Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming

Uirá Kulesza[1], Vander Alves[2], Alessandro Garcia[3]
Carlos J. P. de Lucena[1], Paulo Borba[2]

[1]PUC-Rio, Computer Science Department, Rio de Janeiro - Brazil
{uira, lucena}@inf.puc-rio.br

[2]Informatics Center, Federal University of Pernambuco, Recife - Brazil
{vra, phmb}@cin.ufpe.br

[3]Lancaster University, Computing Department, Lancaster - United Kingdom
garciaa@comp.lancs.ac.uk

Abstract. Object-oriented frameworks are nowadays a common and useful technology used in the implementation of software system families. Despite their benefits, over the last years many researchers have described the inadequacy of object-oriented mechanisms to address the modularization and composition of many framework features, thus reducing the extent to which a framework can be extended. The crosscutting nature of many framework features is identified as one of the main causes of these problems. In this paper, we analyze how aspect-oriented programming can help to improve the design, implementation, and extension of object-oriented frameworks. We propose the concept of Extension Join Points (EJPs) as a way of designing and documenting existing crosscutting extension points. EJPs improve framework extensibility, including superior composability of the framework core functionality with other modules or frameworks. Four case studies of frameworks from diverse domains are presented to illustrate our proposal. This paper also discusses lessons learned on the application of our approach to the development and extension of these frameworks.

1 Introduction

Framework technology plays nowadays a central role in the development of software product lines. Object-oriented (OO) frameworks enable systematic reuse-in-the-large by modularizing and composing one or more recurring features of a given domain, and by offering predictable extension options to the target applications. Framework extension is achieved in different ways, ranging from the selection of optional and alternative features to its integration with other complementary components and frameworks. However, some researchers [2, 19, 20] have recently described the inadequacy of OO mechanisms to address the modularization and composition of many framework crosscutting features, thus reducing the framework variability and integrability. This crosscutting phenomenon manifests itself in several manners: both in the core and variable parts of a framework, and also in the features being integrated as the system evolves.

Hence all framework benefits are hindered if there is no systematic approach to support the encapsulation and extension of crosscutting framework features through their development and instantiation processes. With the emergence of aspect-oriented

programming (AOP) [12], it is important to investigate the suitability of AOP mechanisms to promote enhanced variability and integrability of OO frameworks. AOP supports the encapsulation of crosscutting features into new modular units – the *aspects* – and their composition through the notion of *join points*. Recent work [1, 9, 14, 17] has started to explore the use of aspects to improve the isolation of crosscutting features encountered in the design of frameworks and product lines. Other authors have also examined the influence of AOP on different software integrability scenarios, such as COTS [18] and design patterns [3, 11]. However, there is no methodological foundation to support framework designers while using aspect-oriented mechanisms to improve the variability and integrability of their frameworks in the presence of crosscutting features.

This work briefly revisits well-known problems relative to OO framework modularity (Section 2). We then perform a systematic analysis of how AOP can help to address them, and enhance the framework implementation and extensibility. In particular, we present a systematic approach, based on the concept of Extension Join Points (EJPs), as a unified way of designing and documenting existing crosscutting extension points (Section 3). EJPs provide new means for extending framework core functionality, introducing optional and alternative crosscutting features, and integrating the framework elements with other components and frameworks. We further present a categorization of framework aspects that support the encapsulation of distinct types of crosscutting features. We describe four case studies of frameworks from different domains to illustrate the applicability of our proposal (Section 4). Lessons learned on the design and implementation of these frameworks are also discussed (Section 5).

2 Modularization Problems in OO Frameworks

We now briefly describe three major problems, previously discussed by other authors, showing how they negatively affect framework extensibility from different perspectives. We also illustrate representative symptoms of these problems in the design of the JUnit framework, which is a classical example in the framework literature.

Complexity of Object Collaboration. An OO framework defines a set of abstract and concrete reusable classes implementing a software family architecture. Complex collaborations between these classes must be implemented. These collaborations represent the common functionalities shared by several applications in the framework domain. Each framework class in general has to play different roles, which means that they need to collaborate with different classes in order to implement their different responsibilities [5, 23]. Therefore, understanding and maintaining the framework classes become a difficult task. Moreover, further introductions of framework variations and compositions with other software modules are hindered, as each class role cannot be treated as a separate framework feature as the framework evolves.

In the JUnit framework, for example, the primary purpose of the main classes is to execute a set of test suites and cases, and return a testing report. However, they have to play an additional role: a different feature related to the tracking of the execution of test cases and suites by those classes is superimposed in their code in order to notify GUI classes about the execution state (started, failed, ok, finished) of test cases.

