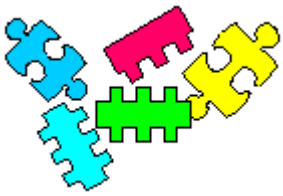
Titel: Proceedings of the GCSE 2000 Young Researchers Workshop

URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/



PROCEEDINGS OF THE

YOUNG RESEARCHERS WORKSHOP

at
GCSE 2000

http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

Organizers:

Elke Pulvermüller, Andreas Speck, Kai Böllert, Detlef Streitferdt
and Dirk Heuzeroth

Workshop Moderator : Jim Coplien

Workshop Panelists:

Paul Bassett, Mario Jeckle, Cristina Lopes, Bertrand Meyer

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop

URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

Aims and Objectives

The workshop aims at providing a platform for young researchers to present their work. These presentations will be commented by experienced panelists. The workshop serves as a forum for the participants to get in contact with other researchers in the field and to become familiar with other approaches and future research topics.

This workshop is part of the Second International Symposium on Generative and Component−Based Software Engineering (GCSE 2000) which is co−hosted with the NET.OBJECTDAYS 2000 and supported by the Working Group Generative and Component−Based Software Engineering of the German "Gesellschaft für Informatik" (GI).

The Papers of the Workshop were published in Proceedings of the
NET.OBJECTDAYS, October 2000

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop

URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

List of Papers

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop

URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

Code Generation from UML models [1]

Ákos Frohner, Zoltán Porkoláb
Department of General Computer Science,
University Eötvös Loránd, Budapest
E−mail: Akos.Frohner@elte.hu, Zoltan.Porkolab@elte.hu

Keywords: UML model, code generation, component, metadata, multi−paradigm
Classification: 3 year´s work (PhD almost completed)

1 Introduction

Creating a generic, object−oriented, component−based, transactional business system, which covers the whole lifecycle, is possible only with integration of commercial tools, component technologies, recently developed class libraries and using code generators. Most of the recently used tools for development techniques are focusing only of one the layers of the model from the code generation point of view. As a consequence the inter−layer connections are lost in the generated code.

2 Design Model Generation

We describe a code generator technique, which uses an analysis model as a starting point and generates the following *model layers* or *sub−models* directly:

- Presentation layer (clients), which provides the user interface.
- Business logic layer. This can be a physically and geographically distributed N−tier architecture:
  - Business objects: correspondents of business entities. They implement the basic behaviour of the entities including persistency.
  - Business processes: a series of operations on the business objects.
- Storage layer: either relational or object−oriented databases, special storage systems and legacy systems.

The sub−models are specialised for the target domain: they contain interfaces, classes and methods specific to the target language.

To allow modifications at each level we give an algorithm to re−generate the sub−models. The formally described graph transformation preserves the modifications made in the sub−model and merges the changes from the analysis model.

3 Code Generation

Based on the experiences with 4GL systems it is obvious the need to provide *customisation* in the generated code as well. The source code can be geenrated in the same manner as the sub−models are generated and re−generated.

The sub−model generation algorithm can be implemented as a transformation on the model graph, but one does not have this clean architecture if the target graph is projected into source code. Although it is possible to reverse engineer source code into model graphs, but it is a never ending battle to follow the changes of the target languages in these tools.

We intend to delegate the actual graph transformations on the source code level to the most appropriate tools, to the compilers itself, thus we offer a multi−paradigm approach [1]. We let the developer to choose the best solution for her or his target language and problem domain:

- Polymorph code sections: the calling part requests a general behaviour, the called code section substitutes a specialised version according to the context. [2]
- Run−time parameterisation of the generated code: the calling part requests a specialised behaviour using a metadata parameter in the routine call.
- Aspect weaving: the modifications (new aspects) of the code are placed in a separated source file, which is woven into the generated code before the compilation. [3]

References

[1] Jim Coplien: Multi Paradigm Design for C++, 1998, Addison− Wesley, ISBN: 0−201−82467−1

[2] Barbara Liskov: Data Abstraction and Hierarchy, SIGPLAN Notices 23, 5. May 1988

[3] Gregor Kiczales: Aspect Oriented Programming, AOP Computing surveys 28(es), 1996 p.154

[1] This work has been supported by the Grants OMFB ALK−00229/98

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop

URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

FlexiGen

a framework design for a flexible generator

Tobias Lindig
Institut für Praktische Informatik und Medieninformatik
Technische Universität Ilmenau
Germany
T.Lindig@gmx.de
http://www.theoinf.tu−ilmenau.de/~tlindig/

Keywords: Framworkdesign FlexiGen, Software−Generator, UML, re−use
Classification: 6 month´s work (diploma thesis)

1 Introduction

The result of my thesis is an design for a White−Box−Framework for the generation of software, which can be used as model in a Model−View−Controller architecture. I named this design *FlexiGen* as a short form for flexible generator. *FlexiGen* decouples processing unit and source of information, permits late dynamic loading and combining of both and enables a later extension or adjustment at minimum expenditure. The design is a combination of the Design Patterns Bridge, Singleton, Adapter, Abstract Factory, Factory Method and Template Method.

2 Motivation

In the near future the software engineering will distinguish between technical architecture and business logic. The technical architecture can be re−used in many projects and the business logic describes the individual peculiarities of a problem. The conflation can be carried out by a generator.

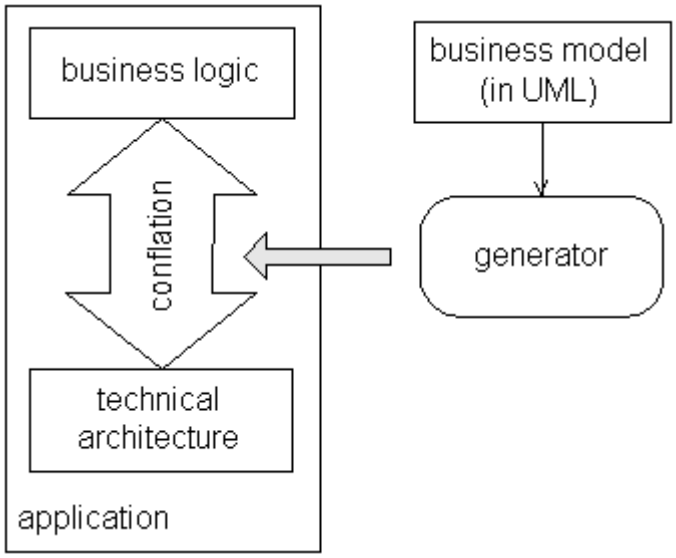


Figure 1: Assignment of generator.

To do so, every supplier of a technical solution has to provide a suitable generator. But the problem is, that there many different CASE−Tools on the market with different interfaces. To support all these CASE−Tools the supplier either needs a generator for each or he simply uses a generator−framework based on *FlexiGen.*

3 Advantage of *FlexiGen*

FlexiGen can be used to build modular generators which get an UML class diagram as input and generates program code for different languages like C++, Java or DCOM as output. It allows to support different input−sources merely by implementing an appropriate adapter module and if do so the new adapter module can be used by all generator modules for the new input immediately.

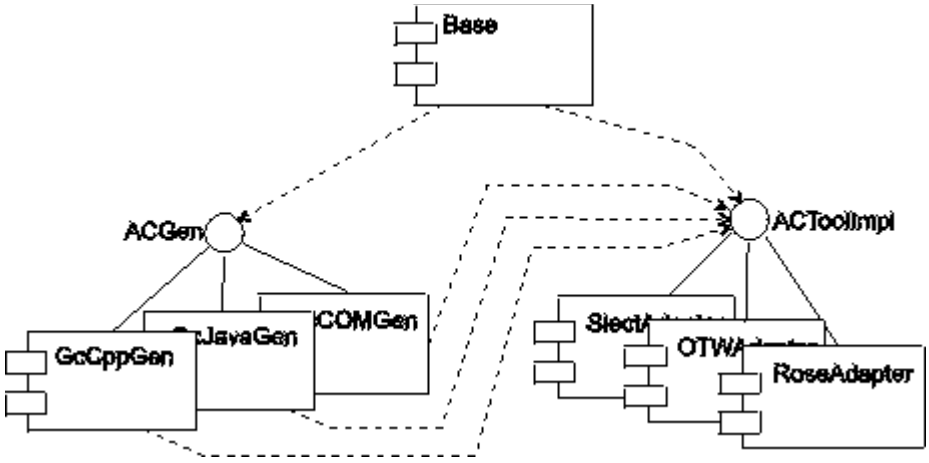


Figure 1: Modules of FlexiGen−based generator

Every adapter module implements the interface "ACToolImpl" that is used by the code generator modules for reading the needed information of the business model.

