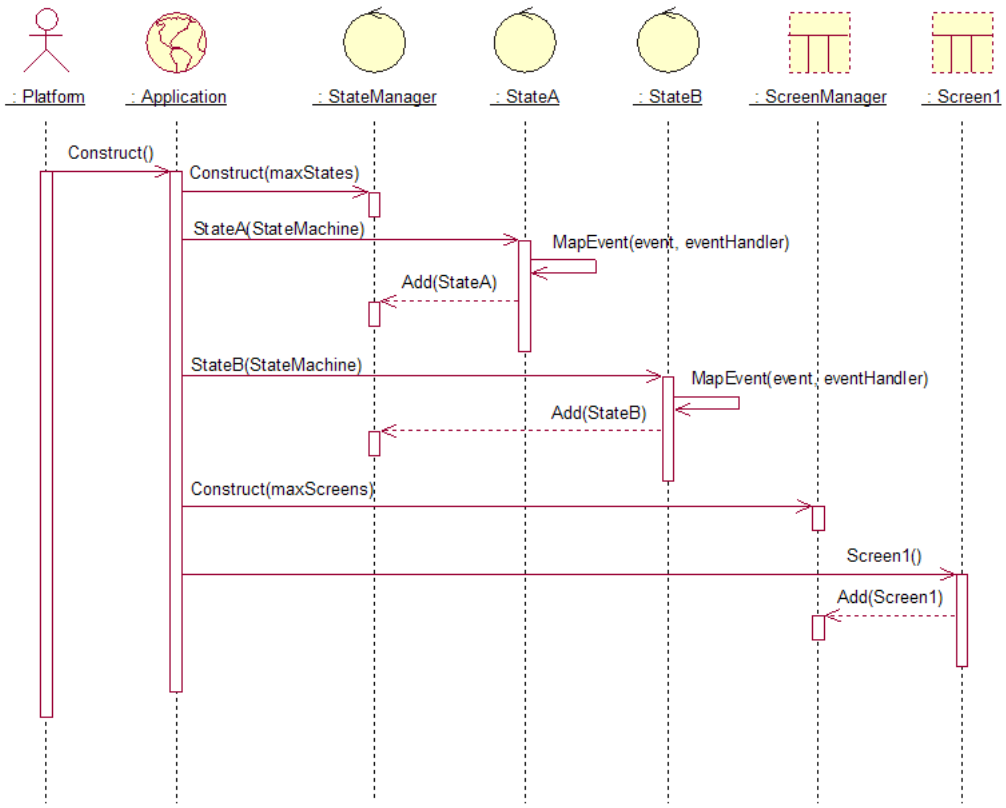


Figura 2. Construção da aplicação

O primeiro método chamado, ao inicializar a aplicação, é o método `Construct` da classe `Application`. A chamada deste método deve estar integrada com a plataforma de desenvolvimento escolhida de forma que ele seja chamado na inicialização da aplicação. Este método é responsável por instanciar os dados da aplicação, bem como todos os estados e telas.

Cada estado criado, é responsável por definir quais os eventos que ele vai tratar e quais os métodos responsáveis por tratar cada evento, através do metodo `MapEvent`.

Na Figura 3 podemos ver a seqüência de inicialização da aplicação. Depois de todos os estados e telas da aplicação serem instanciados, o método `StartApp` da classe `Application` é chamado, o qual deve definir o estado inicial do `StateManager`.

Ao definir o estado inicial, o método `OnEnter` deste estado é chamado, devendo inicializar os dados necessários ao estado e definir qual será a tela apresentada, através do método `ChangeScreen` de `ScreenManager`. O método `ChangeScreen` é responsável por chamar o método `OnInactivate` da tela anterior (caso exista uma

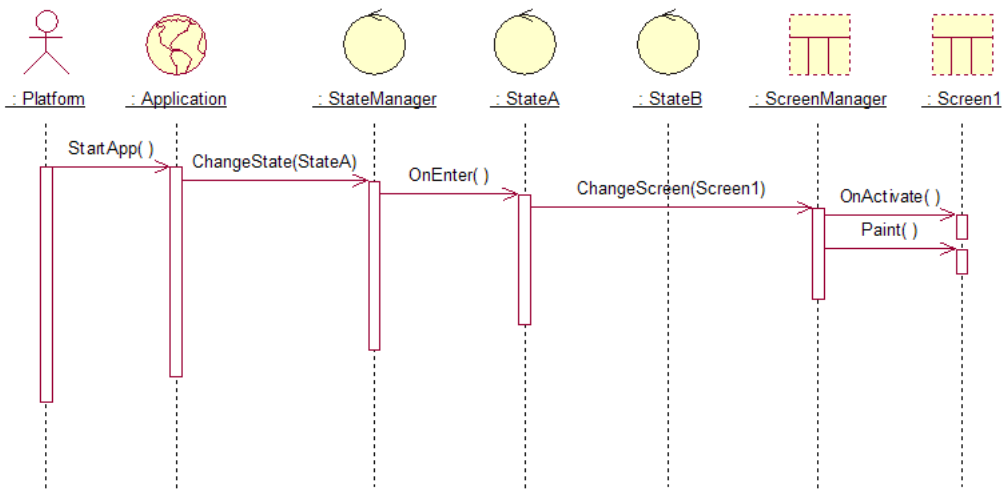


Figura 3. Inicialização da aplicação

tela anterior), para que seus dados sejam removidos da memória, e chamar o método `OnActivate` da tela atual, para que os dados desta tela sejam criados na memória. Depois disto, será chamado o método `Paint` para que a tela atual seja desenhada.

Depois desta inicialização, a aplicação aguardará por eventos para serem tratados pelo estado atual. A Figura 4 mostra o diagrama de seqüência para dois exemplos de tratamento de eventos pela aplicação, um evento de OK, e um evento de EXIT. A seqüência do tratamento de qualquer outro evento é análoga a estes mostrados.

Quando a aplicação recebe um evento, o método `HandleEvent` de `StateManager` é chamado. Este método verifica qual é o estado atual da aplicação e envia este evento para ser tratado, chamando o método `HandleEvent` deste estado. No estado, caso o evento seja um evento de tecla ele será traduzido para um evento significativo, de acordo com a tela que está sendo apresentada. No exemplo do primeiro evento, se a tecla pressionada for a *softkey* da esquerda e, na tela, esta tecla representa a função OK, o evento de *softkey* da esquerda será traduzido para OK.

Depois de traduzido, o estado irá verificar qual é o método responsável por tratar este evento e irá chamá-lo para que execute. A execução do método tratador do evento poderá acarretar em alterações no modelo ou na visualização da aplicação, através de chamadas de métodos de `Application` e `ScreenManager`, respectivamente. Também é possível mudar o estado da aplicação, alterando o estado atual através do método `ChangeState` de `StateManager`.

2.8. Conseqüências

O SMVC oferece as seguintes vantagens:

- Código Modular
  Assim como o MVC, o SMVC desacopla o comportamento do modelo de dados e da visualização. Além disto, o próprio comportamento é modularizado ao dividir o *Controller* em um conjunto de estados.
- Extensibilidade e Manutenabilidade
  Devido ao *Controller* ser implementado como máquina de estados, alterar ou adi-

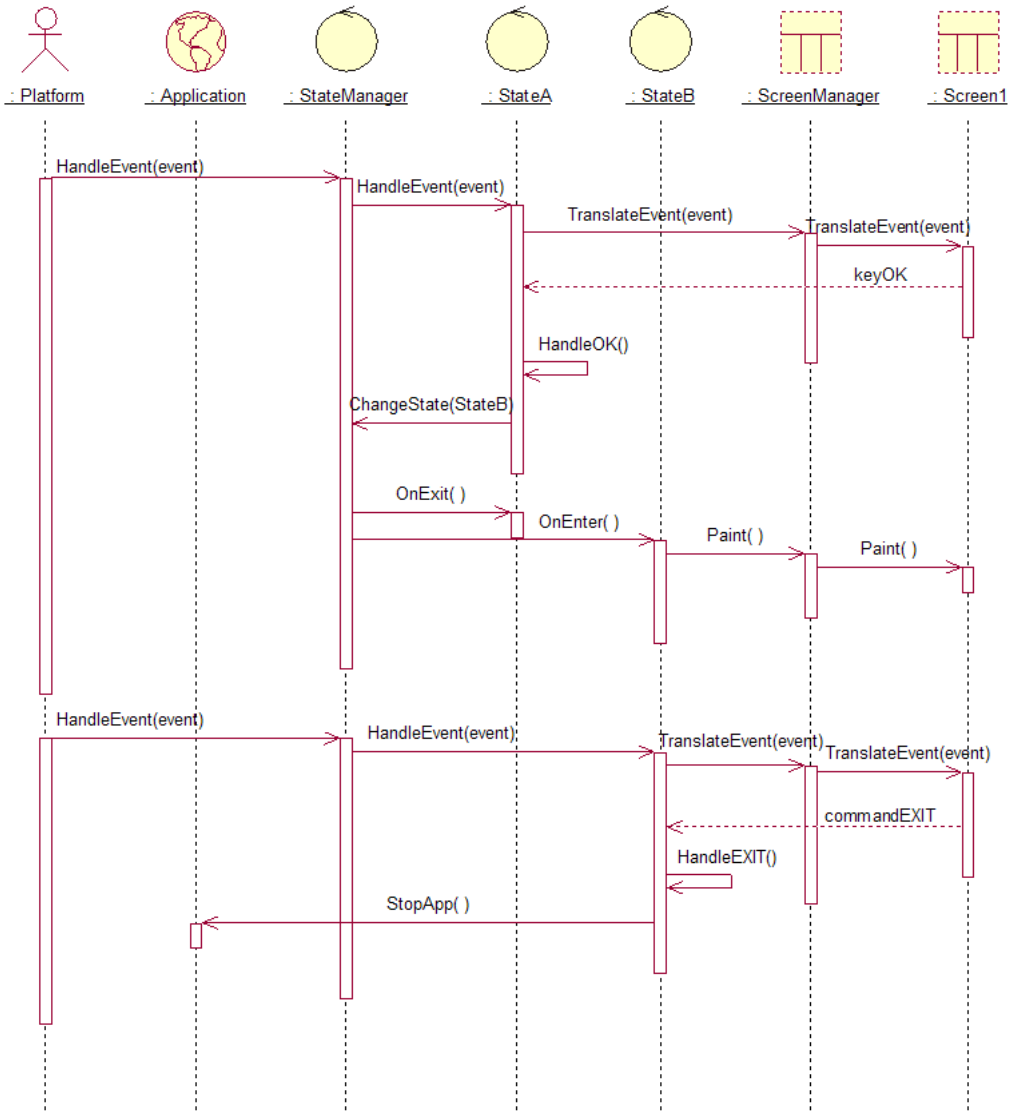


Figura 4. Dinâmica do tratamento de eventos

cionar novas funcionalidades consiste em alterar ou adicionar os estados correspondentes, minimizando bastante o impacto destas mudanças na aplicação completa.

- Reutilização de telas
  Como o código de tratamento de eventos está implementado nos estados (*Controller*), podemos utilizar uma mesma tela em vários estados, evitando a inserção de *flags* no código das telas.
- Redução da memória utilizada
  Os estados da aplicação podem carregar os dados necessários ao seu processamento quando tornam-se ativos e liberar esta memória ao tornarem-se inativos, proporcionando um melhor aproveitamento da memória do dispositivo ao evitar que os dados necessários a todos os estados estejam sempre na memória.
- Descentralização do tratamento de eventos

Os eventos são tratados por métodos específicos de cada estado, evitando-se escrever uma única função para tratar todos os eventos da aplicação.

Em consequência disto, também oferece as seguintes desvantagens:

- Aumento do número de classes
  Com a introdução de `StateManager` e `ScreenManager` além da implementação das classes concretas para os estados e telas, há um aumento no número de classes. Para aplicações pequenas, a Variação 1 (Seção 2.10) do padrão pode ser considerada.
- Duplicação de código
  Eventos tratados em vários estados podem ter seu código de tratamento duplicado na implementação dos estados. Neste caso, é sugerida a Variação 2 do padrão (Seção 2.10), que implementa os estados de forma hierárquica, reusando eventos comuns em níveis de estados intermediários.

2.9. Implementação

Para implementar o padrão SMVC, devemos seguir os seguintes passos:

1. Definir, dentro da plataforma escolhida, de que classe herdaremos a classe `Application`. Por exemplo, em J2ME [J2ME 2007], seria a classe `Midlet` e em BREW [BREW 2007] a estrutura `AEEApplet`.
2. Implementar os dois gerenciadores (`ScreenManager` e `StateManager`). Como estas classes são classes genéricas, deverão ser implementadas completamente desacopladas da aplicação, pois poderão ser reutilizadas nas próximas aplicações desenvolvidas.
3. Modelar a aplicação como uma máquina de estados, verificando quais eventos devem ser tratados por cada estado e quais telas serão apresentadas. Neste ponto, podemos definir se vamos utilizar as variações sugeridas neste artigo.
4. Implementar cada estado modelado, definindo seus métodos tratadores de eventos.
5. Implementar as telas da aplicação.

2.9.1. Exemplo

Foi escolhida a plataforma BREW para demonstração do uso do padrão SMVC. Será mostrada a implementação de uma aplicação tradicional em BREW. Depois mostraremos a implementação do padrão proposto. Esta abordagem visa realizar uma análise comparativa da utilização do SMVC.

Uma aplicação em BREW consiste nos seguintes elementos:

- *Estrutura base da aplicação*
  Deve ser criada uma estrutura para a aplicação que contenha a estrutura `AEEApplet` como primeiro elemento. Isto faz-se necessário visto que `AEEApplet` é a base para qualquer aplicação BREW e é utilizada internamente nas funções de criação da aplicação. Declarando `AEEApplet` como primeiro elemento da estrutura da nossa aplicação permite que se faça um *cast* da estrutura da nossa aplicação para a estrutura `AEEApplet`. Desta forma, pode-se passar a estrutura da nossa aplicação como parâmetro para as funções do framework de BREW.

- *Função de inicialização*
  Esta função é responsável por inicializar (alocar memória) todos os dados da aplicação.
- *Função de finalização*
  Esta função é chamada quando a aplicação termina e é responsável por desalocar toda a memória alocada na função de inicialização.
- *Função de tratamento de eventos*
  É responsável por receber todos os eventos enviados à aplicação. Geralmente é implementada como um grande *switch* que escolhe qual o tratamento adequado, de acordo com o evento recebido.

Abaixo veremos um exemplo de código da estrutura base de uma aplicação BREW, bem como o código das funções de inicialização e finalização da aplicação.

```
typedef struct _HelloWorld
{
    AEEApplet a ; // First element of this structure must be AEEApplet
    AEEDeviceInfo  DeviceInfo; // the hardware device information

    int appScreen; // holds application screen ID
} HelloWorld;


// this function is called when your application
// is starting up
boolean HelloWorld_InitAppData(HelloWorld * pMe)
{
    // Get the device information for this handset.
    pMe->DeviceInfo.wStructSize = sizeof(pMe->DeviceInfo);
    ISHELL_GetDeviceInfo(pMe->a.m_pIShell,&pMe->DeviceInfo);

    return TRUE;
}

// this function is called when your application
// is exiting
void HelloWorld_FreeAppData(HelloWorld * pMe)
{
    // insert your code here for freeing any
    // resources you have allocated...
}
```

Quando a aplicação é inicializada, a função `HelloWorld_InitAppData` será chamada para que os recursos necessários à aplicação sejam alocados. Depois disto, toda a execução da aplicação passa a acontecer na função `HelloWorld_HandleEvent`, que será mostrada a seguir.

```
static boolean HelloWorld_HandleEvent(HelloWorld* pMe,
                                      AEEEvent eCode,
                                      uint16 wParam,
                                      uint32 dwParam)
{
    // switch event code
    switch (eCode)
    {
        // App is told it is starting up
        case EVT_APP_START:
```

```
    // application goes to first screen
    pMe->appScreen = FIRST_SCREEN;
    // send an event to this app to paint the
    // screen
    ISHELL_PostEvent(pMe->a->m_pIShell,
                     HELLOWORLD_CLSID,
                     EVT_USER_REPAINT,
                     0,
                     0);

return(TRUE);

// App is told it is exiting
case EVT_APP_STOP:
// do nothing, just return TRUE
// meaning event was recognized and app
// agrees to be terminated
return(TRUE);

// A key was pressed
case EVT_KEY:
// verify current screen
if (pMe->appScreen == FIRST_SCREEN)
{
    // if key is softkey 1,
    // goto sencond screen
    if (wParam == AVK_SOFT1)
    {
        pMe->appScreen == SECOND_SCREEN;
    }
}
else if (pMe->appScreen == SECOND_SCREEN)
{
    // if key is softkey 1,
    // goto first screen
    if (wParam == AVK_SOFT1)
    {
        pMe->appScreen == FIRST_SCREEN;
    }
    // if key is softkey 2,
    // exit application
    else if (wParam == AVK_SOFT2)
    {
        ISHELL_CloseApplet(pMe->a->m_pIShell, FALSE);
    }
}
// send an event to this app to paint the
// screen
ISHELL_PostEvent(pMe->a->m_pIShell,
                 HELLOWORLD_CLSID,
                 EVT_USER_REPAINT,
                 0,
                 0);
return(TRUE);

case EVT_USER_REPAINT:
```

```
        if (pMe->appScreen == FIRST_SCREEN)
        {
            // Draw first screen
        }
        else if (pMe->appScreen == SECOND_SCREEN)
        {
            // Draw second screen
        }


        //---------------------------------------
        // All other events comes here.
        // Once application becomes complex, this
        // function will become very big.
        //---------------------------------------
    }

    return FALSE;
}
```

O código acima é um exemplo típico de construção de aplicações para dispositivos móveis. Podemos perceber claramente que a variável `appScreen` representa o controle da tela ativa na aplicação. No entanto, analisando o tratamento do evento `EVT_KEY`, visualizamos que esta variável também é utilizada para representar o estado da aplicação.

O crescimento de complexidade desta aplicação irá implicar na adição de mais variáveis como esta para determinar o controle das diversas situações de tela e estado. Esta adição de *flags* proporciona uma pior manutenabilidade e uma maior sucessão a erros, por parte do desenvolvedor, além do excesso de diretivas `if` para verificar estes valores.

Mostraremos abaixo, como resolver estes problemas ao aplicar o padrão SMVC na construção de aplicações para dispositivos móveis.

```
class Application : public AEEApplet
{
public:
    // Application entry point for event handling
    static bool HandleEvent(Application *app,
                            UINT16 evCode,
                            UINT16 wParam,
                            UINT32 dwParam);

    // App memory allocation
    int         Construct();

    // App memory deallocation
    static void FreeAppData(Application *app);

    // Method that is called when app starts
    void StartApp();
    // Method that is called when app ends
    void StopApp();
    // Method that is called when app is suspended
    void SuspendApp();
    // Method that is called when app resumes its
    // execution after being suspended
```

```
    void ResumeApp();

private:
    // State Manager object
    StateManager      *iStateManager;

    // Screen Manager object
    ScreenManager     *iScreenManager;

    // Application data goes here...
};
```

A classe `Application` possui a interface com a plataforma BREW, ao herdar da estrutura `AEEApplet`. Além disto esta classe deve implementar os métodos necessários a sua execução, como os métodos de alocação e desalocação da memória utilizada pela aplicação (`Construct` e `FreeAppData`), bem como o método `HandleEvent`, responsável por tratar todos os eventos recebidos.

```
bool Application::HandleEvent(Application *app,
                              UINT16 evCode,
                              UINT16 wParam,
                              UINT32 dwParam)
{
    UINT16 event, ret = FALSE;

    switch (evCode)
    {

        case EVT_APP_START:
        {
            app->StartApp();
            return(TRUE);
        }

        case EVT_APP_STOP:
        {
            app->StopApp();
            return(TRUE);
        }

        case EVT_APP_RESUME:
        {
            app->ResumeApp();
            return(TRUE);
        }

        case EVT_APP_SUSPEND:
        {
            app->SuspendApp();
            return(TRUE);
        }

        case EVT_KEY_PRESS:
        case EVT_KEY:
        case EVT_KEY_RELEASE:
        // translate the key event in the current dialog
        event = app->iScreenManager->TranslateEvent(aeCode,
```

```
                                                    awParam,
                                                    adwParam);

        // if event is not translated, use it "as is"
        ret =  app->iStateManager->HandleEvent(event,
                                               awParam,
                                               adwParam);
        app->iScreenManager->Repaint();
        return ret;

        // default: send the event to stateManager
        default:
        {
            return app->iStateManager->HandleEvent(evCode,
                                                   awParam,
                                                   adwParam);
        }

    } // switch evCode

    return(FALSE);
}
```

Nesta implementação, o método `HandleEvent` ao receber os eventos de `Start`, `Stop`, `Suspend` e `Resume`, irá chamar os métodos de `Application` responsáveis por tratá-los.

Caso seja um evento de tecla, este evento será primeiramente traduzido pelo `ScreenManager`, de acordo com a tela que está sendo apresentada, e depois enviado ao `StateManager`. Qualquer outro evento será enviado diretamente ao `StateManager`.

O método `HandleEvent` do `StateManager` será então responsável por enviar o evento ao estado ativo, para que seja tratado pelo método correspondente.

Isto descentraliza completamente o tratamento de eventos, evitando a verificação de estados e telas atuais e proporcionando uma maior modularização e extensibilidade do código. Abaixo temos o código do método `HandleEvent` do `StateManager`.

```
int StateManager::HandleEvent(UINT16 evCode,
                              UINT16 wParam,
                              UINT16 dwParam)
{
    int ret = FALSE;

    // look for current state and send event to it
    if(this->iCurrentState)
        ret = this->iCurrentState->HandleEvent(aeCode,
                                               awParam,
                                               adwParam);

    return ret;
}
```

O `StateManager` também possui o método `ChangeState`, que é responsável por alterar o estado ativo, chamando os métodos `OnExit` e `OnEnter` do estado anterior e do novo estado, respectivamente.

```
int StateManager::ChangeState(const UINT16 &aID)
```

```
{
    int ret = FALSE;

    StateBase *state = this->Get(aID);

    if (state)
    {
        // call OnExit from previous state
        if (this->iCurrentState)
            this->iCurrentState->OnExit();

        // change the state
        this->iCurrentState = state;

        // call OnEnter from new state
        this->iCurrentState->OnEnter();

        ret = TRUE;
    }
    return ret;
}
```

A implementação do `StateBase` pode ser vista a seguir. O método `MapEvent` é responsável por mapear eventos em métodos do estado. Este mapeamento pode ser feito utilizando uma tabela de eventos e métodos. Esta tabela associa cada evento tratado pelo estado a um método e pode ser consultada posteriormente para chamar o método desejado.

```
// Event Handler method pointer
typedef bool (StateBase::*EventHandler)(UINT16 wParam, UINT32 dwParam);

int StateBase::MapEvent(UINT16 aEventCode,EventHandler aEventHandler)
{
    int ret = FALSE;
    if ( ( iCurrNumEvents >= 0 ) &&
         ( iCurrNumEvents < this->iMaxEvents ))
    {

        iEventTable[iCurrNumEvents].iEventCode = aEventCode;
        iEventTable[iCurrNumEvents].iEventHandler = aEventHandler;
        iCurrNumEvents++;
        ret = TRUE;
    }
    return ret;
}
```

O método `HandleEvent` é responsável por verificar se o estado trata o evento recebido e chamar o método correspondente.

```
int StateBase::HandleEvent(UINT16 evCode,
                           UINT16 wParam,
                           UINT32 dwParam)
{
    int ret = FALSE;

    for (int i=0; i<iCurrentNumEvents; i++)
    {
        // search for event handler in event table
```

```
        if ((iEventTable[i].iEventCode == evCode) &&
            (iEventTable[i].iEventHandler != NULL))
        {
            // Call event handler for the event evCode
            ret = (this->*(iEventTable[i].iEventHandler))(awParam, adwParam);
        }
    }

    return ret;
}
```

O `ScreenManager` também é implementado de acordo com o padrão de projeto *Manager*. O método `ChangeScreen` é responsável por mudar a tela que está sendo exibida, bem como chamar o método `Repaint` para que a nova tela seja desenhada.

```
int ScreenManager::ChangeScreen(UINT16 aId)
{
    int ret = FALSE;
    ScreenBase *screen = this->GetActiveScreen();

    if (screen != NULL) screen->OnInactivate();

    this->iActiveScreenId = aId;

    screen = this->GetActiveScreen();

    if (screen)
    {
        ret = screen->OnActivate();

        if (ret == TRUE)
        {
            this->Repaint();
        }
    }

    return ret;
}
```

2.10. Variações

Na Variação 1, é possível alterar o padrão SMVC para que possua um único *Manager*. Desta forma, cada tela corresponderia a um único estado. Isto implica num menor número de classes e redução do *overhead*, sendo recomendado para aplicações pequenas.

A Variação 2 consiste em utilizar uma *Hierarchical State Machine* [Samek 2002] para representar o *Controller*. Pertencem a uma *super classe* de estados, os eventos que são tratados da mesma forma em vários estados. Isto faz com que os estados da aplicação herdem destes *super estados*, evitando, assim, a duplicação de código no tratamento destes eventos.

2.11. Usos Conhecidos

O padrão SMVC vem sendo utilizado no desenvolvimento de diversas aplicações para dispositivos móveis no CESAR. Por questões de confidencialidade, não podemos listar nominalmente as aplicações, no entanto é possível ter uma idéia dos domínios de aplicações que utilizaram este padrão:

- Aplicações multimídia;
- Compartilhamento de imagem;
- Sincronização de informações pessoais;
- Aplicações de *slide-show*.

2.12. Padrões Relacionados

- MVC [Krasner and Pope 1998]. O SMVC extende o MVC ao fazer a composição deste padrão com os padrões *State* e *Manager*.

- *State* [Gamma et al. 1994]. O *Controller* do SMVC é implementado como uma máquina de estados usando o padrão *State*.

- *Manager* [Sommerlad 1997]. A *View* e o *Controller* do SMVC utilizam o padrão *Manager* para gerenciar as telas e estados da aplicação.

- *Hierarchical State Machine* [Samek 2002]. Este padrão pode ser utilizado como forma de reusar eventos comuns em níveis de estados intermediários na Variação 2 desse padrão.

- *Observer* [Gamma et al. 1994]. Este padrão e o Controller do SMVC têm objetivos semelhantes. Para cada estado, ambos padrões devem modificar seu comportamento, ou seja, o comportamento de um objeto depende de um estado.

- Strategy [Gamma et al. 1994]. O SMVC e o Strategy se assemelham por terem classes relacionadas que diferem somente nos seus comportamentos. Tais comportamentos são encapsulados e são implementados como uma hierarquia de algoritmos.

Agradecimentos

Este trabalho foi suportado pelo C.E.S.A.R - Centro de Estudos e Sistemas Avançados do Recife.

Agradecemos especialmente a Alexandre Sztajnberg, nosso *shepherd*, pelos comentários e sugestões importantes que proporcionaram melhorias ao nosso padrão.

Referências

Booch, G., Rumbaugh, J., and Jacobson, I. (1998). *The Unified Modeling Language User Guide*. Reading, MA. Addison-Wesley.

BREW (2007). Qualcomm brew - binary runtime environment for wireless. Disponível em http://www.qualcomm.com/brew/.

Forman, G. H. and Zahorjan, J. (1994). The challenges of mobile computing. *IEEE*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Object-Oriented Software*. Addison-Wesley.

J2ME (2007). Sun - java 2 micro edition. Disponível em http://java.sun.com/j2me/.

Krasner, G. and Pope, S. (1998). A cookbook for using the model view controller user interface paradigm in smalltalk-80. In *Journal of Object-Orientated Programming*, volume 1(3), pages 26–49.

Riehle, D. (1997). Composite design patterns. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 218–228, New York, NY, USA. ACM Press.

Samek, M. (2002). *Practical Statecharts in C C++*. CMP Books.

Sommerlad, P. (1997). The manager pattern. In Martin, R., Riehle, D., and Buschmann, F., editors, *Pattern Languages of Program Design 3*. Addison-Wesley.

BulkLoader Pattern

Márcio Santos[1], Uirá Kulesza[2], Carlos José Pereira de Lucena[2]

[1]DATASUS
{marcio.david}@datasus.gov.br

[2] Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
{uira, lucena}@inf.puc-rio.br

Abstract. *This paper describes the BulkLoader design pattern, which aims to minimize the memory amount required by a process that transfers a huge data amount without change the system architectural layers.*

1. Intenção

Este artigo apresenta um padrão de projeto, chamado *BulkLoader*, que tem o objetivo de reduzir a quantidade de memória utilizada em processos, onde há transferência ou a criação de grande quantidade de dados, de forma a não interferir na separação de camadas da aplicação.

2. Exemplo

Esta seção descreve um exemplo de um sistema de informação para gerenciamento de agendas telefônicas. A Figura 1 representa uma arquitetura orientada a objetos deste sistema seguindo o padrão arquitetural *Layer* [2]. De acordo com este padrão, cada camada deve se comunicar com a camada inferior via uma interface bem definida. Tal interface contém o conjunto de serviços que a camada oferece para a camada imediatamente superior. Alguns padrões de projeto foram desenvolvidos de forma a refinar o padrão *Layer*, no contexto de sistemas de informação, tais como: *Service Layer* [1] e *Data Acess Object* (DAO) [1].

A cópia de informações a partir de uma fonte para um destino de dados é bastante comum, contudo quando estamos em um sistema em camadas, este se torna um problema mais complexo devido à especialização de cada camada como visto na Figura 1. Onde fonte/destino de dados é um repositório onde dados são armazenados. Cada camada possui responsabilidades específicas, seja ela negócio, interface com o usuário ou mesmo acesso a dados.

Em geral, são utilizados três tipos de estratégias para o processo de cópia de dados a partir de uma Fonte, são elas: (i) transformar todos os dados da Fonte em objetos em memória; (ii) copiar um a um os objetos da Fonte para o Destino, sendo esse último um repositório de acesso a dados; e (iii) acessar a Fonte de dados a partir da camada de negócios.

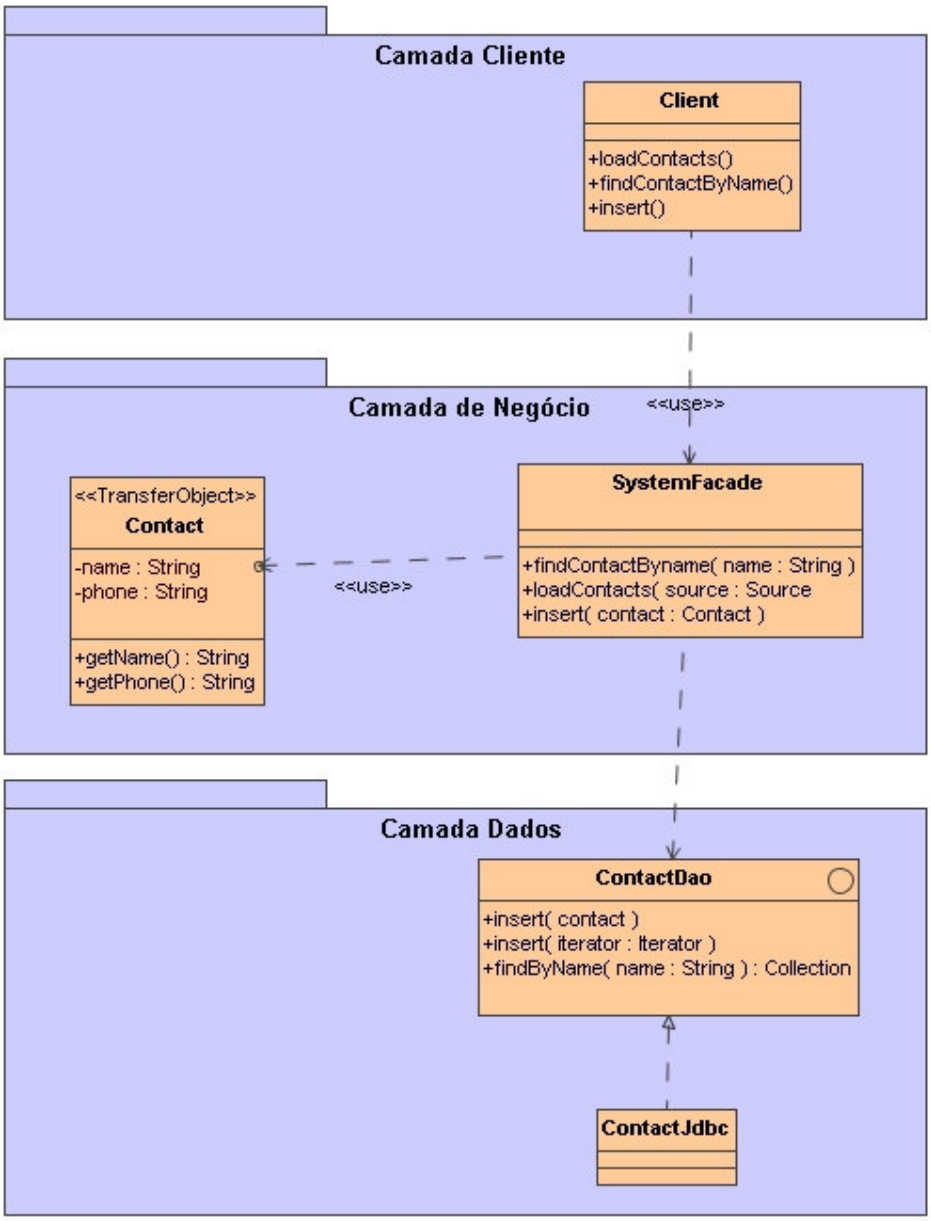


Figura 1: Arquitetura OO em camadas de um sistema de gerenciamento de contatos

A interação entre as camadas neste exemplo funciona da seguinte forma: a classe `Client` é a classe que recebe as requisições do usuário e esta possui uma referência para a fachada (classe `SystemFacade`) que é responsável pelo processamento das informações enviadas pela *camada cliente*. Essa fachada mantém uma referência para um objeto do tipo `ContactDAO` que é responsável pela persistência de dados de um `Contact` no banco de dados.

3. Contexto

Aplicações corporativas são projetadas atualmente seguindo as diretrizes do padrão arquitetural *Layer* [2]. Nesse padrão arquitetural, cada camada oferece serviços para a camada imediatamente superior e requer serviços de camadas inferiores. Dessa forma, requisições do usuário são feitas partindo da camada superior e, em geral, são atendidas pelas camadas inferiores.

No desenvolvimento de sistemas de informação é freqüente a transferência/cópia de dados de uma fonte para um destino, podendo esta transferência ser feita de várias formas:

- Arquivo
  - XML(eXtensible Markup Language)
  - CSV(Comma Separated Value)
- Socket
  - HTTP(Hiper text transfer Protocol)
  - SOAP(Simple Object Acess Protocol)
  - Adhoc
- Mensagens
  - JMS(Java Message Service)
  - E-mail
- Banco de dados

Durante o processo de cópia, normalmente é necessário algum tipo de validação destes dados, seja ela para verificar se os dados estão corretos ou se interessam ou não para a aplicação. Apenas após esse processo de validação, tais dados podem então ser enviados para o destino.

4. Problema

Como copiar uma grande quantidade de dados tendo validação destes dados sem violar a separação de camadas e de forma performática com baixo custo de memória ?

5. Forças

As seguintes forças emergem desse problema:

(1) *Performance*. Em um sistema corporativo geralmente a(s) fachada(s) gerenciam as transações do sistema (através do uso de serviços de frameworks para gerenciamento de transações, tais como, EJB e Spring). Esse processo pode causar uma degradação do desempenho do sistema, pois cada elemento para ser copiado gera uma nova transação, tornando assim o processo muito lento;

(2) *Camadas do sistema*. Uma vez que a aplicação foi desenvolvida em camadas e estas possuem responsabilidades bem definidas, não é interessante para a arquitetura que as responsabilidades das camadas sejam modificadas para atender a funcionalidade de cópia de dados. Ou seja, não é interessante que a *camada Cliente* acesse diretamente os dados armazenados no banco, ou a *camada de negócio* saiba qual a estrutura do arquivo que se está importando. Além disso, sempre que possível é fundamental reusar funcionalidade de classes já implementadas para a arquitetura.

(3) *Uso de memória*. A quantidade de memória utilizada não deve comprometer toda a memória disponível do sistema. Isto pode ocorrer caso a quantidade de dados a serem transferida seja muito elevada. Assim, deve-se restringir o uso de uma grande quantidade de memória, mesmo que isso venha a comprometer a performance do processo de transferência de dados.

(4) *Consistência do processamento*. O processamento deve ser feito de forma atômica, ou seja, se ocorrer alguma falha durante o processamento todos os dados anteriores serão descartados, sendo necessário reiniciar o processo novamente, evitando que a aplicação fique inconsistente. Uma das formas de alcançar esse processamento atômico, é utilizar os serviços de transações disponibilizados por frameworks ou plataformas, tais como, o Spring e EJB.

6. Solução

O padrão *BulkLoader* propõe, para esse problema de transferência de dados, uma solução que realiza a composição de vários padrões conhecidos. Ele adota os seguintes padrões:

(i) o Adapter[4] é usado para abstrair a fonte e o destino dos dados. Tal adapter deve conter um objeto TransferObject [1], que é usado como buffer dos dados sendo transferidos;

(ii) o Iterator [4] usado para percorrer seqüencialmente a fonte dos dados;

(iii) o Strategy [4] que permite definir diferentes estratégias de validação dos dados sendo transferidos. Além de tornar possível a reutilização do Iterator para outras estruturas de dados.

6.1. Estrutura Estática

Na Figura 2 é ilustrada a estrutura do padrão *BulkLoader*. Ele possui 6 participantes:

- `Source` – é responsável por prover uma abstração do tipo físico onde os dados estão armazenados. Possui uma referência para uma instância da classe `TransferObject` , que é utilizada como buffer, evitando a criação de várias instâncias. Isso permite melhorar a performance de duas formas: (i) evitando a criação de várias cópias de objetos; e (ii) evitando o processamento de um coletor de lixo de memória (*garbage collector*) ou outras formas de remoção de instâncias da memória. Esta classe pode ser vista como a implementação do padrão *Adapter* [4] para a fonte de dados;

- `Iterator` – possui a responsabilidade de percorrer a fonte de dados que será copiada. Representa uma implementação do padrão *Iterator* [4];

- `TransferObject` – define uma entidade do sistema, cujos dados estão sendo transferidos a partir de um `Source`. Representa uma implementação do padrão *TransferObject* [1];

- `Validator` – esse sendo responsável pela validação dos dados que estão sendo transferidos do `Source` para o `Target`. Ele é implementado seguindo as diretrizes do padrão *Strategy* [4], de forma a permitir a variação de estratégias de validação. Diferentes tipos de classes de validação podem ser criados em função do objeto `Source` e respectivos `TransferObject` sendo considerados;

• `Target` – representa uma interface para a fonte de dados (*datasource)* onde serão armazenados os dados válidos. Consiste numa implementação do padrão *Adapter* [4];

• `Facade` – sua responsabilidade é gerenciar o processo de cópia e transações de negócio do sistema. Garante dessa forma, que o processo de transferência é atômico, sendo delimitado por uma transação. Representa uma implementação do padrão *Facade* [4].