Riehle et al [23] analyze the problems of complex object collaborations and their impact on framework design and integration. They show how the complexity of an OO

framework increases when its internal classes play different roles. The authors, however, propose the use of role modelling [23], which only partially addresses the plastering of multiple roles into classes as it is focused on the design level.

Inability to Modularize Framework Optional Features. Batory et al [2] discuss the difficulties of the framework technique to modularize optional features. An optional feature is a framework functionality that is not used in every framework instance. They illustrate some alternatives developers typically adopt to deal with this problem, such as: (i) to implement the optional feature in the code of concrete classes during the framework instantiation process; and (ii) to create two different frameworks, one addressing the optional feature and the other one without it. Accordingly, many framework modules need to be replicated just for the sake of exposing optional features. To summarize, the authors argue that frameworks are usually "overfeatured" [5], which means that several non-general functionalities can inevitably be part of the framework.

An analysis of available frameworks (such as JUnit and JHotDraw) makes it evident that the most common practice adopted in the implementation of framework optional features is the use of inheritance mechanisms to define additional behavior in the framework classes. These classes represent an existing framework feature to be extended. In the JUnit framework, for example, inheritance relationships are used to define a specific kind of test case as well as additional extensions to test cases and suites.

Crosscutting Feature Compositions in Frameworks Integration. Mattsson et al [19, 20] have analyzed the problems and causes related to the integration of OO frameworks. For each problem presented, they have also proposed several OO solutions. A combination of two frameworks, as described by the authors, can also be seen as the composition of a new set of features (represented as a framework) in the structure of another framework. As an example, suppose we need to extend the JUnit framework to send specific failures that occur to software developers. A specific test failure report could be send by e-mail to different software developers, every time a specific and critical failure happens. Imagine we have available an e-mail framework to support our implementation. The problem here is how we could implement this functionality in the JUnit framework. It involves the integration of the JUnit and e-mail frameworks. This composition could be characterized as crosscutting since we are interested to send a failure report by e-mail during the execution of the tests.

We have analyzed [17] the framework integration solutions presented by Mattson et al [19, 20]. Many of their OO solutions are invasive and bring several difficulties to the implementation, understanding and maintenance of the framework composition code. Our analysis was based on a case study with feature compositions involving four OO frameworks of varying complexity and addressing concerns from distinct horizontal and vertical domains [6]. The analysis has showed that from the 9 solutions described by those authors, 6 solutions have poor modularity and a crosscutting nature, and require invasive internal changes in the frameworks code.

3 Improving Framework Extensibility with Aspects

This section presents our approach to design and implement OO frameworks with aspects. Our approach deals with the framework modularization problems presented

previously (Section 2.1) by using AOP and the notion of extension join points (EJPs). EJPs also support the disciplined specification of additional opportunities for framework extensions. Sections 3.1 – 3.3 present the proposed approach by respectively describing the central concept of EJPs, describing different uses of aspects to improve framework extensibility, and presenting the achieved benefits. Section 3.4 presents an example of the approach applied to the JUnit framework.

3.1 Extension Join Points

The extension points or hot-spots of an OO framework are typically implemented as abstract classes or interfaces [7]. They allow extending the common collaboration behavior provided by the framework by providing specific implementations for their abstract methods. One of the difficult problems of framework development is that it requires dealing with many common and variable features pertaining to a domain. We can thus notice an increase on framework complexity and on many modularization and composition problems, such as those presented in Section 2.

In our approach, an OO framework specifies and implements not only its common and variable behavior using OO classes, but it also exposes a set of *extension join points* (EJPs) which can be used to also extend its functionality. The idea of EJPs is inspired by Sullivan et al's work [25, 10] on specification of crosscutting interfaces (XPIs). Similar to XPIs, EJPs establish a contract between the framework classes and a set of aspects extending the framework functionality. However, unlike XPIs, EJPs are adopted as a means to increase the framework variability and integrability. Thus, we propose to use the XPI concept in the context of framework development. EJPs can be used to three different purposes:

(i) to expose a set of framework events that can be used to notify or to facilitate a crosscutting integration with other software elements (such as, frameworks or components);

(ii) to offer predefined execution points spread and tangled in the framework into which the implementation of optional features can be included;

(iii) to expose a set of join points in the framework classes that can have different implementations of a crosscutting variable functionality.