4 Further topics of the diploma thesis

A set of research projects and products in the field of the generative and component−based software engineering are introduced and the conversion of the items of a UML class diagram to C++ − program code is described. The introduction section deals with the problems of software development and maintenance; furthermore it describes what is meant by "software quality" and how it can be improved. Special attention is thereby applied to the re−use of software and special techniques which support this procedure.

The paper is written in German and can be downloaded at URL:

http://www.theoinf.tu−ilmenau.de/~tlindig/Diplom/FlexiGen.html

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop
URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

A C++ Implementation of Tabu Search for *k*−Cardinality Tree Problem Based on Generic Programming and Component Reuse

M. Josep Blesa − Fatos Xhafa
Departament de LSI
Universitat Politècnica de Catalunya
Campus Nord, Mòdul C6
Jordi Girona, 1−3
08034−Barcelona, Spain.
E−mail: {mjblesa,fatos}@lsi.upc.es

Keywords: *k−Cardinality Tree, Tabu Search, Generic Programming, Reuse, Components.*
Classification: 1 year´s work (PhD)

1 Introduction

Generic programming and reuse of components have already proved very important to the software process in general. There is an increasing interest to these topics even from the area of combinatorial optimization, where the researchers tends to use *ad hoc* implementations for the problems; this interest can be understood from the advantages of the generic programming and reuse of components such as time savings, flexibility, robustness etc. In combinatorial optimization we are oftenly found with *methods* for sub−optimally solving optimization problems. In this work we concern, the Tabu Search method, a well−known meta−heuristic [2] that has proved successful in sub−optimally solving a whole bunch of optimization problems. The ingredients of such method are the same for any problem to which one would like to apply the method. It is, therefore, quite interesting to have a *generic program* or a kind of *template* for Tabu Search from which one could derive instantiations for any problem of interest. This approach have been considered in our recent work [1] where we give a *skeleton* for the Tabu Search method implemented in C++. Here we show how to obtain an implementation of Tabu Search method for a concrete problem, namely the *k*−Cardinality Tree based on the *generic skeleton* for Tabu Search and component reuse.

The *k*−Cardinality Tree Problem consists in finding, in a given undirected weighted graph, a subtree of minimal weight with *k* edges, introduced by Hamacher et al. [3] and it is important to both theory and industrial applications (oil field leasing, layouts, partitioning, telecommunications and matrix decomposition.) Among several heuristics applied to the problem, there is also the Tabu Search method applied to *k*−Cardinality Tree by Jörnsten and Løkketangen [4].

2 Our results

In this work we give a new implementation of the Tabu Search method for the *k* −Cardinality Tree. One can naturally ask why this new implementation? Our main motivations were, first, to obtain a new implementation with as *little effort* as possible, and, second, we wanted our implementation be *flexible*, so we could easily change several parts of the implementation without affecting the implementation as a whole. We remark that the requirement on the flexibility is very important in the context of Tabu Search method since several ingredients of the method can be implemented in different ways. In order to match these requirements, we address an implementation for the problem based on the *generic skeleton* for the Tabu Search method and component reuse. By instantiating the *k*−Cardinality Tree, we have observed the easy of use of the skeleton, the flexibility, robustness and considerable savings in time. It is generally believed that an efficient implementation of TS should incorporate as much knowledge of the problem as possible. In spite of this we show that our implementation even though obtained through a generic skeleton, matches the results of the specific implementation (Jörnsten and Løkketangen, 1997).

The Design of the Tabu Search Skeleton was done after a careful review of known implementations for the method. The basic ingredients of the method are introduced into C++ classes according to a logical definition in the context of the Tabu Search method. The basic idea behind the skeleton is to allow the user to instantiate any combinatorial problem of interest by only defining the most important problem−dependent features. Elements related to the inner algorithmic functionality of the method are hidden to the user. The classes forming the skeleton are divided into a public and a private part. The private part contains elements related to the inner functionality of the TS method, and the public part contains elements related to the problem−dependent part. The public part must be completed by the user, so after filling the problem−dependent features, the problem can be solved using the Tabu Search method. The generality of our skeleton runs in two directions: the TS skeleton permits the user to instantiate any problem of interest by simply *filling in* the public part and the skeleton itself is generic enough to solve any problem if its problem−dependent elements can be described.

Reusable Components. Although the general idea of the Tabu Search algorithm is well known some details remain unstandardized, for instance the intensification and diversification of the search. In the literature both are freely introduced *ad hoc* to the schema depending on the problem to solved; that leads to subtlety different TS algorithms. Because we wanted a unique generic and powerful enough form of the Tabu Search algorithm, we introduced these functionalities to the basic Tabu Search. The private part of TS skeleton implements the generic Tabu Search algorithm and it is completely reusable for all the instantiations. In a certain sense, the main procedure is the most important reusable component of our approach. Other reusable components are the Statistics (useful statistics), State (information on the state of algorithm), TabuList (efficient use of memory). The instantiation of a concrete problem is done by completing

the public part (by implementing what the problem is, how is a solution, what is a move, etc.) Some of these definitions can be reused from other instantiations, e.g. both the k−Cardinality Tree and the Traveling Salesman are represented by an undirected weighted graph, so in this case, the class representing the problem can be reused.

Different implementations for the problem can also be generated by varying some definitions and structures from an already implemented version e.g. by changing the Tabu List structure, the intensification and/or the diversification strategy etc. The user has to just redefine a couple of methods in a class. So we have two levels of re−usability: in the method itself and in the instantiations. The re−usability of the method is due to the standardization of the Tabu Search algorithm. The second level refers to the instantiations already done.

Experimental Results. Table 1 resumes some of results obtained by our *TSKtree* instantiation with the ones obtained by the implementation of [4]. The instance name *gxx−4−yy* means graphs of *xx* vertices and *yy* indicates the index of the instance (we generated and proved different instances of the same configuration).

instance	#vertices	#edges	Best Fitness of JLTS	Best Fitness of TSKTree
g25-4-01	25	50	219	219
g25-4-04	25	48	620	620
g50-4-01	50	98	466	463
g50-4-03	50	99	580	577
g75-4-02	75	150	307	296
g75-4-03	75	149	421	426
g100-4-03	100	200	412	412
g100-4-04	100	199	442	442
g200-4-04	200	399	310	321
g200-4-05	200	399	380	360
g400-4-03	400	799	302	306
g400-4-04	400	800	315	314
g1000-4-02	1000	1999	321	292
g1000-4-05	1000	2000	284	280

Table 1: Comparison of our *TSKTree* with *JLTS* (Jörnsten and Løkketangen) implementation.

3 Conclusions

We present a C++ implementation of Tabu Search method for the Minimum *k* −Cardinality Tree problem. Tabu Search method is a well known heuristic for sub−optimally solving optimization problems. This method is also applicable to Minimum *k* −Cardinality Tree −an important problem to both theory and industrial applications. Our implementation is based on generic programming and re−usability of several components for Tabu Search method. We have compared our program with other *ad hoc* implementations for the problem and have observed a good performance of our implementation. Moreover, our implementation represents time savings, flexibility and robustness mainly due to the component reuse.