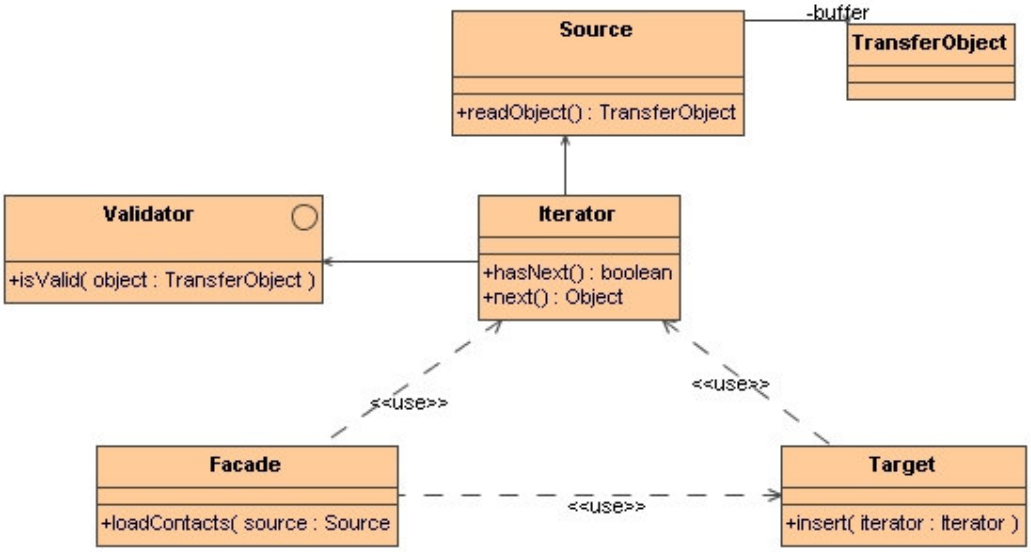


Figura 2: Visão estática da estrutura do padrão.

6.2. Dinâmica

A dinâmica de cooperação dos participantes do padrão *BulkLoader* foi dividida em duas partes para simplificar o seu entendimento. A Figura 3 mostra o diagrama de inicialização do padrão, enquanto a Figura 4 mostra a interação principal do padrão dentro do processo de cópia.

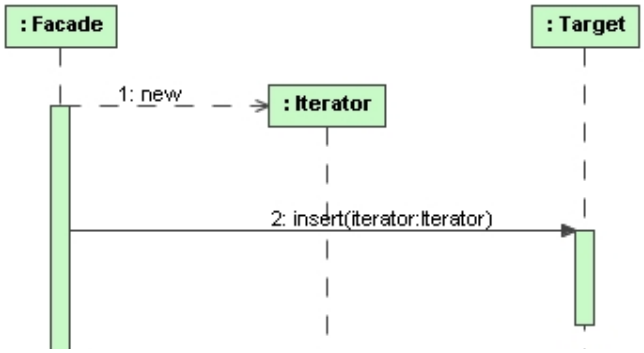


Figura 3: Cenário de Interação ilustrando o processo de inicialização do padrão

Na Figura 3, mostra-se o processo de inicialização do padrão, onde o `Facade` cria o `Iterator` passando como referência os objetos `Validator` e o `Source` dos dados. Em seguida, o `Facade` invoca o método `insert()` de um `Target` passando como parâmetro o `Iterator` criado. Dessa forma delega para o `Target` o processo principal da cópia como pode ser visto na Figura 4.

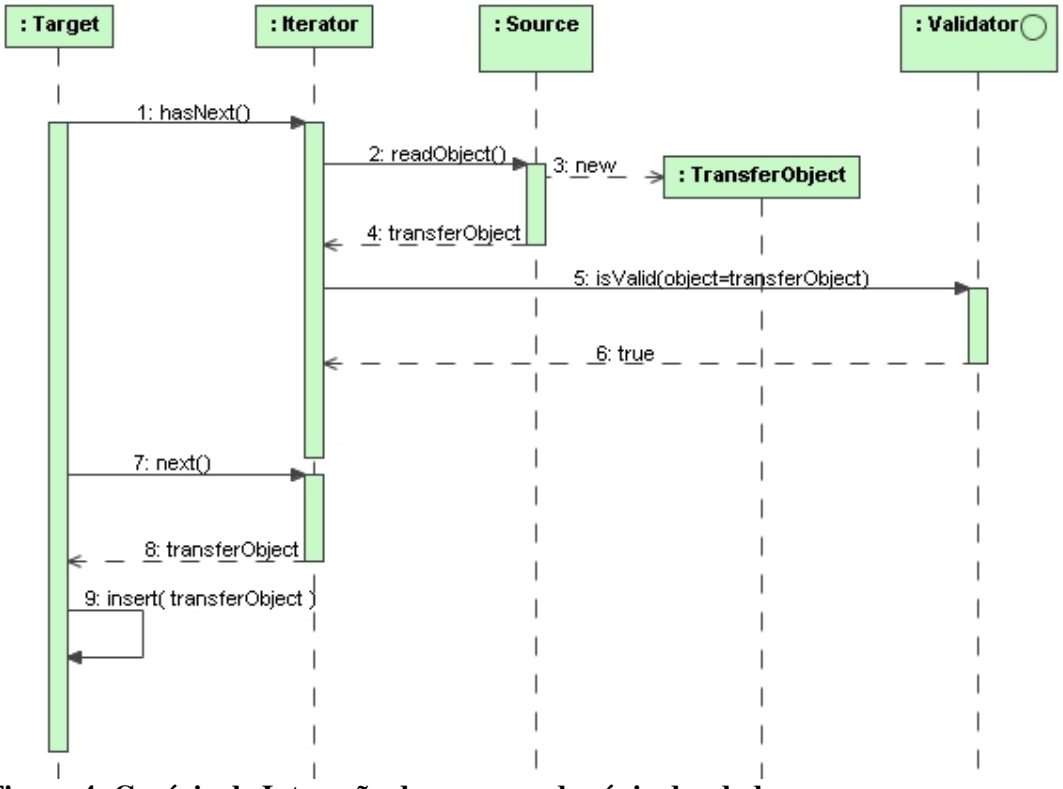


Figura 4: Cenário de Interação do processo de cópia dos dados

Na Figura 4 é ilustrada a interação dos objetos no processo de cópia. O objeto `Target` invoca o método `hasNext()` do `Iterator` com o objetivo de verificar se ainda há algum objeto a ser gravado. O `Iterator` invoca o método `readObject()` do `Source` que por sua vez verifica se o buffer já foi inicializado. Caso o buffer não tenha sido inicializado, o `Source` instancia o `TransferObject` e, em seguida, carrega no mesmo os respectivos valores da transferência de dados. O `Source` então devolve para o `Iterator` a instância do `TransferObject` criada e este por sua vez acessa o `Validator` para validar tal objeto. Caso tal objeto `TransferObject` seja válido, será guardada uma referência para o mesmo, assim retornando `true` para o `Target`. Em seguida, o `Target` invoca o método `next()` do `Iterator` com o objetivo de recuperar a instância de `TransferObject` armazenada. Finalmente, o objeto Target invoca um método para armazenamento do objeto `TransferObject`, tal como o método `insert()`. Esse processo se repete até que o `Iterator` não possua mais nenhum objeto para ser processado.

7. Conseqüências

Benefícios:

- Evita consumo de memória durante o processo de cópia;
- Torna o processo atômico, pois todo o processo ocorre dentro da fachada, e esta é a classe que delimita a transação, além de reduzir o número de transações no sistema durante o processo de transferência dos dados;
- Mantém a separação entre camadas sem violar sua estrutura.

Limitações / Desvantagens

- No contexto de sistemas distribuídos, caso o objeto fonte (`Source`) dos dados esteja localizado em uma máquina remota, é necessário estender a estrutura do padrão para garantir o acesso remoto ao `Source`, através da implementação de um *Proxy* [4].
- Outra desvantagem é o aumento no número de classes.

8. Usos Conhecidos

O padrão *BulkLoader* foi adotado no Sistema SISREG (Sistema de Regulação) [5] do Ministério da Saúde do Governo Federal.

O SISREG é um sistema que foi desenvolvido seguindo as diretrizes do padrão arquitetural Layer[2]. Ele foi desenvolvido usando as seguintes tecnologias: linguagem de programação Java; bibliotecas (APIs) Servlet e JSP; e o framework EJB versão 1.1. O sistema provê uma gama de serviços para seus usuários, tais como: (i) marcação de consultas; (ii) gerenciamento dos leitos hospitalares regulados pelo sistema; (iii) gerenciamento orçamentário das solicitações entre os municípios; e (iv) gerenciamento das solicitações de internação baseadas em laudo médico.

O padrão *BulkLoader* foi instanciado diversas vezes no contexto do sistema SISREG, para implementação das seguintes funcionalidades:

- Processamentos de arquivos financeiros vindo do sistema AIH (Autorização de Internação Hospitalar), para verificação dos procedimentos de internação que foram realmente autorizados. Nesta instância o `source` é um adaptador para o arquivo vindo do sistema AIH e o adaptador destino é o `DAO` de ocorrências de erros do sistema SISREG;

- Sincronização da base de dados de estabelecimentos de saúde do SISREG com os dados do cadastro nacional de estabelecimentos de Saúde (CNES). Nesta instância, o `source` é um adaptador para a API JDBC que acessa a tabela de estabelecimentos de saúde do CNES. O adaptador destino é o `DAO` de estabelecimentos de saúde do sistema SISREG.

- Criação de agendas de consultas. Nesta instância do padrão o source foi implementado não como uma classe que lê objetos provenientes de uma fonte de dados, mas sim como uma classe que define um algoritmo de geração automática de agendas baseado em parâmetros como: intervalo de datas para geração e escala do médico. O adaptador destino é o próprio `DAO` de agenda, disponível no sistema SISREG;

9. Implementação

A seguir será apresentada a implementação da carga de um arquivo em um sistema em camadas utilizando o exemplo descrito na seção Exemplo (Seção 2). Onde será realizado o processamento de um arquivo de contatos no formato CSV *(Comma Separated Values)*, onde cada linha do arquivo contém o nome do contato e o seu telefone.

Na Figura 5 ilustra o diagrama de classes do exemplo (Seção 2), adotando o padrão *BulkLoader.* Os estereótipos representam os participantes do padrão assumido por cada uma das classes apresentadas.

Será realizada a carga de uma grande quantidade de dados a partir do referido arquivo. Neste caso a carga de uma agenda telefônica em formato de texto para um banco de dados relacional. Durante este processo os dados extraídos do arquivo serão validados (o nome do contato não pode estar associado a nenhum contato na base de dados), e tais regras de validação são definidas na classe `ContactManager`.

A seguir serão apresentadas as classes e suas respectivas implementações utilizando-se para isso da linguagem de programação Java.

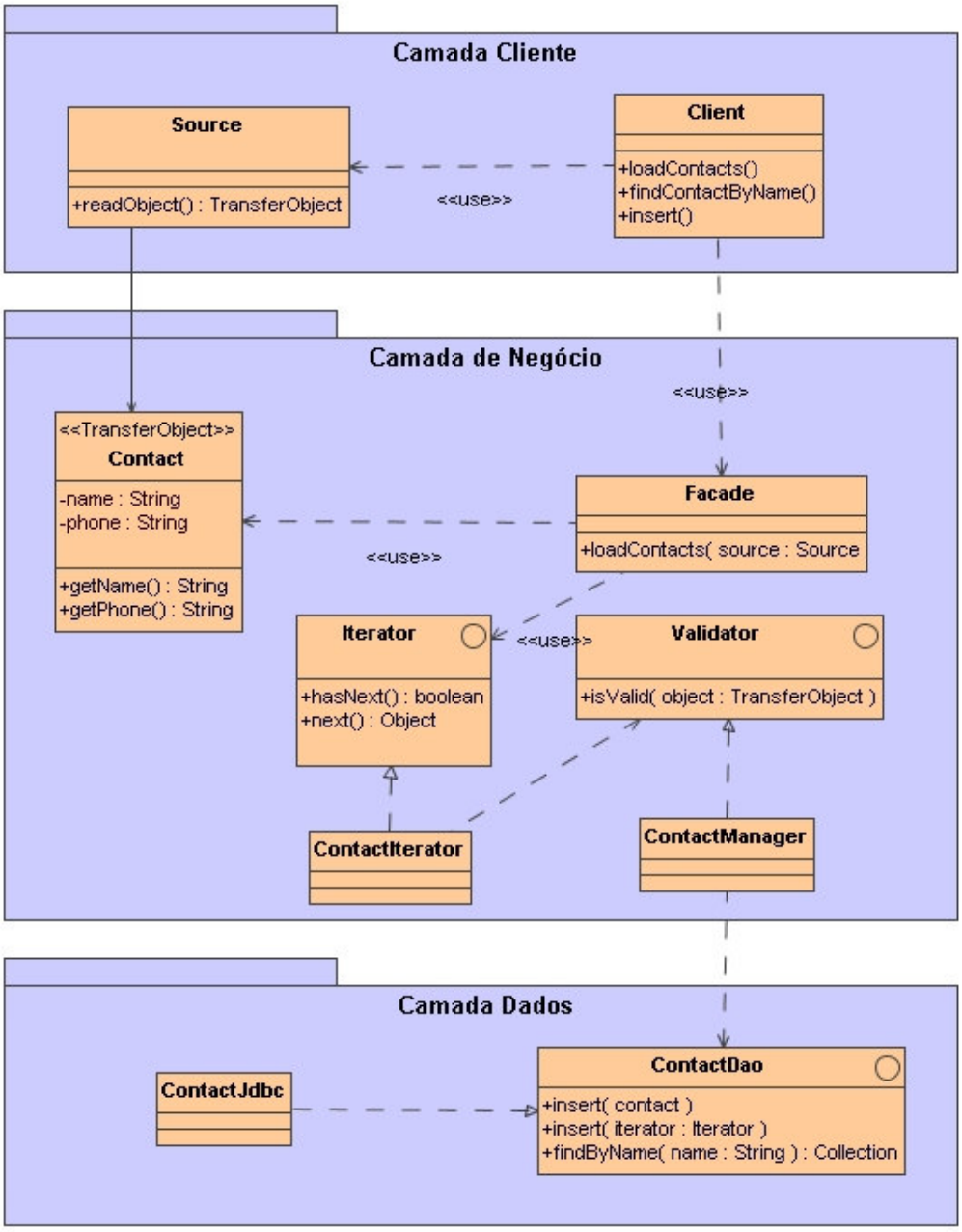


Figura 5: Exemplo de Instanciação do padrão *BulkLoader*

Passo 1: Implementação do Source

Representa o adaptador da fonte de dados que neste exemplo é um arquivo texto. A classe `Source` é responsável por ler o arquivo texto e carregar os dados deste arquivo para o contato, neste exemplo funcionado como um buffer, com o objetivo de minimizar a quantidade de memória a ser utilizada do processamento do arquivo.

```
public class Source {
	private Contact contact;

	private BufferedReader br;

	public Source(BufferedReader br) {
		contact = new Contact();
		this.br = br;
	}

	public boolean loadNext() {
		String entity = null;
		do {
			entity = nextRecord();
			load(entity);
			// eventuais erros de processamento podem ser logados no
			// banco de dados ou em um arquivo separado
		} while (entity != null);

		return  entity != null;
	}

	public Contact readObject() {
		if (!loadNext()) {
			contact = null;
		}

		return contact;
	}

	private String nextRecord() {
		try {
			return br.readLine();// retorna null se acabou o arquivo
		} catch (IOException e) {
	// pode ser gravado no log da aplicacao relativo ao processamento
			return null;
		}
	}

	private void load(String representacaoEntidade) {
	    if (representacaoEntidade != null) {
		StringTokenizer st =
			new StringTokenizer(representacaoEntidade, ",");
		contact.setName(st.nextToken());
		contact.setPhone(st.nextToken());
	    }
	}
}
```

Passo 2: Implementação do Iterator

`Iterator` é uma interface semelhante a interface do *pattern Iterator [4]*. É através desta interface que o adaptador destino (`Target`) solicita referência para o próximo contato a ser persistido na base de dados. Ela possui dois métodos `hasNext()` e `next()`, o primeiro indica se ainda há contatos a serem processados e o segundo retorna a instância do contato a ser processado.

```
public interface Iterator {

   public boolean hasNext();

   public Object next();

}
```

Passo 3: Implementação do TransferObject

`Contact` é a classe que representa os dados de um contato no caso nome e telefone, que serão transferidos do arquivo para o sistema.

```
public class Contact {
      private String name;

      private String phone;

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }

      public String getPhone() {
            return phone;
      }

      public void setPhone(String phone) {
            this.phone = phone;
      }

}
```

Passo 4: Implementação do DAO

Representa a interface do adaptador[4] de destino, específico para a forma como será armazenada a entidade e desta forma seguindo o pattern Dao [1]. Esta interface será utilizada de duas formas uma pelo *ValidaTor* e outra pela *Facade*.

```
public interface Dao  {

      public void insert(Iterator iterator);
      public Contact findByName(String name);
}
```

Passo 5: Implementação de um DAO Concreto

Representa a implementação do adaptador Destino (Target). Neste exemplo consiste na implementação da interface Dao para gravar o contato em um banco de dados relacional utilizado-se da API JDBC do java.

```
public class ContactDaoJdbc implements Dao {

      public Contact findByName(String name) {
            Connection connection =  getConeccao();
            PreparedStatement ps =
    connection.prepareStatement("select * from contatcs where name = ?");
            ps.setString(1, name);
            ResultSet rs = ps.executeQuery();
            Contact contact = null;
            if (rs.first()){
            contact = new Contact();
            contact.setName(rs.getString(1));
            contact.setPhone(rs.getString(2));
            }
            return contact;
      }

      public void insert(Iterator iterator) {
            Connection connection =  getConeccao();
            PreparedStatement ps =
         connection.prepareStatement("insert into contacts values(?,?)");

            while (iterator.hasNext()) {
                  Contact contact = (Contact) iterator.next();
                  ps.setString(1, contact.getName());
                  ps.setString(2, contact.getPhone());
                  ps.addBatch();
            }

      }
...
}
```

Passo 6: Implementação do Iterator Concreto

Esta classe é responsável por percorrer toda a fonte de dados de forma a solicitar a validação dos dados para o `Validator` a medida que tais dados são recuperados do adaptador `Source`.

A dinâmica de funcionamento envolvendo a classe Iterator é a seguinte: (i) o cliente do `Iterator` acessa o método `hasNext();` (ii) tal método recupera um conjunto de dados a partir do objeto `Source;` e (iii) em seguida, delega para o `Validator` a validação de tal objeto. Se o objeto não for válido será recuperado outro no `Source`. Se o objeto for válido, ficará disponível para o método `next()`.

```
public class ContactIterator implements Iterator {
	private Source source;

	private Validator validador;

	private Contact contact;

	public ContactIterator(Validator validador, Source fonte) {
		this.validador = validador;
		this.source = fonte;
	}

	public boolean hasNext() {
		boolean valid = false;

		do {
			contact = source.readObject();
			if (contact==null){
				valid = validador.isValid(contact);
			}
		// eventuais erros de processamento podem ser logados no
		// banco de dados ou em um arquivo separado
		} while (contact != null);

		return valid && contact != null;
	}


	public Object next() {
		return contact;
	}
}
```

Passo 7: Implementação do Validator

O `Validator` define uma interface que representa de forma abstrata a estratégia de validação dos contatos lidos do `Source`. Implementações de tal interface devem ser criadas de forma a definir implementações concretas da estratégia de validação. Dessa forma, a implementação do Validator pode ser caracterizada como uma instanciação do padrão Strategy [4]..

```
public interface Validator {

       public boolean isValid(Contact contact);

}
```

Passo 8: Implementação do Validator Concreto

Define a regra de negócio de validação do objeto `Contact`. Neste exemplo a regra de negócio necessita que não haja dois contatos com o mesmo nome. Para realizar tal validação utiliza-se o adaptador destino, neste caso o `DAO`, para realizar a consulta do contato por nome. Caso não exista algum contato com o mesmo nome, o método `isValid()` retorna `true`, caso contrário retorna `false`.

```
public class ContactManager implements Validator {
      private Dao dao;

      public ContactManager(Dao dao) {
            this.dao=dao;
      }
      public boolean isValid(Contact contato) {
            Contact entidade = consulta(contato.getName());
            return entidade==null;
      }

      private Contact consulta(String nome) {
            return dao.findByName(nome);
      }

}
```

Passo 9: Implementação do Cliente

Classe responsável por receber a solicitação do usuário e montar a requisição a `Facade`. Esta classe tem o conhecimento da localização do arquivo que será processado e desta forma ela consegue construir o adaptador Fonte dos dados.

```
public class Client {

public void load(){
      try {
            Facade facade = getFacade();
            BufferedReader br =
                 new BufferedReader(new FileReader("agenda.csv"));
            Source source = new Source(br);
            facade.load(source);
      } catch (FileNotFoundException e) {
            e.printStackTrace();
      }
}
...
}
```

Passo 10: Implementação da Fachada

Classe responsável por gerenciar as transações e servir de interface única para o cliente. Essa classe também pode implementar regras de negócio do sistema.

```
public class Facade {

   public void load(Source source){
      // Esse método deve ser delimitado por uma transação
      // Tecnologias como EJB (Session Beans) e Spring permitem que se
      // faca isso via arquivos de configuração
      Dao dao = getDao();
      ContactManager contactManager = getManager();
      Iterator ite = new ContactIterator(source,contactManager);
      dao.insert(ite);
   }

   private ContactManager getManager() {
      …
   }

…
}
```

10. Padrões Relacionados

O padrão *BulkLoader* pode ser visto como uma composição de padrões de projeto para resolver o problema de transferência de dados descritos anteriormente. Os seguintes padrões fazem parte dessa composição:

- *Adapter [4]*

  Utilizado para abstrair o tipo de fonte de dados que o `Source` e o `Target` representam.

- *DAO (Data Acess Object)[1]*

  Uma das formas para implementação dos adaptadores `Source` e `Target`

- *Strategy [4]*

  Pode ser utilizado para permitir a variação da estratégia de validação das entidades vindas do `Source`.

- *Iterator [4]*

  Utilizado para percorrer a fonte de dados seja ela o arquivo ou banco de dados ou qualquer outro tipo de meio de armazenamento.

- *Transfer Object[1]*

  É utilizado para armazenar temporariamente os dados, tal qual um buffer.

- *Domain Object [3]*

  Esse padrão pode ser usado para armazenamento dos dados, contudo neste caso o próprio objeto efetuaria as validações mais simples e de consistência do objeto.

Referências

[1] D. Alur, J. Crupi, D. Malks. Core *J2EE Patterns: Best Pratices and Design Strategies.* Prentice Hall, 2001.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal. *Pattern-oriented software architeture: a system of patterns*. John Wiley & Sons Ltd, New York, 1999.

[3] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] SISREG – Sistema de Regulação, Ministério da Saúde, DATASUS, Brazil, 2006. URL: http://www.portalsisreg.epm.br/

SugarLoafPLoP´2007

Pattern Applications

Colaboração entre padrões arquiteturais, de projeto e de interface na construção do framework Athena

Gabrielle D. Freitas[1], **Luciana V. Lourega**[1], **Marcos C. d'Ornellas** [1]

[1]Universidade Federal de Santa Maria – Pós-Graduação em Engenharia de Produção
Av. Roraima, Campus Universitário – 97105-900, Santa Maria, RS

{gabi, lourega, ornellas}@inf.ufsm.br

Abstract. *This paper presents a solution to implement the Athena framework which is dedicated to builder graphical user interfaces. It makes interfaces is a important activity into development process systems, because it through them that the software will be obtain success. A way to builder those interfaces is use frameworks. This way, the Athena framework aims to make easy construct image segmentation applications. In order to implement this framework was used a cooperation between architectural patterns, design patterns and graphical interface patterns. This approach allowed some important characteristics, such as legibility, easy maintenance and reusability of source-code.*

Resumo. *Este trabalho apresenta uma solução para implementar o framework Athena, o qual é dedicado à construção de interfaces gráficas. Produzir interfaces gráficas é uma atividade importante dentro do processo de desenvolvimento de sistemas, uma vez que é por meio delas que o software será bem sucedido. Uma forma de construir essas interfaces é por meio da adoção de frameworks. Assim, o framework Athena tem o objetivo de simplificar o processo de desenvolvimento de aplicações de segmentação de imagens. Para implementar esse framework foi utilizada uma cooperação entre padrões arquiteturais, de projeto e de interface. Essa abordagem permitiu características como legibilidade, fácil manutenção e reusabilidade de código-fonte.*

1. Introdução

Atualmente, a indústria de desenvolvimento de software tem crescido e ganhado destaque, tornando-se um mercado bastante competitivo, no qual a qualidade do software é fator preponderante, influenciando o sucesso ou o fracasso de um sistema. Nesse contexto, o software, para ser considerado de qualidade, deve apresentar atributos, como, facilidade de extensão, flexibilidade, portabilidade e confiança. Dessa forma, o reuso é um princípio importante uma vez que permite a construção de sistemas pela aplicação de unidades bem especificadas e testadas.

Existem diversas formas de reuso, como por exemplo, reuso de código, de projeto, de componentes, de design, de arquitetura, padrões, frameworks, entre outros. Conforme Fayad [Fayad et al. 1999], um framework é um conjunto de classes cooperantes que constroem um projeto reutilizável para uma determinada categoria de software. Com a utilização de um arcabouço de classes (ou framework), tem-se a definição da arquitetura da aplicação, da estrutura geral, da divisão do problema em classes, das responsabilidades

de cada uma dessas classes, da colaboração entre os objetos e também, do fluxo de controle do sistema.

Com a generalização dos usuários, a queda do preço do hardware e a aceitação da Internet ocorreu a popularização do uso dos computadores. Esse fato gerou o crescente interesse no projeto e na implementação de interfaces humano-computador. Nesse contexto, as interfaces estão sendo consideradas as "embalagens" dos softwares e, como conseqüência, se as mesmas forem de fácil utilização e de simples aprendizado, o usuário tende a utilizar a aplicação [Nielsen 1993].

Atualmente, 60% de todo o código em um programa é dedicado a construção de interfaces gráficas. Sendo assim, frameworks facilitam a construção de interfaces gráficas por proverem um conjunto de componentes e mecanismos para combinar esses componentes, a fim de implementar uma interface completa. O framework Athena provê componentes que permitem a criação de interfaces humano-computador no domínio da segmentação de imagens, e, para a construção dessa ferramenta, tornou-se necessário a utilização de diversos padrões.

Dessa forma, o presente trabalho está organizado como segue: na seção 2 é descrito o framework Athena, e na seção 3 são apresentados os principais algoritmos de segmentação que formam o domínio de atuação do Athena. A forma como os padrões arquiteturais, de projeto e de interface foram utilizados na construção do Athena é discutida na seção 4. As considerações finais sobre a pesquisa são apresentadas na seção 5.

2. Framework Athena

De acordo com Schmidt [Schmidt et al. 2004], os frameworks mais utilizados para a construção de interfaces gráficas são *wxWindows*[1], *Java Foundation Classes Swing*[2] *e Qt*[3]. Essas ferramentas apresentam características gerais para a construção de qualquer tipo de interface gráfica, como componentes gráficos, caixas de diálogos, tratamento de eventos, estruturas de dados, dentre outros.

À luz das pesquisas realizadas [Freitas 2006], percebe-se a existência de uma lacuna no campo do processamento de imagens, o que se deve à carência de um framework dedicado à construção de interfaces gráficas para algoritmos de segmentação de imagens. Dessa forma, o Athena tenta preencher essa falha, uma vez que tal framework, além de apresentar a maioria das características de ferramentas semelhantes, por meio da utilização da API Swing, implementa o fluxo de controle comum para as diversas aplicações de segmentação de imagens.

Como vantagens do Athena citam-se ainda o leiaute predefinido das interfaces, o fluxo de controle das aplicações e a arquitetura de sistemas que necessitem de algoritmos de segmentação de imagens na solução de seus problemas. Para tanto, este framework apresenta um conjunto de componentes inter-relacionados os quais permitem que as operações gráficas, comuns ao domínio do problema, estejam mapeadas, bem como providencia a arquitetura básica e o fluxo de controle desse tipo de aplicação.

Portanto, o framework Athena pode liberar o desenvolvedor da tarefa massiva de

[1]Página oficial do framework: http://www.wxwidgets.org.
[2]Disponível em http://java.sun.com/products/jfc.
[3]Página oficial do framework: http://www.trolltech.com/products/qt.

combinar componentes gráficos, permitindo que os mesmos dediquem-se especificamente aos problemas de segmentação de imagens. Dessa forma, o framework Athena provê aos desenvolvedores de ferramentas, no domínio da segmentação de imagens que necessitam de visualização científica de dados, uma forma facilitada para implementar as interfaces gráficas de suas aplicações.

3. Domínio da Segmentação de Imagens

Um dos passos fundamentais no processo de reduzir informação na imagem é a segmentação. Dividir a imagem em regiões é útil para identificar unidades estruturais na cena ou para distinguir objetos de interesse. Assim, a segmentação é descrita como o processo que subdivide uma imagem em partes ou objetos constituintes. Essa é uma técnica análoga ao processo visual humano, o qual separa o objeto principal do fundo da imagem [Russ 1999], constituindo uma etapa decisiva na compreensão da cena.

Segundo Facon [Facon 2001], a segmentação pode ser realizada com base em similaridades, descontinuidades, proximidades ou outras características presentes na imagem em questão. A segmentação para imagens monocromáticas é realizada com base nas propriedades de valores de níveis de cinza, que são a descontinuidade e similaridade [Gonzales and Woods 2000].

Na primeira categoria, descontinuidade, a abordagem é particionar a imagem baseada em mudanças bruscas nos níveis de cinza, que ocorrem nas bordas entres os objetos. A segunda classe de métodos baseia-se em similaridades entre as regiões da imagem, e as principais técnicas são algoritmos de threshold, de crescimento de regiões e de divisão-fusão.

3.1. Limiarização (Threshold)

Nessa modalidade de segmentação, o objetivo é separar os pixels que pertencem ao primeiro plano, ou *foreground*, dos pixels do fundo da imagem, ou *background*. O resultado é uma imagem binária, só com duas classes: o fundo preto e os objetos brancos, ou o contrário. Esse método utiliza o histograma para selecionar um valor de limiar.

Por exemplo, uma imagem de 8 bits, em níveis de cinza, apresenta pixels cujos brilhos variam de 0 a 255. Escolhe-se um valor de limiar (ou threshold) de 127, fazendo com que os pixels que estiverem acima desse valor recebam valor um, e os pixels com valores inferiores a 127 serão transformados para zero (preto) caso a imagem segmentada seja binária (1 bit). A Figura 1 mostra um exemplo de uma imagem segmentada por meio do método de threshold, utilizando a ferramenta Quantiphase [Miranda 2004].

3.2. Detecção de Descontinuidades

A forma mais simples para procurar descontinuidades é por meio da varredura da imagem com uma máscara de convolução (filtragem), na qual os valores numéricos de peso representam o tipo da máscara [Gonzales and Woods 2000]. Esse processo baseia-se não somente no valor do pixel em análise, mas também na vizinhança desse ponto.

Uma borda é o limite entre duas regiões com distribuições distintas de níveis de cinza. Quando tais regiões são homogêneas, a transição entre duas regiões pode ser detectada com base na descontinuidade dos níveis de cinza. Na Figura 2 tem-se um exemplo da operação de detecção de bordas.

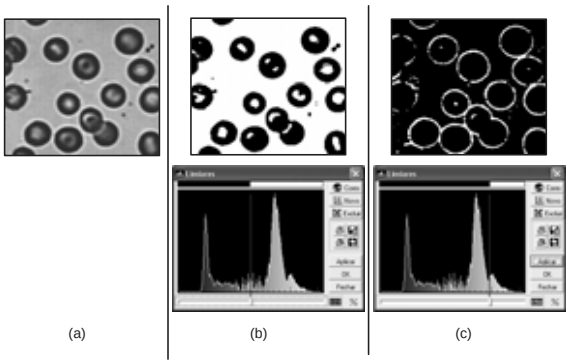


Figura 1. Segmentação por Threshold. (a) Imagem original. (b) Imagem segmentada com um threshold de 127. (c) Imagem segmentada com um threshold de 194.

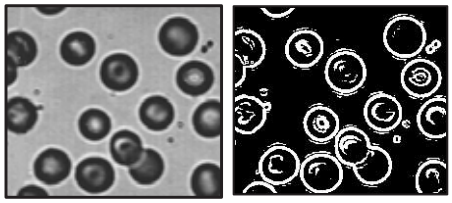


Figura 2. Segmentação por Detecção de Bordas. (a) Imagem original. (b) Imagem segmentada com o operador Frei-Chen de tamanho 3x3 com limiar de 127.

3.3. Crescimento de Regiões

A detecção de regiões em uma imagem pode ser realizada com o objetivo de extrair uma determinada área ou particionar a imagem em um conjunto de regiões distintas [Facon 2001]. Geralmente, as regiões são homogêneas apresentando uma propriedade local constante que pode ser o nível de cinza médio. A técnica de crescimento de regiões agrupa pixels em sub-regiões ou regiões maiores começando por um conjunto de pixels chamados sementes. A partir deles, a região cresce com a adição de pixels desde que esses respeitem o critério de similaridade (nível de cinza). Um exemplo de segmentação por crescimento de regiões pode ser visualisado na Figura 3.

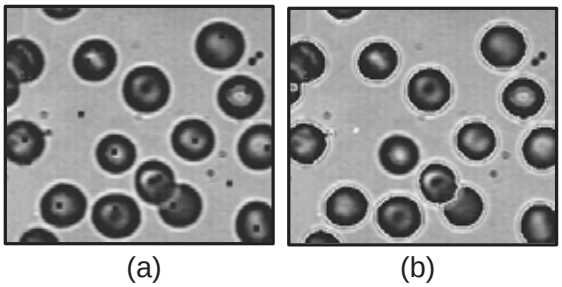


Figura 3. Segmentação por Crescimento de Regiões. (a) Imagem original com sementes. (b) Imagem segmentada.

4. Os padrões empregados para construir o framework Athena

Quando desenvolvedores experientes trabalham em um problema particular, não é comum apresentarem uma solução completamente nova. Geralmente, os desenvolvedores recorrem a problemas similares que tenham sido resolvidos e reusam a essência dessa solução para resolver o novo problema [Buschmann et al. 1996]. A idéia de identificar padrões de problemas para desenvolvimento de sistemas, por meio de catálogos como os de Gamma *et. al* [Gamma et al. 2000] e Buschmann *et. al* [Buschmann et al. 1996], trouxe uma nova forma de se pensar em projetos de software. Atualmente, diversos tipos de padrões são descritos na literatura, como por exemplo, padrões de testes, de interfaces gráficas, de projeto, arquiteturais, entre outros.

Ainda, conforme descrito na seção 1, um framework é uma forma de reuso, tanto de código como de projeto, sendo formado por um conjunto de componentes que interagem para prover a solução comum ao domínio do problema. Um padrão é uma forma de descrever um problema a ser resolvido, uma solução e o contexto no qual a solução trabalha [Buschmann et al. 1996]. Com base nessas definições, é possível afirmar que um framework será composto por diversos padrões e esses por sua vez são mais abstratos que os frameworks [Johnson 1997]. Dessa forma, diversos padrões, entre eles arquiteturais, de projeto e de interface, foram utilizados para construir o framework Athena, possibilitando o reuso de código, da arquitetura e do fluxo de controle das aplicações envolvendo segmentação de imagens. Esses padrões foram encontrados em livros ([Buschmann et al. 1996], [Gamma et al. 2000] e [Tidwell 2006]) e em sites como http://hilside.net.

4.1. Padrão arquitetural *Layer*

De acordo com Buschmann *et. al* [Buschmann et al. 1996], um padrão arquitetural expressa a estrutura e a organização de sistemas de software. Esse tipo de padrão provê um conjunto predefinido de subsistemas, especificando suas responsabilidades e incluindo regras para organizar os relacionamentos entre esses subsistemas.

O contexto do padrão *Layer* é um sistema complexo que exige uma decomposição, onde cada camada seria responsável por tarefas distintas [Buschmann et al. 1996]. Com esse padrão, a mudança em um componente não afetaria outros, pois eles estariam bem encapsulados, cada qual em sua camada. Esse é exatamente o contexto do framework Athena, o qual é um sistema formado por diversas sub-tarefas, como por exemplo, entrada e saída de arquivos, fluxo de controle, interfaces gráficas e bibliotecas utilizadas para desenvolver o framework. Além disso, é importante que cada componente ao ser implementado, apresente a característica de isolamento de mudanças.

O padrão *Layer* foi empregado na arquitetura do Athena a fim de organizá-la evitando a desestruturação dos componentes. Além disso, o *Layer* facilita a implementação, o reuso de camada e de componentes, permite mudanças sem afetar o resto do sistema e possibilita que as dependências sejam mantidas localmente. A escolha desse não foi trivial, tendo em vista que foi necessário um forte embasamento teórico para escolher qual padrão arquitetural se adequaria às propostas do framework.

Sendo assim, o framework Athena é estruturado em três camadas: a camada base formada pela linguagem de programação JAVA e as bibliotecas Swing e JAI; a camada

intermediária, de entrada e saída de arquivos, mensagens e interfaces gráficas; a camada superior constituída pelo componente **principal**.

A base da estrutura é utilizada para implementar os componentes do Athena. O nível intermediário dessa arquitetura tem o objetivo de prover interfaces gráficas aos algoritmos de segmentação de imagens e o topo é responsável pelo fluxo de controle e estrutura do framework. A arquitetura do Athena, organizada conforme o padrão Layer, pode ser visualizada na Figura 4. Essa estrutura segue uma variante do padrão, conhecida como *Relaxed Layered System*, que é menos restritiva sobre o relacionamento entre as camadas, permitindo que cada camada utilize os serviços dos níveis inferiores [Buschmann et al. 1996].