In this context, EJPs document crosscutting extension points for software developers that are going to instantiate and evolve the framework. They can also be viewed as a set of constraints imposed on the whole space of available join points in the framework design, thereby promoting safe extension and reuse. A key characteristic of EJPs is that framework developers and users do not need to learn totally new abstractions to use them, as they can mostly be implemented using the mechanisms of AOP languages (Section 5.1).

3.2 Framework Core and Extension Aspects

Our approach promotes framework development as a composition of a core structure and a set of extensions. A framework extension can be: (i) the implementation of optional or alternative framework features; or (ii) the integration with an additional component or framework. The composition between the framework core and the framework extensions is realized by different types of aspects. Each aspect defines a crosscutting composition

with the framework by means of its exposed EJPs. Next, we describe the main elements of our approach:

(i) *framework core* – implements the mandatory functionality of a software family. Similar to a traditional OO framework, this core structure contains the frozen-spots that represent the common features of the software family and hot-spot classes that represent non-crosscutting variabilities from the domain addressed;

(ii) *aspects in the core* – implement and modularize existing crosscutting concerns or roles in the framework core. They represent the traditional use of AOP to simplify the understanding and evolution of the framework core;

(iii) *variability aspects* – implement optional or alternative features existing in the framework core. These elements extend the framework EJPs with any additional crosscutting behavior;

(iv) *integration aspects* – define crosscutting compositions between the framework core and other existing extensions, such as an API or an OO framework. These elements also rely on the EJPs specification to define their implementation.

Figure 1 shows the design of an OO framework with aspects following our approach. As we can see, both variability and integration aspects can only act in the EJPs provided by the framework and they must respect all the constraints defined by them. This brings systematization to the framework extension and composition with other artifacts.

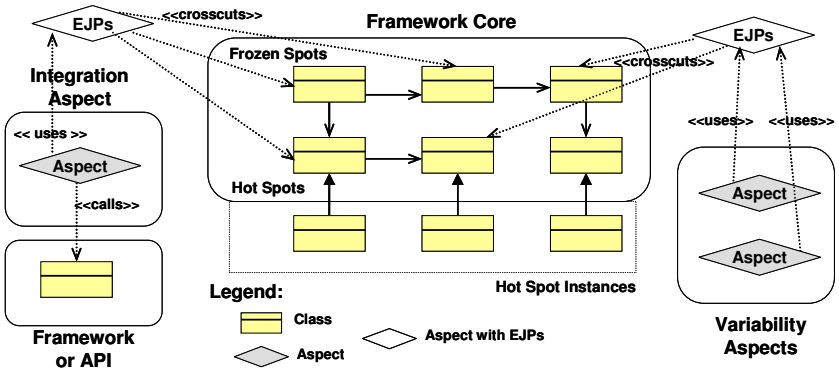

Figure 1. Elements of our Framework Development Approach

3.3 Benefits

Table 1 describes the benefits brought by each type of aspect in our framework development approach. It also indicates how the core, variability, and integration aspects address each of the modularization problems in framework development and evolution discussed in Section 2. As pointed out in the table, the use of internal framework aspects also has a positive impact on the framework variability and integrability. They facilitate the specification of EJPs because core aspects promote modularization of the internal class roles. Therefore, they offer additional join points to be exploited in extension scenarios.

Table 1. Framework Development Approach Elements

Approach Element	Benefits	Modularization Problem Addressed
Aspects in the Core	- Simplify the understanding and evolution of the framework core • Modularize existing crosscutting concerns or roles in the framework core. - Facilitate the design of EJPs	Crosscutting Roles and Concerns
Extension Join Points	- Systematize the framework extension and composition by promoting safe framework reuse • Enable the composition between framework core and extensions. • Encapsulate the framework and exposes only proper join points.	Tight Coupling between Core and Extensions
Variability Aspects	- Facilitate the framework reuse and extension. • Modularize optional and alternative framework features. • Make it possible to plug and unplug optional or alternative features.	Optional or Alternative Features
Integration Aspects	- Facilitate the framework reuse and composition. • Modularize the framework composition with other extensions. • Make it possible to plug and unplug crosscutting framework composition.	Crosscutting Framework Compositions

3.4 JUnit Example

This section illustrates the use of the proposed approach in the context of the JUnit framework. Although JUnit presents a well-modularized architecture, we have found some modularization problems [16] hindering its future extension/evolution. In the context of other complex and large-scale frameworks, these problems can cause architecture erosion after a while. Due to space limitation, we have briefly mentioned JUnit problems in Section 2.

