Tool Support for Feature-Oriented Software Development

FeatureIDE: An Eclipse-Based Approach

Thomas Leich, Sven Apel, Laura Marnitz, Gunter Saake
Department of Computer Science
Otto-von-Guericke-University Magdeburg, Germany
{ leich | apel | lmarnitz | saake }@iti.cs.uni-magdeburg.de

ABSTRACT

Software program families have a long tradition and will gain momentum in the future. Today's research tries to move software development to a new quality of industrial production. Several solutions concerning different phases of the software development process have been proposed in order to cope with different problems of program family development. A major problem of program family engineering is still the missing tool support. The vision is an IDE that brings all phases of the development process together consistently and in a user-friendly manner. This paper focuses on AHEAD, a prominent design methodology and architectural model for feature-based program families. We present our first results on developing an Eclipse-based IDE that supports building program families following the AHEAD architecture model. Starting from current weaknesses and pitfalls in implementing program families we outline several challenges of the feature-based development process. Thereupon, we present our ideas to face these challenges and a resulting integrated tool chain based on Eclipse.

1. INTRODUCTION

In recent years the idea of program families (a.k.a. product lines[1]) has been discussed to overcome the software crisis. The key idea is to build not individual programs, but a family of similar programs. Program family members are grouped by their commonalities. AHEAD is an architectural model and a design methodology to implement program families [4]. The idea of the AHEAD model is to decompose programs into separate modular units (features) and to compose stacks of features to derive a concrete program. When adding new programs to a family existing features of other programs can be reused. This is also known as *step-wise refinement* [12]. The benefit is maintainable, comprehensible software that can easily be reused, configured and extended. An exceptional quality of AHEAD is the unification of features and components, i.e. one component implements exactly one feature.

[1]Although there is a subtle difference between program families and product-lines (see [5]) we use these terms synonymously.

eclipse'05, October 16-17, 2005, San Diego, CA
Copyright 2005 IBM 1-59593-342-5/05/0010…$5.00

The *AHEAD Tool Suite*[2] provides a set of tools that support programming in the AHEAD style. However, using the AHEAD Tool Suite is still a hard challenge, since most of the functionality is loosely coupled and is provided by command-line tools only. To bring the AHEAD model to a widely accepted solution for building software in praxis an adequate IDE tool support is indispensable.

This article focuses on providing IDE support for program family development. Firstly, we sketch out special challenges for building program families using the AHEAD model. Thereby, we do not only focus on design and programming activities but also on the preliminary analysis phase and subsequent configuration support. We perceive *features* as the base concept that is used in all development phases. Consequently, we present a tool-driven concept that improves the overall *Feature-Oriented Software Development (FOSD) Process* to support reuse of information from different development phases. Moreover, we discuss our solutions on how to enforce a consistent development process and we propose generating and checking mechanisms that help to guarantee consistent data in between the phases. Furthermore, we introduce a feedback mechanism that propagates extracted implementation knowledge back to the design and analysis phase. This allows us to use abstract design information in the configuration process in order to overcome the complexity of the configuration process.

2. BACKGROUND

Feature-Oriented Software Development (FOSD) is the overall process of developing software systems in terms of its features. FOSD aims on analysing, designing, and implementing features in program families. Following this idea we utilize for the domain analysis the *Feature-Oriented Domain Analysis (FODA)* [7]. The design is based on step-wise refinement and *collaboration design* [15], and the implementation is based on *Feature-Oriented Programming (FOP)* and *Mixin Layers* [13].

2.1 Feature-Oriented Software Development

FODA: With the domain analysis feature modelling is an appropriate method [7]. The goal of FODA is to analyze the considered target application scenarios and to derive the required and optional features. Since the focus of FODA is

[2]http://www.cs.utexas.edu/users/schwartz/Hello.html

on a domain of applications instead of one application, the resulting features are chosen with regard to a whole family of systems. The results of feature modeling are feature models that describe the features, their relations, constraints, and dependencies [5]. The models express variation points and commonalities of the target-programs in an abstract and implementation independent way. Features are organized in a hierarchical way (see Fig. 1).