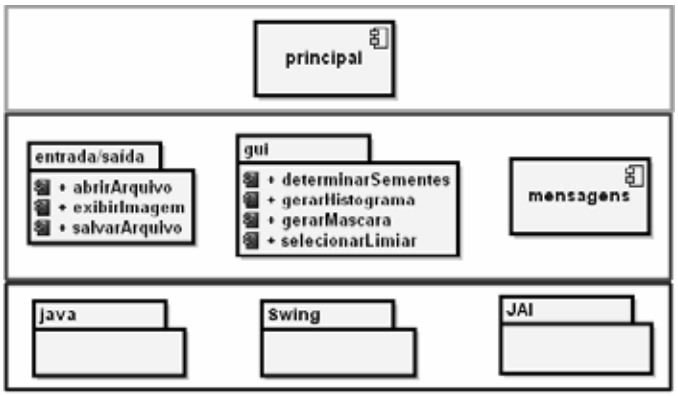


Figura 4. Arquitetura do framework Athena conforme o padrão *Layer*.

4.2. Padrão de projeto *Strategy*

Algumas classes especiais foram criadas, a partir de determinados padrões, para implementar a arquitetura do framework. Nesse sentido, o padrão *Strategy* foi adaptado, originando a classe `strategyComponente`, adotada para definir qual componente o desenvolvedor necessita utilizar. O padrão de projeto *Strategy* tem o objetivo de definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis [Gamma et al. 2000]. *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam.

A escolha do desenvolvedor é realizada com base em uma lista de constantes, chamadas de estratégias, as quais representam os componentes desenvolvidos. A lógica de programação, para a escolha dos componentes, é implementada pelo método `escolheComponente`. Por exemplo, para instanciar o componente **abrirArquivo**, o desenvolvedor precisa escolher a estratégia de número 0 (representada pela constante `abrir`); para o componente **salvarArquivo**, escolhe-se a estratégia 1 (constante `salvar`), e assim sucessivamente. O diagrama que estrutura os componentes do Athena é mostrado na figura 5.

4.3. Padrão de projeto *Façade*

De acordo com Buschmann *et. al* [Buschmann et al. 1996], ao se utilizar o padrão *Layer*, estruturando o sistema em estratos, é necessário definir uma interface de acesso a essas camadas. À luz de suas idéias, o ponto de acesso ao sistema poderia ser implementado utilizando o padrão de projeto *Façade*. Deste modo, outra classe especial, utilizada

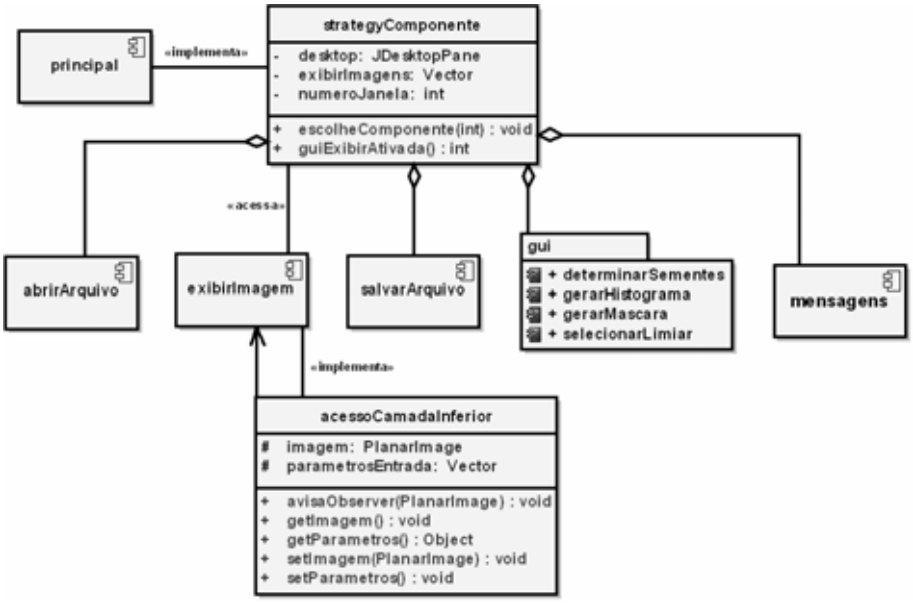


Figura 5. Diagrama simplificado que representa a arquitetura do framework Athena.

na implementação da arquitetura e do fluxo de controle, é a `acessoCamadaInferior`. A partir do padrão *Façade* a classe `acessoCamadaInferior` foi adaptada para facilitar a utilização do Athena pelos desenvolvedores. Esse padrão de projeto tem o objetivo de prover uma interface de acesso unificada para um subsistema [Gamma et al. 2000]. Esse padrão define também uma interface de alto nível que torna o subsistema mais fácil de ser utilizado. Ao se empregar o *Façade* facilita-se a comunicação e minimiza-se as dependências do subsistema. A escolha desse padrão foi simples uma vez que devido ao objetivo de prover uma interface de acesso unificado ao framework, o padrão encaixou-se perfeitamente no problema encontrado.

A classe `acessoCamadaInferior` é necessária para padronizar a forma como os algoritmos de segmentação, implementados pelos desenvolvedores, irão acessar o Athena. Essa classe apresenta o atributo `imagem` (do tipo `PlanarImage`[4]), que armazena a imagem acessada pelo usuário, e o atributo `parametrosEntrada`, do tipo `Vector`, que contem os parâmetros de entrada para tais algoritmos.

Conforme a arquitetura do Athena, ilustrada na figura 5, o fluxo de controle do framework está assim definido: o desenvolvedor escolhe qual componente instanciar por meio da estratégia; o componente escolhido fornece uma interface gráfica que permite ao usuário determinar os parâmetros de entrada para o processo de segmentação. O desenvolvedor implementa o seu algoritmo de segmentação acessando os atributos da classe `acessoCamadaInferior`. O resultado dessa segmentação é uma imagem que deve ser armazenada nessa mesma classe.

Por exemplo, para um algoritmo de crescimento de regiões é necessário que o usuário defina um conjunto de pontos chamados sementes. Esses pontos devem ser armazenados no vetor `parametrosEntrada`. O desenvolvedor, a partir desses

[4]Classe ofertada pela biblioteca JAI que armazena os pixels de uma imagem.

pontos, implementa a lógica para a segmentação, sendo que a imagem resultado é salva no atributo `imagem`. Dessa forma, evita-se que o desenvolvedor tenha que entender toda a implementação do framework, sendo necessário conhecer a classe `acessoCamadaInferior`.

4.4. Padrão de projeto *Observer*

O padrão de projeto *Observer* tem a finalidade de definir uma dependência um-para-muitos entre objetos, de maneira que, quando um objeto muda de estado, todos os seus dependentes serão notificados e atualizados automaticamente. Dessa forma, para atualizar a imagem resultado na tela da aplicação, foi inevitável instanciar um método chamado `avisaObserver`, concebido com base no padrão *Observer* descrito por Gamma *et. al* [Gamma et al. 2000]. Assim, à medida que o atributo `imagem` sofre uma alteração, o método `avisaObserver` informa ao componente **exibirImagem** que a imagem foi modificada e solicita que essa alteração seja mostrada na tela da aplicação.

4.5. Padrão de projeto *Command*

As principais finalidades do framework são: a combinação dos componentes, a utilização/combinação dos componentes implementados fora da estrutura do Athena, o isolamento de mudanças, a facilidade de extensão e manutenção. Para isso, criou-se uma estrutura genérica (ilustrada na figura 6) que é seguida por todos os componentes a fim de alcançar esses objetivos.

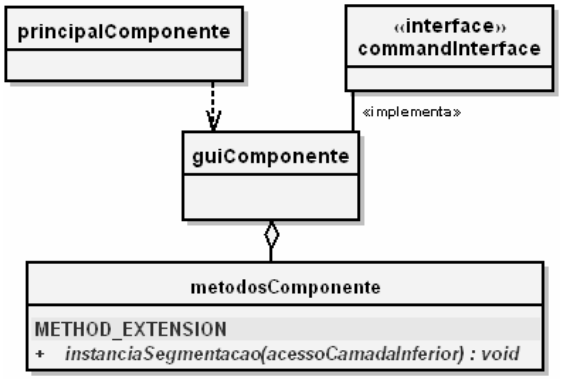


Figura 6. Diagrama de classe que define a estrutura interna de cada componente do Athena

Nessa estrutura, tem-se a classe `guiComponente` que implementa a interface `commandInterface`, uma adaptação do padrão *Command* descrito por Gamma *et. al* [Gamma et al. 2000]. Em Java, as ações de cada elemento gráfico presente, em uma interface, são gerenciados pelo método `actionPerformed` e pelo objeto `ActionEvent` [Welfer 2005]. Entretanto, esse processo sobrecarrega o método `actionPerformed`, fazendo com que todas as ações dos componentes sejam definidas na classe onde `actionPerformed` é implementado. Esta é uma solução possível, contudo é de difícil manutenção e deselegante.

Uma solução para essa problemática é adotar o padrão *Command* para modularizar as requisições. Dessa forma, cada elemento gráfico (um botão, um ícone, um menu, um combo box, entre outros) é implementado como uma nova classe. Cada uma dessas

classes herdam as características da interface pública chamada `commandInterface`, que é o objeto *command*. Entretanto, a escolha do padrão não foi trivial, uma vez que vários padrões poderiam resolver a problemática. O padrão *Command* acaba gerando uma explosão de pequenas classes (uma para cada componente gráfico), mas a legibilidade e a organização do código, providos pelo uso do padrão, tornam essa desvantagem irrelevante. Um exemplo de código para essa solução é apresentado no código 1.

Código 1: Manipulando requisições do usuário de forma mais elegante.

```
01.  //especificação do objeto Command em um arquivo
02.  public interface commandInterface {
03.     public void ExecuteAction();
04.  }
05.  //invocando o objeto Command na guiComponente
06.  public void actionPerformed(ActionEvent e){
07.     commandInterface obj = (commandInterface)e.getSource();
08.     obj.ExecuteAction();
09.  }
10.  //Classe alusiva a um componente gráfico
11.  class BotaoOk extends JButton implements commandInterface{
13.      public void ExecuteAction(){
14.          dispose();
15.     }
16.  }
```

4.6. Os padrões de interface adotados nos componentes do Athena

Os padrões de interface têm a finalidade de auxiliar os projetistas a resolver problemas de design de interfaces Web ou interfaces para aplicações *desktop*. Da mesma forma que os padrões arquiteturais e de projeto apresentam uma solução estruturada para problemas que freqüentemente se repetem, os padrões de interface também provêm soluções para problemas recorrentes, mas no domínio de construção de interfaces gráficas. A identificação desses padrões foi trivial, uma vez que a literatura ([Tidwell 2006]) apresenta diversos exemplos visuais da aplicação dos padrões, o que simplifica e facilita o processo de escolha, por meio da análise do problema e da solução de cada padrão.

O padrão *Titled Sections* é adotado quando existem diversos objetos de informação, sendo que tais objetos precisam ser arranjados espacialmente em uma área limitada. Além disso, o usuário precisa rapidamente compreender a informação e executar a ação dependendo dessa informação. Dessa forma, o padrão indica que todos os objetos devem ser arranjados, com base no conteúdo de cada informação em uma grade. Quando o usuário interage com uma interface com essa organização, ele se sentirá mais confortável para trabalhar [Tidwell 2006].

Para desenvolver o componente **gerarMascara** foi necessário a união dos componentes **selecionarLimiar** e **gerarHistograma** porque o usuário deve escolher um valor de limiar interagindo com o histograma da imagem. Deste modo, percebe-se a importância

de organizar tais elementos em uma única interface empregando o padrão *Titled Sections*, visto que esse padrão tem a finalidade de separar seções por conteúdo [Tidwell 2006].

Assim, ao aplicar o padrão *Titled Sections*, a interface do componente fica separada em duas seções: a superior, que apresenta a configuração do operador de detecção (tipo e tamanho da máscara); a inferior, na qual ocorre a seleção do valor de limiar combinando os componentes **gerarHistograma** e **selecionarLimiar**. A interface do componente é ilustrada na figura 7.

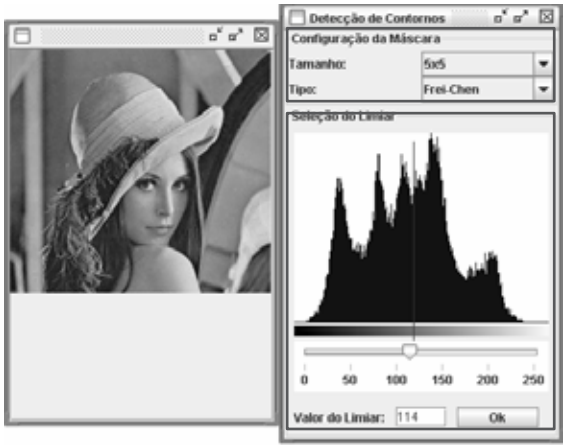


Figura 7. A interface do componente gerarMascara com o emprego do padrão *Titled Sections***.**

O padrão *Fill-in-the-blanks* é utilizado quando um ou mais campos em um formulário devam ser preenchidos. Como é necessário solicitar uma entrada do usuário, geralmente um texto ou número, é importante clarificar qual a informação a ser fornecida pelo mesmo. Esse padrão auxilia a interface a tornar-se auto-explicativa, mostrando para o usuário qual entrada a digitar ou a escolher. Para a construção do componente **selecionarLimiar**, adotou-se esse padrão, visto que o mesmo auxilia o usuário a compreender a forma de interação com o sistema [Tidwell 2006]. A interface gráfica do componente pode ser visualizada na figura 8.

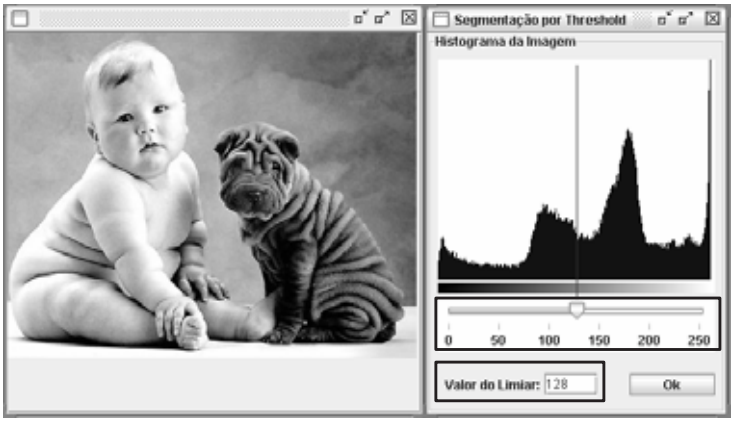


Figura 8. A interface do componente selecionarLimiar emprega o padrão *Fill-in-the-Blanks* **(áreas retangulares).**

O padrão *Dropdown Chooser* estende o conceito de menu usando um painel para exibir um valor mais complexo. O problema a ser resolvido é a situação na qual o usuário tem que fornecer entradas com base em um conjunto de dados. Segundo Tidwell [Tidwell 2006] esse padrão facilita a interação do usuário, a medida que melhora a compreensão da interface.

Já o padrão *Illustrated Choices* é adotado quando se deseja utilizar imagens, ao invés de textos, para mostrar as opções disponíveis. De acordo com Tidwell [Tidwell 2006], a aplicação desse padrão diminui a carga cognitiva do usuário e, além disso, torna a interface mais atraente. Esses dois padrões de interface foram utilizados no componente **abrirArquivo**. O primeiro foi utilizado para permitir a escolha do diretório onde a imagem está armazenada; o segundo, foi aplicado para diminuir a carga cognitiva do usuário, permitindo que o mesmo "veja" qual arquivo será acessado. A figura 9 mostra a interface do componente **abrirArquivo**.

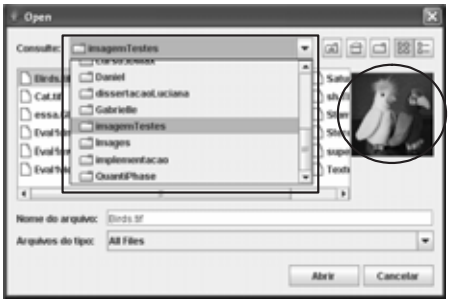


Figura 9. Interface do componente abrirArquivo. A área circulada representa a aplicação do padrão *Illustrated Choices***; a área sob o retângulo mostra o emprego do padrão** *Dropdown Chooser***.**

5. Conclusão

A principal contribuição do trabalho é o própio framework Athena, tendo em vista que o mesmo apresenta soluções aos problemas inerentes à construção de interfaces gráficas no domínio da segmentação de imagens tornando-se uma nova ferramenta, uma vez que não foi possível encontrar um framework semelhante na literatura. Para construir esse framework, tornou-se essencial adotar diversos padrões, dentre eles, padrões arquiteturais, de projeto e de interface.

Nesse sentido, o padrão *Layer* foi escolhido porque permite estruturar o framework em camadas com funções bem definidas, evitando uma estrutura complexa, com diversos componentes sem ligação lógica. Esse padrão propiciou também o isolamento de mudanças em cada camada, o que se torna essencial no ambiente acadêmico de desenvolvimento de sistemas para a segmentação de imagens.

Do mesmo modo, outros padrões foram empregados para definir a arquitetura do framework, como por exemplo, o padrão *Strategy* que simplificou a forma pela qual os desenvolvedores escolhem os componentes. O padrão *Façade* foi importante à medida que possibilitou criar um mecanismo para instanciação dos algoritmos de segmentação de imagens, implementados pelos desenvolvedores, sem obrigá-los a conhecer todos os detalhes de construção do framework.

O padrão *Observer* auxiliou na implementação de um método que permitiu informar o componente **exibirImagem** que uma imagem, resultado da aplicação do algoritmo de segmentação, precisa ser mostrada para o usuário. Outro padrão de projeto importante foi o *Command* o qual pode-se implementar uma solução elegante para o tratamento de requisições dos usuários realizadas via interface gráfica.

Portanto, os diversos padrões que foram utilizados no processo de desenvolvimento do framework Athena possibilitaram o reuso de código, da arquitetura e do fluxo de controle das aplicações envolvendo segmentação de imagens. Não obstante, a aplicação desses padrões permitiu a criação de um framework de forma funcional, garantido a legibilidade, o reuso de código e a facilidade de extensão do framework.

Referências

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., and Stal, M. (1996). *Pattern - oriented software architecture: a system of patterns*. John Wiley & Sons Ltd, New York.

Facon, J. (2001). *Processamento e Análise de Imagens*. Curso de Mestrado em Informática Aplicada. Pontifícia Universidade Católica, Paraná.

Fayad, M., Schmidt, D., and Johnson, R. (1999). *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. Wiley Computer Publishing, New York.

Freitas, G. D. (2006). *Athena: um framework para a construção de interfaces humano-computador no domínio da segmentação de imagens*. Dissertação de mestrado, Programa de Pós-Graduação em Engenharia de Produção(PPGEP)- Universidade Federal de Santa Maria.

Gamma, E., Johnson, R., Helm, R., and Vlissides, J. (2000). *Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos*. Bookman, Porto Alegre.

Gonzales, R. C. and Woods, R. E. (2000). *Processamento de Imagens Digitais*. Edgard Blücher, São Paulo.

Johnson, R. E. (1997). Components, frameworks, patterns. *ACM Special Interest Group on Software Engineering*, pages 10–17.

Miranda, A. N. (2004). *QuantiPhase: um programa de processamento e análise de imagens para a caracterização da composição e homogeneidade de materiais*. Trabalho de conclusão de curso, Curso de Ciência da Computação- Universidade Federal de Santa Maria.

Nielsen, J. (1993). *Usability Engineering*. Academic Press, New York.

Russ (1999). *The Image Processing Handbook*. CRC Press LLC, São Paulo.

Schmidt, D. C., Gokhale, A., and Natarajan, B. (2004). Leveraging application frameworks. *ACM Press. Vol. 2, Issue 5*, pages 66–75.

Tidwell, J. (2006). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly's . Disponível em: http://designinginterfaces.com/.

Welfer, D. (2005). *Padrões de Projeto no desenvolvimento de sistemas de processamento de imagens*. Dissertação de mestrado, Programa de Pós-Graduação em Engenharia de Produção(PPGEP)- Universidade Federal de Santa Maria.

Uma proposta de ambiente para apoiar a utilização de padrões de software e requisitos de teste no desenvolvimento de aplicações

Alessandra Chan[1]*, Maria I. Cagnin[2], José C. Maldonado[1], Rosana T. V. Braga[1]†

[1]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
Caixa Postal 668 – 13560-970 – São Carlos – SP – Brasil

[2]Programa de Pós-Graduação em Ciência da Computação
Centro Universitário Eurípides de Marília
Caixa Postal 2041 – 17525-901 – Marília – SP – Brasil

alechan@icmc.usp.br, istela@univem.edu.br,

jcmaldon@icmc.usp.br, rtvb@icmc.usp.br

Resumo. *O emprego de padrões de software, requisitos de teste, métodos e processos de desenvolvimento na criação de aplicações pode aumentar a produtividade das equipes e a qualidade do produto final. No entanto, há carência por ferramentas que apóiem engenheiros de software no emprego de padrões de software nas diversas etapas de um processo de desenvolvimento e que auxiliem na validação das soluções utilizadas. Assim, este artigo apresenta a proposta de um ambiente Web para desenvolvimento apoiado por padrões e requisitos de teste, com enfoque nos seus requisitos, arquitetura e aspectos de implementação. Durante o desenvolvimento de software, o ambiente apresenta ao engenheiro de software sugestões de padrões que solucionam problemas específicos de cada etapa do processo, juntamente com requisitos de teste para auxiliar a validação das aplicações desenvolvidas com o apoio do ambiente.*

Abstract. *The use of software patterns, test requirements, methods and development processes in the creation of applications can increase teams productivity and the final product quality. However, there is a lack for tools supporting users on the use of software patterns in the many stages of a development process, beyond assisting the validation of the reused solutions. Thus, this article presents a proposal of a Web environment for development supported by patterns and test requirements, with focus on its requirements, architecture, and implementation aspects. During software development, the environment presents to the software engineer suggestions of patterns that resolve specific problems at each process stage. For each pattern, the environment also offers test requirements for assisting the validation of applications developed with the environment support.*

*Apoio financeiro do CNPq
†Apoio financeiro da Fapesp

1. Introdução

Padrões de software constituem uma técnica eficaz de reúso, mas utilizá-los em projetos de desenvolvimento requer um certo custo, principalmente por causa da sua complexidade. Além disso, existe uma grande quantidade de padrões na literatura, utilizando normas distintas de nomenclatura e definição, dificultando a consulta pelo padrão adequado a ser empregado [Pressman 2005]. Assim, é necessário que o projetista possua um conhecimento de diversos padrões para que utilize os mais adequados na solução de problemas.

Atualmente, há uma preocupação com teste dos artefatos resultantes de desenvolvimento com reúso. A atividade de VV&T (Verificação, Validação e Teste) é uma das principais para a garantia de qualidade, minimizando erros e riscos associados ao desenvolvimento [Rocha et al. 2001]. No entanto, essa atividade é uma das mais onerosas da Engenharia de Software [Maldonado et al. 2004, Myers 2004, Pressman 2005] e, no mundo competitivo atual, cresce a importância do desenvolvimento de software de alta qualidade, com preços acessíveis e em tempo reduzido [Chan 2005]. Nesse contexto, uma das dificuldades encontradas é a avaliação da qualidade do padrão que se deseja utilizar, pois há poucos indícios de como ele foi validado e de como validar a solução utilizada. Uma das opções para resolver esse problema é adicionar uma seção no padrão para auxiliar os engenheiros de software a validar a solução por ele proposta [Cagnin et al. 2005].

Outro ponto a ser considerado são os problemas que ocorrem durante o desenvolvimento e manutenção de aplicações, que podem ser evitados por meio da utilização de processos de desenvolvimento disciplinados, métodos, técnicas e ferramentas, além de também colaborarem para construção de aplicações com maior qualidade.

Nesse contexto, ferramentas automatizadas têm um papel importante para que métodos possam ser empregados corretamente, além de apoiar e agilizar o reúso de padrões de software. O teste de aplicações também pode ser auxiliado pela utilização de ferramentas, permitindo maior rapidez e confiança na produção de dados sobre a execução.

Atualmente, ferramentas auxiliam engenheiros de software de diversas maneiras, como por exemplo, para apoiar a implementação, modelagem de diagramas, controle de versão, consulta a padrões de software e teste de software. No entanto, não foram encontrados na literatura ambientes e ferramentas que apoiassem a utilização de padrões durante as diversas etapas de um processo de desenvolvimento, além de apoiar a associação de diretrizes de teste para facilitar a validação das soluções reusadas. Assim, neste artigo é proposto o desenvolvimento de um ambiente que apóie a consulta e aplicação de padrões de software em cada etapa do processo de desenvolvimento de software, documentando o uso dos padrões nas várias atividades, além de informar aos engenheiros de software os requisitos de teste necessários para a validação do padrão utilizado.

Esta seção contém uma breve introdução sobre o contexto e motivação do trabalho proposto. As demais seções deste artigo estão organizadas nos seguintes tópicos: a Seção 2 resume os conceitos básicos que envolvem o ambiente proposto, a Seção 3 cita alguns trabalhos relacionados sobre ferramentas e ambientes encontrados atualmente na literatura, a Seção 4 apresenta a proposta do ambiente e a Seção 5 apresenta as conclusões sobre o assunto tratado e os trabalhos futuros.

2. Conceitos Básicos

Esta seção apresenta um resumo dos conceitos básicos para o entendimento dos temas que envolvem o trabalho proposto. Os seguintes tópicos são abordados: definição de padrões de software e descrição de elementos obrigatórios em sua composição (Seção 2.1), diferença entre processo e método de desenvolvimento (Seção 2.2), descrição de terminologias utilizadas dentro das atividades de VV&T e o conceito de requisito de teste considerado neste trabalho (Seção 2.3).

2.1. Padrões de Software

No final da década de 70, Christopher Alexander introduziu na área da arquitetura as primeiras definições sobre padrões e linguagem de padrões, além de descrever o seu método de documentação [Alexander 1977, Alexander 1979]. Posteriormente, na década de 80, surgiram os primeiros padrões na área de software, com o intuito de captar a estrutura essencial e o raciocínio de uma família de soluções bem sucedidas e comprovadas para um problema recorrente que ocorre dentro de um certo contexto [Appleton 2000]. Além de identificar a solução para um problema, padrões também devem explicar o porquê da necessidade da solução [Appleton 2000].

Padrões têm sido documentados em diferentes formatos [Braga 2003]. No entanto, algumas informações são consideradas essenciais para diferenciar um padrão de uma descrição qualquer de um par "problema/solução", permitindo a sua busca e utilização correta [Meszaros and Doble 1996]. Segundo o padrão *"Presença de Elemento Obrigatório"* (*Mandatory Element Present*), da linguagem de padrões de Meszaros e Doble (1996), os seguintes elementos são obrigatórios em um padrão: "Nome do Padrão", "Problema", "Solução", "Contexto" e "Forças".

Padrões de Software podem trazer benefícios tanto às áreas ligadas diretamente ao projeto e implementação, quanto às áreas de outras disciplinas que fornecem suporte ao desenvolvimento de sistemas. Entretanto, se mal utilizados também podem trazer desvantagens como, por exemplo, a perda de eficiência causada pela adição de classes ou de novas camadas da aplicação e a diminuição da legibilidade e da manutenibilidade por causa do aumento da complexidade do código com a divisão de classes, mensagens, linhas de código e níveis hierárquicos de classes [Santos 2004].

Em razão da grande quantidade de padrões de software encontrados na literatura, é necessário atentar para sua qualidade. Uma das maneiras de selecionar bons padrões é verificar se possuem os elementos obrigatórios e se foram utilizados em pelo menos três aplicações. Além disso, uma outra forma do usuário selecionar e verificar a qualidade de um padrão é adicionar uma seção especial descrevendo como proceder para validar não somente o padrão como também as aplicações criadas a partir dele, conforme sugerido por Cagnin et al. (2005).

2.2. Processos e Métodos de Desenvolvimento

Software é um produto complexo, difícil de desenvolver e testar. Freqüentemente, um software pode apresentar comportamentos inesperados e indesejados, podendo causar sérios problemas e perdas. Assim, pesquisadores têm se esforçado para aumentar a qualidade do software. Uma das hipóteses é que há uma relação direta entre a qualidade do processo e a qualidade do software desenvolvido [Fuggetta 2000].

Um processo de desenvolvimento de software é o conjunto coerente de políticas, estruturas organizacionais, tecnologias, procedimentos, atividades e artefatos que são necessários para entender, desenvolver, implantar e manter um produto de software [Fuggetta 2000, Segrini et al. 2006].

No entanto, não é trivial decidir o que deve ser incluído em um processo de desenvolvimento, pois devem ser consideradas as características da equipe de desenvolvimento e do projeto, o nível de conhecimento em Engenharia de Software, os propósitos da organização, o orçamento disponível, entre outros fatores. Assim, cada organização deve definir os seus processos e melhorá-los constantemente, de acordo com a experiência adquirida durante os projetos [Segrini et al. 2006]. Rocha et al. (2001) consideram três etapas na definição de uma abordagem flexível de processos: definição de processos padrão para a organização, especialização dos processos padrão e instanciação para projetos específicos.

Processos de desenvolvimento, como os propostos por Pressman (2005) e por Murugesan e Ginige (2005), não definem um método para a sua utilização [Bianchini 2005]. Segundo o dicionário [Houaiss 2006], método é o *"procedimento, técnica ou meio de se fazer alguma coisa"*. Um método ensina como desenvolver um software, utilizando como base um conjunto de princípios básicos da Engenharia de Software que abrangem princípios de cada área da tecnologia, incluindo atividades de modelagem e outras técnicas descritivas [Pressman 2005].

Para o emprego correto de processos e métodos, ferramentas e ambientes podem fornecer apoio automatizado ou semi-automatizado. Diversos estudos podem ser encontrados na literatura atual e alguns deles são citados na Seção 3.

Também merecem destaque os estudos voltados para *Web* (*World Wide Web*), pois as suas aplicações têm se tornado muito importantes no mundo de negócios globalizado atual. Assim, também é crescente a preocupação por processos e métodos para desenvolvimento *Web*. No caso de aplicações *Web*, para gerenciar corretamente o desenvolvimento e manutenção, é necessário utilizar técnicas e princípios tradicionais de Engenharia de Software combinados com tratamento dos aspectos específicos da *Web* [Brambilla et al. 2002, Bianchini 2005]. Com relaçao aos métodos de desenvolvimento para *Web*, podem ser encontrados atualmente vários estudos [Koch 2000, Ceri et al. 2000, Conallen 2002].

No contexto do trabalho de Bianchini (2005), esses métodos estão sendo estudados e avaliados para identificar qual o mais apropriado no desenvolvimento de aplicações para a *Web*. Considerando o estado atual do trabalho, o processo de desenvolvimento utilizado por Conallen (2002), pode ser considerado apropriado no desenvolvimento de aplicações *Web* e tem sido empregado em estudos de caso para confirmar essa adequação. O processo proposto por Conallen (2002) é baseado no RUP (*Rational Unified Process*) [Kruchten 2000] e propõe a utilização do WAE (*Web Application Extension*), que define estereótipos para a representação mais clara dos elementos e regras de negócio das aplicações para a *Web*.

Também é importante destacar que, em razão da grande quantidade de padrões de software existentes na literatura atual, é difícil para o usuário identificar o mais adequado a ser utilizado em uma etapa de um processo de desenvolvimento. Repositórios de padrões

podem ser encontrados [Marinho et al. 2003, Bolchini et al. 2002], mas é necessário que o usuário tenha um conhecimento prévio da existência do padrão para saber em qual momento utilizá-lo.

Assim, com relação à apresentação de soluções existentes, um ambiente que, seguindo um processo de desenvolvimento disciplinado, apóie o usuário na organização e exibição de padrões de software previamente cadastrados, pode auxiliar no emprego correto dessas soluções comprovadas, minimizando esforços e melhorando a qualidade do produto de software.

2.3. Requisitos de Teste

Um software deve ser previsível e consistente sem oferecer surpresa aos seus usuários. [Myers 2004]. Mesmo que o processo de desenvolvimento de software utilize uma série de técnicas, métodos e ferramentas, erros no produto ainda podem ocorrer. Assim, um conjunto de atividades, denominadas de Garantia de Qualidade de Software, são introduzidas durante todo o processo de desenvolvimento de software, destacando-se as atividades de VV&T, que visam minimizar riscos e erros associados. O teste é a atividade mais utilizada nesse contexto e constitui um dos elementos para fornecer evidências da confiabilidade do software [Maldonado 1991, Maldonado et al. 2004].

Dentro da terminologia utilizada nas atividades de VV&T, os seguintes termos são diferenciados: defeito, engano, erro e falha. O *defeito* corresponde ao passo, processo ou definição de dados incorretos. O *engano* é a ação humana que produz um resultado incorreto no programa. O *erro* é a diferença entre o dado obtido e o dado esperado. A *falha* é a produção de uma saída diferente da exigida na especificação [Maldonado et al. 2004].

Na atividade de teste é realizada uma análise dinâmica do produto, sendo utilizada para a identificação de erros e eliminação de falhas e defeitos [Maldonado et al. 2004]. Para localizar a maior quantidade possível de falhas e defeitos, testes devem ser conduzidos de maneira sistemática e casos de testes devem ser projetados utilizando técnicas disciplinadas [Pressman 2005].

Segundo Myers (2004), é impraticável e geralmente impossível encontrar todos os erros de um programa. Assim, uma estratégia deve ser estabelecida antes de iniciar os testes [Myers 2004], para que seja coberta adequadamente a lógica do programa e para garantir que as condições do projeto tenham sido cumpridas [Pressman 2005].

Quatro etapas compõem o teste de software: planejamento de teste, projeto de casos de teste, execução e avaliação dos resultados [Maldonado et al. 2004, Myers 2004, Pressman 2005]. Essas etapas devem ser desenvolvidas ao longo do processo de desenvolvimento e geralmente são concretizadas em três fases: teste de unidade, de integração e de sistema [Maldonado et al. 2004].

Casos de teste são criados seguindo os requisitos de teste, que são definidos a partir de critérios de teste, que, por sua vez, são estabelecidos de acordo com as técnicas de teste escolhidas. De acordo com o tipo de informação que se deseja testar, escolhe-se a técnica de teste. Em geral, quatro técnicas são utilizadas: *Teste Funcional*, *Teste Estrutural*, *Teste Baseado em Erros* e *Teste Baseado em Máquinas de Estados Finitos* [Maldonado et al. 2004].

Após escolher a técnica, segue-se um critério de teste para avaliar a adequação do

teste e para estabelecer o conjunto de requisitos de teste que serão utilizados para gerar os casos de teste. Requisitos de teste contêm a idéia do que deve ser testado, não informando como deve ser realizado esse teste [Wilkinson 2003]. Por fim, casos de teste são criados com os dados de entrada que devem ser informados ao software e a descrição das saídas esperadas [Myers 2004].

A atividade de VV&T é uma das mais onerosas no desenvolvimento de software [Rocha et al. 2001]. A associação de requisitos de teste a padrões de software pode auxiliar engenheiros de software na verificação das soluções propostas e reduzir o tempo despendido na atividade de VV&T [Cagnin et al. 2005]. Outra maneira de reduzir o tempo é o desenvolvimento de ferramentas de automatização, que são importantes no suporte à atividade de teste, propiciando maior qualidade e produtividade [Maldonado et al. 2004].

Assim, um ambiente ou ferramenta que apóie e esclareça o usuário quanto à utilização de requisitos de teste ao empregar um padrão de software pode auxiliar a minimizar omissões no teste de suas aplicações e reduzir o tempo da atividade de VV&T.

3. Trabalhos Relacionados

Como mencionado na Seção 1, ferramentas são utilizadas para fornecer apoio automatizado ou semi-automatizado no emprego de métodos de desenvolvimento *Web*. Além de auxiliar no emprego de métodos, ferramentas podem também ser utilizadas para auxiliar engenheiros de software a empregarem padrões de projeto na criação de suas aplicações [Marinho et al. 2003] e para apoiar a atividade de VV&T. Outro recurso que pode ser utilizado são os ambientes, empregados no auxílio de gerência de processos de desenvolvimento de software. Ferramentas e ambientes minimizam a complexidade no desenvolvimento, evitam erros de usuários inexperientes ao utilizarem os métodos ou os padrões, além de prevenir contra a omissão na verificação, validação e teste de aplicações. Alguns trabalhos sobre ambientes e ferramentas podem ser citados, sendo que cada um deles pode fornecer auxílio na área de processo de desenvolvimento de software, padrões de software ou requisitos de teste.

Para apoiar e acompanhar o emprego de processos de desenvolvimento de software têm-se, por exemplo, o ambiente WebAPSEE (*Web Process-Centered Software Engineering Environments*) [Lima et al. 2006] e a ferramenta ODE (*Ontology-based software Development Environment*) [Segrini et al. 2006]. O WebAPSEE é utilizado para auxiliar na modelagem e manutenção de processos, enquanto a ODE é empregada para a definição de processos de ODE [Falbo et al. 2004].

Diversas ferramentas têm sido construídas para fornecer apoio no emprego de padrões de software. Dentre os trabalhos, podemos citar: o repositório de padrões proposto por Marinho et al. (2003) que é integrado ao RUP; o repositório de padrões de projeto para hipermídia e aplicações *Web*, denominado HPR (*Hypermedia Design Patterns Repository*) [Bolchini et al. 2002]; o ambiente FRED (*FRamework EDitor*) [Hakala et al. 2001] utilizado no desenvolvimento Java por meio de prototipação; e o *framework* GREN (Gestão de REcursos de Negócio) [Braga 2003] utilizado para auxiliar no desenvolvimento de aplicações utilizando a linguagem de padrões GRN (Gestão de Recursos de Negócio) [Braga 2003].

Para auxiliar na automatização de teste são encontradas na literatura ferramentas como: a Proteste [Price and Zorzo 1990], que fornece apoio ao teste estrutural de progra-

mas em Pascal; a Atac (*Automatic Test Analysis for C*) [Horgan and Mathur 1992], que apóia a aplicação dos critérios estruturais de fluxo de controle e de dados em programas C e C++; a JaBUTi (*Java Bytecode Understanding and Testing*) [Vincenzi et al. 2003], utilizada para o teste de programas Java; a PokeTool (*Potential Uses Criteria Tool for Program Testing*) [Chaim 1991], que apóia a aplicação dos critérios de teste Todos-Nós, Todos-Arcos e Potenciais-Usos [Maldonado 1991]; e a Proteum [Delamaro 1993] que apóia o teste de mutação para programas desenvolvidos na linguagem C.