The main purpose of the JUnit framework is to allow the design, implementation and execution of a set of test suites and cases for any Java application. It is especially useful to implement unit tests, but it can also be used to implement integration tests between modules. The JUnit framework implementation is composed of the following components:

(i) Tester: defines the framework classes responsible for specifying the basic behavior to execute test cases and suites. The main hot-spot classes available in this component are `TestCase` and `TestSuite`. The framework users extend these classes in order to create specific test cases to their applications;

(ii) Runner: this component is responsible for offering an interface to start and track the execution of test cases and suites. JUnit provides three alternative

implementations of test runners: a command-line based user interface (UI), an AWT based UI, and a Java Swing based UI;

(iii) Extensions: responsible for defining functionality extending the basic behavior of the JUnit framework. Examples of available extensions include: a test suite to execute tests in separate threads, a test decorator to run tests repeatedly, and a test setup class that allows specifying initial and final configuration of specific tests.

Following our approach, in the JUnit OO framework we can consider the classes of the Runner and Tester components as the *framework core* (Figure 2). They provide the main functionalities needed to execute the test cases and suites, as follows: (i) the definition of a test case or suite to be executed; (ii) the execution of a selected test case or suite; and (ii) the collection and visual presentation of the test results. The JUnit core offers three abstract classes (`TestCase`, `TestSuite` and `BaseTestRunner`) representing traditional extension points of the framework.

The JUnit framework core must expose its extension join points in order to allow the composition of crosscutting extensions into its basic functionality. Figure 2 presents the `TestExecutionEvents` aspect, which exposes a set of EJPs of the JUnit framework. It exposes the following join points: (i) test case execution; (ii) test suite execution; and (iii) initialization of test runners. We have chosen these join points because they represent relevant events in the test execution functionality of the JUnit core that can be of interest to framework extensions.

Different extensions can be implemented to add new functionality into the JUnit EJPs. We can assume that the tracking of the test case execution by the Runner component is not initially addressed by the framework core. It is necessary to codify an *aspect in the core* working as an observer. Since this is an aspect in the core, it can be codified to intercept directly framework classes (such as presented in Figure 2) or it could reuse the join points exposed by the `TestExecutionEvents` aspect mentioned above and notify the instance of the Runner executing about the current state of the test case (initiated, finalized, failed). Figure 2 shows three aspects (`ObserverTestExecution`, `AWTUIObserver`, `TextUIObserver`) addressing this functionality considering different Runner classes available.

The implementation of the functionality of the original JUnit Extension component can also benefit from its EJP. Different variability aspects, as presented in Figure 2, can be codified to add the testing extensions into the JUnit EJPs, such as: (i) to run test cases or test suites repeatedly (RepeatAllTest aspect); (ii) to execute them in separate threads (ActiveTestSuite aspect); and (iii) to introduce some additional behavior before or after the test case or suite (TestCaseDecorator and TestSuiteDecorator aspects). In this case, we could implement easily these aspects by reusing the join points exposed in the aspect TestExecutionEvents. It is also possible to codify aspects to affect just specific test cases or suites defined to test an application. Finally, the JUnit EJPs can also be used to compose it with other OO frameworks. Figure 2 shows, for example, the MailNotification integration aspect responsible for monitoring the test execution, building specific test reports and sending them by e-mail to specific developers. An e-mail framework could be composed with the JUnit framework to provide that functionality by means of an integration aspect.