Acknowledgment

We thank the Barcelona Team of Mallba Project for support while conducting this research. We are grateful to A. Løkketangen, M. Ehrgott and H. Hamacher for providing us with their benchmarks and executable codes for *k*−Cardinality Tree.

References

1 M.J. Blesa and F. Xhafa. A C++ Implementation of a Skeleton for Tabu Search Method. Technical Report LSI−00−47−R, Dept. de LSI, UPC, 2000.

2 F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Op. Res.*, 5:533−549, 1986.

3 H.W. Hamacher, K. Jörnsten, and F. Maffioli. Weighted K−Cardinality Trees. Technical Report 91.023, Dept. di Elettronica, Politecnico di Milano, 1991.

4 K. Jörnsten and A. Løkketangen. Tabu Search for Weighted *k*−Cardinality Trees. *Asia−Pacific J. of Op. Res.*, 14(2):9−26, 1997.
Research partially supported by the ALCOM−FT (IST−99−14186) and CICYT Proj. TIC1999−0754−C03.

Full paper may be downloaded at URL:

http://www.lsi.upc.es/~fatos/papers/ktree10Ag.ps

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop
URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

Olena: a Component−Based Platform for Image Processing, mixing Generic, Generative and OO Programming

Alexandre Duret−Lutz
Epita R&D Laboratory
14−16 rue Voltaire
94276 Le Kremlin−Bicêtre
France
alexandre.duret−lutz@lrde.epita.fr

Keywords: generic programming, lazy compilation, bridge static−dynamic
Classification: MSC student work

1 Introduction

This paper presents Olena, a toolkit for programming and designing image processing chains in which each processing is a component. But since there exists many image types (different structures such as 2D images, 3D images or graphs, as well as different value types) the platform has been designed with genericity and reusability in mind: each component is written as a generic C++ procedure, a−la STL.

Other libraries, such as Khoros [Kon94] have a different approach where a processing component contains an implementation for each type supported by the library. This makes code maintenance hard and this prevents easy addition of new image types.

Still, Olena is not only a generic component library [Jaz95], it shall contains additional tools such as a visual programming environment (VPE). Those tools may be programmed in a classical object−oriented fashion (using operation and inclusion polymorphism) which may seems antagonist with the generic programming paradigm used in the library.

Section 2 outlines the architecture of Olena and elaborates more on the designs problems resulting from the use of generic components. Section 3 presents the solution chosen to address these problems.

2 Overview of the platform architecture

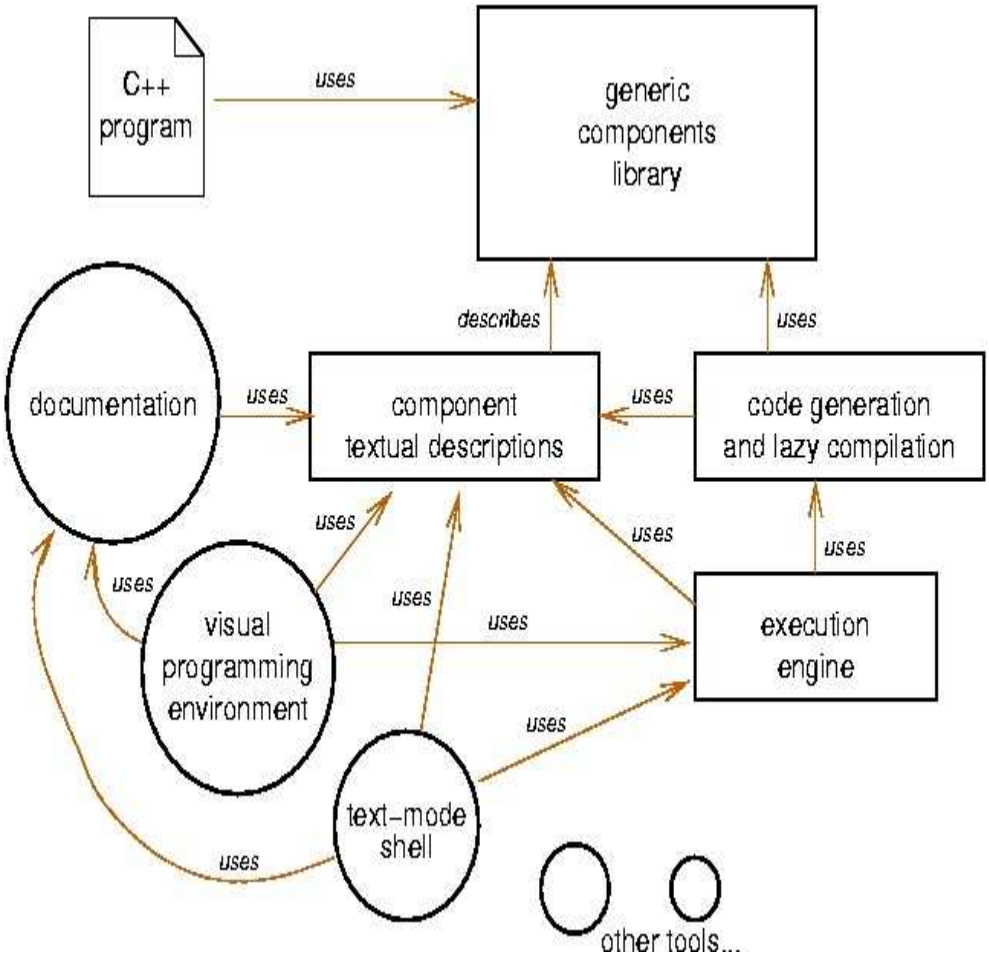


Figure 1: The Olena architecture.

Figure 1 shows the various parts of the architecture. A text−mode shell interface is needed to call the components and to prototype a processing, either interactively or via a script. Alternatively, a visual programming environment can be used to define the data flow diagrams for a processing chain. In each of these environment, components can be grouped to form reusable composites components (such as the one shown in figure 2); and once a processing has been successfully prototyped, the corresponding C++ code can optionally be generated.

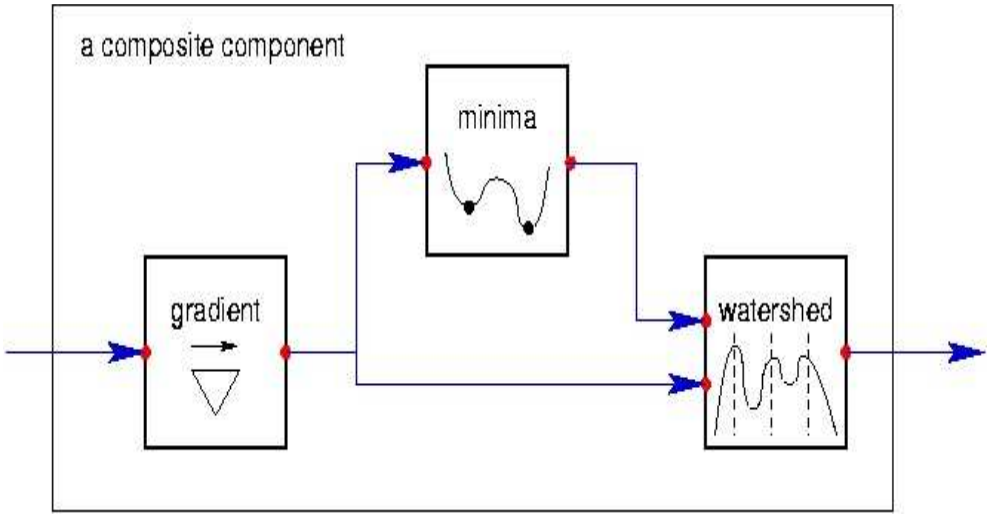


Figure 2: An image processing chain example. Here a composite component made of three atomic components.

Moreover, for each component the representation in the environment (visual aspect in the VPE, option switches in the text−mode shell) as well as on−line help shall be generated automatically. These require additional information that is recorded by a database of textual description of each components.