Apesar das inúmeras ferramentas e ambientes encontrados atualmente para auxiliar individualmente o emprego de processos de desenvolvimento de software, ou a aplicação de padrões de projeto ou o apoio ao teste de software, não foram encontradas ferramentas ou ambientes que auxiliassem engenheiros de software a acompanhar a execução dos seus projetos apoiando a aplicação de padrões nas diversas etapas de um processo de desenvolvimento de software. Além disso, ainda não existem ferramentas ou ambientes que incluam uma seção para auxiliar na validação de padrões, uma vez que a proposta é recente.

Atualmente existem aplicações *Web* para acompanhamento de projetos, mas não foram encontradas aplicações *Web* que apresentem ao usuário os padrões de software mais adequados para serem empregados na execução de uma etapa do processo de desenvolvimento. Em geral, repositórios, como o HPR, apenas servem como uma base para consulta sendo necessário que, ao deparar-se com um problema, o usuário lembre da existência do padrão para poder consultá-lo.

4. Proposta do Ambiente

Nesta seção descreve-se a proposta de um ambiente *Web* de apoio ao uso de padrões de software e requisitos de teste durante o processo de desenvolvimento de software [Chan 2005]. Esta seção está dividida nos seguintes tópicos: a Seção 4.1 apresenta os requisitos do ambiente, a Seção 4.2 explica a arquitetura adotada no desenvolvimento do ambiente e a Seção 4.3 contém a descrição dos aspectos de implementação considerados no trabalho proposto.

4.1. Requisitos do Ambiente

O ambiente proposto é destinado a engenheiros de software que desejam acompanhar o andamento do projeto de suas aplicações utilizando padrões de software durante o processo de desenvolvimento e reusando requisitos de teste associados a esses padrões de software. Resumidamente, o ambiente possui duas funções principais: cadastro de processos de desenvolvimento (e suas respectivas fases, atividades, artefatos, padrões e requisitos de testes) e uso desses processos em projetos de desenvolvimento. Assim, o usuário tem no ambiente o suporte necessário para definir os processos de desenvolvimento da organização e utilizá-los em projetos concretos. O desenvolvimento de aplicações *Web* pode ser realizado utilizando o ambiente, contanto que seja cadastrado e utilizado um processo que trate das peculiaridades desse tipo de aplicações. Além de permitir o desenvolvimento para a *Web*, o ambiente é uma aplicação *Web* permitindo que usuários compartilhem padrões de software, melhorando a qualidade de seus projetos.

Como pode ser observado no modelo conceitual do ambiente (Figura 1), existem dois tipos de usuários: o engenheiro de software e o administrador. O *engenheiro de*

software é capaz de executar as funcionalidades de: cadastro, gerência, instanciação e acompanhamento da execução de processos de desenvolvimento; criação e gerência de projetos; criação, gerência e instanciação de padrões de software; criação e gerência de requisitos de teste; criação e gerência de diagramas de classes e de todos os elementos necessários na construção desses diagramas; geração de arquivos XMI; impressão de relatórios; e busca dos elementos cadastrados no ambiente. Além de executar todas as funcionalidades disponibilizadas para um engenheiro de software, o *administrador* é capaz de cadastrar, visualizar e gerenciar as contas dos usuários e é o único capaz de alterar as informações de um processo cadastrado no "Repositório do Ambiente", que é descrito com mais detalhes na Seção 4.3.

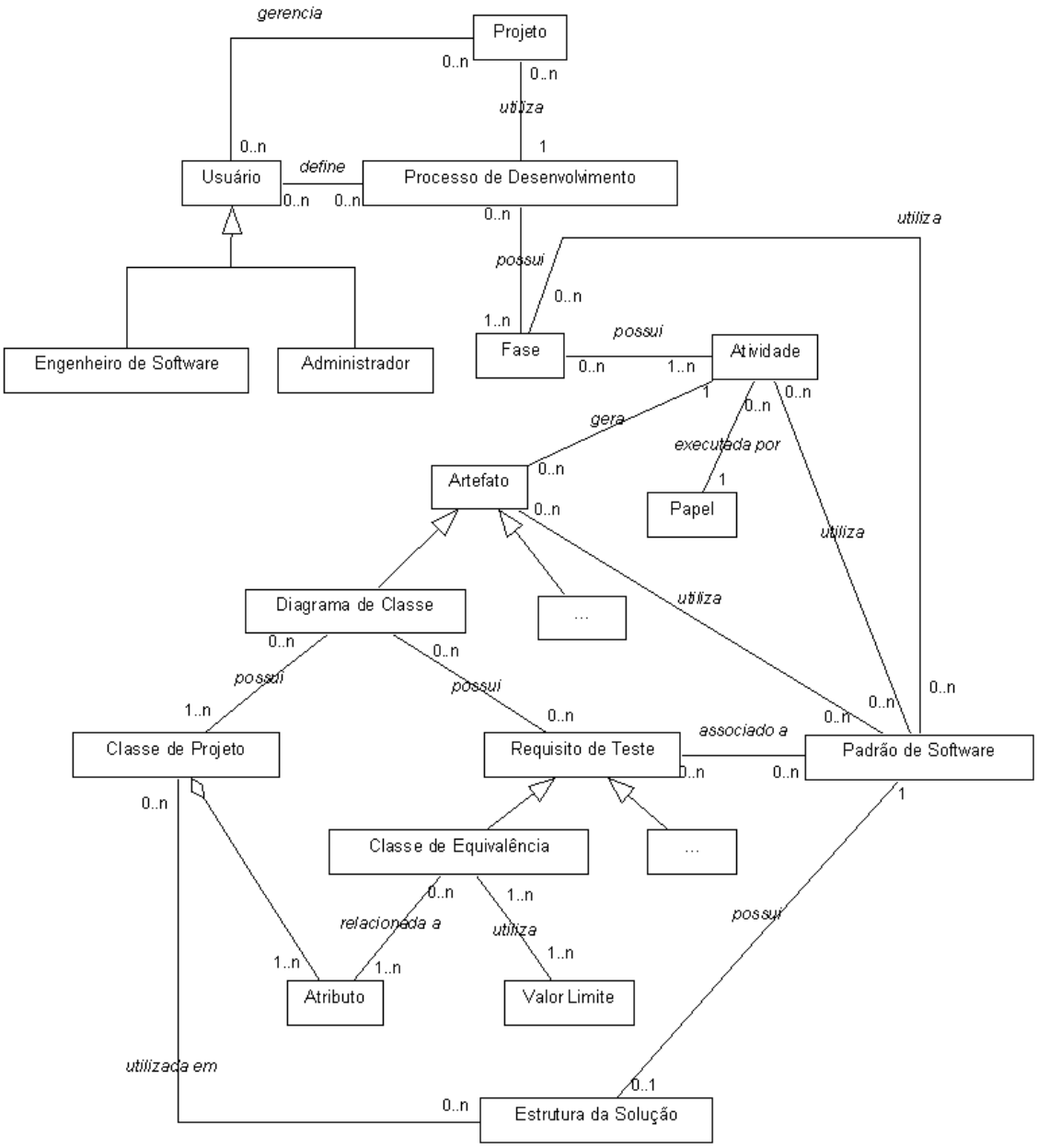


Figura 1. Modelo Conceitual do Ambiente.

Como mencionado na Seção 2.2, uma abordagem flexível sobre processos envolve três etapas: a definição, especialização e instanciação do processo. Assim, o ambiente proposto apóia o cadastro de *processos de desenvolvimento*, permitindo que os usuários escolham seus elementos e informem a obrigatoriedade de cada um deles, dando a liberdade de definir e especializar de acordo com suas necessidades. Também fornece apoio à instanciação de processos, por meio de projetos a eles associados, permitindo o acompanhamento da execução das fases e atividades para a produção de artefatos necessários ao projeto. Deve-se observar que no modelo conceitual da Figura 1 não estão incluídos os conceitos pertinentes à instanciação de processos.

Com relação à estrutura do processo de desenvolvimento, ela aborda elementos básicos do RUP, no entanto, não são utilizados todos eles para que o usuário tenha uma maior liberdade na definição dos seus processos de desenvolvimento. Cada processo é composto de *fases* que, por sua vez, são compostas de atividades. A execução de uma atividade é atribuída a uma pessoa que desempenha um *papel* no desenvolvimento do produto. Uma *atividade* descreve os procedimentos de como o trabalho deve ser realizado e pode receber artefatos de entrada e produzir artefatos de saída. Um *artefato* é um produto de trabalho gerado com a execução de uma atividade, por exemplo, modelos, elementos de modelo, código-fonte e documentos [Conallen 2002].

O ambiente proposto auxilia na produção de artefatos que são utilizados para identificar a execução das atividades e, conseqüentemente, das fases de um processo de desenvolvimento. Inicialmente, o apoio é fornecido para a criação de diagramas de classes, sendo acrescentado incrementalmente o auxílio ao desenvolvimento de outros artefatos. Também pode ser utilizado para armazenar informações a respeito dos artefatos produzidos, como por exemplo, onde estão localizados e os padrões de software utilizados, conforme discutido na Seção 4.3.

Padrões de software são cadastrados no ambiente e associados a uma fase ou atividade do processo de desenvolvimento. Assim, quando o usuário estiver em uma determinada fase ou atividade dentro de um projeto que siga tal processo, uma lista de padrões de software é sugerida a ele, permitindo que visualize quais padrões podem ser aplicados naquele momento e registrando no ambiente o uso desse padrão. Pode-se também atribuir a cada padrão uma identificação do domínio ao qual pertence, para facilitar o uso desse padrão em projetos de um domínio em particular.

A representação conceitual de um padrão de software por meio de classes de projeto e associações é realizada na *estrutura da solução*. Para serem utilizados nos diagramas de classes criados com o apoio do ambiente, padrões de software precisam ser instanciados, ou seja, o usuário deve informar os nomes dos elementos que compõem a estrutura da solução do padrão no contexto do projeto. Na Tabela 1 é apresentada a descrição do caso de uso "Instanciar Padrão de Software", incluindo a descrição dos passos para executar essa operação. O ambiente permite que mais de um padrão seja instanciado em uma fase ou atividade, já que padrões (de projeto por exemplo) podem ser usados em conjunto para solucionar um problema.

Seguindo a proposta feita por Cagnin et al. (2005), os padrões de software cadastrados no ambiente possuem uma seção especial com descrição de requisitos de teste que podem ser utilizados para validar esse padrão e, conseqüentemente, a aplicação sendo

Tabela 1. Caso de Uso "Instanciar Padrão de Software"

Caso de Uso: Instanciar Padrão de Software	
Atores Principais: Administrador ou Engenheiro de Software	
Interessados e Interesses: Engenheiro de Software: deseja criar uma instância de um padrão de software para ser utilizado em um diagrama de classes. A instanciação é realizada para que os elementos de um padrão de software sejam renomeados adequadamente no contexto do projeto.	
Pré-Condição: Administrador ou Engenheiro de Software autenticado no ambiente.	
Pós-Condição: Uma instância de um padrão de software é criada e associada a um diagrama de classes.	
Fluxo Básico	
Ator	**Sistema**
1. O usuário está preenchendo o formulário de inserção de itens em um diagrama de classes, clica na opção de "Procurar" do campo "Padrão de Software", seleciona um padrão de software da lista apresentada e clica no botão "Ok".	**2.** O ambiente verifica se foi selecionado um dos padrões de software da lista.
	3. O ambiente exibe a página de instanciação de padrão de software com o formulário que deve ser preenchido. São apresentadas todas as classes da estrutura da solução e para cada uma delas é apresentado o campo "Nome da Classe Instanciada". São apresentados todos os relacionamentos da estrutura da solução e para cada um deles é apresentado o campo "Nome do Relacionamento". Também são exibidos dois botões, sendo eles "Salvar" e "Cancelar".
4. O usuário preenche os campos do formulário de cadastro.	
5. O usuário clica no botão "Salvar" para enviar as informações.	**6.** O ambiente verifica se todos os campos "Nome da Classe Instanciada" foram preenchidos.
	7. O ambiente salva as informações da instância do padrão de software.
	8. O ambiente exibe a página anterior, ou seja, o diagrama de classes que estava sendo alterado pelo usuário.

desenvolvida. Esses requisitos de teste podem ser reusados em conjunto com os padrões de software, ou seja, podem ser reaproveitados pelo usuário na elaboração e execução da atividade de VV&T.

Classes de projeto são cadastradas previamente para poderem ser empregadas nos diagramas de classes e nas estruturas da solução de padrões de software. O usuário pode criar requisitos de teste para validar os *atributos* de uma classe de projeto contida em uma estrutura da solução de um padrão. Em sua primeira versão, o ambiente permite o uso somente de *classes de equivalência* e *valor limite*, mas outros tipos de requisitos de teste podem ser incluídos com a evolução do ambiente. Caso o usuário opte por utilizar um outro ambiente ou ferramenta para continuar o desenvolvimento de sua aplicação, é fornecida a opção de exportação dos diagramas de classes produzidos pelo ambiente proposto, sendo armazenados em um arquivo no padrão XMI (*XML Metadata Interchange*) [World Wide Web Consortium 2005]. Assim, o usuário pode importar para essas outras ferramentas ou ambientes o arquivo XMI criado. Também é permitido que o usuário importe para o ambiente proposto os diagramas de classes no formato XMI.

4.2. A Arquitetura

Na Figura 2 é apresentada a arquitetura do ambiente proposto. É utilizada a arquitetura de três camadas e o padrão arquitetural MVC (*Model-View-Controller*) [Krasner and Pope 1988]. Na **Camada de Apresentação**, a **Visão** é responsável por armazenar as informações enviadas pelo **Navegador do Cliente** e enviá-las para o **Controlador**, que por sua vez, é responsável por processar as informações recebidas, transformando-as em solicitações para a **Camada de Aplicação**. Na **Camada de Aplicação**, o **Modelo** realiza o processamento da lógica do negócio, enviando os dados para a **Camada de Persistência**, ou requisitando informações dela. A **Camada de Persistência** realiza a busca por dados e o armazenamento de informações no **Banco de Dados** do ambiente.

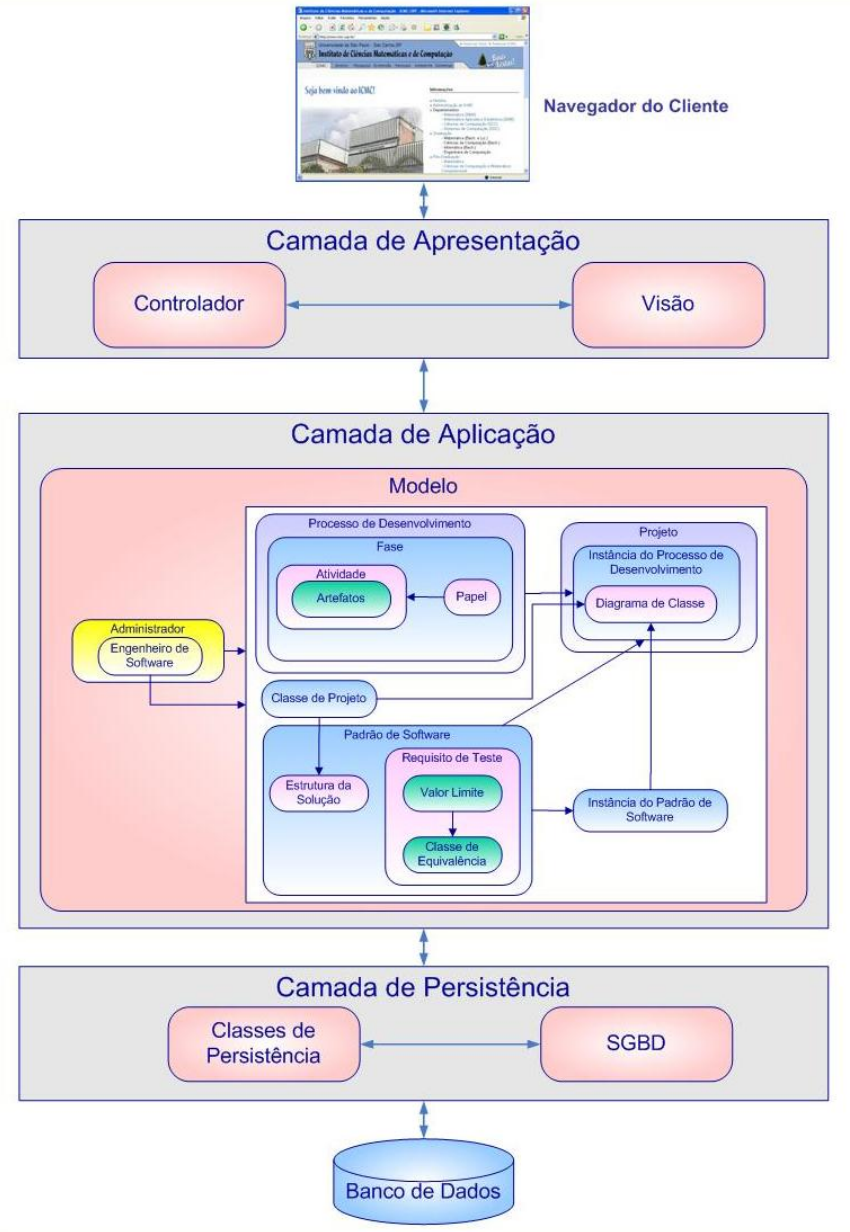


Figura 2. Modelo da arquitetura do ambiente.

4.3. Aspectos de Implementação

O ambiente proposto neste artigo é uma aplicação para a *Web* e está sendo implementado em Java [Sun Microsystems, Inc. 1999], por ser uma linguagem de programação orientada a objetos, simples, robusta, interpretada, portável, distribuída, de arquitetura neutra, segura, de alto desempenho, *multithreaded* e dinâmica [Sun Microsystems, Inc. 1999].

Por ser uma aplicação *Web*, o processo de desenvolvimento proposto por Conallen (2002) está sendo utilizado para o projeto do ambiente. Para permitir que aplicações *Web* sejam modeladas nos diagramas de classes criados pelo ambiente, os estereótipos propostos pelo WAE [Conallen 2002], que é uma extensão da UML (*Unified Modeling Language*) [OMG's 2006], estão sendo disponibilizados para a modelagem das características específicas desse tipo de aplicações.

Com relação ao armazenamento de processos de desenvolvimento no ambiente, para que um usuário tenha a flexibilidade de escolher entre manter restrito o acesso aos processos por ele criados ou permitir o compartilhamento com os outros usuários do ambiente, dois tipos de repositórios de processos de desenvolvimento são oferecidos: o "Repositório do Ambiente" e o "Repositório do Usuário". No primeiro, todos os usuários do ambiente podem visualizar e utilizar os processos nele cadastrados, no entanto, não podem alterá-los. No segundo, apenas os donos do repositório podem utilizar e alterar os processos nele cadastrados.

Como mencionado na Seção 4.1, o ambiente auxilia na produção de artefatos. Como o ambiente está sendo desenvolvido utilizando uma abordagem incremental, a princípio, esse auxílio restringe-se apenas à criação de diagramas de classes, sendo que outros tipos de artefatos podem ser criados separadamente e informados ao ambiente por meio do nome do arquivo gerado e do caminho para recuperá-lo. Futuramente, outros tipos de apoio ao desenvolvimento de artefatos podem ser incorporados ao ambiente.

À medida que os artefatos são produzidos ou informados, o ambiente considera as atividades e, conseqüentemente, as fases como executadas para que o usuário possa acompanhar a evolução do seu projeto. É oferecida uma opção para que o usuário informe que o projeto foi finalizado. Após informar a finalização, o ambiente automaticamente realiza a verificação das fases e atividades marcadas como executadas. Caso alguma fase ou atividade obrigatória não tenha sido marcada como executada, o ambiente alerta o usuário sobre a impossibilidade de finalizar o projeto. No entanto, se apenas fases ou atividades opcionais não tiverem sido executadas, o ambiente apenas avisa sobre quais delas não foram executadas e sobre o sucesso na finalização do projeto.

Como mencionado na Seção 2.1, alguns elementos são considerados obrigatórios em um padrão. Assim, o ambiente permite, para cada padrão de sofware, o cadastrado dos atributos: "Nome do Padrão", "Problema", "Solução", "Contexto", "Força", "Estrutura da Solução", "Requisito de Teste", "Exemplo", "Contexto Resultante", "Raciocínio", "Padrões Relacionados", "Usos Conhecidos" e "Etapa do Processo de Desenvolvimento". É por meio do atributo "Etapa do Processo de Desenvolvimento" que o ambiente reconhece quais padrões sugerir para o usuário em uma fase ou atividade.

O campo "Estrutura da Solução" é opcional e seu conteúdo é empregado na instanciação do padrão, permitindo a criação de partes do diagrama de classes. Por exemplo, se uma das fases do processo for a "Modelagem do Sistema" e uma de suas atividades

for a “Criação do Diagrama de Classes”, ao empregar um padrão nessa atividade, a estrutura de sua solução pode ser importada, instanciada e as classes passam a fazer parte do diagrama de classes. Se outros padrões também forem aplicáveis nessa mesma atividade, eles podem ser instanciados e as novas classes devem ser incorporadas ao diagrama de classes final.

Caso o campo “Estrutura da Solução” não tenha sido informado pelo engenheiro de software, o padrão não pode ser instanciado em detalhes, sendo utilizado apenas nas fases e atividades que não envolvem a criação de diagrama de classes. Mesmo não sendo utilizado na construção de artefatos, é importante informar os padrões utilizados nas fases e atividades para que sejam documentados no ambiente, permitindo o acompanhamento da evolução do projeto em conjunto com o emprego dos padrões e fornecendo a base para futuras análises estatísticas que podem auxiliar os engenheiros de software no desenvolvimento de outros projetos.

O campo “Requisito de Teste” permite ao usuário informar os requisitos de teste para auxiliar na validação dos padrões de software cadastrados no ambiente. Considerando que a estratégia de teste utilizada pelo usuário depende de diversos fatores, como por exemplo, custos e cronogramas, não está sendo considerada uma abordagem específica para a criação dos requisitos de teste para a validação dos padrões de software, ou seja, o usuário tem a liberdade na elaboração e associação dos requisitos de teste. Devido ao caráter incremental no desenvolvimento do ambiente, em um primeiro momento, o suporte ao cadastro de requistos de teste deve ser realizado apenas para os critérios Análise do Valor Limite e Particionamento de Equivalência.

Destaca-se que cada Classe de Equivalência é considerada como um requisito de teste no ambiente. Assim, o usuário pode utilizar apenas o critério Particionamento de Equivalência, mas não pode utilizar somente o critério Análise do Valor Limite. Podem ser cadastrados valores limites, mas para que possam ser visualizados como requisitos de teste, devem ser geradas as classes de equivalência desses valores. A funcionalidade para gerar automaticamente classes de equivalência a partir de valores limites é oferecida pelo ambiente.

5. Conclusão

Processos e métodos de desenvolvimento, padrões de software, ferramentas e ambientes têm o objetivo comum de apoiar engenheiros de software no desenvolvimento de aplicações. Explorando o suporte comum a esses temas, o ambiente proposto neste artigo têm por objetivo fornecer flexibilidade para o usuário cadastrar processos de desenvolvimento e acompanhar a sua execução, sugerir automaticamente padrões de software para serem empregados nas fases e atividades do processo de desenvolvimento escolhido pelo usuário, além de também apoiar a atividade de VV&T por oferecer requisitos de teste para validar os padrões cadastrados no ambiente.

Com o aumento da quantidade de padrões existentes, cresce também a dificuldade na visualização e escolha dos padrões mais adequados a serem empregados em um projeto. Muitas vezes, engenheiros de software sequer recordam a existência de um padrão de software e da solução por ele proposta. Assim, ao sugerir automaticamente os padrões cadastrados e relacionados a uma fase ou atividade de um processo de desenvolvimento, o ambiente pode minimizar as chances do usuário deixar de utilizar um padrão de software

por não lembrar a existência da solução, além de auxiliá-lo na visualização das soluções existentes. No entanto, um ponto a ser considerado é a qualidade dos padrões armazenados no ambiente. Futuramente planeja-se estabelecer um filtro no cadastro ou um critério de remoção para que apenas padrões de software válidos sejam mantidos e repassados entre os usuários, evitando que soluções inválidas comprometam os projetos.

Outro problema é a falha na validação de padrões de software. Na maioria das vezes, quem utiliza os padrões não são as pessoas que os desenvolveram. Assim, é importante para o engenheiro de software ter diretrizes de como testar a solução proposta pelo padrão. Seguindo a sugestão de Cagnin et al. (2005), o ambiente proposto permite a associação de requisitos de teste a padrões de software para auxiliar usuários na validação dos padrões de software utilizados, podendo minimizar o tempo despendido na atividade de VV&T.

Uma vez que os padrões de software tenham sido acrescentados ao ambiente, novos processos de desenvolvimento incorporados ao ambiente também podem fazer uso dos padrões, já que ao incluir um processo de desenvolvimento é possível associar os padrões existentes a cada uma de suas fases ou atividades. Além disso, havendo requisitos de teste associados ao padrão, esses requisitos são automaticamente válidos no contexto do novo processo.

O ambiente está sendo desenvolvido de maneira incremental, fornecendo apoio, em um primeiro momento, à construção de diagramas de classes e ao cadastro de requistos de teste utilizando os critérios Análise do Valor Limite e Particionamento de Equivalência. Assim, para as fases e atividades que não envolvem a construção de diagramas de classes, o usuário pode utilizar o ambiente para acompanhar e controlar o emprego de padrões. No entanto, em trabalhos futuros planeja-se adicionar apoio à construção de outros artefatos para que o suporte ao emprego de padrões torne-se mais efetivo gradativamente. Também espera-se que, além de apoio ao cadastro a outros critérios de teste, o ambiente também forneça um suporte maior à validação de padrões de software utilizados no desenvolvimento de aplicações, como por exemplo, a criação de casos de teste.

Referências

Alexander, C. (1977). *A Pattern Language*. Oxford University Press.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Appleton, B. (2000). Patterns and software: Essential concepts and terminology. Online.

Bianchini, S. L. (2005). Avaliação de Metodologias de Desenvolvimento de Sistemas *Web*. Master's thesis, ICMC/USP, São Carlos/SP - Brasil. em andamento.

Bolchini, D., Garzotto, F., Paolini, P., Lowe, D., Cantoni, L., Nanard, J., Rossi, G., Schwabe, D., and Ruggeri, R. (2002). HPR - Hypermedia Design Patterns Repository. Online.

Braga, R. T. V. (2003). *Um Processo para Construção e Instanciação de Frameworks Baseados em uma Linguagem de Padrões para um Domínio Específico*. PhD thesis, ICMC/USP, São Carlos/SP - Brasil.

Brambilla, M., Comai, S., and Fraternali, P. (2002). Hypertext Semantics For Web Applications. In *SEBD Italian National Conference on DataBase Systems*, Portoferraio - Italy.

Cagnin, M. I., Braga, R. T. V., Germano, F., Chan, A., and Maldonado, J. C. (2005). Extending Patterns with Testing Implementation. In *SugarLoafPlop'2005, V Confer-*

encia Latino-Americana em Linguagens de Padrões para Programação, Campos do Jordão/SP - Brasil. Submetido.

Ceri, S., Fraternali, P., and Bongio, A. (2000). Web modeling language (WebML): a modeling language for designing web sites. In *9th International World Wide Web Conference*, pages 1–22, Amsterdam.

Chaim, M. L. (1991). Poke-tool: Uma Ferramenta Para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados. Master's thesis, DCA/FEEC/UNICAMP, Campinas/SP - Brasil.

Chan, A. (2005). Um Ambiente de Apoio ao Uso de Padrões de Software e Requisitos de Teste no Desenvolvimento de Aplicações *Web*. Master's thesis, ICMC/USP, São Carlos/SP - Brasil. em andamento.

Conallen, J. (2002). *Buildind Web Applications with UML*. Addison-Wesley, 2nd. edition.

Delamaro, M. E. (1993). Proteum - Um ambiente de teste baseado na analise de mutantes. Master's thesis, ICMC/USP, São Carlos/SP - Brasil.

Falbo, R. A., Ruy, F. B., Pezzin, J., and Moro, R. D. (2004). Ontologias e Ambientes de Desenvolvimento de Software Semânticos. In *JIISIC'04 - IV Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento*, Madri - Espanha.

Fuggetta, A. (2000). Software Process: A Roadmap. In *ICSE'00 - Future of Software Engineering Track*, pages 25–34, Limerick - Ireland.

Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A., and Viljamaa, J. (2001). Architecture-Oriented Programming Using FRED. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 823–824, Washington/DC - USA. IEEE Computer Society.

Horgan, J. R. and Mathur, A. P. (1992). Assessing Testing Tools in Research and Education. *IEEE Software*, 9(3):61–69.

Houaiss, A. (2006). Dicionário Houaiss da Língua Portuguesa. Online.

Koch, N. (2000). *Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process*. PhD thesis, Ludwig-Maximilians University, Munich - Germany.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model view controller user interface paradigm in Smalltalk-80. In *Journal of Object-Orientated Programming*, volume 1, pages 26–49.

Kruchten, P. (2000). *The Rational Unified Process: An Introduction*. Addison-Wesley, 2th. edition. 298 p.

Lima, A., Costa, A., França, B., Reis, C. A. L., and Reis, R. Q. (2006). Gerência Flexível de Processos de Software com o Ambiente WebAPSEE. In *SBES'06 - XIII Sessão de Ferramentas do SBES*, pages 97–102, Florianópolis/SC - Brasil.

Maldonado, J. C. (1991). *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. PhD thesis, DCA/FEE/UNICAMP, Campinas/SP - Brasil.

Maldonado, J. C., Barbosa, E. F., Vincenzi, A. M. R., and Márcio Eduardo Delamaro, Simone Rocio Senger Souza, M. J. (2004). Introdução ao Teste de Software. Nota Didática.

Marinho, F., Santos, M., Pinto, R. N., and Andrade, R. (2003). Uma Proposta de um Repositório de Padrões de Software Integrado ao RUP. In *SugarLoafPlop Proceeding 2003, The Third Latin American Conference on Pattern Languages of Programming*, pages 277–290, Porto de Galinhas/PE - Brasil.

Meszaros, G. and Doble, J. (1996). MetaPatterns: A Pattern Language for Pattern Writing. In *PLoP'1996 - Proceedings of the 8th Pattern Languages of Programs Conference*, Monticello/Illinois - USA.

Myers, G. J. (2004). *The art of software testing*. John Wiley & Sons, Inc., 2th. edition.

OMG's (2006). UML Resource Page. Online.

Pressman, R. S. (2005). *Engenharia de Software*. McGraw-Hill, 6th. edition.

Price, A. M. and Zorzo, A. (1990). Visualizando o Fluxo de Controle de Programas. In *SBES'1990 - IV Simpósio Brasileiro de Engenharia de Software*, Águas de São Pedro/SP - Brasil.

Rocha, A. R. C., Maldonado, J. C., and K, C. W. (2001). *Qualidade de Software: Teoria e Prática*. Prentice Hall, 1th. edition.

Santos, M. S. (2004). Uma Proposta para a Integração de Modelos de Padrões de Software com Ferramentas de Apoio ao Desenvolvimento de Sistemas. Master's thesis, UFC, Fortaleza/CE - Brasil.

Segrini, B. M., Bertollo, G., and Falbo, R. A. (2006). Evoluindo a Definição de Processos de Software em ODE. In *SBES'06 - XIII Sessão de Ferramentas do SBES*, pages 109–114, Florianópolis/SC - Brasil.

Sun Microsystems, Inc. (1999). The Java Language: An Overview. Online.

Vincenzi, A. M. R., Wong, W. E., Delamaro, M. E., and Maldonado, J. C. (2003). JaBUTi: A Coverage Analysis Tool for Java Programs. In *Sessão de Ferramentas do 17º Simpósio Brasileiro de Engenharia de Software*, Manaus, AM, Brasil.

Wilkinson, G. (2003). Tests Without Specs. Professional Tester Magazine.

World Wide Web Consortium (2005). Extensible Markup Language Metadata Interchange (XMI). Online.

A Process to Create Analysis Pattern Languages for Specific Domains*

Rosana T. V. Braga[1], **Reginaldo Ré**[2], **Paulo Cesar Masiero**[1]

[1]Instituto de Ciências Matemáticas e de Computação – ICMC
Universidade de São Paulo - USP
Caixa Postal 668 – 13.560-970 – São Carlos – SP – Brazil

[2]Universidade Tecnológica Federal do Paraná
Campus Campo Mourão
Caixa Postal 271 – 87301-005 – Campo Mourão – PR – Brazil

rtvb@icmc.usp.br, reginaldo@utfpr.edu.br, masiero@icmc.usp.br

Abstract. *Pattern languages are a powerful instrument through which knowledge about a specific domain can be documented. When composed of analysis patterns, they can help novice developers to model systems belonging to a wide variety of applications in a particular domain. We show a process to systematically produce a pattern language for a specific domain. The process starts with the identification of the domain functionality, then patterns are created to solve individual problems found in the domain, and finally relationships among patterns are established. Two analysis pattern languages created by the authors are used to illustrate the proposed process.*

1. Introduction

Software patterns document proven solutions for common problems that occur during software development [Gamma et al. 1995], so that they can be reused by inexperienced developers when facing the same problems. The grouping of patterns into a pattern language improves reuse even more, as they can lead to the design of complete applications [Brugali and Menga 1999]. A pattern language is a structured collection of patterns that support each other to transform requirements and restrictions into an architecture [Coplien 1998]. A pattern language represents the temporal sequence of decisions that lead to the complete design of an application, so it can become a method to guide the development process [Brugali et al. 2000].

In this work, we are particularly interested in using pattern languages to help novices to model specific-domain systems. So, the patterns of a pattern language can be used as a guide through which he can find the solutions to each problem he faces when modeling an application in a certain domain. Each pattern solves a problem and results in a context that is used as input to other patterns of the pattern language. But how are pattern languages developed? How are the patterns defined in a way that they can be easily applied when modeling applications in a specific domain?

Our research group has developed three pattern languages [Braga et al. 1999, Ré et al. 2001, Pazin et al. 2004], after studying many others available in the literature [Alexander et al. 1977, Aarsten et al. 2000, Beck and Cunningham 1987,

*Financial support from FAPESP

Brown and Whitenack 1996, Cunningham 1995, Meszaros and Doble 1998]. During the submission of our pattern languages for publishing, a common question among reviewers was about how we have created the pattern language. In fact, there is not much written about this process, as reported in the Related Work Section. The development of a pattern language requires a deep knowledge about the domain. Nevertheless, a systematic process could be very useful to achieve a meaningful pattern language.

Thus, considering the absence of a well-defined and detailed process to create pattern languages, we found important to describe our process, so that other domain experts can use it to document their knowledge about particular domains in the form of a pattern language. We consider that, if we have pattern languages for as many domains as possible, software development will be eased, as developers will have a starting point for modeling their applications. Furthermore, to leverage reuse to lower abstraction levels, frameworks can be built based on pattern languages, as proposed in another work of our research group [Braga and Masiero 2002b], in such a way that the framework and the corresponding pattern language are used together to produce domain-specific applications [Braga and Masiero 2002c]. Also, tools can be developed to help the instantiation of this framework based only in the knowledge about the pattern language [Braga and Masiero 2003, Pazin 2004, Shimabukuro et al. 2006], allowing the development to concentrate in the system requirements instead of design and implementation details.

This paper describes a process for creating a domain-specific pattern language, composed of analysis patterns that can be further used to model concrete applications in the same domain. Section 2 summarizes related work regarding writing pattern languages. Section 3 gives an overview of the process. In Section 4 we describe the first step of the process, which aims at producing a domain class model. In Section 5 we show how to define an initial set of patterns, that will be refined to produce the pattern language. In Section 6 we present the process for creating a pattern language graph, to show the order in which the patterns are used and their interaction. In Section 7 we give details of how to describe each pattern individually. In Section 8 we show how to validate the pattern language. Finally, in Section 9, we make our concluding remarks.

2. Related Work

The work of Meszaros [Meszaros and Doble 1998] provides useful guidelines for pattern writing, including several problems and solutions for pattern language writing, but the focus is on the patterns format and disposition of the patterns throughout the pattern language, i.e., nothing is mentioned about how to discover the patterns based on the knowledge about a particular domain, or how to organize them or to delimit their scope.

Cunningham [Cunningham 1994] provides several hints about the sequence of activities to perform when conceiving a pattern language, based on his experience in writing the CHECKS pattern language [Cunningham 1995]. Though short, his hints are easy to understand and helped to guide us in the creation of our process.

Buschmann and others [Buschmann et al. 1996] describe a process for pattern mining that focuses on the creation of individual patterns that will be possibly joined to form a pattern system. Although a pattern system ties its constituent patterns together, it does not have the compromise of being complete like a pattern language, which needs to

have at least one pattern available for every aspect of the construction and implementation of software systems, with no gaps or blanks [Buschmann et al. 1996]. Although they do not present a process for pattern language creation, they stress the importance of having complete pattern languages to cover substantial part of the design space of the respective domains. The rules of thumb that they present to mine new patterns are often applicable when describing each pattern of a pattern language. For example, "find at least three examples where a particular recurring design or implementation problem is solved effectively by using the same solution schema" is a guideline that can help finding candidate patterns to form the pattern language.

3. Process overview

Analysis pattern languages for specific domains could be straightfully developed, without the need of an application domain class model. However, a domain class model is useful during the development of a pattern language, as the patterns represent a well-succeeded structure of solutions for problems in the same context [Riehle and Zullighoven 1996]. Our process presents a set of activities that, from a domain class model, obtained through the first phase of our general process, aims at creating a pattern language. The activities to be conducted for the creation of the pattern language, the resources needed to execute them, and the results obtained for each activity are presented in Figure 1.