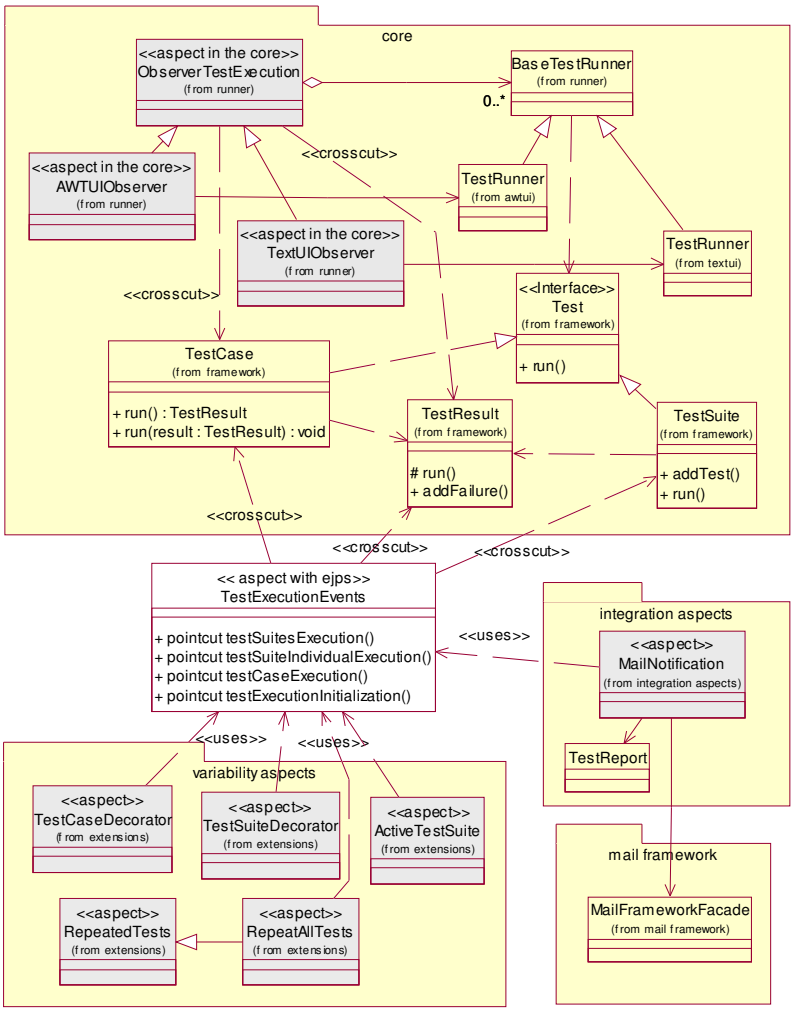

Figure 2. Aspect-Oriented Design of the JUnit Framework

4 Case Studies

Our approach has emerged from our experience in different domains, through a process of continuous interaction and refinement between case studies and the approach itself. In this context, the approach was employed in the development of frameworks in the following domains: (i) J2ME games [1]; (ii) multi-agent systems (MASs) [9, 14]; and (iii) measurement support for product quality control [17]. Table 2 summarizes the framework core, EJPs, variability and integration aspects of the three case studies. Due to space limitation, in this section we describe our experience only in the domain of J2ME Games. For a complete description of the implementation of EJPs and framework extensions for these case studies, please refer to [15].

Table 2. Case Studies Overview

Domain	Framework Core	EJPs	Variability and Integration Aspects
J2ME Games	*Game engine*, a state machine defining the game core structure and workflow, including: (i) handling of user interaction and elapsed time: (ii) game actors state update; (iii) game actors rendering; (iv) game screen management.	- Image initialization and usage; - Drawing of specific images; - Game startup and changing screens.	Variability and Integration aspects were defined to implement: (i) *croma* optional feature; (ii) alternative drawing feature; (iii) optional image loading optimization feature.
MASs	Our AspectT agent framework implements the core internal structures and behaviors of autonomous software agents, including: (i) the knowledge elements – beliefs, actions, plans, goals; (ii) the management of goals and plan execution; (iii) the thread scheduling; (iv) adaptation of agent knowledge.	- Reception and sending of messages; - Reception of external stimulus; - Initialization of goals; - Exceptions in plan executions; - Initialization and finalization of plan/action executions.	Integration aspects were defined to integrate the framework core with: (i) 2 alternative inter-agent communication platforms; (ii) 2 alternative code mobility platforms; Variability aspects were included to enhance the agent behavior with: (i) optional learning capabilities; (ii) different roles and collaboration protocols.
Measurement for Quality Control	The Measurement Framework defines a process for quality control composed of the following steps: (i) product data collection phase; (ii) data analysis and product categorization phase; (iii) actuation phase – actions are performed over the products according to their categorization.	- Activation of triggers (event of product processing initialization); - Activation of sensors (event of product data collection); - Activation of actuators (event of product processing finalization).	Different integration aspects were defined to compose the measurement framework with other ones, as follows: (i) a GUI framework presenting visually information about the measurement process; (ii) a Statistical framework calculating statistical data about the measurement process; and (iii) a Persistence framework storing information about the items processed and the statistical data.

4.1 J2ME Games

J2ME games are mainstream mobile applications of considerable complexity [1]. Their overall structure and behavior are defined by a framework known in this domain as the *game engine*. Essentially, this is a state machine whose state change is driven by elapsed time and user input through the device keypad. State changes affect the state of various drawing objects (*game actors*) and how they interact. Then, these objects are drawn

again after such state changes. Typical hot-spots of this framework include some abstract classes defining basic drawing capability for game actors.