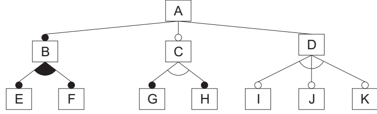

Figure 1: Example feature tree

Step-wise Refinement and Collaboration Design: Step-wise refinement and the collaboration design are methods to design software incrementally, using minimal building blocks, and starting from a minimal base [13]. The possibility of exchanging, adding and removing such building blocks, also called *layers*, yields reusability, extensibility, and customizability. Features[3] are basic building blocks that satisfy intuitive user-formulated requirements on the software system. Batory et al. have mapped this concept to the object-oriented world [3, 13]. They have observed that a new *software feature* often extends or modifies numerous existing classes. Based on this observation, they perceive features as *collaborations of class fragments*, also referred to as *roles*. Figure 2 shows a result of the step-wise refinement and collaboration design, a stack of collaborations (features).

Classes are arranged vertically ($C_1 - C_3$). Collaborations are arranged horizontally and span several classes ($L_1 - L_3$). Several features of a software system result in a stack of collaborations. Collaborations with the same interfaces are easily exchangeable. They are instances of large-scale components [3].

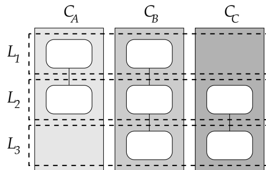

Figure 2: Stack of collaborations

FOP: *Mixin Layers* are one appropriate technique to implement collaborations in a step-wise manner. As mentioned, the basic idea is that features are often implemented by a collaboration of class fragments. A Mixin Layer is a static component encapsulating fragments of several different classes (Mixins) so that all fragments are composed consistently. Advantages are the high degree of modularity and the easy composition. The AHEAD Tool Suite, including the *Jak* language, implements AHEAD for Java. FeatureC++[4] implements the AHEAD model for C++. Using the *Jak* language or FeatureC++ Mixin Layers are represented by directories of the file system. Mixins are represented by included source files. Therefore, Mixin Layers have no textual representation at code level. An equation file specifies which features are required for a program configuration. It defines ordered feature collections as algebraic expressions. Those Mixins found inside the directories are assigned to be members of the enclosing Mixin Layers.

[3]We use the terms feature and layer as synonym for collaboration.

[4]http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

Design Rule Checking: Type checking is provided by the underlying language, but it does not catch deep semantic composition violations. Not all combinations of features are semantically correct. Selecting a feature may enable or disable the selection of other features. Design rule checking (DRC) [4] helps to overcome this problem. Using an attributive grammar the programmer specifies these checks. Attributes are value annotations and predicates over these annotations that determine which syntactically-legal combinations of tokens are semantically correct.

3. CHALLENGES OF FOSD

The following considerations are based on our experiences in developing FeatureC++ [2] and in building program families in the area of embedded databases [9] and middleware [1]. We see three major fields of interest: (1) the challenges regarding the missing tool support of the FOSD process, (2) missing support for coding Mixin Layers, and (3) challenges according to further improvements of FOP.

3.1 Supporting the FOSD-Process

There is a broad spectrum of different sources of information during the development process of a program family. Without integrated software visualization tools for displaying, connecting, and managing the different sources, as well as navigation support, the handling of program families with more than 1,000 features is not feasible. Moreover, to avoid failures and to enforce a consistent development process an automatic generation and checking of models is necessary. We consider two major problems of the FOSD process:

- *Inconsistent states between development phases:* FOSD is a phase-oriented development process. Without having tool support a lot of information regarding the different phases has to be redundantly fed into the different development phases. This causes failures and inconsistent states. One example is that information of FODA are not connected to the design and implementation phases, e.g. constraints, dependencies, annotations represented in feature diagrams are not present in other phases. In subsequent phases the programmer has to integrate and evaluate these relationships manually. This is errornous and leads to redundant work. Furthermore, software engineering is seldom a straightforward process. Often programmers discover deeper relations between features not until the implementation phase. This encountered information has to be back propagated to the previous software engineering phases.
- *Dealing with the complexity of the configuration:* Building concrete family members using AHEAD or FeatureC++ requires a lot of implementation knowledge. This is because of features interact with each other mainly at implementation level. Program families consisting of many features (> 100) are hard to handle.

Moreover, relationships and dependencies between them are not catchable by humans. During the development of embedded middleware and database systems we observed that due to the high variability in this field – with millions of possible configuration variants – the configuration process becomes impossible without tool support. Dealing with these problems the configuration process has to be lifted to a more abstract level in order to handle the complexity.