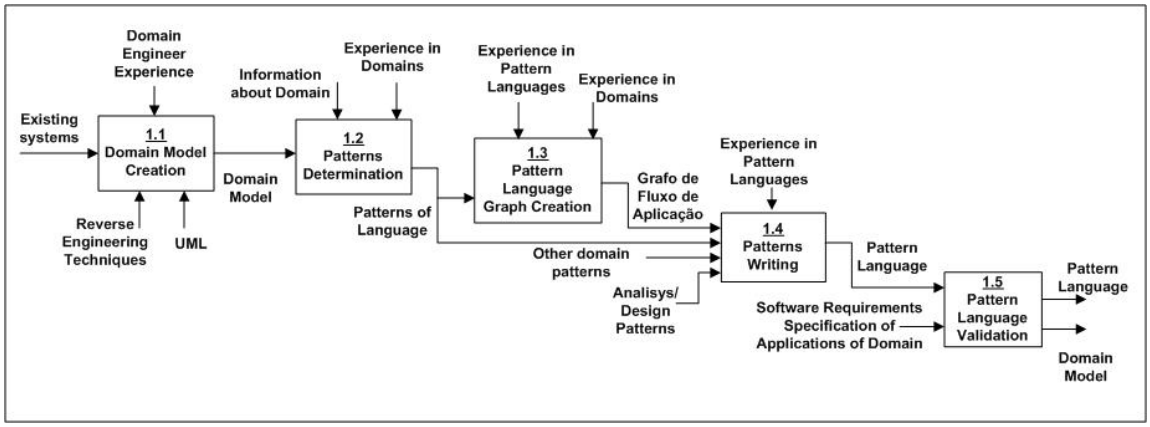


Figure 1. Pattern Language Creation Process

4. Starting point: a domain analysis model

Patterns are usually documented based on software development practice. Consequently, to build a pattern language that covers applications in a certain domain, it is necessary to observe the solutions that are commonly employed to solve recurring problems in that domain. Thus, the starting point for creating a pattern language is to obtain a model for the target domain, i.e., a model that captures the functionality present in the majority of applications in that domain (step 1.1 of Figure 1). Independently of how the information about the domain is obtained, our process states that both static and dynamic models should be produced at the end of this step. The static model can be expressed using an object oriented notation, as for example a UML [Rational 2000] (Unified Modeling Language) class diagram, showing only classes, attributes, and methods that are common to all applications in the domain. Generic names should be assigned to these classes, attributes and

methods, and they should be properly explained in the respective domain glossary. Relationship among classes also need generic names that reflect semantic aspects inherent to the domain. The dynamic model can be expressed, for example, by UML sequence diagrams, which show the communication between objects to implement system operations. Again, they should be defined with generic names. UML use case diagrams can also be created to show dynamic aspects of the domain, focusing on the behavior that is common to all applications.

The target domain model can be obtained by several means: a domain analysis can be conducted, using techniques such as those described by Gomaa [Gomaa 1996] or those selected by Prieto-Diaz [Prieto-Diaz and Arango 1991]; a reverse engineering can be done in existing applications of the domain, similarly to what has been done by Ré et. al. [Ré et al. 2001] for the on-line auctions domain; or the practical experience in the development of applications in a particular domain can be used, as occurred during the creation of the GRN pattern language [Braga et al. 1999]. In the last two cases, experience about a domain can be obtained by building or reverse engineering several systems in the target domain, at least three as recommended by Roberts and Johnson [Roberts and Johnson 1998], obtaining intermediate models that represent each of the three systems. These different models of applications in the same domain can then be generalized to produce a domain analysis model [Ré et al. 2001]. They can be compared with each other and, if a certain element is present in the three systems, there is a high probability that it will be part of the domain model. As a matter of fact, a difficult decision needs to be made at this point by the domain engineer, about whether or not each element is common to the domain. This decision involves other factors, as for example the personal experience of the domain engineer and specific goals to be achieved with the resulting domain model.

Tools that provide automated mechanisms to reverse engineer legacy systems could be useful in this step. More than obtaining models of the system in higher abstraction levels, these tools could also help finding existing patterns in code. However, this work did not consider the aid of these tools, so all the work was manually done. This issue could be target of future work.

To illustrate this step, we consider the on-line auction domain, for which we have created a pattern language [Ré et al. 2001]. Three existing systems were (manually) reverse engineered: Arremate.com[1], iBazar[2], and eBay[3]. These systems were active when the reverse engineering was conducted, but nowadays iBazar has been incorporated to eBay. Intermediate models with dozens of classes were obtained for each of them. Figure 2 illustrates part of the domain analysis model obtained at the end of this step, with seventeen classes representing the main functionalities of an auction. The complete model contains forty classes.

Notice that the domain model contains, besides entities with their attributes, methods, and relationships, more abstract operations that denote the behavior inherent to an entity, which are useful to better understand the domain concepts. Operations are more than methods, as they reflect how system events are treated by the software (probably

[1] `http://www.arremate.com`
[2] `http://www.ibazar.com.br`
[3] `http://www.ebay.com`

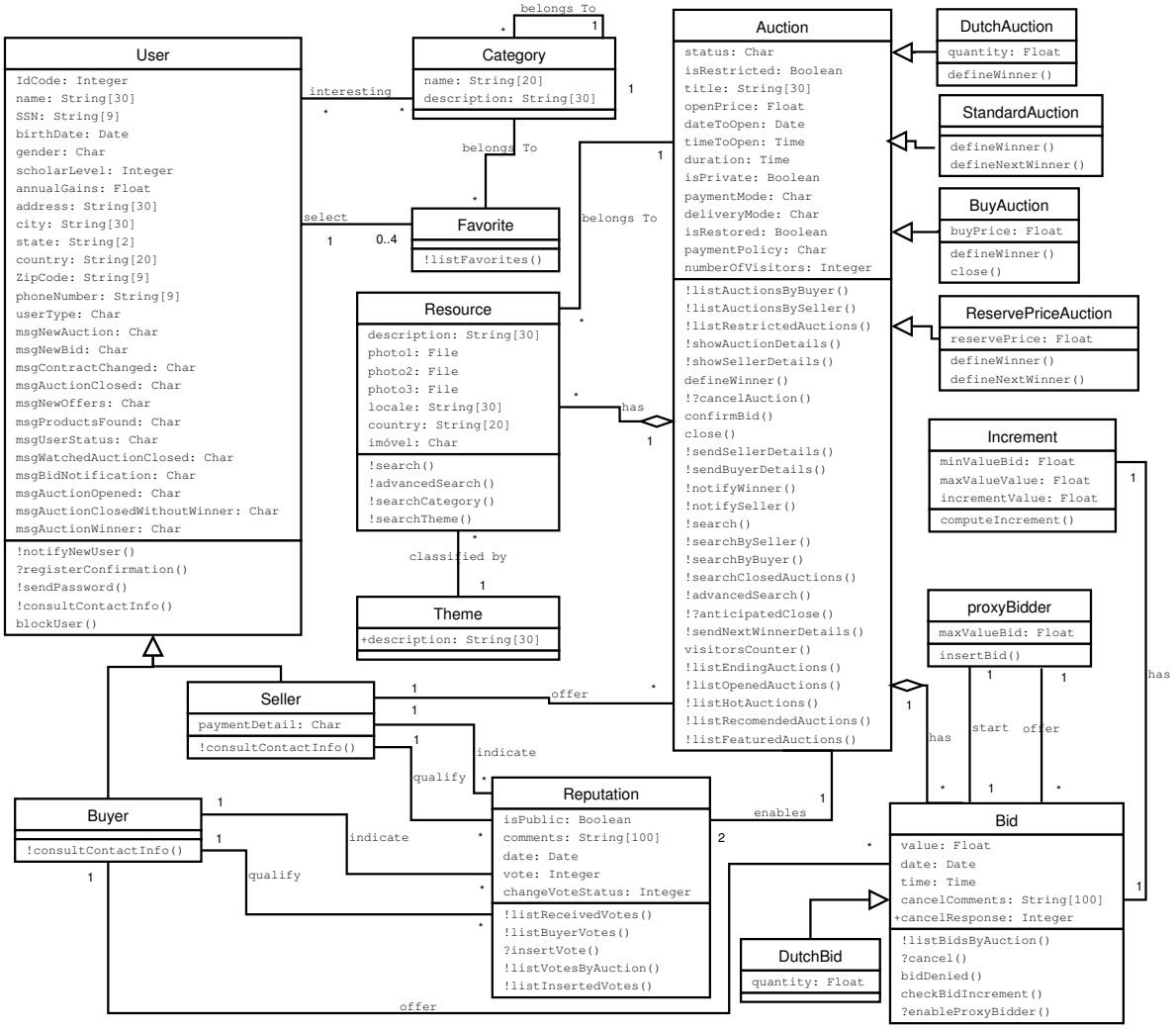


Figure 2. Partial Domain Model for On-line Auctions

invoking more than one method of several different classes). In our notation, extended from UML, we use special characters before operation names to denote certain types of behavior. For example, we use the interrogation (?) and exclamation (!) marks to denote input and output operations, respectively.

5. Partitioning the domain model into a initial list of patterns

The domain analysis model resulting from the previous step is used as basis for identifying the patterns that compose the pattern language (step 1.2 of Figure 1). This activity is often dependent on the knowledge and experience about patterns possessed by the pattern language developer. However, some guidelines should be followed so that the patterns are defined in a uniform way and with higher possibility of being reused:

1. Existing patterns in the literature should be analyzed, as some of them are likely to be present in the domain analysis model. Pattern repositories should be searched, specially with automatic tools, to ease this task. When a pattern is found, the problem solved by it should be specialized to the specific domain, originating a new pattern that should be assigned a name reflecting the domain-specific

problem. Durign the creation of the pattern language for the online auction [Ré et al. 2001], several patterns were found in the literature to represent items to be dealt with by the application, as for example the patterns ITEM DESCRIPTION[Coad et al. 1997] and TYPE-OBJECT [Johnson and Woolf 1998]. So, these patterns were specialized to the resource being auctioned and originated the first pattern, which was named IDENTIFY THE RESOURCE.

2. Other analysis pattern languages, for similar domains, should be studied and their patterns should be observed, e.g., their documentation and relationship. This contributes to enhance the knowledge about patterns and improves the chance of developing more correct and reusable patterns. At this point, a format for the patterns can be chosen or, at least, two or three possible formats can be identified. In the Online Auction (OA) example, the GRN pattern language [Braga et al. 1999] was used as basis for formatting the patterns.
3. The pattern definition should begin by identifying the basic classes of the domain model obtained in the previous step. Basic classes are those involved in basic or main system functions represented in the domain model, i.e., those that are present in all systems belonging to the domain. This concept is equivalent to the "core types" defined by Cheesman and Daniels [Cheesman and Daniels 2001] or to the frozen spots of a framework [Buschmann et al. 1996]. For example, in the OA domain model of Figure 2, classes Buyer, Seller, Resource, Auction, and Bid are basic, as they appear in any instances of this domain. Other classes present in the domain model are complementary classes, as they can appear in certain systems but not in others.
4. Basic classes identified using the previous guideline should be studied in order to discover groups of two or more classes that are responsible for important system functions. This can be done based on the domain class model itself, for example by highlighting them with a different color or creating smaller models relative to each function. For example, in Figure 2 we can group classes around Bid (DutchBid, ProxyBidder and Increment), as they all refer to behavior regarding bids. It must be observed that this is an incremental process, so later on it could be changed if necessary. It must also be noticed that the classes belonging to a group are not necessarily all basic classes, as will be explained in the next guideline. These groups of classes will represent the main patterns of the pattern language, as long as each pattern should refer to a specific function performed in the domain. This improves reuse, because patterns with short, well defined, and focused problem/solution pairs are created.
5. Differently from basic classes, complementary classes add improvements to a pattern, or add a different function not present in the domain model. Complementary classes usually represent functionalities that are optional for the correct functioning of systems in the studied domain. So, they are more likely to become optional patterns, i.e., patterns that can be applied or not when modeling a particular system, or they can be joined to existing patterns to form pattern variants, so that they are considered as optional pattern elements. Again, the pattern language developer has to decide how to establish which classes make a pattern. The possibility of creating optional patterns allows them to be ignored during the usage of the pattern language, in case the functionality they offer is not necessary in the particular application. For example, in the OA domain, the Favorite class is a complemen-

tary class, as it does not appear in all auctions that were investigated in the reverse engineering. So, the pattern language author has two alternatives: he could create a separate pattern to include this behavior, or he could add an optional element in an existing pattern (even if it is a mandatory pattern). In our case, we have chosen to create a separate pattern, named ENABLE FAVORITE. It should be highlighted that the concepts of basic and complementary classes are equivalent to the concepts of mandatory and optional features in domain analysis, specially for product lines engineering.

6. Each pattern should be named - a task that can be eased by observing its functions. This name is important, as it abstracts the pattern content and allow its identification and usage by other analysts. Meszaros and Doble [Meszaros and Doble 1998] suggest several conventions for pattern naming, as for example, to use an evocative pattern name, a noun phrase name, or a meaningful metaphor name. In our example, we have chosen to name patterns with phrases.
7. After identifying and naming patterns, a table can be constructed containing the pattern name, the problem solved by the pattern, and a summary of the proposed solution, , as suggested by Meszaros and Doble [Meszaros and Doble 1998]. This table helps the pattern language developer to keep consistency and to follow each pattern goal during the remaining steps of the process. See the initial list for the OA patterns in Table 1.

In summary, the resulting artifacts for this step are a list of patterns, together with a general description for each of them and information about the classes that compose them. These results will be used in the remaining steps to write the individual patterns.

6. The pattern language graph

In step 1.3 a graph is defined to show the patterns interaction or the patterns application flow. Basically, the graph contains pattern names and the order in which they can be applied, showing also which patterns are mandatory or optional. This information can be obtained in the summary table produced in the previous step. The graph has to show how patterns are disposed and how they are applied to obtain the class model for a specific application. The order in which patterns are applied is usually also shown as a special pattern component, named “Next Patterns”, which is defined during the pattern writing (step 1.4).

It is important to notice that the graph shows only the interaction among patterns, i.e., the order in which patterns are applied during the modeling of applications, but it is not intended to show how the resulting system works, i.e., it is not a flowchart. The graph presents a coherent way in which to apply or operate the patterns to achieve the desired solution.

Besides being influenced by the optional patterns, the patterns application flow is also influenced by other pattern elements, such as their variants, optional classes, and elements. Cases may occur in which: a) the application of a certain pattern implies the inclusion of certain elements in other patterns; b) the application of a pattern requires the application of another pattern; c) one pattern should be chosen among several patterns; d) the application of a pattern excludes the application of another pattern; or e) a pattern can only be applied if another pattern has been previously applied. For example, in Figure 2,

Table 1. Summary of the pattern language

Pattern	Problem
IDENTIFY THE RESOURCE (1)	How do you represent the business resources auctioned by the system?
ENABLE FAVORITE (2)	How does your application allow that resource categories of more interest be established for the customers?
AUCTION THE RESOURCE (3)	How do you handle the different types of resource auctions performed by your application?
PROMOTE NEGOTIATION (4)	How does your application support the negotiation among auction trading parties?
HANDLE BIDS (5)	How does your application deal with the different types of bids related to the several types of auction?
MANAGE THE AUCTION HOUSE (6)	How does your system manage the rules followed by the auction house involved in the auctioning process?
MANAGE REPUTATION (7)	How can your application provide subsidies for the parties to evaluate each other trustability?
HANDLE REFUNDING (8)	How can your application provide ways to refund fees that were unduly charged?
ENABLE MESSAGES (9)	How does your application manage the messages sent to customers?
HANDLE ADVERTISEMENT (10)	How does your application manage auction advertising?

the Dutch Bid class belongs to a variant of pattern HANDLE BIDS, and it can only be used if the Dutch Auction variant of pattern AUCTION THE RESOURCE has been used. So, a dependency can exist among patterns and/or pattern variants application.

A strategy that can be followed when determining the patterns application order is to start with patterns that represent the most basic domain functionalities and gradually add patterns that represent more specific functionalities. Finally, the optional patterns are added, because they represent problems not always present in domain applications. However, the best place to place an optional pattern should be carefully analyzed, as it often depends on the application of a mandatory pattern or should be applied immediately after it. As mentioned before, a mandatory pattern can have optional elements, and thus cases can occur in which the inclusion of an optional element of a mandatory pattern can also imply in other further dependencies.

The OA Pattern Language graph is shown in Figure 3. The main language patterns are split between two categories. The first is composed of required patterns – (1)IDENTIFY THE RESOURCE, (3)AUCTION THE RESOURCE, (5)HANDLE BIDS, (6)MANAGE THE AUCTION HOUSE, (7)MANAGE REPUTATION, (8)HANDLE REFUNDING, (10)HANDLE ADVERTISEMENT – which should be always applied, as they represent the essential requirements of an OA system. The second category is formed by patterns that are only desirable – (2)ENABLE FAVOURITES, (4)PROMOTE NEGOTIATION, (9)ENABLE MESSAGES – but not strictly necessary.

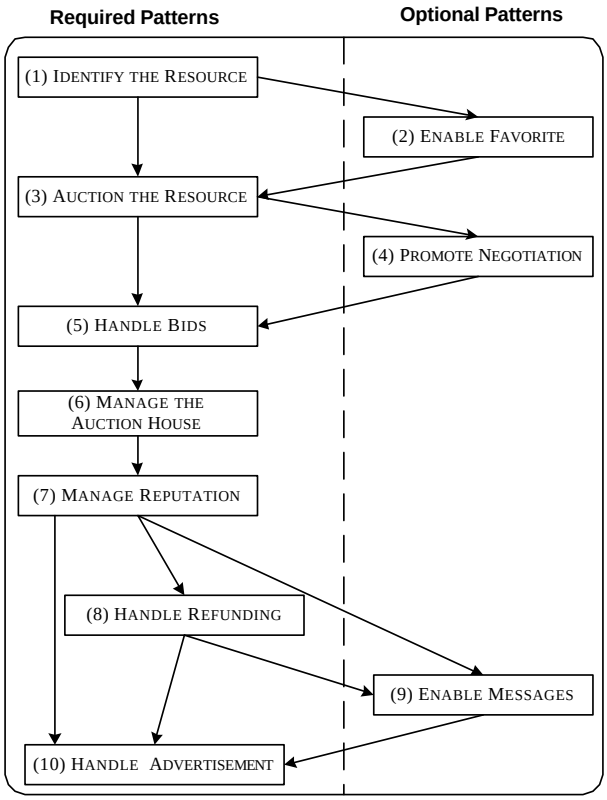


Figure 3. Pattern Language Application Graph

7. Details of each pattern

A pattern is much more than a class structure and its description: it presents all the context information in which it can be applied, the problem it solves, as well as the forces that act to form the solution [Fowler 1997]. So, the pattern writing activity (step 1.4) has to be carefully conducted so that each pattern can be correctly reused. The guidelines proposed by Meszaros [Meszaros and Doble 1998] are very useful to produce well-written patterns, so we recommend their use and reinforce them with some basic recommendations for writing concise patterns.

The pattern language developer needs to establish a format for describing each pattern. However, independently of the format chosen, for analysis patterns there are some elements that are more likely to be present, such as “name”, “context”, “problem”, “structure”, “participants”, “related patterns”, and “next patterns”. The same format should be used for all patterns, although some elements might be optional. There are

several proposals in the literature for structuring patterns, among which we can mention the Alexander format [Alexander et al. 1977], from which it was abstracted the Portland Pattern Form; the Coplien format [Coplien and (eds) 1996]; and the GoF Pattern Form [Gamma et al. 1995].

Having chosen an appropriate format, the pattern writing begins based on the general pattern description and the information about the classes that compose the pattern, obtained in step 1.2. This means that the pattern solution is the first item to be written, as it was already defined in step 1.2 and, thus, it is easier to work with. The problem solved by the pattern was also previously identified, so now it can be written and refined, if necessary. The forces are the next item to be described, based on the problem/solution pair. Considerations about the context in which the solution is applicable are made, trying to raise questions about the many issues that lead to the solution and that could be modified to attend other requirements or different contexts. At this time, pattern variants can be discovered and included in the pattern language.

To improve the pattern description, the developer can search other patterns in the same (or correlate) domain, so that analogies can be done to reuse the experience of other pattern developers. Moreover, during this search the developer can find other patterns for which the pattern being written is a variant, or patterns that complement it or can be joined to other pattern languages. Thus, this searching process is important to ensure that the pattern language references all co-related existing patterns, supplying alternative solutions in case the current pattern is not applicable. The developer can also search for analysis and design patterns that enhance the proposed solutions, complementing the pattern language with references to other patterns or adopting the patterns as a real part of the pattern language.

The documentation of each pattern is a time consuming task, usually demanding several iterations to obtain a satisfactory result. While the patterns are being written in detail, other patterns can be identified, as well as different relationships that might require to alter the pattern graph. The pattern community recommends that every pattern language be submitted to a writers' workshop, for example by submitting it to a PLoP (Pattern Languages of Programs) Conference, where the developer can improve it based on the opinions of other experienced pattern authors. The pattern mining process suggested by Buschmann and others [Buschmann et al. 1996] also contains this specific step of making a writers' workshop.

To illustrate this step we show, in Figure 4, part of a pattern of the OA Pattern Language, which is responsible for handling the different types of online auctions. This pattern has several optional elements, as for example the various types of auction. During the pattern instantiation, the application developer chooses the types that fit the business rules of the specific online auction being developed. Also, there are variants that can be applied if necessary. Notice that there is a "Following Patterns" section to guide the pattern language user during the instantiation process.

8. Validation

The pattern language validation (step 1.5) finishes the process of pattern language creation from a domain class model. The goal is to validate the pattern language through its application to different systems of the domain. Basically, this activity consists of studying

AUCTION THE RESOURCE

Context

Your application deals with resources that have already been identified and categorized. The resource auction may be considered as a property transference, in which a resource owned by a party becomes owned by another party. When a trade is done through an auction, the resource is put on sale by a trading party and several other parties try to buy it for the lowest possible price. There are several types of auctions that provide various options for the trading parties, each with its own rules to define which of the buying parties will be the winner.

Problem

How do you handle the different types of resource auctions performed by your application?

Forces

- Information about the participants must be stored, both to supply the information needed for the trading process and for the system functioning.
- It is important that several auction types be available, observing those that are more appropriate to certain types of resource, the quantity of auctioned resources, efficiency, and restrictions imposed by certain auction types, considering the environment in which the trade occurs: the Internet.

⋮

Therefore:

Create classes to represent the different auction types and distinguish the roles played by participants.

Structure

Figure 1 shows the class diagram for the AUCTION THE RESOURCE pattern.

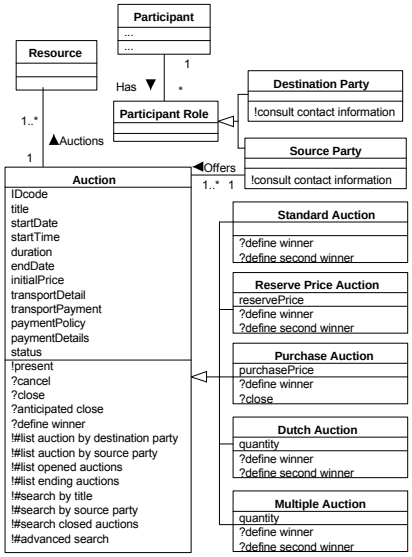


Figure 1: Structure diagram for AUCTION THE RESOURCE

Participants

- *Participant*: Represents the party - organization or person - who intends to auction or acquire a resource. It has two specialized classes: *Source Party* and *Destination Party* (see below). Notice that the same participant can play both roles in different auctions. This is guaranteed by the use of the ROLES pattern. The `statusLogin` attribute indicates whether the participant has supplied its `IDcode` and `password` and, therefore, can effectively participate in the auctions. It is important to notice that passwords need special treatment during design, through a security policy , but we are not considering such issues in this pattern language. This class has some basic attributes, but other attributes may be added, depending on the particular instance of the participant.
- *Participant Role*: Represents the role played by the participant in a specific auction, which can be *Source Party* or *Destination Party* (see below).

⋮

Example

Figure 2 presents an example of the AUCTION THE RESOURCE pattern adopted by the online auction site Arremate.com, which uses `Multiple Auction` for both single and multiple product items. Arremate.com also uses two other auction types: `Reserve Price Auction` and `Winner Auction`, which is an instantiation of the `Purchase Auction`. eBay uses `Dutch Auction` to trade multiple product items and `Standard Auction` to trade only one item. It also provides `Reserve Price Auction` and `Purchase Auction`. iBazar has only `Standard Auction` and `Reserve Price Auction`.

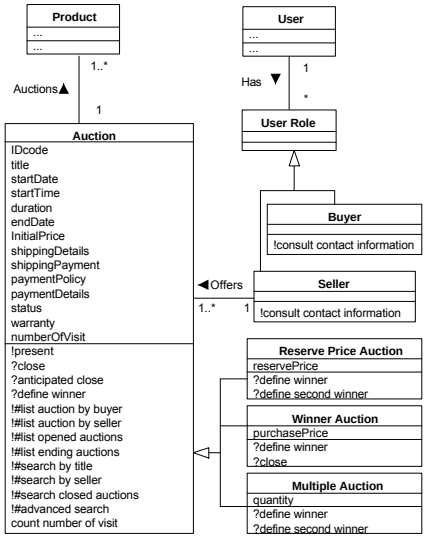


Figure 2: AUCTION THE RESOURCE example

Variants

To make the pattern language useful to different auction types, new classes representing the auction rules may be added as specializations of the `Auction` class. These new auction types may need new attributes and methods.

In some systems `Standard Auction` is replaced by `Multiple Auction`. In this case `Multiple Auction` is used both for auctions of a single item and for several items.

⋮

Related Patterns

This pattern is an application of patterns ASSOCIATION-OBJECT and TIME ASSOCIATION. It is also a combined application of patterns SPECIFIC ITEM-TRANSACTION and PARTICIPANT-TRANSACTION. The different auction types can be implemented using the STRATEGY design pattern.

Following Patterns

After applying the AUCTION THE RESOURCE pattern try to use the PROMOTE NEGOTIATION pattern. If it is not applicable, then use the HANDLE BIDS pattern.

Figure 4. Example of a Pattern in the Online Auction Pattern Language

the requirements documents of applications being modeled, studying and applying the pattern language based on the requirements document, and evaluating the class model, comparing the desirable requirements with the application class model.

An important aspect to be considered here is the fact that other applications, rather

than those used to build the pattern language, must be modeled using the pattern language to enhance its validation and, also, to improve the language itself, as new features can be incorporated to it.

However, it should be clear that a complete validation of the pattern language is a very difficult task, as a great number of applications would have to be modeled to guarantee its uselfulness. Our experience shows that the validation process should try to model a set of applications that use, at least once, each pattern and pattern variant of the pattern language.

Figure 5 shows part of a model obtained through the application of the OA Pattern Language, as a first step for the development of a specific online auction system (Arremate.com). The tags show the roles played by each class in the corresponding patterns. Notice that, in this particular OA system, only reserve price and standard auction are allowed, and almost all patterns were applicable (except pattern #9).

9. Concluding Remarks

In this work we present a systematic process to obtain a pattern language for a specific domain. This pattern language is composed of analysis patterns that can be applied when modeling applications in that domain. It helps novices to model applications, as each pattern contains insights about the problems to solve in the domain, as well as the solutions that better solve these problems.

Representing domain knowledge through pattern languages is an effective way of easing systems modeling. Our research group has conducted an experiment in which students and professionals had to produce the analysis models of information systems using or not a pattern language to support them. The results have shown that better models were produced in less time when using the pattern language, when compared to those groups that did not use it. These experiments are described in detail elsewhere [Braga et al. 2003].

After creating the pattern language, it is easier to develop one or more frameworks that can be used to instantiate the patterns and to create concrete applications. This can be done by following the process proposed in another work [Braga and Masiero 2002b]. A tool to automatically instantiate such framework can also be built [Braga and Masiero 2002a, Braga and Masiero 2003]. The construction of product lines for a specific domain can also be easied using such an approach, as the pattern language could be considered as a domain-specific language used to model concrete members of a product family.

Even though we are aware that a pattern language should be complete, as mentioned in Section 2, we can not guarantee that pattern languages created using our process will have this feature. We provide some conditions for that, such as delimiting the scope of the pattern language and reverse engineering at least three meaningful applications in the domain, so that the minimum common functionalities are available. But, of course, new functionalities can appear after the pattern language is created. Thus, we can say it is complete in a certain moment in time and within a certain scope, but we cannot ensure its everlasting completeness.

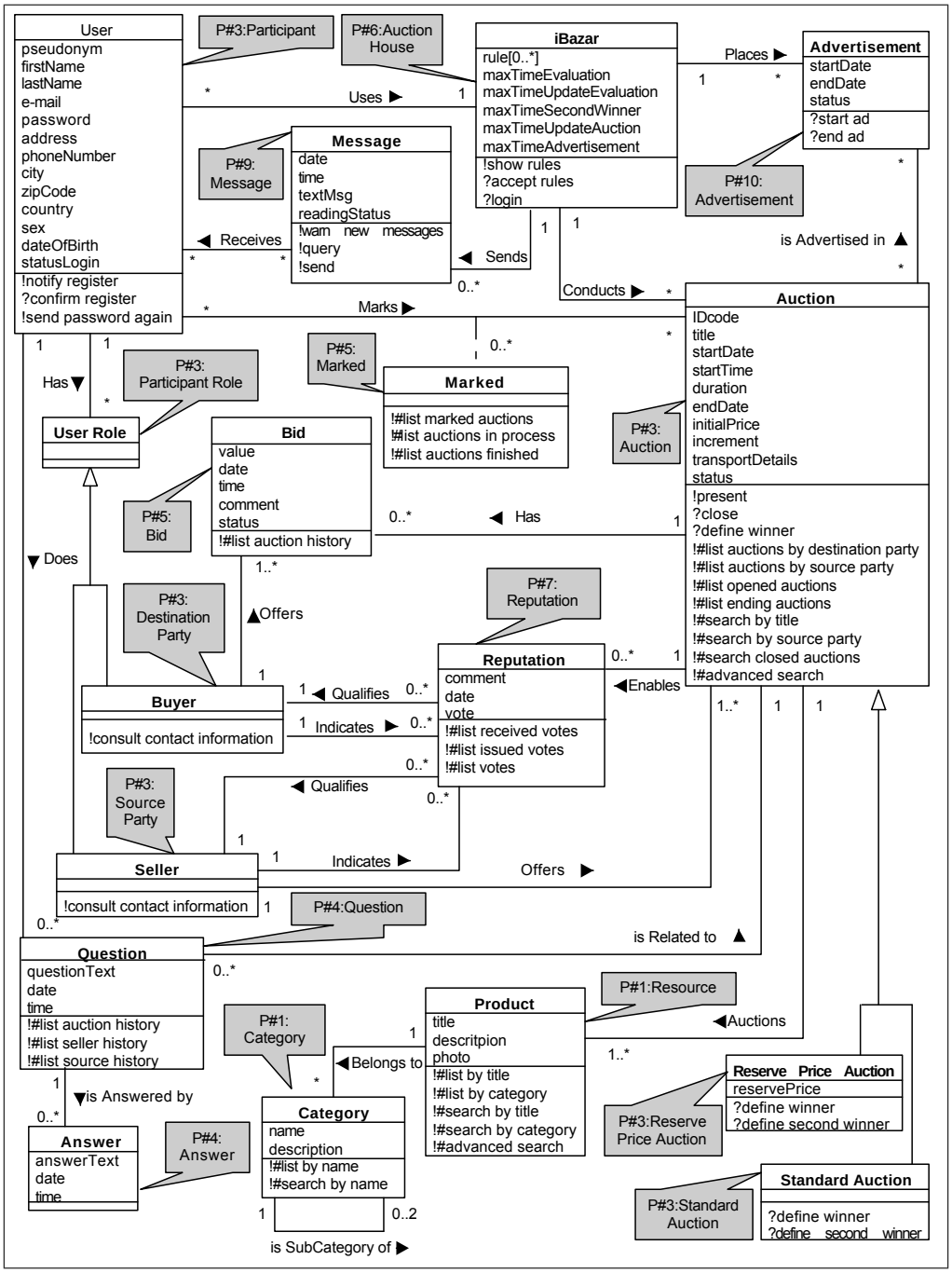


Figure 5. Example of a System modeled using the Online Auction Pattern Language

References

Aarsten, A., Brugali, D., and Menga, G. (2000). *A CIM Framework and Pattern Language*, pages 21–42. Domain-Specific Application Frameworks: Frameworks Experience by Industry, M. Fayad, R. Johnson, –John Willey and Sons.

Alexander, C. et al. (1977). *A Pattern Language*. Oxford University Press, New York.

Beck, K. and Cunningham, W. (1987). Using pattern languages for object-oriented

programs. Relatório Técnico n. CR-87-43. available on January 2007 at the URL: `http://c2.com/doc/oopsla87.html`.

Braga, R. T. V., Germano, F. S. R., and Masiero, P. C. (1999). A pattern language for business resource management. In *6th Pattern Languages of Programs Conference (PLoP'99)*, Monticello – IL, USA.

Braga, R. T. V., Germano, F. S. R., and Masiero, P. C. (2003). Experiments on pattern language-based modeling. In *17º SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE (SBES 2003)*, Manaus – AM, Brazil.

Braga, R. T. V. and Masiero, P. C. (2002a). GREN-Wizard: a tool to instantiate the GREN framework. In *Caderno de Ferramentas do 16o Simpósio Brasileiro de Engenharia de Software (SBES 2002)*, pages 408–413, Gramado-RS.

Braga, R. T. V. and Masiero, P. C. (2002b). A process for framework construction based on a pattern language. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 615–620, IEEE Computer Society, Oxford-England.

Braga, R. T. V. and Masiero, P. C. (2002c). The role of pattern languages in the instantiation of object-oriented frameworks. *Lecture Notes on Computer Science*, 2426-Advances in Object-Oriented Information Systems:122–131.

Braga, R. T. V. and Masiero, P. C. (2003). Building a wizard for framework instantiation based on a pattern language. *Lecture Notes on Computer Science*, 2817-Evolution of Object-Oriented Information Systems:95–106.

Brown, K. and Whitenack, B. G. (1996). *Crossing Chasms: A Pattern language for Object-RDBMS Integration, The Static Patterns*, pages 227–238. Addison-Wesley. in Vlissides et al., 1996 Pattern Languages of Program Design 2.

Brugali, D. and Menga, G. (1999). Frameworks and pattern languages: an intriguing relationship. *ACM Computing Surveys*, 32(1):2–7.

Brugali, D., Menga, G., and Aarsten, A. (2000). *A Case Study for Flexible Manufacuring Systems*, pages 85–99. Domain-Specific Application Frameworks: Frameworks Experience by Industry, M. Fayad, R. Johnson, –John Willey and Sons.

Buschmann, F. et al. (1996). *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley.

Cheesman, J. and Daniels, J. (2001). *UML Components*. Addison-Wesley.

Coad, P., North, D., and Mayfield, M. (1997). *Object Models: Strategies, Patterns and Applications*. Yourdon Press, 2 edition.

Coplien, J. O. (1998). *Software Design Patterns: Common Questions and Answers*, pages 311–320. Cambridge University Press. in L. Rising - The Patterns Handbook: Techniques, Strategies, and Applications.

Coplien, J. O. and (eds), D. S. (1996). *Pattern Languages of Program Design*. Addison-Wesley, Reading-MA.

Cunningham, W. (1994). Tips for writing pattern languages. Available on January 2007 at the URL: `http://www.c2.com/cgi/wiki?TipsForWritingPatternLanguages`.

Cunningham, W. (1995). *The CHECKS Pattern Language of Information Integrity*, pages 145–155. Addison-Wesley. in J. Coplien and D. Schmidt (eds.) - Pattern Languages of Program Design.

Fowler, M. (1997). *Analysis Patterns*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Gomaa, H. (1996). Reusable software requirements and architectures for families of systems. *Journal of Systems and Software*, pages 189–202.

Johnson, R. E. and Woolf, B. (1998). *Type Object*, pages 47–65. Addison-Wesley. in Martin, R.C.; Riehle, D.; Buschmann, F. Pattern Languages of Program Design 3.

Meszaros, G. and Doble, J. (1998). *A Pattern Language for Pattern Writing*, chapter 29, pages 529–574. Reading-MA, Addison-Wesley.

Pazin, A. (2004). GAwCRe: Um gerador de aplicações baseadas na web para o domínio de gestão de clínicas de reabilitação (in portuguese). Master's thesis, Universidade Federal de São Carlos, São Carlos – SP.

Pazin, A., Penteado, R., and Masiero, P. C. (2004). SiGCli: A pattern language for rehabilitation clinics management. In *4ª Conferência Latino-Americana em Linguagem de Padrões para Programação (SugarLoafPlop)*, Porto das Dunas - CE, Brasil.

Prieto-Diaz, R. and Arango, G. (1991). *Domain Analysis and Software System Modeling*. IEEE Computer Science Press Tutorial.

Ré, R., Braga, R. T. V., and Masiero, P. C. (2001). A Pattern Language for Online Auctions. In *8th Pattern Languages of Programs Conference (PLoP'2001)*, Monticello – IL, USA.

Rational, C. (2000). Unified Modeling Language. available on January 2007 at the URL: `http://www.rational.com/uml/references`.

Riehle, D. and Zullighoven, H. (1996). *Theory and Practice of Object Systems*, volume 2, chapter Understanding and Using Patterns in Softwre Development. John Wiley & Sons, New York – NY, USA.

Roberts, D. and Johnson, R. (1998). *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, pages 471–486. Pattern Languages of Program Design 3, Martin, R.C., Riehle, D. , Buschmann, F. – Addison-Wesley.

Shimabukuro, E. K., Masiero, P. C., and Braga, R. T. V. (2006). Captor: Um gerador de aplicações configurável (in portuguese). In *Anais da XIII Sessão de Ferramentas do XX Simpósio Brasileiro de Engenharia de Software*, pages 121–128, Florianópolis, SC, Brazil.

POREI: Patterns-Oriented Requirements Elicitation Integrated – Proposta de um Metamodelo Orientado à Padrão para Integração do Processo de Eliciação de Requisitos

Kleber Rocha de Oliveira[1,2], Mauro de Mesquita Spínola[2]

[1]Tecnologia em Sistemas de Informação – Faculdades Integradas de Bauru (FIB)
17056-100 – Bauru – SP – Brasil

[2]Departamento de Engenharia de Produção – Universidade de São Paulo (USP)
Cidade Universitária – Caixa Postal 61.548 – 05508-900 – São Paulo – SP – Brasil

kleber@softvip.com.br, mauro.spinola@poli.usp.br

Abstract. *The requirements elicitation is essential to the success of software development projects. Many papers have been written that promulgate specific elicitation methods. However, none have yet modeled elicitation in a way that makes clear the critical role played by situational knowledge. This paper presents a unified model of the requirements elicitation process that emphasizes the applying the concepts of patterns as it transforms the current state of the business requirements and the situation to an improved understanding of the requirements and, potentially, a modified situation. The values of this model are: (1) an improved understanding of elicitation helps analysts improve their elicitation efforts and (2) as we improve our ability to perform elicitation, we improve the likelihood that systems we create will meet their intended customers' needs.*

Resumo. *A atividade de eliciação de requisitos é essencial para o sucesso de projetos de software. Muitos artigos difundem apenas os métodos de eliciação de requisitos. Entretanto, é incomum encontrar artigos que modelem a eliciação como um meio de estabelecer a compreensão e o entendimento de problemas nas diversas situações. Este artigo apresenta um modelo do processo de eliciação de requisitos que prioriza a aplicação dos conceitos de padrões (patterns) na compreensão das necessidades da organização. Os principais benefícios do modelo são: (1) Melhorar a compreensão no processo de eliciação e (2) Como podemos evoluir nossas habilidades na execução das atividades de eliciação, tornando dessa maneira mais compreensíveis necessidades dos usuários.*

1. Introdução

Analistas experientes possuem maior facilidade na atividade de construção de software por adquirirem conhecimentos de soluções recorrentes que podem ser aplicadas em diversas situações similares. Tais soluções podem ser documentadas adequadamente no

formato de padrões, sendo que, um padrão pode ser visto como a descrição de uma solução de um problema recorrente em um determinado ambiente para facilitar a sua utilização diversas vezes, sem, no entanto implementar a solução da mesma forma duas vezes [Sommerville and Sawyer 1997].