The kernel of the platform is a generic library which provide generic components and data. A component is a processing, and is written as a C++ generic procedure. Data are the entities that transit between components in the processing chain; the library provides image classes, neighborhoods, value types, histograms, *etc*. This library can be used directly by the end user in a C++ program, independently of the other tools of the platform. A component that adds a constant value to an image can be written as follow.

```
template< class Aggregate_Model >
void add (Aggregate_Model& input, typename
Aggregate_Model::data_type value)
{
  typename Aggregate_Model::iterator_type iter =
input.create_iterator ();

  for (iter.first (); !iter.is_done (); iter.next ())
    iter.current_item () += value;
}
```

The library is written fully in generic programming in order to achieve good performances, using techniques such a those described in [Vel99]. The iterator pattern used in the sample code is an adaptation to generic programming of the classical OO design pattern, and is well know in generic programming; but other design patterns can also be adapted [Ger00a].

An issue with generic components is that they lead to many instances (several binary variants). As long as it is used directly in a C++ program this is not a problem because the compiler will determinate and compile only the needed variants. But in interpreted environments such as the VPE, instances needed are not known until run−time. Moreover the number of available image types prevents the compilation of all variants once and for all (anyway since we want to allow the user the adds new types, this would not be a good answer). We detail our solution in the next section. Type constraints on the arguments types are also described for each component, because the interpreted environments will perform type checking at run−time.

3 Technical Solutions environment

As we said, at the VPE (or any other use interface) level only abstract data are handled. The only requirement is that in order to perform type checking, we need a `type()` method which return the real type of the data as a string.

The abstract type used is not part of the generic library since it makes sense only for the VPE. Figure 3 shows how a set of concrete data types can be grouped in a hierarchy. We encapsulate each data of type `T` in a class `DataProxy<T>` which inherit from the superclass `AbstractData`. This an application of the *external polymorphism pattern* [Cle96].

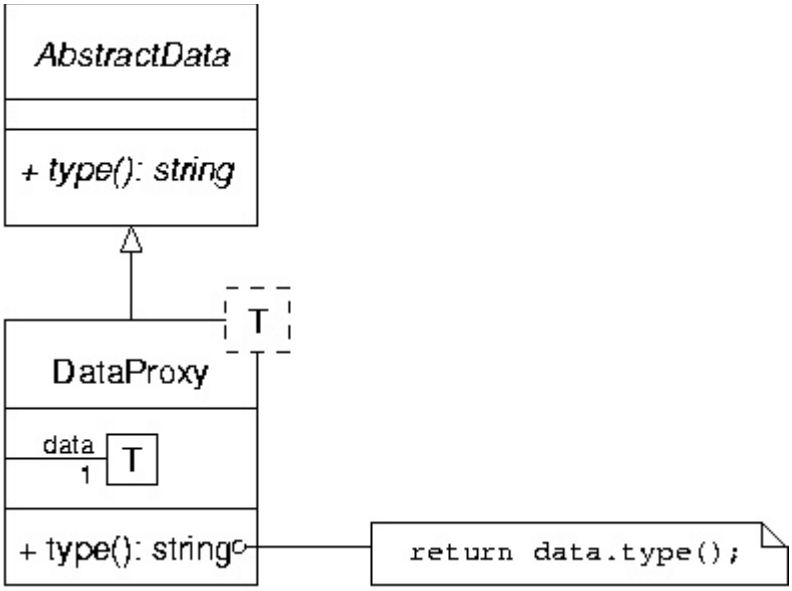


Figure 3: The `AbstractData` and `DataProxy` classes.

Thus, the VPE can handle abstract data, but in order to apply the algorithm, the data should be downcast back to its concrete type. If the set of types is short and fixed, this can be done using a `switch` construction. Unfortunately neither conditions are true: there exists a lot of types and the users can add new ones. Our solution is to use lazy compilation.

Data flow is managed by an Execution Engine Figure 2). When all the inputs of a components are ready (*i.e.* the upstream components have completed their work) the processing can start. But before it begins, several steps must be performed:

- type checking must be done on the inputs, this is done easily using the string−type of data and the rules given by the textual description of the component.
- input data must be casted to their real types (which are known as strings, at run−time).
- the generic algorithm must be instantiated for these types

The last two steps cannot be done at run−time. Therefore we generate, with help from the textual description, the code for an intermediate procedure which performs these two tasks; this code is compiled as a dynamic library, it is then loaded by the Execution Engine and finally executed.

As an exemple, consider the "add" component. "add" takes two inputs − let's label them `input1` and `input2` − and produce one output − named `output`. If this component receive a `Image2D<Float>` object and a `Float` value as inputs, the following code will be generated:

```
void add__Image2D_Float_ (Component& cmp)
{
  /* downcast the data */
  Image2D<Float> input1& =
    static_cast< DataProxy< Image2D< Float > >& >(cmp.entry
("input1")).data;
  Float input2& =
    static_cast< DataProxy< Float>& >            (cmp.entry
("input2")).data;
  Image2D<Float> output& =
    static_cast< DataProxy< Image2D< Float > >& >(cmp.entry
("output")).data;

  /* apply the algorithm */
  add  (input1, input2, output);
}
```

This procedure takes the OO version of the component as arguments and extracts its inputs and output with their concrete types in order to call the right variant of the "add" component. Finally, compiled variants of algorithms are cached, and therefore successive calls to these algorithms are not penalized by the slow compilation process. More over caching politics (such as component sharing by a group of user)can be applied.

4 Conclusion

The design presented here combine thee advantages of components oriented designs (flexibility, reusability) of both dynamic approach (ability to assemble component interactively) and static approach (good performance) thanks to the bridge between static and dynamic worlds we presented. Still, Olena is a work in progress. So far, our work has been focused mainly on the generic library [Ger00a] but the solution presented in the previous paragraph have been tested in practice.

References

[Cle96] Chris Cleeland, Douglas C. Schmidt and Timothy H. Harrison *External Polymorphism.* In Proceedings of the 3rd Pattern Languages of Programming Conference, Allerton Park, Illinois, September 4−6, 1996. http://www.cs.wustl.edu/~cleeland/papers/.

[Ger00a] Thierry Géraud, Yoann Fabre, Alexandre Duret−Lutz, Dimitri Papadopoulos−Orfanos, and Jean−François Mangin. *Obtaining genericity for image processing and pattern recognition algorithms.* In Proceedings of the 15th International Conference on Pattern Recognition (ICPR'2000), Barcelona, Spain, September 2000. To appear.

[Ger00b] Thierry Géraud and Alexandre Duret−Lutz. *Generic programming redesign of patterns.* In Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000), Irsee, Germany, July 2000. http://www.coldewey.com/europlop2000/.

[Jaz95] Mehdi Jazayeri *Component programming − a fresh look at software components* In Procedings of the 5th European Software Engineering Conference (ESEC'95), pages 457−478, September 1995.

[Kon94] K. Konstantinides and J.R. Rasure. *The Khoros software development environment for image and signal processing.* IEEE Transactions on Image Processing, vol. 3, no. 3, 1994, 243−252.