The game engine must also expose its EJPs in order to allow the composition of crosscutting extensions in its basic functionality. Some interesting EJPs are the following: (i) how images are initialized and used; (ii) drawing of specific images; (iii) game startup and changing screens. We have chosen these EJPs because they represent relevant events that can be of interest when extending the game engine core workflow.

Based on these EJPs, we provide implementations for optional and alternative features. For example, EJP (i) can be composed with a *variability aspect* to implement the *croma* optional feature (decorative images of the game screen, for example clouds rolling in the background). This feature is optional in the product line comprising a number of devices, since some are resource-constrained and thus this feature should not be selected for such devices. Accordingly, in the game assets, such feature amounts to images files in the resource directory; in the code, they are declared, and loaded into fields, which are updated in their state, and then drawn, this comprising a number of different code blocks in different classes. Therefore, the implementation of this optional feature is crosscutting. By exposing this crosscutting nature within EJPs and implementing the feature within a variability aspect, we provide an appropriate way to document this crosscutting in design and to implement it modularly.

As another example, EJP (ii) can be composed with integration aspects to implement the alternative features for drawing some images. Specific images may be drawn at various locations and, under certain circumstances, may be transformed (rotated, flipped), which may be accomplished by using fresh new images or by transforming the original ones by calling device proprietary drawing API. By exposing these EJPs and composing them with integration aspects, we provide modular implementation for the interaction between the core game and the device-specific API.

Lastly, EJP (iii) can be composed with variability aspects to implement an optional optimization feature: images are loaded on demand when changing to the next screen. This is also a policy for resource-constrained devices; other devices just load all images at once during game startup. By exposing these EJPs, we can explicitly show where the optional optimization might be composed.

5 Discussion and Lessons Learned

This section provides some discussion and some lessons learned based on our experience on applying our proposed approach (Section 3) to several frameworks [1, 3, 9, 14, 15, 16, 17], such as those ones described in Section 4.

5.1 EJPs Documentation

An important issue to consider when developing the framework EJPs is how to document them. The way they are documented can help developers to implement more easily their framework extensions. Different ways of documentations can be used which complement each other. We advocate the use of a combination of programming language, textual and visual documentation to make EJPs explicit.

The XPI proposed approach presents the documentation of exposed join points of an application using AspectJ [13] source code directly [10] or a textual representation of elements composing the XPI [25]. The documentation based on AspectJ source-code

defines aspects declaring public pointcuts that expose the join points of an application. Many invariants that must be respected by the extensions can also be codified in AspectJ to guarantee their automatic verification [10]. This way of documentation for the EJPs is very useful because developers that are interested in extending the framework can directly reuse those public pointcuts. Besides, the syntactic specification of the invariants also helps to control inappropriate interactions between the framework and extensions in the EJPs. In the JUnit example in Section 3.4, the `TextExecutionEvents` aspect represents the syntactic specification of the framework EJPs specifying a set of public pointcuts related to the testing execution in the framework core. The documentation of EJPs based on source code can be complemented with textual based representation. Sullivan et al [25] present a XPI textual representation which can also be used to document our EJPs.

Although Sullivan et al [10, 25] XPI proposal can be useful to document the EJPs, new properties must also be considered in their documentation. A framework crosscutting extension can be codified to affect only join points of specific hot-spot instances. In the JUnit framework, for example, only specific test cases or test suites of a framework instance can be considered to be extended. In order to address these situations, we also need to document specific join points that will be completely available only after the framework instantiation. The pointcuts specifying these join points need be customized with the name of framework instance classes. Thus, it is necessary to distinguish between EJPs allowing changes to the framework internal classes and those ones allowing direct extensions of concrete implementations of hot-spots in a framework instance. The source code based documentation, for instance, can present examples of how the pointcuts that affect directly hot-spot classes can be adapted to affect only specific hot-spot instances.

Many OO framework documentation techniques (such as, cookbook approaches) emphasize the use of examples to show how to instantiate the framework extension points. In the case of EJPs, we believe it is also fundamental to present examples in how they can be used to compose any additional functionality in the framework. First, because AOP has not became a widely adopted technology yet. Second, because crosscutting composition is supposed to be more difficult to understand than composition based on inheritance. Section 5.2 explores the documentation of examples of framework crosscutting extensions using implementations based on traditional design patterns. Finally, the documentation of EJPs could also offer any visual representation to make them more explicit. Currently, we are analyzing if it is possible to use aSideML crosscutting interfaces [4] to offer a UML-based notation to represent the EJPs.