Na Engenharia de Software, a idéia de padrões encapsulam as melhores soluções baseadas em anos de desenvolvimento de aplicações, observação e experiência. Para encontrar a melhor solução, o desenvolvedor deve entender o problema, o contexto e as forças que governam esse problema [Gause and Weinberg 1990]. Dessa forma, os padrões ajudam na construção de sistemas confiáveis, seguindo os passos de outras construções de sistemas de sucesso [Harrison, Foote and Rohnert 1990].

Alexander (1979) postula que a proposição por trás dos padrões diz respeito ao fato de que a qualidade dos sistemas de software pode também ser medida objetivamente. Ele sabia que as estruturas não poderiam estar separadas do problema que tentavam resolver. Portanto, em sua pesquisa para identificar e descrever consistência da qualidade de projeto, ele percebeu que teria que observar diferentes estruturas projetadas para resolver o mesmo problema [Gamma *et.al.* 1995]. A esta concepção, como sabido, deu-se o nome de padrões, na qual definiu como "uma solução para um problema em um determinado contexto" [Alexander 1979].

Assim como em outras áreas, a aplicabilidade dos conceitos de *patterns* na Engenharia de Requisitos vêm sendo aprimorada nos últimos anos, e os resultados de cada solução são armazenados para que possam ser reutilizados em novos projetos, mas habitualmente de forma não estruturada [Shalloway and Trott 2004]. Portanto, esta pesquisa visa propor um arcabouço baseado em padrões, relacionados ao processo de eliciação[1] de requisitos de software, baseado nos preceitos que levaram Gamma *et al.* a criar soluções padronizadas às diversas situações encontradas na Análise Orientada a Objeto. O intuito é aplicar o modelo na integração dos métodos de eliciação de acordo com cada situação, proporcionando ao engenheiro de requisitos a compreensão dos problemas. Este modelo recebe a denominação POREI – Patterns-Oriented Requirements Engineering Integrated.

2. Patterns e a Produção do Software

No âmbito da engenharia, o arquiteto Alexander observou a semelhança nas soluções pertinentes à arquitetura urbanista, na qual foi suficiente para que pudesse postular o famoso conceito de *patterns*, através da trilogia "A timeless way to builting, A pattern language e The Oregon Experiment" que constituem o ideário coeso do arquiteto, na qual defendia a seguinte proposição:

> *(…) Cada padrão descreve um problema que se coloca, vez por outra, em nosso entorno, e traz em si mesmo o núcleo da solução para esse problema, de tal forma que se possa*

[1] *Atividade também é identificada pelo verbo "elicitar", porém, não se recomenda a utilização deste termo, uma vez que, a mesma deriva-se de "elicit", proveniente da língua inglesa,e ausente (até 2007) no idioma português, segundo a Academia Brasileira de Letras. Debates sobre o termo têm mostrado que "elicitar" é na verdade uma composição de vários verbos da língua portuguesa, como segue: eliciar + clarear + extrair + descobrir, ou seja, tornar explícito, obter o máximo de informação para o conhecimento do objeto em questão [Leite 1989]. O verbo "eliciar" é o que mais se aproxima do termo inglês, portanto, tem sido recomendado sua utilização no meio acadêmico, inclusive optado nesta artigo.*

> *utilizar essa solução mais de um milhão de vezes, sem necessidade de repeti-la nunca da mesma maneira* [Alexander 1979].

O conceito foi adotado e expandido posteriormente por Gamma *et. al.* (1995) na obra Design Patterns, onde aborda tais doutrinas para solucionar problemas relacionados a análise orientada a objetivo na construção de software. Pesquisas direcionadas a produção de software baseadas nos estudos de *patterns* tem sido amplamente difundida, visto os resultados obtidos pelo cientista computacional Richard Gabriel, que vislumbrou caminhos na aventura teórica de Alexander.

No elástico mundo do conhecimento, não é incomum a migração de enunciados e de princípios de um campo da ciência para explicar fenômenos de outra origem epistemológica. E se a concepção arquitetural se alicerça em analogias, parece razoável admitir uma interpretação inversa, onde um certo paradigma teórico do pensamento arquitetônico possa estimular a interpretação de problemas de projetos de outra área da criação contemporânea [Buschmann 1996] [Gause and Weinberg 1990].

No prefácio do livro de Gabriel (1996), Christopher Alexander deixa transparecer surpresa com o fato, mas também alguma mágoa pela incompreensão de seus pares:

> *"(...) O que teve de fascinante para mim, na verdade, inteiramente surpreendente, foi que no ensaio dele (Gabriel), um cientista da computação, para mim um desconhecido, e com quem nunca havia me encontrado, me pareceu entender mais sobre o que tenho feito e o que venho tentando em meu próprio campo, do que meus próprios colegas arquitetos"*. (Alexander, prefácio [Gabriel 1996]).

Gabriel (1996) percebe e põe em evidência naquele conjunto de ensaios a relação possível entre o método gerador de formas e estruturas de Alexander e a oportunidade de propor uma acepção no campo dos sistemas orientados ao objeto. Isso, em essência, se dá pelo reconhecimento de que, em um caso e outro, o processo é o de associação de "entidades" que funcionam como blocos de uma linguagem. Nos dois casos se está diante de processos que projetarão por analogias. Após a detalhada interpretação do pensamento de Alexander, lança algumas bases para o desenvolvimento de uma teoria própria, de certa maneira apontando já os caminhos de uma interface cognitiva distinta, isto é, uma diferente forma de construir o conhecimento através de um processo de simulação que, no caso, se vale do isomorfismo entre entidades arquitetônicas e partes de uma linguagem computacional. Na concepção de linguagem (informática) de Gabriel estão presentes, entre os principais aspectos da "teoria alexandriana":

i) A capacidade de geração de padrões como partes intercambiáveis, capazes de metamorfoses conforme a atividade e a posição geométrica que ocuparem no programa;
ii) A geração de softwares caracterizados pela autonomia semântica entre suas partes;
iii) A construção de softwares habitáveis, ou seja, configurados por linhas de código compreensíveis por um grande número de pessoas da comunidade informática;
iv) O desenvolvimento de softwares que possam evoluir a partir de um crescimento incremental e;

v) Quando necessários, softwares complexos poderiam ser estruturados através de conexões com outros programas pré-existentes.

De muitas maneiras, a idéia sintetizada como linguagem de padrões, vis-à-vis o sentido de totalidades, sugere um processo de sucessivos acoplamentos, de partes maiores ou menores, na conformação da estrutura do software ou de um sistema. Pode-se então falar que isso revela um modo onde estrutura e organização convergem para a idéia de rede, que se realiza nos muitos planos de coordenação e controle do processo.

Até mesmo em jogos, os conceitos de patterns são aplicáveis. Wright (2001), criador do SimCity, afirma que sua inspiração original para o jogo foram as 256 regras de design contidas no livro "A Pattern Language", [Alexander 1979], cada qual baseada em um aspecto do comportamento humano. A idéia básica é que o projeto de construção deve refletir aspectos desse comportamento em diferentes escalas.

A aplicação de patterns para o entendimento do problema e captura dos requisitos de um sistema vêem sendo difundido entre diversos grupos de pesquisa, onde inclusive se produz certa quantidade de artigos sobre o tema e suas aplicabilidades, de forma satisfatória. Um grupo que tem forte destaque nesta linha de pesquisa é a alemã Deutsches Elektronen-Synchrotron (DESY) Sócios da Associação de Helmholtz, é um centro de pesquisa nacional apoiado por fundos de dívida pública, localizada em Hamburg e Zeuthen (Brandenburg), reconhecidamente um dos principais centros de aceleração gravítica (prótons e elétrons) no mundo [Deutsches 2006].

Portanto, propor um modelo para orientar o levantamento de requisitos, baseado nas práticas primárias e corolários de patterns se torna plenamente viável, pois além de agilizar o processo de eliciação dos requisitos de software, pode aumentar a qualidade dos documentos gerados no projeto, inclusive observar falhas nos processos organizacionais, conseqüentemente sugerir mudanças e melhorias.

3. Reuso Aplicado a Engenharia

Reuso de artefatos em geral tem sido um objetivo primordial em Engenharia de Requisitos. A própria utilização do termo reuso é uma demonstração do quanto reuso é essencial e também do quanto ele não está sendo atingido. Todas as tradicionais disciplinas de Engenharia estão tão intrinsecamente baseadas na grande quantidade de reuso de elementos, que o termo reuso nem é mencionado. Afinal reuso é parte integrante de praticamente tudo que se faz em Engenharia. Estas cláusulas comuns formam uma boa definição de Engenharia, pois estão relacionadas com a criação de soluções eficientes, para problemas práticos, através da aplicação de conhecimento científico, construindo coisas a serviço da condição humana [Sawyer , Sommerville and Viller 1997].

Para atingir este objetivo, uma Engenharia utiliza conhecimentos científicos sobre domínios tecnológicos que estão codificados de uma forma que seja diretamente útil para um engenheiro. Deste modo este conhecimento codificado provê respostas para questões que ocorrem comumente na prática. Ou seja, este conhecimento deve ser reutilizado para a geração de soluções [Czarnecki and Eisenercker 2002].

Engenharia também pode ser entendida pela distinção entre trabalho criativo e trabalho rotineiro. Trabalhos rotineiros são aqueles que envolvem a solução de problemas conhecidos e, portanto facilitam a reutilização de grande parte de outras

soluções já aplicadas a problemas similares. A construção de uma rodovia por engenheiros civis é geralmente um trabalho rotineiro. A menos que o relevo da região seja tão diferente que requeira soluções novas. Nestes casos, onde soluções novas são necessárias, o trabalho a ser realizado é denominado de trabalho criativo. Engenharia está fortemente relacionada com trabalhos rotineiros [Sawyer , Sommerville and Viller 1997].

4. Reutilização de Requisitos

Uma das características de um processo de desenvolvimento maduro de software é a reutilização de artefatos gerados em seus ciclos iniciais e intermediários [Sommerville and Sawyer 1997] [Paulk 1993]. No caso da reutilização de requisitos deve-se contemplar explicitamente o modelo de processos, assim como ter um suporte automatizado. Os principais motivos de aplicar técnicas de reutilização em Engenharia de Requisitos, ao menos neste ponto de vista, não é simplesmente reduzir custos e aumentar a qualidade, que evidentemente são fundamentais, mas também ter boa performance em produtividade, cumprindo dessa forma os prazos estabelecidos pelos contratos.

A introdução sistemática da reutilização em Engenharia de Requisitos pode levar a uma troca radical entre os planejamentos habituais sobre o processo a seguir, conduzindo a fazer uma Engenharia de Requisitos baseadas em componentes, de forma similar a como está sendo produzido em nível de implementação.

Oportunamente, poderia alegar situações em que o Engenheiro de Requisitos dispõe de um repositório suficientemente rico, o processo de eliciação-análise-documentação-validação vai ser substituído por outro em que, as funções que necessitam de um cliente, se obtiverem os requisitos selecionados em uma série de requisitos, nos quais relacionamos rastreabilidade, indicam que os componentes de software deveriam organizar-se para implementar um sistema que dê ao cliente a solução de seus problemas. Esta situação seria parecida com que a produz quando se adquire um produto modular pré-fabricado: o cliente escolhe de um catálogo em função de suas necessidades, conhecendo desde o primeiro momento o custo de sua escolha [Sommerville and Sawyer 1997].

5. Modelo POREI: Patterns-Oriented Requirements Elicitation Integrated

O modelo apreciado na figura 1, assim como os preceitos de Alexander (1979), cada padrão descreve um problema que ocorre repetidamente no nosso ambiente e, portanto, descreve o cerne da solução desse problema, de tal forma que pode-se utilizar essa solução diversas vezes repetitivamente, sem nunca fazê-la duas vezes do mesmo modo [Alexander 1979].

Para que os processos sejam mais reutilizáveis, organizações precisam expressar elementos comuns e variáveis dentro de um processo. *Frameworks* fornecem um mecanismo para obter esta reutilização e são bem apropriados para domínios onde várias aplicações similares são construídas várias vezes, partindo-se apenas de idéias [Hollenback and Frakes 1996). Pesquisas voltadas para *patterns* também têm mostrado que eles são ferramentas efetivas para a reutilização [Meszaros and Doble 1998] [Gamma *et. al.* 1995].

Portanto, tais pesquisas anteriores serviram para aplicar a idéia de padrões de projeto a padrões de requisitos. Aqui é descrita uma estrutura para catalogar e descrever padrões de projeto. Como se vê na figura abaixo, foram identificados 20 padrões de requisitos, contemplados por muitas pesquisas, mas não aplicados de forma estruturada.

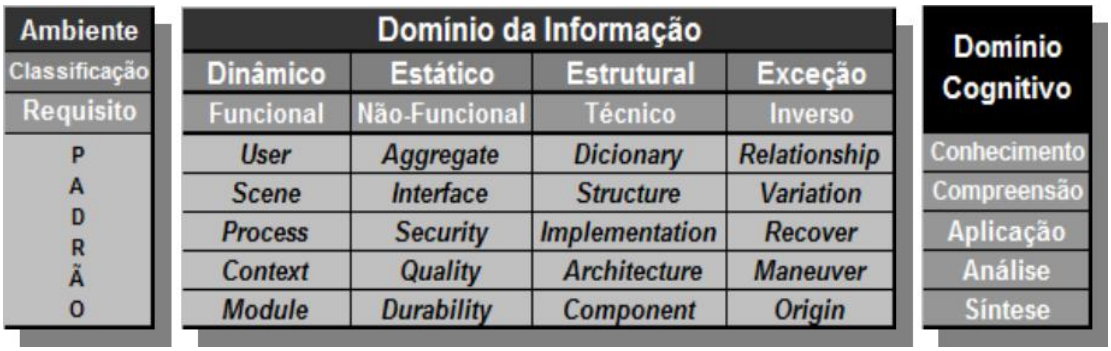


Figura 1. Modelo de Padrão de Requisitos[2]

Existem diversos formatos ou *templates* para a descrição de padrões de requisitos de softwares. Alguns são quase puramente textuais escritos em prosa livre, enquanto outros são mais estruturados [Gamma *et.al.* 1995]. Embora haja tantas opções, não existe um formato padronizado pela comunidade de software, principalmente porque diferentes tipos e domínios de padrões podem exigir diferentes maneiras de apresentar tais padrões. Mesmo assim, há certo consenso geral sobre elementos essenciais que devem ser contemplados e comunicados por qualquer padrão, independentemente do formato utilizado [Gamma *et.al.* 1995] [Shalloway and Trott 2004].

5.1. Definição dos elementos do modelo

As notações gráficas, embora sejam importantes e úteis, não são suficientes. Elas simplesmente capturam o produto final do processo de projeto como relacionamento entre os casos de usos e demais diagramas, Para reutilizar os requisitos, devemos também registrar decisões, alternativas e analises de custos e benefícios que levaram a ele. É evidente que exemplos concretos reforçam o conhecimento sobre o tema e o problema [Gamma *et.al.* 1995] [Shalloway and Trott 2004].

Assim como o conceito de *Design Patterns*, será descrito os padrões de requisitos usando um formato consistente, ou seja, cada padrão é dividido em seções de acordo com suas características que envolvem: o Nome, a Identificação, o Problema, a Solução, o Ambiente, as Forças, e principalmente os Exemplos. Tal gabarito fornece uma estrutura uniforme às informações, tornando os padrões de requisitos mais fáceis de aprender, comparar e usar.

A relação dos padrões de requisitos catalogados neste modelo, assim como suas classificação e intenções são listados na tabela 1 que segue:

[2] *O Ambiente, a Classificação e os Tipos de Requisitos que fazem parte do modelo apresentado (figura 1), não foram criados pelos autores, ou seja, são contemplados e utilizados plenamente pela comunidade de software. São princípios que refletem o que tem sido aprendido sobre projetos em sistemas de informação, de alta qualidade para problemas específicos.*

Tabela 1. Catálogo de padrões de Requisitos

Padrão	Descrição
User	Reconhecer usuários que terão direito de uso do sistema da informação.
Scene	Transmitir realidade visual ou a atmosfera dos locais onde decorre a ação.
Process	Esboçar maneiras pela qual se realiza uma operação, segundo determinadas normas, método ou técnica.
Context	Discriminar idéias de uma funcionalidade, expondo o grau de formalidade ou de intimidade entre as fases.
Module	Organizar unidades planejadas à determinadas proporções e destinada a reunir-se ou ajustar-se a outras unidades análogas, de várias maneiras, formando um todo homogêneo e funcional.
Aggregate	Indicar relação entre as partes (de um todo) não-funcionais de um sistema, em que cada uma delas mantém a autonomia e a consistência próprias.
Interface	Inferir sobre dispositivos, físico ou lógico, que faz a adaptação entre o sistema e o usuário, à respeito da usabilidade.
Security	Ilustrar sobre dispositivos, físico ou lógico, relativos à segurança dos dados (ativos intangíveis).
Quality	Qualificar as propriedades, atributos e condições dos recursos que envolvem a construção do sistema da informação e quantificar uma escala de valores que permite avaliar e, conseqüentemente, aprovar, aceitar ou recusar, qualquer dispositivo do sistema.
Durability	Combinar tempo útil aplicado, de certo dispositivo, físico ou lógico, ao sistema que proporcione performance satisfatória e garantia dos processos.
Dicionary	Definir glossário de termos, sigla, nomenclaturas aplicados aos processos de negócios.
Structure	Inventariar estrutura física pela qual as informações irão fluir.
Implementation	Escolher tecnologias (ferramentas e recursos), método e processo de implementação do sistema.
Architecture	Rotear informações, elucidando todos os pontos de abastecimento e reorganização dos pacotes de dados e estrutura das informações.
Component	Propor uso de componentes de conexão aos dispositivos, lógicos e físicos, do sistema.
Relationship	Relacionar ocorrências dos fluxo alternativos e restrições ao cenários do sistema.
Variation	Exprimir possíveis variações das exceções ocorridas nos fluxos das informações em função do contexto.
Recover	Estruturar métodos e práticas contingentes relativos a recuperação de falhas no sistemas.
Maneuver	Detectar desvios nos fluxos de informações e identificar suas alternativas.
Origin	Planificar origens das exceções, ou seja, esboçar o modo na qual ocorre sua concepção.

Os padrões podem variar de acordo com sua granularidade e no nível de abstração. Pois cada padrão tem suas peculiaridades e torna-se necessário organizá-los de maneira a fazer sentido sua aplicação. Portanto, nós classificamos os padrões de requisitos por dois critérios (figura 1). O primeiro critério, baseado no conhecimento do Domínio da Informação, reflete o tipo de requisito identificados em qualquer ambiente. Os padrões podem, nessa visão, ter a características de serem dinâmicos, estáticos, estrutural e inversos. Os padrões dinâmicos estão relacionados com os requisitos

funcionais de um sistema da informação. Os padrões estáticos estão relacionados com os requisitos não-funcionais do sistema, e que refletem a qualidade do sistema. Já os padrões estruturais estão ligados aos requisitos técnicos, e por fim, os padrões de exceção, relacionados aos requisitos de exceção, ou seja, expõem os fluxos alternativos, assim como regras e exceções à regra básica de parte do sistema.

O segundo critério, chamado Domínio Cognitivo, especifica se o padrão se aplica ao Conhecimento, Compreensão, Aplicação, Análise e Síntese. Na tabela 2 segue as descrições desses Domínios Cognitivos.

Tabela 2. Domínios Cognitivos

Conhecimento	São informações de idéias e fenômenos armazenados, podendo ser especifico, geral, abstrato ou estrutural. Designa modo de operações e técnicas gerais do tratamento de temas e problemas.
Compreensão	Grau de entendimento ou percepção de algo que está sendo transmitido sem necessariamente relacioná-la com outras matérias ou ver todas as suas implicações.
Aplicação	Uso da abstração em situações específicas e concretas. Podem apresentar-se sob forma de idéias gerais, normas de procedimento ou método geral. Poderá ser ainda princípios, leis, teorias que devem ser recordadas e aplicadas.
Análise	Divisão de uma comunicação em seus elementos ou partes constituintes, de modo que a relativa hierarquia de idéias apareça claramente ou a relação entre as idéias expressas se evidencie. Busca identificação destas conexões e interações entre elementos.
Síntese	Envolvem a reunião, ordenação e combinação de segmentos, partes, elementos, em um padrão ou estrutura anteriormente não especificadas. Exige a combinação de partes da experiência prévia com novo material.

Existem diversas maneiras de se organizar os padrões, pois as maiorias dos padrões devem ser usados em conjunto. Por exemplo, o padrão *User* é frequentemente usado com o *Scene* e o *Interface*. Alguns padrões resultam em requisitos semelhantes, embora tenham intenções diferentes. Outros padrões são alternativos: O *Quality* pode ser um padrão alternativo para o *Architecture*.

Outra forma, ainda, de organizar padrões de requisitos é de acordo com que eles mencionam outros padrões no modelo de relacionamento. A figura 2 ilustra estes relacionamentos graficamente.

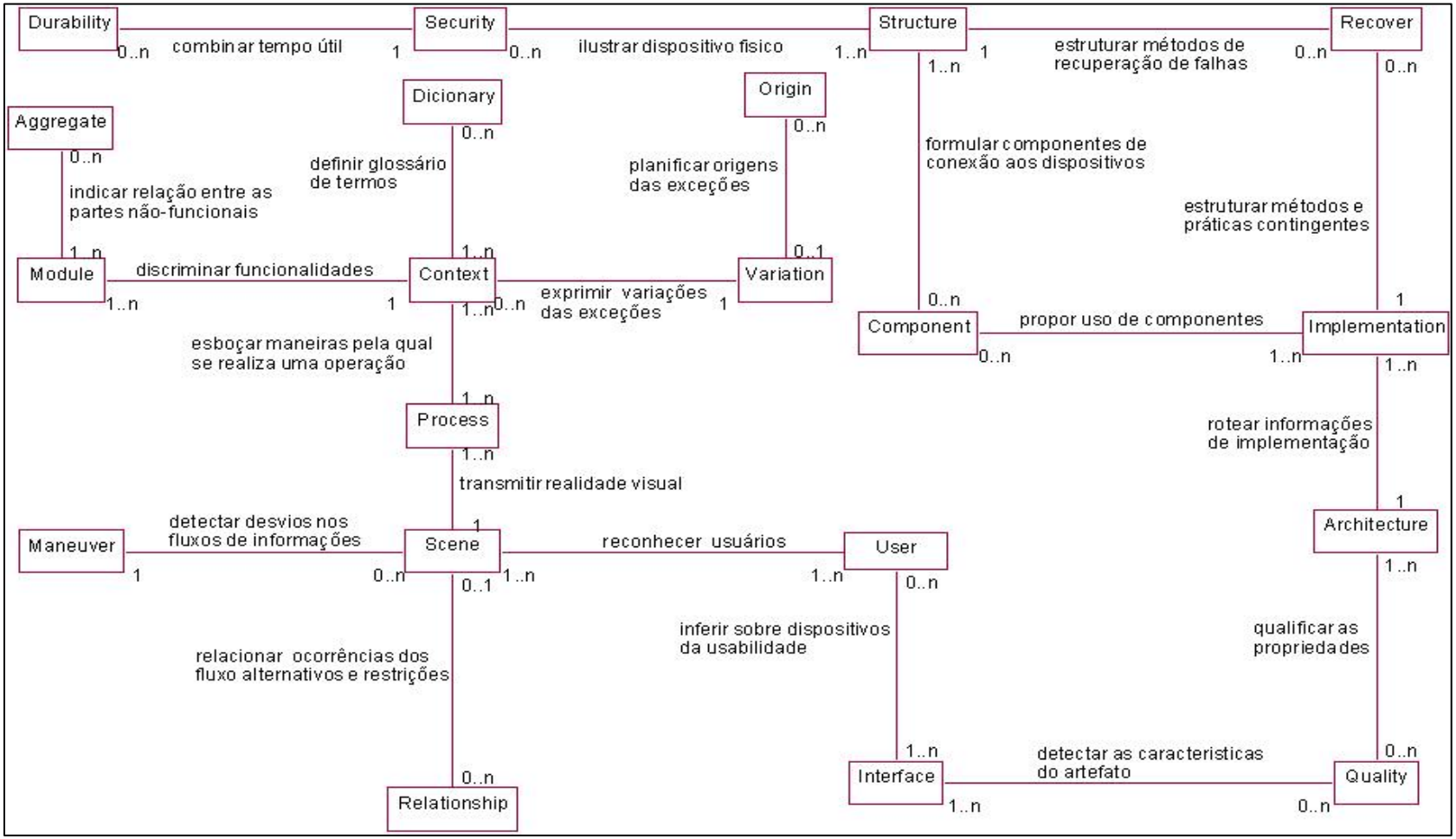


Figura 2. Relacionamento dos Padrões de Eliciação de Requisitos.

5.2. Como os padrões solucionam problemas de projeto

Os padrões de requisitos solucionam muitos dos problemas que os engenheiros de requisitos enfrentam diariamente, e de muitas maneiras diferentes. Apresentaremos a seguir vários problemas e como os padrões de requisitos solucionam:

- Definindo os usuários (stakeholders): Basicamente todos os sistemas sofrem o impacto da participação de pessoas, empresas (fornecedores, clientes, parceiros, terceirizados etc) e até coisas (servidores, outros sistemas, tecnologias em geral etc). Com este padrão é possível identificar usuários ou coisas que sofrem influência direta e indiretamente do sistema e definir seu papel, responsabilidades e relacioná-los aos cenários afins.

- Eliciando os cenários: Os cenários são seqüências de interações entre o sistema entre o sistema e seus atores. Um conjunto de cenários pode dar uma boa descrição de como o sistema Será sempre usada descrição mínima como a descrição do sistema antes de entrar no cenário (pré-condições), o fluxo de eventos, as exceções, atividades paralelas e as descrições dos estados do sistema após a atividade do cenário. O padrão favorece a reutilização destes fluxo de eventos e suas ligações, reduzindo dessa forma o trabalho de eliciar determinados requisitos.

- Definindo a qualidade do produto: Muitos dos requisitos seguem normas de qualidade de produtos e processos, como as publicadas pela ISO, SEI/CMU, NBR, BS entre outros órgãos regulamentadores espalhados pelo mundo. A qualidade com que as informações chegam ao seu receptor é muito importante, portando tratar desses itens é fundamental para agradar aos usuários e patrocinadores dos projetos. Os padrões neste caso auxiliam na escolha da norma e seu conjunto de critérios de acordo com o histórico de aplicabilidade como se fosse uma “jurisprudência” na área de Engenharia de Software.

- Quantificando o tempo útil da tecnologia: Quanto tempo leva para certa tecnologia se defasar? Quanto não tem profissionais com conhecimento necessário para manipulá-lo? Ou quando o fornecedor deixa de produzir tal ferramenta? Ou talvez quando as informações não chegam da forma, velocidade e consistência, que deveria chegar ao seu receptor? Portanto o padrão de requisitos, de maneira análoga a percepção do usuário em relação seu carro, sua casa, poderá criar critério que determinem a validade de determinadas tecnologias em relação ao produto que está comprando. Esses limites seriam determinados levando em conta critérios como estrutura necessária para garantir a satisfação dos usuários, tempo de resposta a uma determinar tomada de decisão, e obviamente deveriam ser calibradas com o passar do tempo através de aplicação de benchmarking.

- Familiarizar com os termos, siglas e conceitos desconhecidos: Técnica que procura descrever os símbolos de uma linguagem na área de Engenharia de Requisitos dá-se o nome de Léxico Ampliado da Linguagem (LAL) [Leite *et.al.*, 1997]. A idéia central do LAL é a existência da linguagem da aplicação. Esta idéia parte do princípio que no universo de informações existe uma ou mais culturas e que cada cultura (grupo social) tem sua linguagem própria. Portanto, o principal objetivo (e desafio) a ser perseguido pelos engenheiros de requisitos é a identificação de palavras ou frases (peculiares) ao meio social da aplicação sob estudo. Somente após a identificação dessas frases e palavras é que se procurará seu significado. A estratégia de eliciar é ancorada na sintaxe

da linguagem, gerar um glossário indexado que a possibilita de confrontar seus significados e rastrear suas aplicabilidades dentro de um contexto definido.

6. Conclusão

O modelo, que faz parte de uma pesquisa de doutorado, vislumbra os benefícios que a padronização pode fornecer quando se estabelece uma estrutura aplicável a qualquer situação. Embora seja evidente a necessidade de uma boa documentação que oriente o profissional em sua jornada no processo de eliciação dos requisitos, a proposta de utilizar os conceitos de padrões traz mais eficiência na identificação dos elementos chaves do ambiente de informação e proporciona a reusabilidade com mais qualidade, mapeando a solução de problemas recorrente a produção de software.

A rastreabilidade é também um item a ser observado, pois muitas dessas soluções dependem de outros artefatos para gerar o resultado esperado, e o modelo proposto traz essa ligação encapsulada em cada um dos padrões proposto.

Enfim, as mesmas necessidades e desafios encontrados em outras áreas da Engenharia são refletidos na área de construção de Sistemas de Informações, portanto não se devem fechar os olhos para as idéias e soluções que se encontram próximos e passíveis de exploração, experimentação, adaptação.

Referências

Alexander, C. (1979) “The Timeless Way of Building”, Oxford University Press.

Buschmann, F. et al.(1996) “ Pattern Oriented Software Architecture: A System of Patterns”, John Wiley & Sons.

Czarnecki, K., Eisenercker, U.W.(2002) “Generative Programming”,Addison-Wesley.

Deutsches.(2006) “Elektronen-Synchrotron: DESY, http://www.desy.de, Agosto.

Gabriel, R. P. (1996) “Patterns of Software: Tales from the software community”, Oxford: Oxford University Press.

Gamma, E. et al.(1995) “Design Patterns: Elements of Reusable Object-Oriented Software”, Reading, MA : Addison-Wesley.

Gause, D. C., Weinberg, G. M.(1990) “Are Your Lights On? How to Figure Out What the Problem Really Is”. 1ed. USA : Dorset House Publishing Co. Inc., 157 p.

Harrison, N.; Foote, B.; Rohnert, H.(1999) “Pattern Languages of Program Design”, Addison-Wesley.

Hollenbach, C. ; Frakes, W.(1996) “Software Process Reuse in an Industrial Setting”, Fourth international Conference on Software Reuse, Orlando, Florida, IEEE Computer Society Press, Los Alamitos, CA, pp 22-30.

Leite, J.C.S.P. (1989) “Viewpoint Analysis: A case Study”, IWSSD'89 Fifth International Workshop on Software Specification and Design. (Pittsburg, Pensylvania, USA) 1ed.USA : ACM Sigsoft Engineering. Proceedings, may, p111-119.

Leite, J.C.S.P. et al.(1997) “Enhancing a Requirements Baseline with Scenarios”, ISRE'97 Third International Symposium on Requirements Engineering. (Annapolis, Maryland, USA) 1ed.USA: IEEE CSP, Los Alamitos, CA.Proceedings, p 44-53.

Meszaros, G.; Doble, J. (1998) “A Pattern Language for Pattern Writing”, Reading, MA : Addison-Wesley.

Paulk, M. C. et al.(1993) “Capability Maturity Model for Software”, Version 1.1. Technical Report CMU/SEI–93–TR–024, Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu, Junho.

Sawyer, P.; Sommerville, I.; Viller, S. (1997) “Requirements Process Improvement Through The Phased Introduction of Good Practice”, Software Process – Improvement and Practice, http://www.comp.lancs.ac.uk, Junho.

Shalloway, A; Trott, J.(2004) “Explicando padrões de projeto : uma nova perspectiva em projeto orientado a objeto”, tradução Ana M. de Alencar Price. Porto Alegre: Bookman.

Sommerville, I.; Sawyer, P. (1997) “Requirements Engineering (A Good Practice Guide)”, 1ed. England : John Wiley & Sons Ltd, 391p.

Wright, R.(2001) “Game design: theory and practice”, Interview in ROUSE III, Plano, Texas: Wordware Publishing.

Aplicando Padrões de Projeto em Computação Móvel

Mauro Strelow Storch, André Rauber Du Bois, Adenauer Correa Yamin

[1] Escola de Informática - Universidade Católica de Pelotas(UCPEL)
Pelotas – RS – Brasil

{mstorch,dubois,adenauer}@ucpel.tche.br

Resumo. *As rede de computadores estão em constante evolução, causando também uma evolução de toda a estrutura da computação que atua sobre elas. Hoje em dia existem vários recursos computacionais conectados em rede, e pesquisadores tentam tirar proveito do poder computacional disponível nas redes de larga escala. Uma nova abordagem para se aproveitar esses recursos seria o uso da mobilidade de código, ou programas móveis. Um programa móvel pode iniciar sua execução em uma máquina da rede e depois mover-se para outra máquina onde continua a sua execução. Apesar das vantagens adquiridas com o uso da computação móvel, esse tipo de sistema ainda é muito difícil de programar. O objetivo deste trabalho é implementar padrões de projeto que tornem mais fácil a programação de sistemas com mobilidade de código. Foram identificados padrões de computação móvel e estes foram modelados como padrões de projeto do tipo Template Method. A grande vantagem desses padrões é que o programador não precisa se preocupar com a programação de baixo nível desse tipo de sistema, ele apenas escolhe o padrão de projeto que descreve o comportamento móvel desejado. Para testar os padrões desenvolvidos nesse trabalho, implementou-se uma agenda colaborativa distribuída que usa os padrões de projeto identificados para implementar toda logística de mobilidade de código do sistema.*

Abstract. *Nowadays almost all computing resources are connected in networks, and researchers are trying to take advantage of the computational power avaliable in large scale networks. A promissing aproach is to use code mobility, or software mobility. A mobile program can initiate its execution in a host and then move itself to another host where it continues its execution. Despite the advantages acquired with mobile computation, this kind of system is still very difficult to program. The objetive of this work is to design Java classes that encapsulate common patterns of mobile computation. These classes should help in the development of systems that use code mobility. We have identified three common patterns of mobile computation and implemented them as Template Method design patterns. The advantage of these patterns is that the programmer does not need to worry about the low level details of programming mobile systems, he just has to choose the mobility pattern that describes the desired mobile behavior. To test the mobility patterns developed in this work, a distributed meeting scheduler was implemented that uses the design patterns identified to implement all the code mobility of the system.*

1. Introdução

As redes de computadores estão em constante evolução, principalmente com a disseminação da Internet. Junto à evolução das estruturas de rede, há também a evolução do software que atua sobre essas estruturas. A idéia de poder compartilhar informações através da rede abriu um leque muito grande de opções para utilização de seus recursos. A partir dai surgiram linhas de pesquisa, não só para o compartilhamento de informações, mas também para o compartilhamento de todos os recursos que uma rede de computadores pode oferecer. Uma abordagem para o compartilhamento de recursos em redes de larga escala seria o uso de *mobilidade de código* ou *programas móveis* [Fuggetta et al. 1998]. Um programa móvel possui a capacidade de mover-se entre as máquinas de uma rede e executar suas instruções em qualquer uma delas. Desta forma um programa móvel pode utilizar os recursos disponíveis localmente em cada uma das máquinas da rede, em uma perspectiva muito mais flexível que aquela explorada com componentes distribuídos em localizações pré-fixadas.

Além de especificar o algoritmo a ser executado, um programa móvel também deve descrever vários outros aspectos de *coordenação* do aplicativo, e.g., como o programa será dividido entre as várias máquinas do sistema, quando partes do programa devem ser movidas, comunicação, sincronização etc. Dessa maneira, a implementação de programas que utilizam computações móveis é tão ou mais difícil do que a implementação de sistemas distribuídos tradicionais. O objetivo deste trabalho é apresentar padrões de projeto para computação móvel que facilitem a programação desse tipo de sistema. Os padrões apresentados neste artigo facultam ao programador especificar esse tipo de sistema usando um maior nível de abstração.

Em [Du Bois et al. 2005b] foram identificados três padrões como de uso recorrentes na computação móvel.

Neste artigo é apresentada a modelagem desses padrões como um *template method*. Sendo assim, estes padrões foram implementados como classes abstratas na linguagem de programação *Java* [Java 2006]. Essas classes abstratas permitem que o programador não se preocupe com os aspectos de baixo nível da coordenação de programas móveis. Ao desenvolver uma aplicação móvel o programador deve apenas escolher o padrão que melhor descreve a aplicação e estender a super-classe do padrão implementando métodos abstratos que descrevem as computações que serão executadas *localmente* nas máquinas que a computação móvel irá visitar. Aspectos como sincronização e comunicação das computações móveis são herdados da classe pai.

Este artigo é organizado da seguinte maneira: na Seção 2 os conceitos de Padrões de Projeto, incluindo o padrão de projeto *template method*, e computação móvel são revisados. Em seguida, na Seção 3, os padrões de projeto para programação móvel são apresentados. Para demonstrar a usabilidade dos padrões desenvolvidos, a implementação de uma agenda colaborativa distribuída é descrita na Seção 4. A agenda usa os padrões de projeto identificados para implementar toda a comunicação de código móvel do sistema. Finalmente os trabalhos relacionados e as conclusões são discutidos nas seções (Seção 5) e (Seção 6) respectivamente.

2. Fundamentos

2.1. Padrões de Projeto

Historicamente os Padrões de Projeto foram identificados pelo arquiteto Christopher Alexander no final dos anos 70, que fez a seguinte afirmativa: *Cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira.* Embora Alexander estivesse falando de padrões de construções civis, o que ele diz é verdadeiro em relação aos padrões de projeto utilizados na computação. No centro de ambos os tipos de padrões estão as soluções para os problemas em seu devido contexto.

Um padrão descreve uma solução para um problema que ocorre com freqüência durante o desenvolvimento de software, podendo ser considerado como um par ”problema/solução”[Bushamnn and Meunier 1995]. O uso de padrões proporciona um vocabulário comum para a comunicação entre projetistas, criando abstrações num nível superior ao de classes e garantindo uniformidade na estrutura do software [Gall et al. 1996].