[Vel99] Todd L. Veldhuizen *Techniques for scientific C++*.August 1999. http://extreme.indiana.edu/~tveldhui/papers/techniques/

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop

URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

A Technique for Variability and Decision Modeling Facilitating the Incremental Introduction of Product Line Engineering

Dirk Muthig
Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D− 67661 KaiserlsauternAddress, Germany
muthig@iese.fhg.de
http://www.iese.fhg.de/Staff/muthig/

Keywords: Product Line Engineering, Technology Transfer, Variability Modeling
Classification: 3−4 year´s work (PhD complete in Summer/Fall 2001)

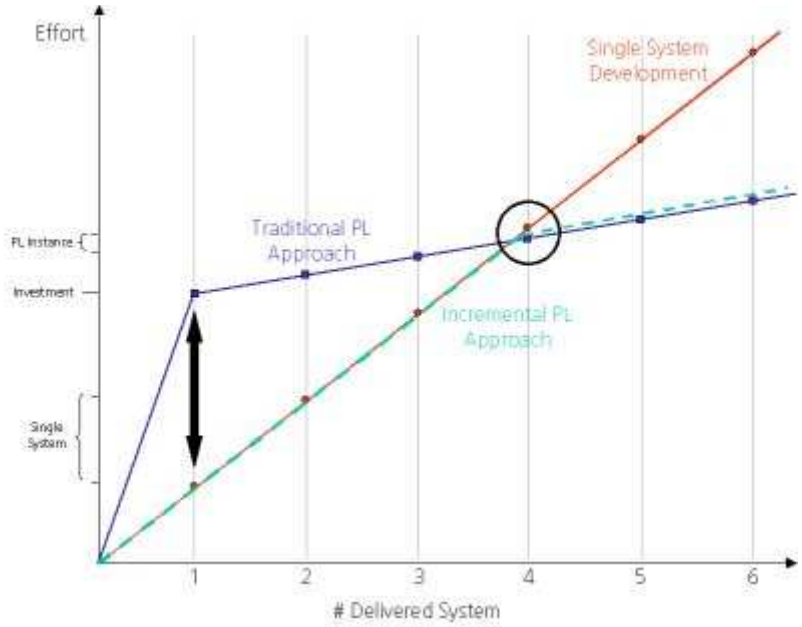
1 Introduction

Reuse is seen as a solution for many problems that software development organizations have. Unfortunately, it is hard to achieve in practice. The reason is that there is no general solution that has been proven to work in many different environments. Product line engineering is a solution that works in environments that develop multiple systems in the same application area. Its key idea is to define and build a product line infrastructure, which is generic and thus can be reused across systems in the same application area.
Unfortunately, existing product line engineering approaches are big endeavors that require one big and radical shift within an organization: nearly all processes and products, as well as the structure of the organization itself, must be adapted with respect to product line development. Such a shift towards product line engineering embodies huge risks for an organization that are difficult to control. Consequently, product line engineering is not an option for many organizations although it promises to tackle and solve their major problems.
In this thesis, a technique for modeling variabilities and decision modeling is defined and validated that enables the incremental introduction of product line engineering into running organizations without high up−front investments as it is typically required by traditional product line approaches. The main goal of an incremental introduction is to minimize and control the risks of introducing product line engineering and, thus, to enable more organizations to profit from the offered benefits.

2 Problem

The potential market for product line engineering approaches is huge because most of the environments develop product lines:

- They typically develop applications in one application domain (in which they are experts).
- They start with one product and then develop successively other products (or variants of former products). This is true independent whether the environment develops individual systems for different customers, one standard application, which is adapted to diverse contexts, or one professional application that supports all features in a domain from which variants with a limited functionality are derived .



Though there is this market for product line engineering approaches, they are hard to find in practice. The reason is that existing product line approaches require high and risky up−front investments without providing sufficient support for managing the related risks. The investments include adaptions of and changes to nearly all development activities and used products. Initially, this introduces new problems instead of immediately tackling and solving the concrete problems existing in an environment. Additionally, the details are often missing that are esstential in eventually applying an approach in a project environment. Last but not least, the existing approaches do not sufficently describe strategies and ways for introducing the approaches but describe the final (ideal) solutions only. By focusing on the final solutions, the approaches ignore the fact that environments cannot stop their current activities and start with something new. A smooth transition from single system development towards product line engineering (including the new and different way of thinking in terms of common and variable things) is crucial for a successful introduction of product line engineering into an organization.

A way out of the dilemma (i.e., the mismatch between existing product line approaches and existing organizations) is to integrate the product line apporach as much as possible with the existing practices, that is, change existing practices only where necessary and only incrementally. In the figure, the effort typically required for delivering systems in a product line is illustrated. Three effort lines for three approaches are shown. The traditional product line approach needs only a minimal effort for delivering a particular instance. In comparison with a single system approach (red line), the up−front investment pays back after a number of systems has been delivered (rule of thumb: three to four systems). The incremental approach proposed in this thesis aims at the green line: following the effort distribution of single system development but at one point in time benefit from the product line characteristics of the delivered systems.

3 Product Line Infrastructure

A product line approach that aims at minimizing the changes to an organization must be customizable, especially with respect to the used types of generic assets. Generic assets are part of the product line infrastructure. They capture common and variable characteristics of a system family and, thus, are the assets (re)used and tailored during application engineering. In this thesis, a technique for modeling variabilities is developed that modifies the assets already used for modeling single systems in an organization by extending their meta models. Newly introduced elements (often for each type of variability, an individual element must be defined) capture points of variation in an explicitly visible manner and they encapsulate the knowledge needed to instantiate this point of variation (i.e., how the assets must be modified to represent each single alternative it provides). An investigation of the published product line approaches shows that a product line infrastructure contains, besides generic assets, a decision model that captures the domain knowledge needed to systematically employ a product line infrastructure. In this thesis, a technique for modeling decisions is developed that complements the technique for modeling variability. Due to the limited space, no more details of the two techniques are presented here (see [1] and [2] for details)

4 Validation

The techniques for variability and decision modeling are validated within projects and case studies.The incremental introduction of product line engineering using these techniques is validated in an experiment in which two groups of software developers are involved. The first group gets the specifications of three variants of a component, which are required for three members of a product line. Then, a generic component covering all three variants is created and, then, instantiated (traditional product line approach). The second group gets specification by specification and, thus, subsequently develops generic components covering variant 1 only, variant 1 and 2, and finally all three variants. The expected outcome is, on one hand, that the effort used by the second group does not

significantly differ from the effort of the first group, and, on the other hand, that the generic components covering all three variants are of similar quality. Then, it has been shown that without the up−front investment, product line concepts can be exploited (the investment is distributed (equally) among the developed variants).

5 Conclusion

In the context of an applied research institute, it is of particular interest to identify problems of industrial organizations. The interaction with industry in the context of product line projects (project work and acquistion) has shown that the necessary up−front investment and the required organizational changes hinders most organizations to go towards product line engineering. However, the benefits of product line approaches is exactly what they are looking for. Hence, the main point of this thesis is to create the basis for, as well as to validate, an alternative way towards product line engineering and, thus, to enable also small companies to profit from the underlying concepts.

References

[1] Atkinson, C., Bayer, J., Muthig, D, Component−Based Product Line Development: The KobrA Approach, accepted by the First International Software Product Line Conference, Denver, 2000

[2] Bayer, J., Muthig, D., and Widen, T. Customizable Domain Analysis In Proceedings of Generative and Component−Based Software Engineering Conference, September 1999

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop

URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

Requirements for Modeling Software Product Lines

Kai Böllert
Ilmenau Technical University
P.O. Box 100565, 98684 Ilmenau, Germany
kai.boellert@theoinf.tu−ilmenau.de
http://www.theoinf.tu−ilmenau.de/~kaib/

Detlef Streitferdt
Ilmenau Technical University
P.O. Box 100565, 98684 Ilmenau, Germany
detlef.streitferdt@theoinf.tu−ilmenau.de
http://www.theoinf.tu−ilmenau.de/~streitdf/

Keywords: Software Product Lines, Object−Oriented Analysis and Design
Classification: PhD work, first year

1 Introduction

Today software development produces two types of software: individual software and standard software. Individual software is built on special order and tailored to the specific requirements of one customer. Standard software, on the other hand, is supposed to meet the needs of many customers. Once developed standard software is produced in a mass production process by duplicating the distribution medium. The main problems of both types of software are: individual software causes high development costs because it takes a long time to implement the software; standard software is cheaper, but seldom meets exactly the customers' requirements.

A solution to these problems is batch production of software systems: given a customer's requirements, a generator automatically assembles a tailored system from prefabricated components. Some advantages of this method are: short production times, resulting systems are cheaper than individual software, the functionality of the systems match the customers' requirements better than standard software, as a consequence users need fewer training hours and the systems consume less resources (cf. Figure 1). The economic setting for batch production is a product line. A product line describes a "group of products that are closely related because they function in a similar manner" [10, p. 298]. Software product lines are "developed from a common set of core assets" [3, p. 3].