5.2 Implementation of Variability and Integration Aspects

In our approach, all framework crosscutting extensions are attached to the framework core using integration and variability aspects. Each aspect introduces a crosscutting behavior in a specific set of EJPs. The aspects play a specific role related to the way they extend the framework. They can, for example, play the role of observers of internal framework events to notify any external OO extension. They can mediate the communication between the framework classes and other OO extensions on particular EJPs. They can also decorate EJPs with new and optional implementation of features.

Thus, many aspects can play the role of traditional design patterns [8] with the aim of extending the framework core.

Hanneman & Kiczales [11] have demonstrated how the implementation of many design patterns can be well modularized using aspects. The implementation of many integration and variability aspects can follow the Hanneman & Kiczales' guidelines based on the played role by the framework extension being composed. In the JUnit example showed in Figure 2, the variability aspects decorate the execution of test suites and test cases with new optional features, whereas the `MailNotification` integration aspect mediates the communication between the framework testing elements and a mail framework. Examples of aspect implementations based on design patterns can work as an effective documentation of the available ways to extend the EJPs.

Other issues relative to the implementation of variability and integration aspects deserve attention but are not explored in this paper due to space limitation, such as: (i) the composition of aspects on the same EJPs – since different aspects can add new functionality on the same EJPs, it is necessary to determine if there is any execution order of them or any conflict on their composition on the same points; and (ii) the modularization of the common and specific code of aspects [17] – also a clear separation between the extensions affecting directly the framework and those affecting only specific framework instances could be done.

5.3 Finding EJPs

One of the main difficulties in the development of OO frameworks is to find their flexible points or hot-spots. Domain analysis methods [6] and experience on the development of applications in the same domain are techniques used to find commonalities and variabilities of a framework. EJPs can also be considered framework hot-spots. They represent flexible points in the execution of specific framework scenarios that can have a crosscutting extension inserted. EJPs are modeled as events or transition states occurring during the execution of the framework functionalities. Thus, these events or transitions states are dependent on the domain and functionalities being addressed by the framework. Table 2 shows a set of EJP examples derived from our case studies in different domains.

While some heuristics to find EJPs such as the identification of relevant events and transition states in the framework functionalities can help, we believe that current domain analysis methods and techniques need to be extended to support the modeling of crosscutting relations between features in early development stages. This can anticipate the modeling of EJPs. An EJP could be modeled as the integration point of the crosscutting relationship between two features [16].

6 Related Work

Our concept of EJPs is inspired by Sullivan et al's work [25] on specification of crosscutting interfaces (XPIs). XPIs abstract crosscutting behavior, isolating aspect design from base code design and vice-versa. Continuing this work, Griswold et al show how to represent XPIs as syntactic constructs [10]. EJPs play a similar role to XPIs, but specifically in the context of framework development, by exposing a set of framework events for notification and crosscutting composition, and by offering predefined execution points for the implementation of optional and alternative features.

Feature oriented approaches (FOAs) have been proposed [24] to deal with the encapsulation of program features that can be used to extend the functionality of existing base program. Batory et al [2] argue the advantages that feature-oriented approaches have over OO frameworks to design and implement product-lines. Mezini and Ostermann [21] have identified that FOAs are only capable of modularizing hierarchical features, providing no support for the specification of crosscutting features. These researchers propose CaesarJ [22], an AO language that combines ideas from both AspectJ and FOAs, to provide a better support to manage variability in product-lines. The work of those authors has a direct relation to our work, since we believe that the design of product-line architectures may benefit from the composition and extension of different frameworks using integration and variability aspects.

In the middleware domain, Zhang and Jacobsen [26] propose the Horizontal Decomposition method (HD), a set of principles guiding the definition of functionally coherent core architecture and customizations of it. The core is customized with aspects implementing orthogonal functionality. Unlike our approach, which uses EJPs to achieve bi-directional decoupling of the core from its extensions in the framework context, HD has a principle explicitly embracing obliviousness, whereby the core should be completely unaware on the aspects.

7 Conclusions

In this paper we presented an approach for the design and implementation of traditional OO frameworks with aspects. Our approach addresses the modular implementation of framework optional features and enables framework crosscutting composition with other OO extensions. The exposition of only specific framework join points brings systematization to the process of extension and composition of the framework. We have presented some case studies that demonstrate the benefits brought by the approach. Furthermore, some initial guidelines to the use and adoption of the approach were discussed. As a future work, we plan to refactor other existing object-oriented frameworks to validate quantitatively the benefits of our approach.