Os Padrões de Projeto são classificados, de acordo com a granularidade e nível de abstração, em três categorias diferentes [Gamma et al. 1995]. São elas:

- **De Criação**: Criar ou instanciar objetos.
- **Estrutural**: Reunir objetos existentes.
- **Comportamental**: Prover uma maneira de manifestar comportamento flexível (variável).

Os padrões de projeto para computação móvel descritos na seção 3 são modelados como *Template Methods* e classificados como padrões do tipo comportamental. Basicamente um padrão *Template Method* é uma classe abstrata que define um método *gabarito* e descreve o esqueleto de um algoritmo, postergando a definição de alguns passos para as sub-classes. Este padrão permite que sub-classes redefinam certos passos de um algoritmo sem mudar sua estrutura.

2.2. Computação Móvel

Computação Móvel pode ser definida de diferentes formas dependendo da área. Quando falamos de *hardware*, associamos esse termo a mobilidade física dos equipamentos, como *notebooks* e *laptops*. Porém na área de *software* chamamos Computação Móvel quando um programa se move entre equipamentos interligados por uma rede de computadores [Cardelli 1999].

Este artigo trata sobre Computação Móvel na área de *software*, ou seja, *mobilidade de código*. Nesse caso, uma computação móvel pode iniciar sua execução em um nodo de uma rede e em certo ponto ser movida para um outro nodo da estrutura distribuída dando continuidade a sua execução. Assim a computação pode ser executada localmente em vários nodos, utilizando o máximo dos recursos disponíveis na rede.

A mobilidade de software traz também vantagens para os usuários de dispositivos móveis. Um usuário de um dispositivo de pouco poder computacional pode enviar um programa para ser executado nos recursos computacionais existentes em uma rede e reconectar novamente mais tarde para receber os resultados dessa computação.

3. Padrões de Projeto para Computação Móvel

3.1. Formas Recorrentes de Computação Móvel

Em [Du Bois et al. 2005b], foram identificados três formas recorrentes de computação móvel que ocorrem em sistemas de aquisição de informações distribuídos [Callan 2000]. Nesse tipo de sistema várias bases de dados são analisadas em busca de alguma informação em comum. Este tipo de aplicação é considerada uma *Killer Application* para a computação móvel [Fuggetta et al. 1998]. Os padrões identificados são:

- *Mmap*: Descreve o multicast de computações aos nodos de uma rede.
- *Mfold*: Descreve uma computação que visita uma lista de nodos executando instruções e recolhendo valores.
- *Mzipper*: Descreve uma computação que visita os nodos de uma rede buscando um valor comum em todos os nodos.

No artigo citado, os padrões são identificados e implementados em uma extensão para mobilidade de código da Linguagem funcional Haskell [Haskell 2006]. Os padrões são implementados como *Esqueletos de Mobilidade*, i.e., *funções de alta ordem*, que encapsulam padrões recorrentes de computação móvel. Uma das principais vantagens dos Esqueletos para Mobilidade é facilitar a programação de aplicações móveis. Essa facilidade ocorre pois o programador não se preocupa com os aspectos de baixo nível da mobilidade de código, mas apenas com a implementação da computação que será executada *localmente* nos nodos da rede.

Um dos problemas dos *esqueletos de mobilidade* é o fato de eles estarem modelados e implementados em uma linguagem de pesquisa, usando um paradigma de programação de difícil aceitação no mercado de desenvolvimento de software. O objetivo desta seção do artigo é modelar e implementar os padrões identificados em [Du Bois et al. 2005b] usando a linguagem de programação Java. Os nomes dos padrões foram inspirados em funções típicas de linguagens funcionais cujo comportamento, embora não comporte distribuição e/ou mobilidade, assemelham-se aos padrões propostos. Dessa maneira pretendemos tornar mais fácil a programação de sistemas com mobilidade de software pois os padrões estarão descritos, modelados e implementados usando um paradigma de programação largamente utilizado no mercado. Para isso os padrões serão modelados como padrões de projeto do tipo *template method* e implementados usando a linguagem de programação Java.

3.2. Estrutura para Mobilidade dos Padrões

Para desenvolver os Padrões de Projeto para Computação Móvel, foi necessária a implementação de uma estrutura baseada em *Sockets* para mover e executar programas remotamente utilizando os recursos existentes na linguagem de programação Java.

A estrutura é composta basicamente de três componentes:

1. Interface `Execute`- Interface Java que descreve como executar objetos remotamente. Possui um único método abstrato `executar()` que deve ser implementado por todos os objetos a serem executados remotamente.
2. Classe `RemoteCreate`- Esta classe possui dois métodos estáticos para a criação remota de objetos. O método `createS` recebe como argumentos uma máquina

remota e um objeto do tipo `Execute`, e move uma instância do objeto para a máquina remota. Assim que o objeto é movido, o seu método `executar()` é chamado. O resultado da chamada a `executar()`, um objeto do tipo `Serializable`, é enviado de volta para a máquina que chamou o método `createS`. A classe `RemoteCreate` possui também um método `createA` que é uma versão assíncrona do método `createS`, ou seja, executa o objeto remotamente sem esperar por uma resposta.

3. Servidor `JMServer` - É um servidor presente em todas as máquinas do sistema. Ele recebe objetos enviados pelo métodos `createS` e `createA` e os executa automaticamente.

A estrutura implementada é bem mais simples que outras existentes em java, e.g., RMI [Grosso 2001] e Voyager [Voyager 2006], disponibilizando somente os recursos básicos necessários para o desenvolvimento dos padrões de projeto.

3.3. O Padrão `Mmap`

O padrão mais simples de computação móvel identificado é o `Mmap`, também chamado de *Multicast*. Esse padrão define uma aplicação que envia uma computação a todos os *hosts* de uma lista passada como parâmetro. O `Mmap` retorna uma lista, do mesmo tamanho da lista de `hosts`, que contém o resultado da execução das computações, como pode ser visto na Figura 1.

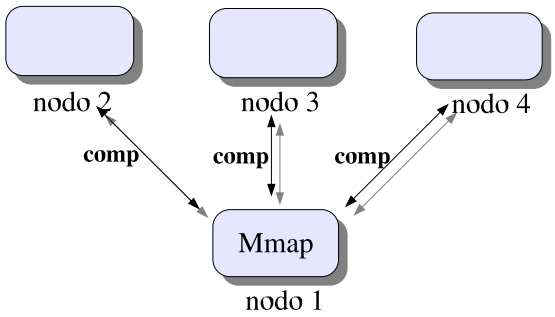


Figura 1. mmap - Multicast

A Figura 2 apresenta o diagrama UML da modelagem do `Mmap` como um padrão de projeto *Template Method*. A classe abstrata `Mmap` possui um atributo privado `hosts` que contém os nodos da rede a serem visitados e é inicializado através do construtor da classe. O único método que o programador deve implementar quando estende a classe `Mmap` é o método abstrato `executar()` que descreve o que a computação deve fazer em cada uma das máquinas que visita. O método `goMmap()` é o método *gabarito* da classe, que deve ser chamado para que o `Mmap` seja ativado.

Na Figura 3 um exemplo simples de uso do `Mmap` é apresentado. A classe `Ola` herda as funcionalidade de mobilidade do padrão `Mmap` e a sua execução consiste em visitar uma lista de *hosts* e imprimir a string `'Ola!!'` em cada *host*. Como o resultado retornado pela computação remota é irrelevante, ele é simplesmente ignorado no exemplo.

3.4. O Padrão `Mfold`

Este padrão descreve uma computação que visita uma lista de nodos em uma rede, executando uma computação em cada nodo e combinando os resultados produzidos us-

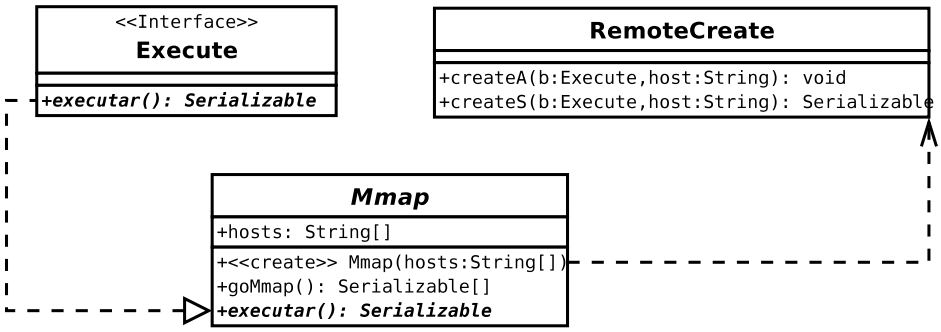


Figura 2. Diagrama de Classes do Padrão `Mmap`

```
class Ola extends Mmap{

    Ola(String[] hosts){ super(hosts); }

    public String executar(){
        System.out.println("Ola!!");
        return "Ok";
    }

    public static void main(String[] args){
        String [] hosts =  (...) // lista de hosts

        Ola oi=new Ola(hosts);
        oi.GoMmap();
}
}
```

Figura 3. Exemplo de uso do `Mmap`

ando um operador. Quando atinge a última máquina a ser visitada, o `Mfold` devolve o resultado da computação para a máquina inicial. A Figura 4 ilustra o funcionamento deste padrão.

A Figura 5 apresenta o diagrama UML da classe que implementa o padrão `Mfold`. Da mesma forma que o `Mmap`, este foi modelado como uma classe abstrata onde foi definido um método gabarito (`goMfold()`) responsável pela mobilidade e dois métodos abstratos, `executar()` e `operador()` que devem ser implementados quando a classe `Mfold` for estendida. O método `executar()` descreve a computação que será executada em cada nodo e o método `operador()` é responsável por combinar os resultados produzidos em um acumulador. O construtor da classe `Mfold` recebe como argumen-

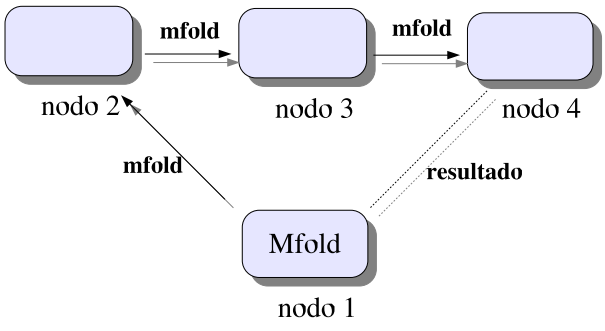


Figura 4. Mfold - Sistema de Aquisição de Informações

tos um valor inicial para o acumulador e a lista de nodos a serem visitados. O método gabarito `goMfold()` é usado para iniciar a execução do `Mfold`.

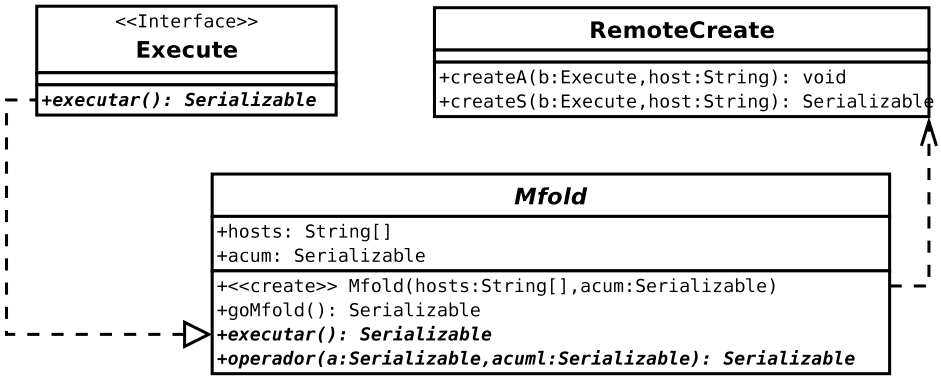


Figura 5. Diagrama de Classes do Padrão `Mfold`

O programa da Figura 6 usa o padrão `Mfold` para visitar nodos de uma rede e coletar seus nomes. O construtor da classe `Coletor` recebe como argumentos uma lista vazia, usada como acumulador e a lista de nodos a serem visitados. O método `executar()` usa `getHostName()` para pegar o `hostname` e o método `operador()` combina os nomes de todas as máquinas visitadas em uma string. O método `goMfold()` dentro do `main` inicia a execução do `Mfold` e retorna uma string contendo o nome de todas as máquinas visitadas pela computação móvel.

```
class Coletor extends Mfold{

   Coletor(String[] hosts){
      super(hosts,"");
   }

   public Serializable executar(){

      String res="";
      try{
         res=InetAddress.getLocalHost().getHostName();
      }catch(Exception e){System.out.println(e);}
      return res;
   }

   public Serializable operador(Serializable incluir,
                                Serializable acumulador){
      return (acumulador + " " + incluir);
   }


   public static void main(String[] args)
                     throws Exception{
       (...)
       String [] hosts =  (...) // lista de hosts
       Coletor c=new Coletor(hosts);

       resultado=c.goMfold();
       System.out.println(resultado);
    }
}
```

Figura 6. Exemplo de uso do Padrão `Mfold`

Registra-se que algumas aplicações implementadas com o padrão `Mfold` também

podem ser implementadas usando o padrão `Mmap`, porém os programas terão um comportamento operacional completamente diferente, como pode ser visto nas figuras 1 e 4, i.e., no `Mmap` o controle da aplicação retorna sempre para a máquina inicial e o padrão `Mfold` sempre executa a *continuação* da computação na próxima máquina a ser visitada.

3.5. O Padrão `Mzipper`

O Padrão `Mzipper` descreve uma computação móvel que tenta encontrar um valor que seja satisfatório a todos os nodos de uma rede. O valor é testado com um predicado em cada nodo, e caso o predicado falhe, a computação é movida para o nodo inicial da rede, onde um novo valor é gerado e a busca recomeçada. A Figura 7 ilustra o comportamento do padrão.

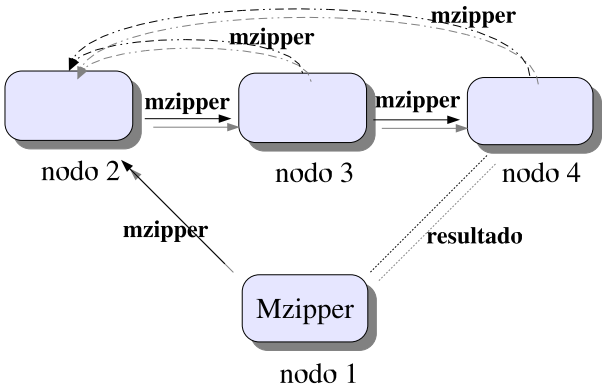


Figura 7. Comportamento do `Mzipper`

A Figura 8 mostra o digrama UML do padrão `Mzipper`. No construtor é passada a lista de nodos que a computação deverá visitar. O método `goMzipper()` é o método gabarito, e os método abstratos `executar()` e `predicado()` serão implementados pelas sub-classes. O método `executar()` é sempre chamado no primeiro nodo da lista para gerar um valor inicial. Esse valor é validado nos demais usando o método `predicado()`. Quando o predicado falha, a computação é movida novamente para o primeiro nodo e um novo valor é gerado usando o `executar()`. O `Mzipper` termina quando todos os nodos concordaram com um valor ou quando o nodo inicial não pode mais gerar valores para serem comparados.

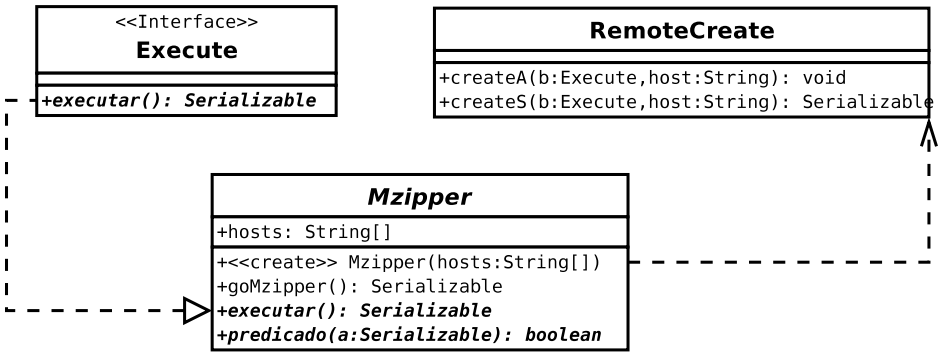


Figura 8. Diagrama de Classes do Padrão `Mzipper`

O programa da Figura 9 apresenta um programa simples que tenta encontrar um horário livre em agendas distribuídas nas máquinas de uma rede. O método `executar()` busca um horário livre na máquina inicial. Este horário é então testado em todas as máquinas usando o método `predicado()`. Se uma das máquinas não possui esse horário livre, a computação volta para a primeira máquina e chama o método `executar()` novamente para que este gere um novo horário.

```
class Agenda extends Mzipper{

  ...

  public Horario executar(){
    return (horarioLivre());
  }
  public boolean predicado(Horario horaAtual){
    if (verificaHorario(horaAtual)) return true;
    else return false;
  }

  ...

  public static void main(String[] args){
    Serializable resposta;
    Agenda agenda=new Agenda();

    resposta=agenda.goMzipper();
    if(resposta!=null){
      System.out.println("Hora livre nas agendas:" + resposta);
    }
  }
}
```

Figura 9. Exemplo de uso do padrão `Mzipper`

4. Estudo de Caso: Agenda Colaborativa Distribuída

Nesta seção descrevemos a implementação de uma aplicação móvel que usa os padrões de projeto descritos para implementar toda logística de computações no sistema. A aplicação é uma *agenda colaborativa distribuída* que consiste em uma agenda que possui uma lista de compromissos e permite agendar compromissos com outras agendas distribuídas pela rede. O objetivo da aplicação é disparar uma computação móvel que visita agendas em máquinas remotas tentando achar um horário disponível em todas as agendas para marcar um compromisso. Quando o horário é achado, este deve ser comunicado a todas as agendas que participam do sistema. Essa aplicação apresenta dois padrões de mobilidade. O primeiro é a idéia de uma computação que *visita nodos da rede e realiza ações*, ou seja, visitar os nodos procurando pelo horário. Na seção 3 vimos dois padrões que apresentam esse comportamento, i.e., o `Mfold` e o `Mzipper`. O segundo padrão de computação móvel é a idéia de enviar uma computação para todas as máquinas, ou seja `Mmap`, informando o horário do compromisso. Nas próximas seções, duas diferentes implementações da agenda colaborativa são apresentadas usando os padrões descritos anteriormente.

A Figura 10 mostra a janela principal da agenda distribuída. Esta possui dois botões que representam os dois Padrões de Projeto para Programação Móvel utilizados para validação dos horários. Logo abaixo dos botões há uma lista onde encontram-se os endereços IP das máquinas onde estão rodando as outras agendas (aplicações iguais a

esta). Essa lista pode ser modificada através de dois botões (add, remove), e é nela que o usuário indica com quais agendas deseja marcar o evento, ou seja, os padrões somente irão visitar os endereços selecionados.

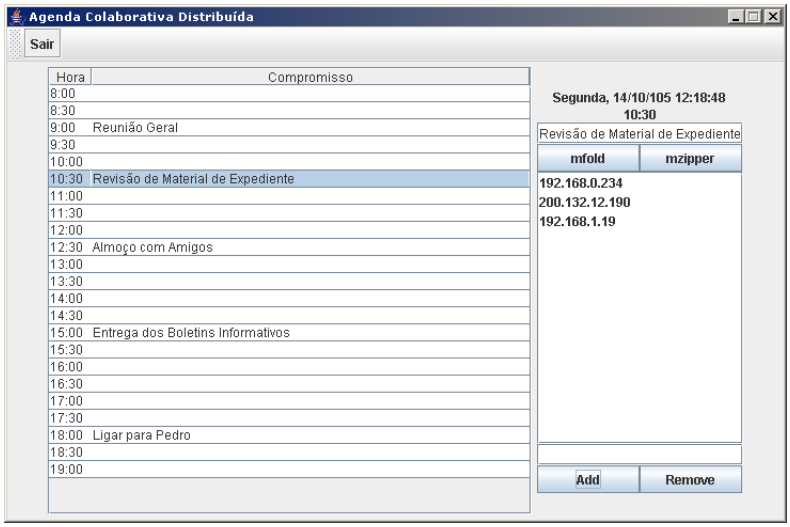


Figura 10. Janela Principal da Aplicação

4.1. Agendamento com o Padrão Mfold

O agendamento de eventos em agendas remotas usando o padrão `Mfold` é acionado com um *click* no botão `mfold` da aplicação.

O objetivo do `Mfold` na agenda é visitar uma lista de máquinas e computar a intersecção das listas de horários livres de todas as máquinas. A agenda implementada estende a super-classe `Mfold` implementando os métodos `operador` e `executar`. O método `operador()` retorna a intersecção de dois arrays com horários disponíveis e o método `executa()` é usado para ler localmente em cada máquina um arquivo que contém os horários livres de cada agenda. Depois de executar o método `goMfold()`, o `Mfold` visita todas os hosts pegando os horários livres de cada agenda (`executar()`) e computando a intersecção desses horários (`operador()`). No final o `Mfold` retorna os horários livres comuns à todas as máquinas. O resultado é apresentado ao usuário que escolhe um dos horários para a reunião. O horário selecionado é então enviado para todas as agendas através do `Mmap`, que faz o *multicast* de uma computação que faz o agendamento em todos os hosts do horário escolhido.

4.2. Agendamento usando o Padrão `Mzipper`

Na implementação da agenda distribuída a super-classe `Mzipper` é estendida de forma parecida com o programa apresentado na Figura 9. O método `executar()` é implementado de forma a buscar um horário livre na agenda da máquina em que é chamado, sendo somente executado na primeira máquina a ser visitada. Após chamar `executar()`, o `Mzipper` visita as outras máquinas testando o horário livre com o método `predicado ()`. `predicado()` recebe como argumento o horário livre corrente e compara este com os horários livres da máquina sendo visitada. Se o horário sendo pesquisado também está livre na máquina corrente, o `Mzipper` visita a próxima

máquina da lista. Caso contrário, o `Mzipper` volta para a primeira máquina e gera um novo horário para ser pesquisado usando novamente `executar()`. No final temos como resposta um horário que está livre em todas as máquinas, ou um valor nulo , que indica que não existe um horário livre comum à todas as agendas visitadas. Se o `Mzipper` consegue achar um horário livre, a reunião é marcada em todas as agendas usando o `Mmap`, assim como foi feito na implementação com o `Mfold`.

4.3. Comparação da aplicação dos padrões

Apesar da mobilidade necessária na Agenda distribuída poder ser descrita usando os padrões `Mzipper` e `Mfold`, o padrão de mobilidade presente nas diferentes implementações é bem diferente. Na implementação usando o `Mzipper`, a computação move pelas máquinas carregando apenas *um horário* para ser agendado. Toda a vez que esse horário é negado por uma das agendas a computação volta para a primeira agenda e pede um novo horário. Já na implementação usando o `Mfold`, a computação carrega uma *lista de horários* livres, e vai cruzando essa lista com as listas encontradas em cada uma das máquinas que visita. A implementação usando o `Mfold` parece ser mais oportuna para a agenda distribuída já que a lista de horários a ser carregada é pequena. Dessa maneira a computação não precisa voltar para a máquina inicial toda a vez que um horário é negado por uma máquina remota. Quando a lista de valores a serem comparados nas máquinas remotas é grande, por exemplo um banco de dados, fica praticamente impossível carregar toda a base de dados junto com a computação móvel. Nesse caso o padrão `Mzipper` é o mais adequado.

5. Trabalhos Relacionados

Os padrões apresentados neste texto modelados como *Template Methods* , podem ser assciados os *Algortithmic Skeletons* [Cole 1989], que são abstrações para programação paralela, geralmente implementadas em linguagens funcionais como funções de alta ordem, que encapsulam padrões de paralelismo, comunicação e/ou sincronismo de tarefas. Abstrações de mais alto nível para a programação distribuída também estão surgindo, como por exemplo *Behaviours* na linguagem Erlang [Erlang 2006]. Computação móvel é um campo relativamente mais novo mas que vem crescendo. Existem várias linguagens móveis, e.g., [Conchon and Fessant 1999, Du Bois et al. 2005a, Wojciechowski 2000, Knabe 1995, Voyager 2006, Cardelli 1995], porém poucas abstrações de alto nível para programação foram desenvolvidas. Podemos destacar os *Mobility Skeletons*, descritos na Seção 3, nos quais se baseam este trabalho.

Em [Wojciechowski 2000], uma plataforma para computação móvel baseada em agentes é apresentada. A plataforma utiliza uma linguagem chamada *Nomadic Pict* [Wojciechowski 2000] que estende a linguagem *pict* [C.Pierce and Turner 1997] com primitivas para mobilidade. As primitivas são divididas em duas classes: primitivas de *baixo* e *alto nível*. As primitivas de *baixo nível* descrevem migração e sincronização de computações, como por exemplo a primitiva `migrate to` que move a computação corrente para um outro nodo da rede. As primitivas de *alto nível* são implementadas usando as primitivas de baixo nível e fornecem uma maior abstração para a comunicação de computações baseada no nome dos agentes e não em sua localização. Os padrões apresentados neste artigo fornecem uma abstração para programação ainda maior pois cada padrão encapsula vários aspectos de um algoritmo para comunicação de computações.

Outros trabalhos já estudaram padrões de projeto para computação móvel, principalmente na área de agentes, e.g. [Lima et al. 2004], porém estes trabalhos focam mais na modelagem do sistema do que na facilidade de programação e reutilização de código.

6. Conclusão e Trabalhos Futuros

Este trabalho apresentou um estudo sobre a especificação e implementação de padrões de projeto (*design patterns*) que representam formas recorrentes de mobilidade de código. Mais especificamente, foram descritos, usando padrões de projeto do tipo *Template Method*, formas de mobilidade que ocorrem em *sistemas de aquisição de informações distribuídos*, assim como descrito em [Du Bois et al. 2005b]. As contribuições do trabalho foram: apresentar as formas de mobilidade em um paradigma de programação e notação largamente utilizada na indústria de software - facilitando assim o seu entendimento, implementar e demonstrar o uso dos padrões através de exemplos. Os padrões de projeto apresentados facilitam a programação de software móvel já que são formas reusáveis de código: basta o programador entender o padrão descrito para que ele possa reusar o código dos padrões através de herança. Além disso, quando o programador usa os padrões, ele só precisa especificar as computações que serão executadas *localmente* em cada nodo da rede. Toda a distribuição e sincronização das tarefas é herdada da classe-pai. Para testar os padrões desenvolvidos nesse trabalho, foi implementada uma aplicação, uma agenda colaborativa distribuída, que usa os padrões de projeto identificados para implementar toda a comunicação de código móvel do sistema.

Existem várias linhas para projetos futuros. A estrutura implementada para mobilidade dos padrões é baseada em *Sockets* e portanto é necessário que a classe do objeto móvel esteja presente em todas os nodos da estrutura. Isso é um problema pois aumenta o conjunto comum de software em todos os nodos para que o sistema móvel funcione. O problema pode ser solucionado modificando a estrutura de mobilidade para que esta faça a atualização remota de classes ou através do uso de uma estrutura pervasiva como por exemplo o *EXEHDA* [Yamim 2004] ou *Voyager* [Voyager 2006]. Uma outra área de projeto futuro seria a identificação e implementação de outros padrões recorrentes de computação móvel através da análise de programas móveis já existentes ou baseados em outros trabalhos, e.g. [Lima et al. 2004].

A estrutura de mobilidade e o código fonte dos Padrões de Projeto para Programação Móvel são de domínio público e podem ser acessados em [Strelow Storch 2006].

Referências

Bushamnn, F. and Meunier, R. (1995). *A system of Patterns*. New York, NY, USA: ACM Press / Addison Wesley Publishing Co.

Callan, J. (2000). Distributed information retrieval. pages 127–150. Advances in information retrieval in Kluwer Academic Publishers.

Cardelli, L. (1995). A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 286–297, New York, NY.

Cardelli, L. (1999). Mobility and security. In *Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation*, pages 3–37, Marktoberdorf, Germany.

Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman.

Conchon, S. and Fessant, F. L. (1999). Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA.

C.Pierce, B. and Turner, D. N. (1997). Pict: A programming language based on the pi calculus. Technical report, Computer Science Department, Indiana University.

Du Bois, A. R., Trinder, P., and Loidl, H.-W. (2005a). *m*Haskell: mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254.

Du Bois, A. R., Trinder, P., and Loidl, H.-W. (2005b). Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288.

Erlang (2006). Erlang. WWW page, http://www.erlang.org/.

Fuggetta, A., Picco, G., and Vigna, G. (1998). Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361.

Gall, H. C., Klosch, R. R., and Mittermeir, R. T. (1996). Application patterns in reengineering: Identifying and using reusable concepts. In *6th International Conference on Information Processing and Management of Uncertainty in Knowlege-Based Systems*, pages pp. 1099–1106.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.

Grosso, W. (2001). *Java RMI*. O'Reilly.

Haskell (2006). Linguagem Haskell. http://www.haskell.org.

Java (2006). Linguagem Java. http://java.sun.com.

Knabe, F. C. (1995). *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie mellon University.

Lima, E. F. A., Machado, P. D., Sampaio, F. R., and Figueiredo, J. C. A. (2004). An approach to modelling and applying mobile agent design patterns. *ACM Software Engineering Notes*, 29(4).

Strelow Storch, M. (2006). Padrões de projeto para computação móvel. http://atlas.ucpel.tche.br/˜mstorch.

Voyager (2006). Voyager System. http://www.recursionsw.com/voyager.htm.

Wojciechowski, P. T. (2000). *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Wolfson College, University of Cambridge.

Yamim, A. C. (2004). *Arquitetura para um Ambiente de Grade Computacional Direcionado às Aplicações Distribuidas, Móveis e Conscientes do Contexto da Computação Pervasiva*. Tese de doutorado, Univesidade Federal do Rio Grande do Sul.

Utilização de Padrões para Otimizar a Automação de Testes Funcionais de Software[1]

Rafael Braga de Oliveira[1,3], Francisco Nauber Bernardo Góis[3], Jerffeson Teixeira de Souza[2], Pedro Porfírio Muniz Farias[1]

[1]Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321 – Fortaleza – CE – Brasil

[2]Universidade Estadual do Ceará (UECE)
Av. Paranjana, 1700 - Fortaleza – CE - Brasil

[3]Serviço Federal de Processamento de Dados (SERPRO)
Av. Pontes Vieira, 832 - Fortaleza – CE - Brasil

{rafael.oliveira, francisco.gois}@serpro.gov.br,
jeff@larces.uece.br, porfirio@unifor.br

Resumo. *A automação de testes funcionais tem se tornado um evidente atrativo para equipes de desenvolvimento de software. Tal fato se deve principalmente à grande redução de custo observada a médio e a longo prazos com o uso desta prática. Este artigo propõe a aplicação de padrões para otimizar a automação de testes funcionais de software, introduzindo benefícios como o aumento da reusabilidade e da manutenibilidade de scripts de teste, e a facilidade de inclusão de novos casos de teste. O uso de padrões na automação de testes funcionais representa uma aplicação inovadora de padrões e um diferencial em relação às técnicas de automação de testes funcionais citadas na literatura.*

Palavras-chave: automação de testes; testes funcionais; teste de *software*, padrões de projeto.

Abstract*: The functional testing automation has become a real interest to software development teams, mainly because of the great cost reduction observed on medium and long terms with the use of this practice. This article proposes the application of patterns to optimize the software functional testing automation, introducing benefits as the increase of reusability and maintainability of tests scripts, and the facility of including new test cases. The use of patterns on functional testing automation represents an innovative application of patterns and a improvement in relation to the techniques of the functional testing automation mentioned on the literature.*

Keywords: testing automation; functional testing; software testing, regression testing, design patterns.

1. Introdução

Para minimizar o custo e proporcionar maior qualidade no desenvolvimento de *software*, inúmeros estudos ressaltam a importância de um processo de teste efetivo [1, 2, 3, 4, 16]. Desta forma, quanto mais eficiente e mais eficaz for o teste, menor será o custo dos reparos e maior será a qualidade do produto.

[1] The authors thank SERPRO for supporting this work.

Evidentemente, quanto mais tarde um defeito é encontrado maior é o custo de sua correção, podendo, em casos extremos, causar danos irreparáveis. Isto amplia o incentivo à adoção de mecanismos eficazes e eficientes para a realização de testes efetivos.

Testar um software é uma tarefa meticulosa e pode se tornar cansativa. Neste contexto, a automação de parte dos testes é uma alternativa para proporcionar a entrega de produtos mais confiáveis ao cliente.

Automatizar testes corresponde a desenvolver um novo código, portanto exige um esforço adicional em relação a testes realizados manualmente. Normalmente, o planejamento e a elaboração de testes automatizados requerem mais tempo do que o necessário para testes manuais. A principal vantagem é que a execução dos testes automatizados é muito mais rápida e torna-se possível repetir a realização dos testes num baixo custo e numa velocidade bastante superior. Segundo Fewster [2], testes manuais que levariam horas para serem concluídos, podem ser executados em minutos, quando automatizados.

Os testes podem ser divididos em testes caixa-branca, onde temos acesso ao código fonte do programa, e testes caixa-preta, onde não se conhece a estrutura interna do sistema. Os testes caixa-preta são realizados navegando na interface do sistema, introduzindo dados e selecionando opções, com o objetivo, normalmente, de verificar se as funcionalidades (testes funcionais) estão implementadas de acordo com as especificações.

Os testes caixa-preta podem ser automatizados através da criação e na execução de *scripts* de teste utilizando ferramentas *Record and Playback*. Estas ferramentas permitem a criação de *scripts* na forma de programas [3], os quais simulam ações de um usuário sob a interface do sistema. No contexto deste artigo, a automação de testes funcionais se limita à criação e à execução de *scripts* de teste utilizando tais ferramentas.

Para sistemas de grande porte, podem ser necessárias centenas de *scripts* para implementar a automação dos testes. Algumas técnicas, como *Data-driven* e *Keyword-driven* [2], têm sido propostas no intuito de tornar os *scripts* mais manuteníveis. Também têm sido desenvolvidos *frameworks* de forma a organizar e estruturar o uso destes *scripts* [10, 11, 12].

Neste artigo, apresenta-se um *framework* denominado FuncTest, que, além de aplicar as duas técnicas citadas, faz o uso de padrões [6, 7, 8] para otimizar o projeto de automação de testes funcionais. Foram utilizados a arquitetura MVC [6], o padrão DAO (*Data Acess Object*) [8, 19] e, em duas situações, o padrão *Factory Method* [5].

Scripts de teste são, de fato, programas. Portanto, podem se beneficiar da utilização de padrões. Todavia, como normalmente são gerados automaticamente ou produzidos a partir de trechos de códigos gerados automaticamente, não foram encontrados registros da utilização de padrões em *frameworks* usados na automação de testes funcionais. Recentemente, em [21], foram utilizados padrões para automatizar testes unitários (caixa-branca).

Além de contribuir para a estruturação e organização de *scripts* de teste, o *framework* FuncTest, no contexto da automação de testes funcionais construídos através de ferramentas *Record and Playback*, representa um enfoque inovador do uso de padrões.

Com a utilização do padrão de arquitetura MVC, associa-se cada passo de um caso de teste, através de uma tabela, ao seu *script* correspondente. Isto permite, como vantagem, uma independência entre casos de teste e *scripts*. Esta independência possibilita que um projetista desenvolva os casos de teste enquanto outro implementador, com experiência no desenvolvimento de *scripts*, encarrega-se de desenvolvê-los.

O padrão DAO utilizado acrescentou o benefício de fornecer transparência no acesso aos dados e permitir que os dados de teste persistam em bases de dados distintas. A geração dos objetos DAO foi implementada através de uma aplicação do padrão *Factory Method*. O padrão *Factory Method* também foi utilizado para a seleção dos *scripts* de teste.

O *framework* FuncTest se encontra em uso por especialistas de teste de uma grande empresa estatal de desenvolvimento de *software*. O processo de desenvolvimento utilizado na equipe é uma adaptação do RUP aderente ao nível 2 do CMMI. Uma versão preliminar do *framework* foi premiada em um congresso promovido pela empresa em 2006. A versão atual, além de aperfeiçoar a aplicação de padrões, contempla a utilização de técnicas relevantes para a literatura, bem como permite configurar a automação através de arquivos XML.

Na próxima seção, discorreremos sobre técnicas para a criação de *scripts* funcionais de teste, evidenciando-se problemas que serão tratados em seções subseqüentes através do uso de padrões. Na seção 3, abordaremos os padrões utilizados. Na seção 4, apresentaremos o *framework* proposto. Por fim, na seção 5, serão apresentados a conclusão e os trabalhos futuros.

2. Técnicas para Criação de *Scripts* Funcionais

Através de ferramentas *Record and Playback*, podemos gerar *scripts* automaticamente através da gravação de ações de usuário sobre a interface da aplicação ou simplesmente programar os *scripts*.

Normalmente, os *scripts* gerados pelas ferramentas *Record and Playback* deverão ser alterados ou, até mesmo, ser inteiramente programados. Por exemplo, devem ser excluídos comandos desnecessários inseridos pela ferramenta e utilizadas boas práticas de programação possíveis, como a inclusão de comentários para esclarecer a lógica do código.

Em [2], são apresentadas as seguintes técnicas para construção de *scripts*:

- *Scripts* Lineares;
- *Scripts* Estruturados;

- *Scripts* Compartilhados;
- *Scripts Data-Driven*;
- *Scripts Keyword-Driven*.

2.1 *Scripts* Lineares

Os *scripts* lineares são aqueles desenvolvidos utilizando-se unicamente a técnica *Record and Playback*. Portanto, são gravados durante a execução de um teste manual. Estes *scripts* conservam todos os comandos realizados durante a gravação. O uso desta técnica não exige conhecimento de programação por parte do testador. Entretanto, pode limitar o reuso e a manutenção dos *scripts* gerados.

Seguem algumas das limitações observadas:

- *scripts* longos e ilegíveis: normalmente um único *script* para cada caso de teste;
- presença de dados "*hard-coded*": ocorre quando o *script* possui dados de teste em seu código;
- *scripts* pouco coesos: *scripts* que realizam outras atividades além de ações sobre a interface de usuário, que é o seu principal propósito;
- vulneráveis a mudanças do sistema sob teste.

2.2 *Scripts* Estruturados

A criação de *scripts* estruturados, assim como na programação estruturada, pressupõe a utilização de instruções de controle, como seleções e interações. Além disso, um *script* pode chamar outro *script*. Este mecanismo pode ser usado para dividir *scripts* grandes em *scripts* menores e mais gerenciáveis, melhorando o reuso e a manutenibilidade dos *scripts*.