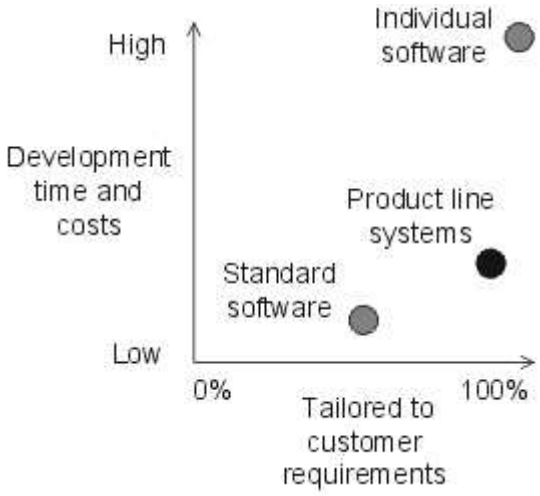


Figure 1: Benefits of product lines for customers

One important task in the development of a software product line is the design of the product line model, i.e. modeling the reusable assets. This paper examines, which methodical support is necessary to design product line models, in particular object−oriented product line models. Additionally, a short overview of existing object−oriented product line methods is given. The paper concludes by identifying future directions of this work.

2 Requirements for Product Line Methods and Models

A product line model describes a family of related software systems and, thus, expresses the commonalities and differences (or variabilities) of all family members. One member may differ from another one in both functional and technical requirements, for example business processes and objects or distribution and persistence mechanisms.

Various stakeholders make use of a product line model:

- marketeers who sell systems built from the product line
- customers who would like to buy such a system
- developers who model and implement the product line
- administrators who install and run the system

In the following requirements for product line methods and models are derived from the point of view of these four stakeholders.

2.1 From the Marketeers' and Customers' Point of View

Marketeers and customers are primarily interested in the functionality or features of a product line. With respect to each stakeholder the model of a product line has to answer either the question "What functionality (at which price) can I offer the customer?" or "What are the features I can choose from to configure my system?"

> *Requirement 1:* A product line method needs a systematic procedure to identify and describe the features of a product line.

While configuring a system, features cannot be chosen arbitrarily. Rules in the model determine if two features can be combined in a system and, when one feature is chosen, if there are other features that must also be chosen.

> *Requirement 2:* A product line model must define dependencies between features.

To describe features and their interdependencies, feature diagrams [9] and use cases [7] have proven to be useful.

2.2 From the Developers' Point of View

Developers refine the features of a product line into a design and finally into an implementation. A key task within this process is to manage the inherent complexity of product lines, so that they remain understandable and maintainable. Therefore, it is necessary to connect features with their refined design elements (traceability). For example, features may link to the following elements of the UML (Unified Modeling Language): activities, packages, classes, attributes, or operations.

> *Requirement 3:* A product line model must link features with their respective design elements.

Systems built from a product line differ in their static structure as well as in their runtime behavior. Hence, variability can be found not only in class diagrams but also in activity and sequence diagrams.

> *Requirement 4:* A product line model must consider both structural and behavioral variability.

2.3 From the Administrators' Point of View

Administrators install and run a system built from a product line, so they are mostly interested in that the system consumes not much resources and users need few training hours. A product line system can fulfill these requirements by implementing and offering users only those features that the customer has chosen to be part of the system. This is called zero−overhead rule [4].

> *Requirement 5:* A product line model must be modularized in a way, so that the generator can produce lean, zero−overhead systems.

To achieve this, the principle "separation of concerns" [5, p. 211] has to be applied while modeling product lines. In the context of product lines, the concerns are features. Therefore, in addition to decompose a product line by objects, the product line has to be decomposed into a second dimension−−−by features. Tools are required to help maintaining the consistency of the resulting product line model.

3 Overview of Object−Oriented Product Line Methods

By the time of writing, two object−oriented product line methods exist: FeatuRSEB and KobrA. Both use an extended version of the UML as notation for product line models.

FeatuRSEB [6] is the successor of the Reuse−Driven Software Engineering Business (RSEB) [8]. Based on the Object−Oriented Software Engineering Method [7], RSEB defines processes and methods to develop product lines. FeatuRSEB integrates concepts and diagrams from the Feature−Oriented Domain Analysis Method [9] into RSEB.

KobrA [1] is an instance of PuLSE, the Product Line Software Engineering Methodology [2]. PuLSE defines a process component and a number of abstract technical components to develop product lines. KobrA refines the abstract components of PuLSE, so that product lines can be developed using object−oriented techniques.

4 Summary and Outlook

The paper has derived requirements for product line methods and models from the point of view of various stakeholders. In addition, two object−oriented product line methods were briefly introduced.

Next in the PhD work is the evaluation and assessment of these methods with respect to the identified requirements. The aim is to improve and extend those methods and, finally, to integrate them into one coherent method.

References

[1] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component−based product line development: The KobrA approach. In *Proceedings of the 1st Software Product Line Conference*, 2000. (To appear.)

[2] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean−Marc DeBaud. PuLSE: A methodology to develop software product lines. In *Proceedings of the 5th Symposium on Software Reusability*, pages 122−131, 1999.

[3] Paul Clements and Linda Northrop. A Framework for Software Product Line Practice, Version 2.7, July 1999. Available at http://www.sei.cmu.edu/plp/framework.html.

[4] Krzysztof Czarnecki, Ulrich W. Eisenecker, and Patrick Steyaert. Beyond objects: Generative programming. Position paper at the ECOOP '97 Workshop on Aspect−Oriented Programming, 1997. Available at http://www.prakinf.tu−ilmenau.de/~czarn/aop97.html.

[5] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice−Hall, Englewood Cliffs, NJ, 1976.

[6] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In P. Devanbu and J. Poulin, editors, *Proceedings of the 5th International Conference on Software Reuse*, pages 76−85. IEEE Computer Society Press, 1998.

[7] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object−Oriented Software Engineering: A Use Case Driven Approach.* Addison−Wesley, 1992.

[8] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison−Wesley, 1997.

[9] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature−oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI−90−TR−021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990. Available at http://www.sei.cmu.edu/domain−engineering/FODA.html.

[10] Philip Kotler and Gary Armstrong. *Principles of Marketing.* Prentice−Hall, Englewood Cliffs, NJ, 7th edition, 1996.

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop
URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

Introducing Separation of Concerns to Product Line Engineering

Joachim Bayer
Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6
D−67661 Kaiserslautern, Germany
bayer@iese.fhg.de
http://www.iese.fhg.de/Staff/bayer/

Keywords: Separation of Concerns, Product Line Engineering, Product Line Infrastructures
Cassification: Ph.D. Proposal accepted, Planned Submission in Winter 2001/2002

1 Introduction

Product line engineering aims at improving development efficiency for families of related systems by facilitating large−scale reuse. This is achieved with a reuse infrastructure that can be used to derive the product line members. The infrastructure contains all assets that are used to construct, use, and evolve the product line. These assets contain aspects that are common to all product line members and aspects that vary among them. Such an infrastructure can only be used coherently when all known forces that influence the product line infrastructure are modeled explicitly. Separation of concerns is an approach that has been promoted for single system development to achieve such an explicit handling of forces.
The idea of separation of concerns is to describe and handle the important and critical aspects of a software system separately. This leads to reduced complexity in the description of the individual aspects. Separation of concerns during software development is realized by having a set of development models, in which each model describes the software system from a different perspective. The models together provide a view on the system that contains more information than the added information of the individual models. It improves the comprehensibility of the system, supports evolution and maintenance, enables consistency checking, facilitates reuse, and enables traceability among models.
The need for separation of concerns has long been recognized in software engineering. The concept was introduced at different software development life cycle stages. In its beginning, separation of concerns was realized at code level and expressed as the need for modularization [9]. Many approaches for separation of concerns have been proposed. Recent examples are Aspect−Oriented Programming [6], Subject−Oriented Programming [5], and Multi−Dimensional Separation of Concerns [12]. Other approaches for separation of concerns include view based requirements engineering (e.g., in [8], [11]), architectural views (the most influencing paper here was [7]), as well as object−oriented analysis and design (Catalysis [4] provides means to compose design models; in OORAM [10]

concerns are modeled as roles that can be synthesized).
However, the existing approaches for separation of concerns do not take into account the special situation that product line engineering methods impose [2]. Because product line engineering is concerned with the development of a reuse infrastructure that encompasses numerous products, it is a long−term investment that has numerous stakeholders over time. The stakeholders all have concerns on the product line infrastructure, some of which might contradict each other. Because a product line infrastructure encompasses the full life cycle of a product line, concerns must be usable and reusable at different development stages. Additional problems arise when the set of models used in the product line infrastructure are customized, as it is the case in PuLSE, the product line engineering approach developed at IESE [3]. Its customization includes the definition of model sets for the different development stages. Therefore, separation of concerns must provide the possibility to choose the used paradigm and set of models to model the concerns.