Acknowledgements. We would like to thank the members of Software Productivity Group at Federal University of Pernambuco for valuable suggestions for improving this paper. This research was partially sponsored by FAPERJ (grant No. E-26/151.493/2005), CNPq (grants No. 552068/2002-0, 481575/2004-9 and 141247/2003-7), MCT/FINEP/CT-INFO (grant No. 01/2005 0105089400), and European Commission Grant IST-2-004349: European Network of Excellence on AOSD (AOSDEurope).

References

[1] V. Alves, P. Matos, L. Cole, P. Borba, G. Ramalho. “Extracting and Evolving Mobile Games Product Lines”. Proceedings of SPLC'05, LNCS 3714, pp. 70-81, September 2005.

[2] D. Batory, Rich Cardone, and Y. Smaragdakis, Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC), pp. 227-248, Denver, August 1999.

[3] N. Cacho, et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proceedings of AOSD'06, Bonn, Germany,2006.

[4] C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena. Taming Heterogeneous Aspects with Crosscutting Interfaces. Journal of the Brazilian Computer Society, 2006 (to appear).

[5] W. Codenie, et al. "From Custom Applications to Domain-Specific Frameworks", Communications of the ACM, 40(10),October1997.

[6] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley,2000.

[7] M. Fayad, D. Schmidt, R. Johnson. Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, September 1999.

[8] E. Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[9] A. Garcia. From Objects to Agents: An Aspect-Oriented Approach. PhD Thesis, Computer Science Department, PUC-Rio, April 2004.

[10] W. Griswold, et al, "Modular Software Design with Crosscutting Interfaces", IEEE Software, Special Issue on Aspect-Oriented Programming, January 2006.

[11] J. Hannemann, G. Kiczales. Design Pattern Implementation in Java and AspectJ. Proceedings of OOPSLA'02, 2002, pp.161-173.

[12] G. Kiczales, et al. Aspect-Oriented Programming. Proc. of ECOOP'97, Finland, 1997.

[13] G. Kiczales, et al, "Getting Started with AspectJ," Comm. ACM, vol. 44, pp. 59-65, 2001.

[14] U. Kulesza, et al. "A Generative Approach for Multi-Agent System Development". In "Software Engineering for Multi-Agent Systems III". LNCS 3390, pp. 52-69, 2004.

[15] U. Kulesza, et al. "Implementing Framework Crosscutting Extensions with XPIs and AspectJ", Technical Report, PUC-Rio, Brazil, April 2006.

[16] U. Kulesza, A. Garcia, F. Bleasby, C. Lucena. "Instantiating and Customizing Product Line Architectures using Aspects and Crosscutting Feature Models". Proceedings of the Workshop on Early Aspects, OOPSLA'2005, San Diego,2005.

[17] U. Kulesza, A. Garcia, C. Lucena. "Composing Object-Oriented Frameworks with Aspect-Oriented Programming", Technical Report, PUC-Rio, Brazil, April 2006.

[18] A. Kvale, et al. A Case Study on Building COTS-Based System using Aspect-Oriented Programming. Proceedings of SAC'2005, pp. 1491-1498.

[19] M. Mattson, J. Bosch, M. Fayad. Framework Integration: Problems, Causes, Solutions. Communications of the ACM, 42(10):80–87, October 1999.

[20] M. Mattsson, J. Bosch. Framework Composition: Problems, Causes, and Solutions. In [7], 1999, pp. 467-487.

[21] M. Mezini, K. Ostermann: "Variability Management with Feature-Oriented Programming and Aspects". Proceedings of FSE'2004, pp.127-136, 2004.

[22] M. Mezini, K. Ostermann. "Conquering Aspects with Caesar". Proc. of AOSD'2003, pp. 90-99, March 17-21, 2003, Boston, Massachusetts, USA.

[23] D. Riehle, T. Gross. "Role Model Based Framework Design and Integration". Proceedings of OOPSLA'1998, pp. 117-133, Vancouver, BC, Canada, October 18-22, 1998.

[24] Y. Smaragdakis, D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM TOSEM, 11(2): 215-255 (2002).

[25] K. Sullivan, et al. Information Hiding Interfaces for Aspect-Oriented Design, Proceedings of ESEC/FSE´2005, pp.166-175, Lisbon, Portugal, September 5-9, 2005.

[26] C. Zhang, H. Jacobsen. "Resolving Feature Convolution in Middleware Systems". Proceedings of OOPSLA'2004, pp.188-205, October 24-28, 2004, Vancouver, BC, Canada.