Embora estes *scripts* sejam mais flexíveis, não estão isentos de dados "*hard-coded*" e as chamadas entre eles trazem uma dependência que desfavorece o reuso dos mesmos.

2.3 *Scripts* Compartilhados

Scripts compartilhados são aqueles utilizados por mais de um caso de teste. O uso desta técnica visa identificar tarefas repetitivas que possam ser reutilizadas. O compartilhamento destes *scripts* pode ser feito entre casos de teste de um mesmo sistema ou de diferentes sistemas. Embora esta técnica aumente o reuso dos *scripts*, os mesmos ainda podem apresentar dados "*hard-coded*" e chamadas que os tornam dependentes.

2.4 *Scripts* Data-driven

Uma das técnicas amplamente utilizadas e essenciais para tornar a automação mais reutilizável é a técnica *Data-driven*. Esta técnica propõe a independência entre o código do *script* e a massa de dados utilizada durante o teste, evitando dados "*hard-coded*". Para aplicá-la, os dados de entrada deverão ser eliminados do corpo do *script* e inseri-los em arquivos de dados independentes ou tabelas. A principal vantagem desta independência entre o código dos *scripts* e os dados de teste é permitir que o *script* seja reutilizado para vários conjuntos de dados de entrada. Além disso, novos testes podem ser adicionados sem o conhecimento da linguagem de programação de *script* correspondente.

A ilustração seguinte (Figura 1), adaptada de [2], apresenta um exemplo simplificado do uso desta técnica.

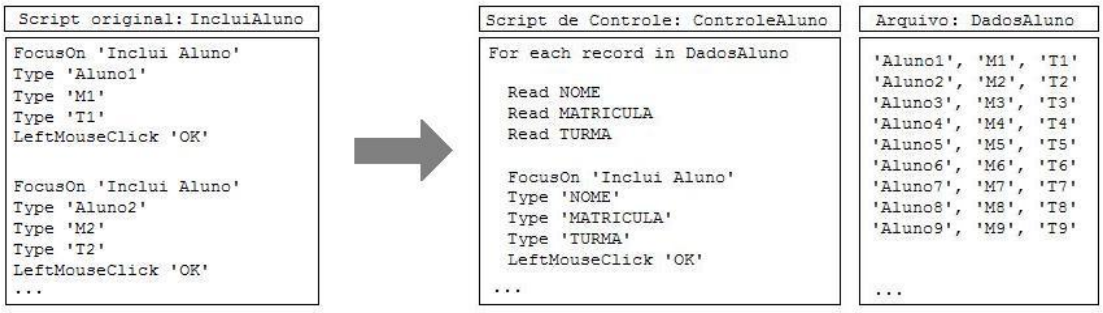


Figura 1 – Exemplo Simplificado de Aplicação da Técnica Data-Driven

2.5 *Scripts Keyword-driven*

A técnica *Keyword-driven* propõe a modularização dos *scripts*, de maneira que cada módulo seja representado por uma *keyword* e possua um *script* a ele associado. As *keywords* podem representar eventos simples, como um clique num botão, a serem aplicados na interface do sistema [12], ou eventos mais complexos como, por exemplo, o processamento de uma opção do sistema que envolva navegar em várias telas. Independente da complexidade do módulo, as *keywords* representam ações, de maior ou menor complexidade, sobre a interface do sistema.

A figura 2, adaptada de [2], ilustra como esta técnica pode ser aplicada. No exemplo, temos três casos de teste. Cada linha do caso de teste possui uma *keyword* e os dados a ela associados. Para cada *keyword*, existe um *script* de suporte especifico, o qual será responsável por realizar suas ações correspondentes.

Associando-se a técnica *Keyword-driven* à técnica *Data-driven*, podemos garantir a independência tanto dos dados quanto das ações de teste [10].

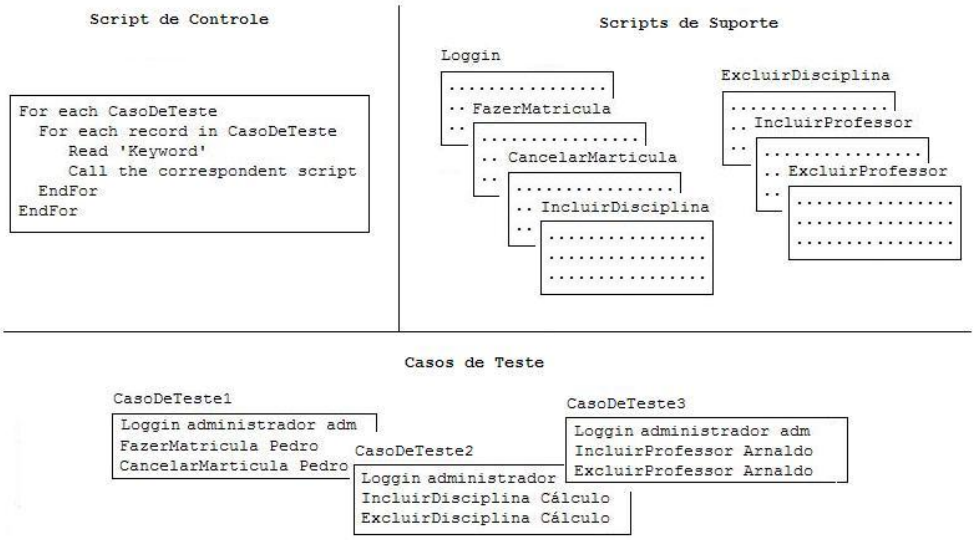


Figura 2 – Exemplo Simplificado de Aplicação da Técnica Keyword-driven

É importante entender que o projeto de automação de testes deve ser tratado como qualquer outro projeto de desenvolvimento de *software*. Portanto, a estruturação dos testes automatizados pode ser significativamente aprimorada através do uso apropriado de padrões. Na próxima seção, serão indicados os padrões utilizados na formulação do *framework* proposto.

3. Utilização de Padrões

O *framework* FuncTest faz uso dos padrões *Factory Method* e DAO. Nesta seção, descreveremos brevemente tais padrões para, posteriormente, evidenciarmos suas aplicações no *framework*.

3.1 Padrão *Factory Method*

O padrão *Factory Method* [5], também conhecido como *Virtual Constructor*, é especificado utilizando dois níveis: um nível abstrato e um nível concreto.

No nível abstrato, especifica-se a utilização de um construtor virtual, o *Factory Method*. Assim, lidando, neste nível, com a construção virtual de objetos, ainda sem antecipar a classe dos objetos que serão criados. No nível concreto, são criadas classes que estendem as classes abstratas, implementando apropriadamente o construtor virtual.

No diagrama da Figura 3, tem-se, no nível abstrato, um objeto da classe abstrata Creator para a criação de objetos da classe abstrata *Product*. Como ambas as classes são abstratas, não é possível, neste nível, antecipar a classe concreta que será utilizada.

Especifica-se, então, o método abstrato *FactoryMethod()*, que prevê a devolução do produto desejado como resultado. Este método funciona como um construtor virtual.

No nível concreto, tem-se as classes *ConcreteProduct*, subclasse da classe *Product*, e a classe *ConcreteCreator*, subclasse da classe *Creator*.

A classe *ConcreteCreator* implementa o construtor virtual através do método *FactoryMethod()*, que retorna um objeto da Classe *ConcreteProduct*.

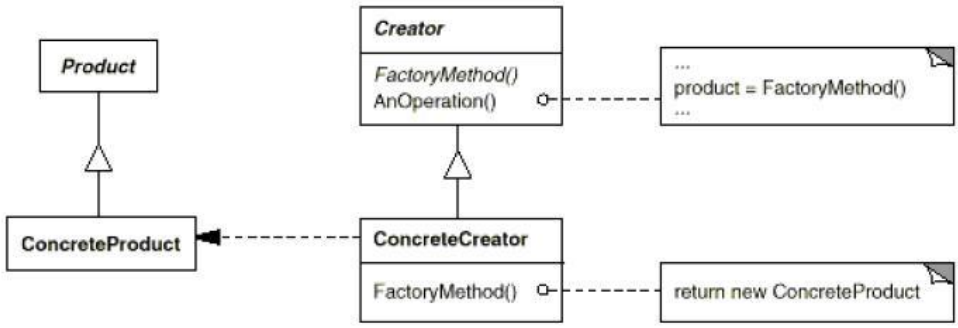


Figura 3 - Diagrama de Classes do Padrão Factory Method [5]

3.2 Padrão DAO

O padrão DAO encapsula o modo de acesso aos dados, tornando a obtenção dos dados transparente para as classes de negócio, e permite a utilização de fontes de dados distintas. Este padrão elimina a necessidade de conhecimento prévio da fonte de dados e dos tipos de *drivers* e *interfaces* utilizados para acesso à persistência.

No diagrama da Figura 4, é apresentado o relacionamento entre os participantes deste padrão.

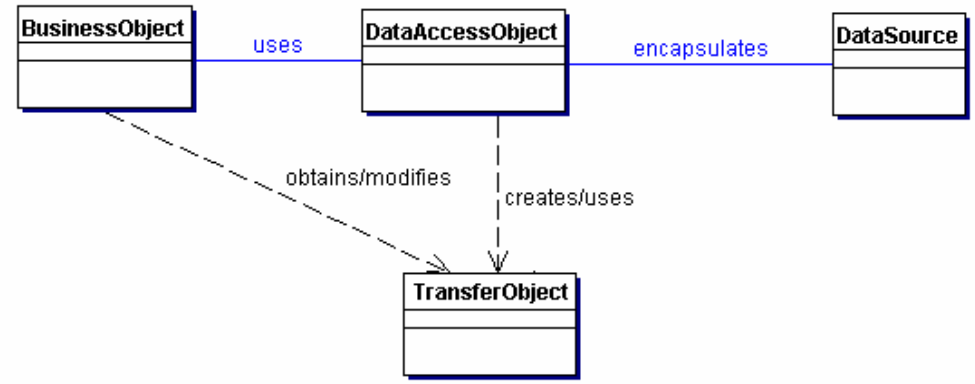


Figura 4 – Diagrama de Classes do Padrão DAO [19]

A classe *BussinessObject* é a classe de negócio que utilizará o padrão através de chamada à classe *DataAccessObject*. A classe *DataAccessObject* implementa a forma de acesso as dados, retornando um objeto da classe *TransferObject*. A classe

DataSource representa a forma de acesso que é encapsulada pela classe *DataAccessObject* e, normalmente, refere-se a uma classe que implementa a interface JDBC, no caso de um sistema J2EE.

4. O *Framework* FuncTest

O *framework* FuncTest está sendo utilizado dentro de um processo de desenvolvimento aderente ao nível 2 do CMMI com o objetivo de melhorar a produtividade na automatização de testes funcionais.

O FuncTest foi desenvolvido com a utilização da ferramenta *Rational* XDE *Tester* [13, 15, 17], mas poderá ser adaptado a ferramentas similares que também gerem código Java.

O *Rational XDE Tester* utiliza uma instância do Eclipse, um ambiente de desenvolvimento integrado (IDE - *Integrated Development Environment*). Nela, cada *script* é gravado ou programado como uma classe *Java*. A ferramenta dispõe de um mapa de objetos que registra uma pontuação para cada objeto do *browser*, baseado em suas características. Isto permite que a ferramenta identifique os objetos durante a execução de um *script*.

O FuncTest permite que os *scripts* contenham somente ações de interação com interface de usuário. Desta forma, outras responsabilidades, como chamada a outros *scripts*, acesso a dados e controle de erros, são desvinculadas dos *scripts*, fortalecendo o reuso e a manutenibilidade dos mesmos.

Nossa proposta prevê a criação de suítes de teste que contém um número arbitrário de casos de teste. Cada caso de teste é composto por vários *steps*. Aplicando-se a técnica *Keyword-driven*, associa-se cada passo do caso de teste a uma *keyword*.

Utilizando a arquitetura MVC, um controlador associa cada *keyword* ao *script* que deverá ser executado.

A arquitetura do FuncTest (Figura 5) permite o controle independente de chamada dos *scripts*. Desta forma, evitamos que um *script* chame outro *script*, eliminando o acoplamento entre eles. A seqüência de execução dos *scripts* é dada através da seqüência de *steps* do caso de teste.

Como propõe a técnica *Data-driven*, os dados de teste são mantidos desvinculados dos *scripts*. Assim, eles poderão ser mais facilmente mantidos e reutilizados. Cada *step* está associado aos dados de teste necessários á sua execução.

O *framework* permite que erros, os quais interromperiam a execução da suíte de testes, possam ser manipulados por *scripts* especificamente construídos com esta finalidade.

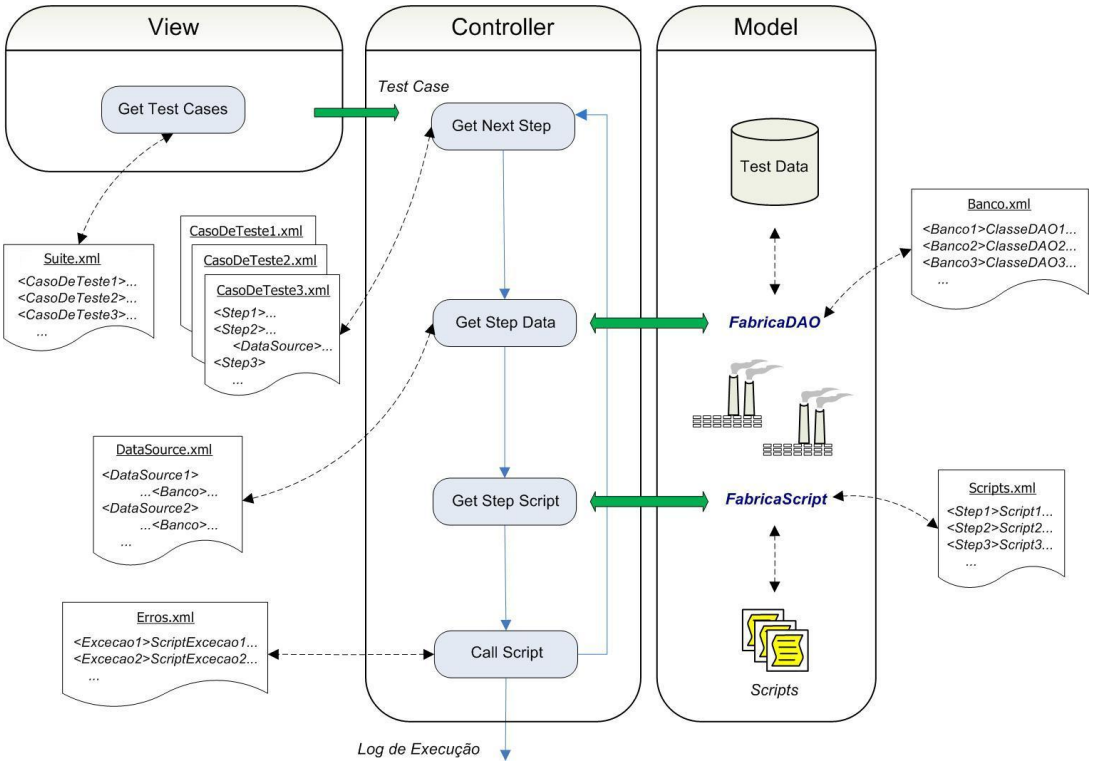


Figura 5 – Diagrama de Classes do Padrão DAO [19]

Descreveremos na seção 4.1 os pacotes do *framework* distribuídos segundo a arquitetura MVC.

O *framework* é configurado através de um conjunto de arquivos XML. Assim, só é necessário manipular código java na elaboração dos *scripts*. Na seção 4.2, descreveremos os arquivos de configuração do *framework*.

O funcionamento do FuncTest será abordado na seção 4.3, através de um diagrama de seqüência. Na seção 4.4, será realizada uma correspondência entre as classes preconizadas nos padrões e aquelas implementadas no *framework*.

4.1 Pacotes do *Framework*

O *framework* é formado basicamente pelos pacotes *Model*, *View* e *Controller*, e um conjunto de arquivos de configuração. O diagrama de classes a seguir (Figura 6) mostra uma visão resumida dos referidos pacotes, contemplando as suas principais classes.

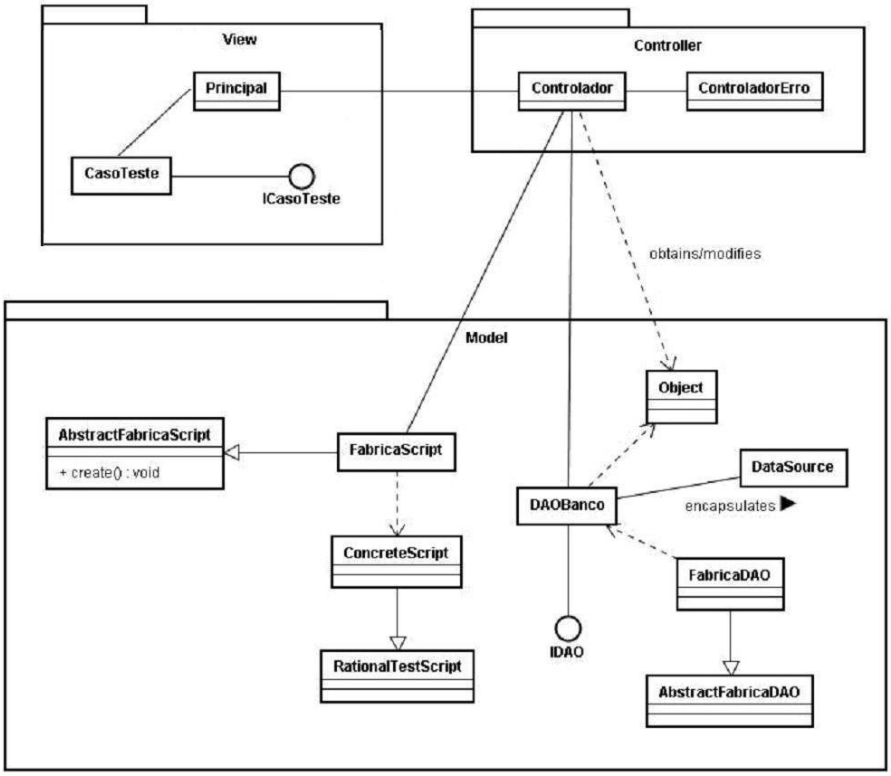


Figura 6 – Diagrama de Classes Resumido do FuncTest

O funcionamento do *framework* será detalhado adiante, onde poderá ser esclarecido o papel de seus participantes.

4.2 Arquivos de Configuração

Os arquivos de configuração usados pelo FuncTest seguem o formato XML. São eles:

- Suite.xml;
- NomeDoCasoDeTeste>.xml;
- Scripts.xml;
- Erros.xml;
- Banco.xml;
- DataSource.xml.

Os dois primeiros arquivos, Suite.xml e <NomeDoCasoDeTeste>.xml, persistem os dados da camada de visão.

Os arquivos Scripts.xml e Erros.xml estão associados à camada de controle. O arquivo Scripts.xml representa a tabela onde o Controlador associa cada *step* do caso de

teste ao *script* que será executado. O arquivo Erros.xml representa a tabela que registra os *scripts* de tratamento de erro a serem chamados pelo Controlador de erros.

A camada de modelo é constituída por um conjunto de *scripts* e seus respectivos dados.

Em cada *step*, está indicado seu respectivo *datasource*. O arquivo Datasource.xml indica as informações associadas a cada *datasource*, incluindo o nome do SGBD a ser utilizado. Para cada SGBD, o arquivo Banco.xml informa uma classe correspondente no padrão DAO.

A estrutura de cada um desses arquivos será detalhada a seguir.

Suite.xml

É formado pelo conjunto de *tags* denominadas CasoTeste. Cada uma desta *tags* possuirá duas seções:

Nome: representa o nome do caso de teste;

Arquivo: representa o nome do arquivo XML com a descrição do caso de teste.

```
<CasoTeste>
        <Nome>NomeDoCasoDeTeste1</Nome>
        <Arquivo>NomeDoCasoDeTeste.xml</Arquivo>
</CasoTeste>
<CasoTeste>
        <Nome>NomeDoCasoDeTeste2</Nome>
        <Arquivo>NomeDoCasoDeTeste.xml</Arquivo>
</CasoTeste>
```

Quadro 1 - *Template* para criação do arquivo Suite.xml

<NomeDoCasoDeTeste>.xml

Para cada caso de teste, deve ser criado um arquivo correspondente. Estes arquivos serão formados pelo conjunto de *tags* denominadas s*tep*. Cada *tag step* possuirá quatro elementos:

Nome: representa o nome do *step*;

DataSource: esta *tag* é opcional. Ela define uma fonte de dados (*datasource*) para os dados de teste. É definida quando o *step* contemple a inclusão de dados na interface de usuário;

Numero: esta *tag* é opcional. Corresponde ao número de vezes que o *script* será executado. Deverão existir dados de teste distintos a serem utilizados para cada execução do *script*.

```
<step>
        <Nome>Step01</Nome>
        <Tipo>Script</Tipo>
</step>
<step>
       <Nome>Step02</Nome>
       <DataSource>NomeDataSource00</DataSource>
       <Numero>000</Numero>
</step>
```

Quadro 2 - *Template* para criação dos arquivos <NomeDoCasoDeTeste>.xml

Scripts.xml

Contém uma *tag* indicando o *script* correspondente a cada *step* dos casos de teste.

```
<scripts>
        <Step01>Script01</Step01>
        <Step02>Script02</Step02>
</scripts>
```

Quadro 3 - *Template* para criação dos arquivos Scripts.xml

DataSource.xml

Para cada *step* cuja *tag* <Tipo> possua o valor "*Datasource*", deverá ser descrita a respectiva fonte de dados. O arquivo Datasource.xml é um repositório que contém as descrições das diversas fontes de dados utilizadas. No template do Quadro 5 temos um *datasource* denominado <NomeDataSource01>. As fontes de dados são descritas por *tags* que representam a *string* de conexão, o nome do banco, a tabela de dados, o *login* e a senha do *datasource*, o nome do *driver*, o número de colunas a ser selecionado na tabela e o nome da coluna para a ordenação da consulta.

```
<DadosDeTeste>
        <NomeDataSource01>
                <Conexao>StringConexao</Conexao>
                <Tabela>NomeTabela</Tabela>
                <Login>Login</Login>
                <Driver>NomeDriver</Driver>
                <Senha>Senha</Senha>
                <Colunas>NumeroColunasTabela</Colunas>
                <Banco>Banco01</Banco>
                <Ordenado>ColunaParaOrdenacao</Ordenado>
        </NomeDataSource01>
</DadosDeTeste>
```

Quadro 4 - *Template* para criação dos arquivos DataSource.xml

Banco.xml

Para cada banco de dados, contém uma *tag* <NomeBanco> que indica a classe DAO correspondente.

```
<bancos>
       <ORACLE>DAOOracle</ORACLE>
       <SQLSERVER>DAOSQLServer</SQLSERVER>
       <DB2>DAODB2</DB2>
       <ACESS>DAOAccess</ACESS>
       <INTERBASE>DAOInterbase</INTERBASE>
       <MYSQL>DAOMySQL</MYSQL>
       <SYBASE>DAOSybase</SYBASE>
</bancos>
```

Quadro 5 - *Template* para criação dos arquivos Banco.xml

Erros.xml

É formado por um conjunto de *tags*, cada uma associando uma Exceção ao *script* que será executado para o tratamento correspondente.

```
<erros>
       <Excecao1>Nome.do.Script.que.trata.a.Excecao1 </Excecao1>
       <Excecao2>Nome.do.Script.que.trata.a.Excecao2 </Excecao2>
</erros>
```

Quadro 6 - *Template* para criação dos arquivos Erros.xml

A apropriada configuração dos arquivos XML permitirá a associação dos passos dos casos de teste aos *scripts* a serem executados com os respectivos dos dados de teste, se for o caso.

4.3 Uso dos Padrões na Automação de Testes Funcionais

Temos duas utilizações do *Factory Method* no *framework* FuncTest. Na primeira utilização, temos uma *Factory* para criação de *scripts*. Neste caso, o *framework* delega à *Factory* a decisão de qual *script* será criado. Na segunda utilização, temos uma *Factory* para geração de um Objeto DAO. Objeto DAO é utilizado para recuperar os dados de teste.

Os diagramas de classe a seguir (Figuras 7 e 8) evidenciam respectivamente os dois usos do padrão *Factory Method* no *framework*.

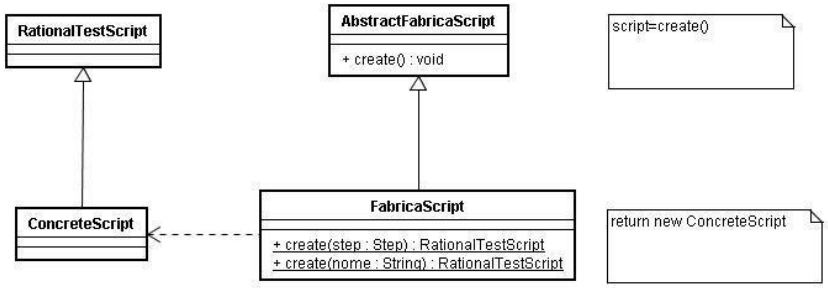


Figura 7 – Uso do Padrão Factory Method para a criação de *scripts* de teste

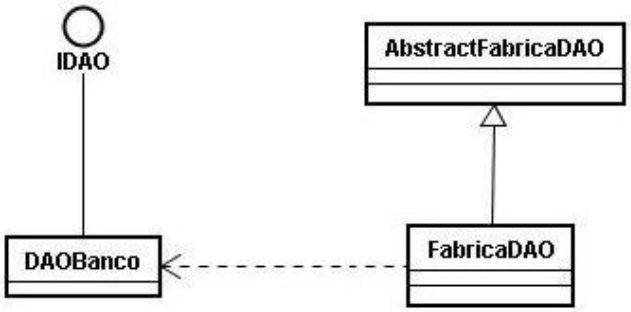


Figura 8 – Uso do Padrão Factory Method para a criação dos objetos DAO

Na tabela 1, é apresentada uma correspondência entre os participantes do padrão *Factory Method* e as classes do *framework* que implementam este padrão.

Participantes do Padrão *Factory Method*	**Participantes Correspondentes no FuncTest**	
	Fábrica Script	**Fábrica DAO**
Product	RationalTestScript	IDAO
ConcreteProduct	ConcreteScript	DAO
Creator	AbstractFabricaScript	AbstractFabricaDAO
ConcreteCreator	FabricaScript	FabricaDAO

Tabela 1 – Correspondência entre participantes do *Factory Method* e participantes do FuncTest

O diagrama de classes abaixo (Figura 9) evidencia o uso do padrão DAO no *framework*.

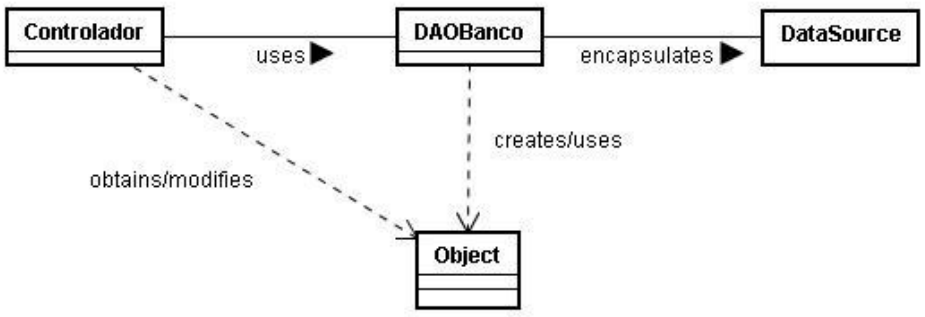


Figura 9 – Uso do Padrão DAO para selecionar a massa de teste

Na tabela 2, é apresentada uma correspondência entre os participantes do padrão DAO e as participantes do *framework* que implementam este padrão.

Participantes do Padrão DAO	Participantes Correspondentes no FuncTest
BusinessObject	Controlador
DataAccessObject	DAOBanco
DataSource	DataSource
TransferObject	Object[]

Tabela 2 – Correspondência entre participantes do DAO e participantes do FuncTest

4.4 Funcionamento do *Framework*

O funcionamento do FuncTest será apresentado com base no Diagrama de Seqüência a seguir (Figura 10).

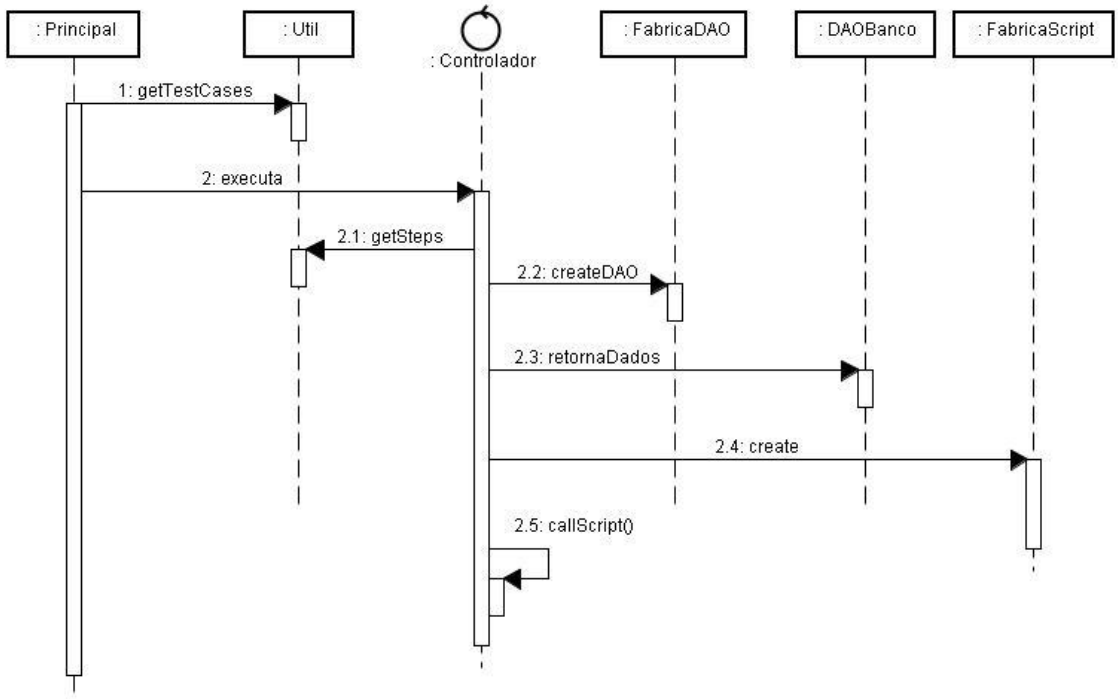


Figura 10 – Diagrama de Seqüência Simplificado do framework FuncTest

A classe *Principal* inicia o processo de automação requisitando à classe *Util*, responsável por deserializar os arquivos XML, os casos de teste contidos no arquivo Suite.xml. Em seguida, através de um *loop*, para cada caso de teste, a classe *Principal* cria um objeto *casoTeste*, instância da classe *CasoTeste*, e solicita a sua execução ao método *executa()* da classe *Controlador*, passando o objeto criado.

Para cada objeto casoTeste, a classe *Controlador* requisita à classe *Util* a leitura o arquivo <NomeDoCasoDeTeste>.xml recuperando os *steps* do caso de teste correspondente.

Em cada *Step*, se existir uma *tag* <DataSource>, serão recuperadas as informações da fonte de dados armazenadas no arquivo Datasource.xml. Dentre estas informações, consta, na *tag* <Banco>, o SGBD utilizado.

Em seguida, é solicitada à classe *FabricaDAO* a criação do objeto *DAOBanco* apropriado ao SGBD informado. Então a classe *Controlador*, através do método *retornaDados()*, solicita ao objeto *DAOBanco* os dados de teste a serem utilizados no *script*.

Após recuperar os dados do banco, a classe *Controlador* solicita à classe *FabricaScript* o *script* associado ao *step* corrente. A *FabricaScript*, então, retorna o *script* informado no arquivo Scripts.xml. Portanto, existe um construtor virtual de objetos DAO, e outro para a criação de objetos *Script*.

Caso tenha sido informado o número de execuções necessárias ao *script*, é feito um *loop* que irá repetir a execução do *script* para os dados retornados no objeto DAO correspondente.

Havendo falha durante a automação, a classe *Controlador* requisita o tratamento do erro à classe *ControladorErro*, a qual escolherá o *script* informado no arquivo Erros.xml.

Na seção 5 serão apresentados a conclusão e os trabalhos futuros.

5. Conclusão

Scripts gerados a partir de ferramentas *Record and Playback* normalmente são pouco reutilizáveis e pouco manuteníveis. De forma a minorar este problema, *frameworks* que utilizam as técnicas *Data-driven* e *Keyword-driven* têm sido propostos para a automação de testes funcionais.

Neste artigo, foi apresentado um *framework* para a automação de testes sistêmicos funcionais denominado FuncTest, que, além das técnicas *Data-driven* e *Keyword-driven*, utiliza a arquitetura *MVC* e os padrões *Factory Method* e DAO para aprimorar a manutenibilidade e a reusabilidade de projetos de automação. O uso de padrões em *frameworks* para automação de testes funcionais conduzidos através de ferramentas *Record and Playback* representou um enfoque inovador da utilização de padrões.

A experiência de utilização do *framework* mostra que este é efetivo quando automatizamos telas de entrada, consulta e saída de dados. Funcionalidades que envolvem a execução de processos *batch* ou telas com regras de negócio complexas nem sempre são passíveis ou viáveis de automação.

Nos casos onde a interação entre usuário e a tela da aplicação é facilmente mapeada, o *framework* se torna uma ferramenta ágil e efetiva. Utilizando o *framework* em testes de regressão, diversos erros já foram encontrados nas aplicações testadas. As dificuldades encontradas na utilização do *framework* estão relacionadas ao processo de configuração dos arquivos XML.

O uso do FuncTest num processo de desenvolvimento aderente ao nível 2 CMMI trouxe, dentre outros, os seguintes benefícios ao projeto de teste:

- desacoplamento, através da arquitetura MVC, entre passos do caso de caso de teste e *scripts* a serem executados;
- independência entre *scripts* e dados de teste, segundo a técnica *Data-driven*;
- modularização dos *scripts*, através da técnica *Keyword-driven*, com o controle independente de chamada de suas chamadas e conseqüente redução do acoplamento entre eles;
- facilidade de inclusão de novos casos de teste nas suítes de teste;
- transparência no acesso aos dados e independência em relação ao SGBD utilizado, obtidas através do padrão DAO;
- controle de erros centralizado;
- melhoria no tempo de execução de testes, uma vez que *scripts* de tratamento de erro são invocados automaticamente reduzindo as paradas por exceções;
- configuração do *framework* através de um conjunto de arquivos XML, tornando necessária a manipulação código java apenas na elaboração dos *scripts*;
- melhoria da reusabilidade e da manutenibilidade de *scripts*;
- melhor legibilidade de código.

A utilização do *framework* exige, obviamente, as seguintes contrapartidas:

- curva de aprendizado para utilização do *framework*;
- custo para a configuração do *framework*.

Estão sendo realizadas medições que indiquem a produtividade obtida com a utilização do *framework*.

Além da análise dos resultados obtidos com as medições, como trabalho futuro, adaptaremos o *framework* para a utilização da técnica *model-based testing* [9, 14], possibilitando a geração automática de casos de teste.

6. Referências Bibliográficas

[1] MYERS, Glenford J. The Art of Software Testing. New York: John Wiley & Sons, Second Edition, 2004.

[2] FEWSTER, Mark, GRAHAM, Dorothy. Software Test Automation. Addison-Wesley Professional; 1st edition, 1999.

[3] DUSTIN, Elfriede. Effective Software Testing: 50 Specific Ways to Improve Your Testing. Addison-Wesley Professional; 1st edition, 2002.

[4] DUSTIN, Elfriede, RASHKA, Jeff, PAUL, John. Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley Professional; Bk&CD Rom edition, 1999.

[5] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[6] BUSCHMANN, Frank, MEUNIER, Regine, ROHNERT, Hans, SOMMERLAD, Peter, STAL, Michael. Pattern-Oriented Software Architecture: A System of Patterns. New York: John Wiley & Sons, 1996.

[7] SCHMIDT, Douglas, STAL, Michael, ROHNERT, Hans, BUSCHMANN, Frank. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. October 2000.

[8] ALUR, Deepak, MALKS, Dan, CRUPI, John. Core J2EE Patterns: Best Practices and Design Strategies, 2 Ed. California: Sun Microsystems, 2003.

[9] DALAL, S. R., JAIN, A., KARUNANITHI, N., BELLCORE, N., LEATON, J. M., LOTT, C. M., PATTON, G. C., HOROWITZ, B. M. Model-Based Testing in Practice. International Conference on Software Engineering, 1999.

[10] FANTINATO, Marcelo, et al. AutoTest – Um Framework Reutilizável para a Automação de Teste Funcional de Software. Simpósio Brasileiro de Qualidade de Software, 2004.

[11] Framework automation with IBM Rational Functional Tester: Data-driven. Disponível em http://www-128.ibm.com/developerworks/rational/library/05/1108_kelly/.

[12] Framework automation with IBM Rational Functional Tester: Keyword-driven. Disponível em http://www-128.ibm.com/developerworks/rational/library/06/0523_kelly/.

[13] Data Driven Testing: How to Create a Data Driven Test with XDE Tester. Disponível em http://www-128.ibm.com/developerworks/rational/library/384.html.

[14] I. K. El-Far and J. A. Whittaker, "Model-Based Software Testing". Encyclopedia of Software Engineering (edited by J. J. Marciniak). Wiley, 2001

[15] Testing Java and Web applications with IBM Rational XDE Tester. Disponível em http://www-128.ibm.com/developerworks/rational/library/390.html.

[16] PRESSMAN, Roger. Engenharia de Software, 5ª Edição. McGraw-Hill, 2002.

[17] Testing IBM Workplace with IBM Rational XDE Tester. Disponível em http://www-128.ibm.com/developerworks/lotus/library/xde-tester/.

[19] Core J2EE Patterns - Data Access Object. Disponível em http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.

[20] Kaner, C., "Improving the Maintainability of Automated Test Suites", Proceedings of the Thenth International Quality Week, 1997.

[21] Meszaros, Gerard. XUnit Test Patterns: Refactoring Test Code. Prentice Hall, 2007.

SugarLoafPLoP'2007 Proceedings

Patrocínio

Apoio

Realização

ISBN978-85-87837-13-4