2 Goal of the Ph.D. Work

The goal is to define and validate techniques for the separation of concerns that is devised to support product line engineering and fulfills the following requirements:

- It must be applicable to the product line full life cycle, that is, it must be able to reuse concerns and to capture different definition for concerns for the different life cycle stages
- It must be fully customizable, that is, it must be independent of the used model set and of the underlying paradigm
- It must be able to handle variabilities in models. If concerns are realized differently for different product line members that can lead to contradicting models that realize the same concern. These inconsistencies can not be resolved; a technique for separation of concerns for product line must be able to handle such situations.

3 Approach

The work will contain three facets: theoretical contribution, validation, and tool support. The theoretical contribution encompasses techniques for the definition, separation, and composition of concerns, as well as support for identifying concerns. A concern is a goal, anticipation, or expectation a stakeholder has on a system. A concern is expressed as a set of requirements associated with the stakeholder. Concerns are persistent over the life cycle of the product line, but the requirements they impose in the different life cycle phases are different. A concern is realized in a number of models consisting of a number of model ele−ments that together are able to express how the system fulfills the requirements imposed by the concern.
A concern is described by a set of models on one hand and the stakeholders with their interests, goals, or expectations on the concern on the other. A model set itself is defined

by defining the relationships of its constituting models based on overlap. Two models are said to overlap, if they designate common aspects. This can be a model element that appears in several models (e.g., methods appear in class diagrams and in collaboration diagrams). More complex relations among models can be expressed in transformations (e.g., methods are derived from use cases). The description of the overlap is necessary to capture the semantics and overall meaning of the model set and to compose the separated concerns in order to derive a complete description. The model sets depend on the domain and the life cycle phase. Therefore, the definition of overlap must be customized. The overlap definition is also the basis for consistency checks and traceability.
Each goal or concern a stakeholder has on a model set must explicitly be stated. This is broken down to the models that serve a certain purpose within the set. Concerns can be related to each other, too. This is done by hierarchically composing model sets. Using the same mechanism that is used to relate the models and concerns in one life cycle stage, different stages can be related.
For validation, two case studies will be performed. The first will be done in the context of PuLSE. The focus is on the scope and the architecture of a product line in the domain of stock market applications. The technique will be applied to provide a basis for the traceability support.
The second case study is a KobrA [1] case study, in which the domain of library systems is being modeled to provide a complete application example of the method. This case study does not contain information on concerns. After the case study has been finished, parts of it will be redone based on the technique for separating concerns. To validate the benefits, the two case studies will be compared in detail.
Tools support is essential for the success of the proposed technique. There is a great amount of information that must be captured and kept consistent in order to model concerns as described above. Therefore, the thesis will provide a proof of concept in order to show that tool support can be provided. The approach is to add support for the separation and composition of concerns to existing software engineering environments (SEE). A prototype will be created that handles concerns and their relations to models that reside in a SEE. Possible ways of linking the prototype to the SEE are the XML Metadata Interchange Specification (XMI) from the OMG or application programming interfaces provided by the SEE. The prototype will provide means to export composed models.

4 Summary and Outlook

This Ph.D. is aiming at introducing the notion of separation of concerns to product line engineering. It will thereby support the coherent and consistent creation and evolution of product line infrastructures.
The work on the theoretical contribution is finished and the case studies are being performed.

References

[1] C. Atkinson, J. Bayer, and D. Muthig. Component−Based Product Line Development: The KobrA Approach. In *Proceedings of the First Software Product Line Conference (SPLC−1)*, Denver, Colorado, August 2000.

[2] J. Bayer. Towards Engineering Product Lines Using Concerns. In *Workshop on Multi−Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, June 2000.

[3] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.−M. DeBaud. PuLSE: A methodology to develop software product lines. In *Proceedings of the Symposium on Software Reuse (SSR'99)*, May 1999.

[4] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison−Wesley, 1998.

[5] W. Harrison and H. Tarr. Subject−Oriented Programming (A Critique of Pure Objects). In *Proceedings of the 8th Conference on Object−Oriented Program−ming: Systems, Languages, and Applications (OOPSLA'93)*, pp. 411−428, Washington, D.C., September 1993.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Medea, C. Lopes, J.−M. Loingtier, and J. Irving. Aspect−oriented Programming. In *Proceedings of the 11th European Conference on Object−Oriented Programming (ECOOP'97)*, pp. 220−242, Jyväskylä, Finland, June 1997.

[7] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, November 1995, pp. 42−50.

[8] J.C.S.P. Leite and P.A. Freeman. Requirements Validation through Viewpoint Resolution. *IEEE Transactions on Software Engineering*, 17(12):1253−1269, 1991.

[9] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053−1058, December, 1972.

[10] T. Reenskaug, P. Wold, and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning Publications Co, 1995.

[11] G. Spanoudakis, A. Finkelstein, and D. Till. Overlaps in Requirements Engineering. *Automated Software Engineering*, 6(2):171−198, April 1999.

[12] P. Tarr, H. Ossher, W. Harrison, S. Sutton. N Degrees of Separation: Multi−Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

Titel: Proceedings of the GCSE 2000 Young Researchers Workshop
URL: http://i44w3.info.uni.karlsruhe.de/~pulvermu/workshops/GCSE2000_Young_Research/

Multi−Design Application Frameworks

Stefan Hanenberg
Department of Mathematics & Computer Science
University of Essen
Schützenbahn 70, 45117 Essen, Germany
Email shanenbe@cs.uni−essen.de
URL: http://www.cs.uni−essen.de/dawis/shanenbe/

Keywords: Application Frameworks, Aspect−Orientation, Design Patterns
Classification: 1 year´s work

1 Introduction

Application frameworks allow to reuse design and implementation of software. Although they are meanwhile an essential basis for the rapid implementation of large software systems, the reuse of design is quite restrictive. Multi−design application frameworks are combinations of object−oriented and aspect−oriented frameworks, which allow a more flexible reuse by separating design as a concern.

2 Application Frameworks

Application frameworks are meanwhile an essential basis for the rapid implementation of large information systems. They contain domain knowledge that has to be adopted by the application developer. By using a framework developers reuse its design and implementation.

The flexible artifacts within a framework that can or even must be adapted by developers are called *hot spots* [Pre95], those that can't or shouldn't be changed are *frozen spots*. Usually the hot spots are abstract classes or interfaces which must be extended or implemented by the application developer. The main essence of frameworks is, that hot spots are invoked from within the framework itself (cf. [JoF88]) using certain design patterns like strategy, template method etc. (cf. [Gam+95]).

In fact application frameworks not only contain an abstract design and implementation of a domain, they also contain numerous concerns [Dij76] which influenced the framework's design and its functionality. For example a framework based on Enterprise JavaBeans contains numerous aspects [Blv98] and therefore can not be used for client−sided applications, even though it may depict the needed domain.

3 Aspect−Orientation

Aspect−oriented programming (AOP) [Kic+97] extends the object−oriented paradigm by introducing aspects as a new model element. Aspects are elements to express concerns, which are able to crosscut the structure of another model [CzE00]. Software systems can be developed using object−oriented programming languages and extended by binding aspects to them. Within the aspect−oriented terminology the oo−programs are called *primary structure*, the process of binding aspects is called *weaving*. The points at which an aspect crosscuts another model element are called *join points*, the references to join points are called *pointcuts.*

4 Aspect Frameworks

Aspect frameworks are collections of abstract and concrete aspects. Like object−oriented frameworks they also contain hooks and templates, but in addition to methods, abstract pointcuts may be overwritten. That means although certain aspects are already implemented, it is not fixed which elements of a primary structure they crosscut. So abstract aspects allow to reuse the concern they depict for numerous primary structures.

5 Separating Design Patterns as Concern

Design patterns become lost on implementation level within object−oriented applications and frameworks [Sou95]. If programming languages supply implementations of some patterns (e.g. class `Observable` and interface `Observer` in Java) then these patterns become part of the inheritance structure of some classes and there is no way to extend them without the included patterns. It developers decide to implement patterns on their own, it is hardly possible to identify these patterns in the model of an application. If would be more desirable to identify patterns on model level and reuse the patterns' implementations. This can be done using aspect−oriented techniques.

In [HFU00] we showed, how certain design patterns can be encapsulated using aspects and introduced our aspect framework *sideFrame* [Side00] based on the aspect language aspectJ (cf. [LoK98]). This white box aspect framework contains pre−build design patterns which can be weaved to an existing primary structure by overriding abstract methods and pointcuts. The main benefit is, that by weaving design−aspects the implementations of patterns can be reused and on model level these patterns can be identified. Additional, these patterns can be weaved whenever needed without staining the inheritance structure of the participating classes.

By encapsulating design patterns into aspects there is no need any longer to keep certain design patterns within an object−oriented framework when not strictly prescribed by the domain. Instead object−oriented frameworks should additionally be supplied with an

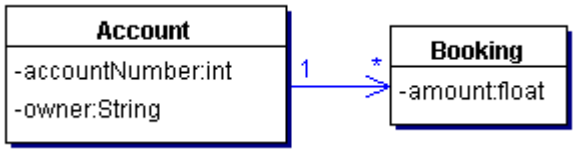
aspect framework which allows developers to add specific patterns. The main consequence of this approach is, that object−oriented frameworks become more flexible, because application developer decide which patterns to use in what situations. Another effect of separating design patterns as concerns is, that documenting frameworks becomes more easier of the fact, that interfaces within the framework become more understandable.

6 Example

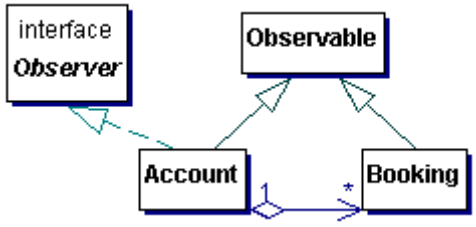
We assume a quite simple financial framework in Java, which supplies two (concrete) classes `Account` and `Booking`. `Account` has two attributes `owner` and `id` for the owner's name and the account number. A booking just consists of an `amount`, which is in this simple example of type float (figure 1a).
There may be two different kinds of customers for this framework:

1. developers writing client−applications for managing accounts on a single machine
2. developers writing internet applications for managing account via internet

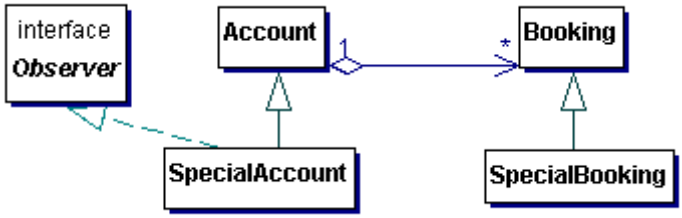
The first type of customer writes a GUIs which needs to observe `Account`− and `Booking`−instances, the other does not. For the framework provider it means either to supply the observable−functionality within the framework and to satisfy the first type's needs (figure 1b), or to neglect it and to offer simple interfaces appropriate for the second type of customer. If the observer−property is neglected, the first customer has to adopt it (figure 1c) by implementing the observer−pattern on his own.



a)



b)



c)

Figure 1: a) simple framework, b) including oberserver / observable, c) adopted simple framework

When separating design concerns within an aspect framework and supplying it in addition, the framework can satisfy both customers' needs. Adding observable− and observer−properties to `Booking` and `Account` means in sideframe to extend the generic aspects of the participants within the observer pattern (`GenericSubject` and `GenericObserver`) and set `Account` and `Booking` as join points (figure 2). The generic aspects contain the needed behavior and interaction of the pattern like attaching the observer to the subject and informing the observer whenever the subject's state changes. So the framework's design can be adopted to the developers' needs.

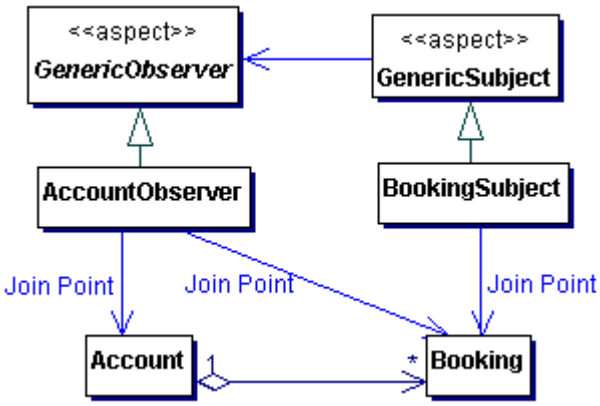


Figure 2: Adopting simple Multi−Design Framework

7 Future Works

In the future we plan to analyze how the structure of aspect−oriented frameworks can be described with well known techniques like metapatterns. In addition we plan to extend sideframe with architectural patterns.

References

[Blv98] Gregory Blank, Gene Vayngrib, *Aspects of Enterprise Java Beans*, Aspect−Oriented Programming (AOP) Workshop at ECOOP'98. 1998

[CzE00] Czarnecki, Krzysztof und Ulrich W. Eisenecker: *Generative Programming: Methods, Tools, and Application*, Addison−Wesley, 2000

[Dij76] Dijkstra, Edsger W.: *A Discipline of Programming*. Prentice−Hall. 1976

[Gam+95] Gamma, Erich, Helm, Richard, Johnson, Ralph E., Vlissides, John: *Design Patterns: Elements of Reusable Object−Oriented Software*, Addison−Wesley. 1995.

[HFU00] Hanenberg, Stefan, Franczyk, Bogdan, Unland, Rainer, *Aspect Frameworks: How aspect−orientation an smoothly support the evolution process of software*, submitted

[JoF88] Johnson, Ralph E. , Foote, Brian , Designing reusable classes. Journal of Object−Oriented Programming, 1(5), 1988, pp. 22−35

[Kic+97] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean−Marc Loingtier und John Irwin: *Aspect−Oriented Programming*. In: Aksit, Mehmet (Hg.): Proceedings of ECOOP'97. Springer−Verlag. 1997

[LoK98] Cristina Videira Lopes and Gregor Kiczales, Recent Developments in AspectJ, Aspect−Oriented Programming (AOP) Workshop at ECOOP'98. 1998

[Pre95] Pree, W.; *Design Patterns for Object−Oriented Software Development*. Addison−Wesley, Reading, MA, 1995

[Side00] SideFrame: Simple Design Pattern Framework, Homepage, http://www.cs.uni−essen.de/dawis/research/sideframe/

[Sou95] J. Soukup, *Implementing Patterns*. In: J.O. Couplin, D.C. Schmidt (eds.): Pattern Languages of Program Design, Addison−Wesley, 1995.