3.2 Supporting FOP

Reading or writing code is still the most used way of building and understanding software. The main tasks of tool support in programming code is to point the programmer in an easy way to the right location of the source code and help him/her to read, modify, and/or write the code. Although the languages Java and C++ and their feature-oriented extensions Jak and FeatureC++ only differ in a handful of key words, it is still a hard challenge to use them. In object-oriented IDEs the class view is the dominating element of representation. Instead, with FOP, the main subject of interest are features. Features contain several software artifacts that contribute to the feature's functionality.[5] Standard object-oriented IDE-functionality as Eclipse supports for Java e.g. source code completion, syntax checks on demand, class navigation and debugging support, are not available for FOP. Furthermore, FOP yields some special problems:

- *Handling the complexity of feature interactions:* Features interact with each other in many ways. These interactions result in dependencies between different features in that way that features exclude or require other features. The programmer must take great care in identifying every interaction before he can implement or modify a feature. In AHEAD and FeatureC++ features have no direct textual representation. Therefore, dependencies are not explicitly represented. This complicates the development process.

- *Supporting DRC specification:* As mentioned in Section 2 the specification of semantic correctness is enforced by design rules using an attributive grammar. Unfortunately, the programmer has to map the knowledge of feature dependencies of the analysis manually to the design and implementation phase. This a non-trivial task, not user friendly, and in most cases redundant work for the programmer.

3.3 Combining AOP and FOP

Aspect-Oriented Programming (AOP) is a prominent technique to localize, separate, and modularizes crosscutting concerns [8]. Thus, it is also adequate for building program families. The idea behind AOP is to implement so called orthogonal features as *Aspects*. This prevents the known phenomena of code tangling and scattering. The core features are implemented as components, as with common design and implementation methods. Using join point specifications (*pointcuts*), an aspect weaver brings aspects and components together. There are several discussions on separating crosscutting concerns using AOP and FOP [11, 10, 2]. All current approaches focus on language support. Developing FeatureC++, we discovered that visualizing dependencies of aspects and features in different context views would be extremely helpful for the programmer. These kinds of views can help to control the powerful combination of AOP and FOP. However, there are first attempts to achieve that for pure AOP, e.g. the AcpectJ Visualiser[6].

A second very important issue is the support of debugging functionality. AHEAD and FeatureC++ are source-to-source code transformation systems. The mapping of the runtime-debugging code in the original source code is a hard challenge, especially when integrating AOP support.

4. IDE SUPPORT FOR FOSD

Since Eclipse is being widely used by a growing user community and it is open to trying out new ideas, we implemented FeatureIDE as an Eclipse 3.0 plug-in. Each component of the workbench is extensible and customizable via the plug-in interconnection model that supports any number of named extension points and any number of extensions to one or more extension points in other plug-ins. Moreover, the elementary components of an IDE, such as the workbench, file navigator, wizards, text editor, component-version management, and publishing services are already available. Besides the standard functionality we have used the GEF and Visualiser plugin. Due to these facts the implementation process was relatively easy and straightforward.

4.1 FOSD-Process

A major goal of the FeatureIDE project is to handle the complexity of the program family development process. Taking an ambitious stance, we claim that the ultimate goal of our work is to become the preferred way of developers of looking at software in all phases. Therefore, we utilize visualization and interaction techniques, e.g. detail and overview, detail on demand, and graphical hints. Most of the functionality is quite similar to the standard Java IDE. We believe only this will help to bring FOP to an widely accepted programming paradigm.

4.1.1 Preventing Inconsistent States

Preventing inconsistent states is enforced by *mapping functions* that connect all phases of the development process. The central elements of these mapping functions are the features and their relationships and constraints. Doing so, the results of FODA are propagated to the design phase and all features are mapped to collaborations. An heuristic function generates a half-order of layers (see Fig. 3).

This propagation helps the programmer during the design process to define a concrete sequence of the layer stack. Furthermore, the additional relations are transferred into the corresponding layers. If, e.g., two features exclude each other, this relation is also known in subsequent phases (e.g. implemented layers). Doing so, it is not allowed to use or refine the functionality of these layers. As a result of the design phase the relationships between features are further refined. Using this information the implementation structure e.g. file directory folders, implementation files, and DRC-file are generated.

[5] Although the FOP concept allows the refinement of other fragments like UML-diagrams or documentation we concentrate on implementation units, i.e. classes in this paper.

[6] http://www.eclipse.org/ajdt/visualiser/main.html

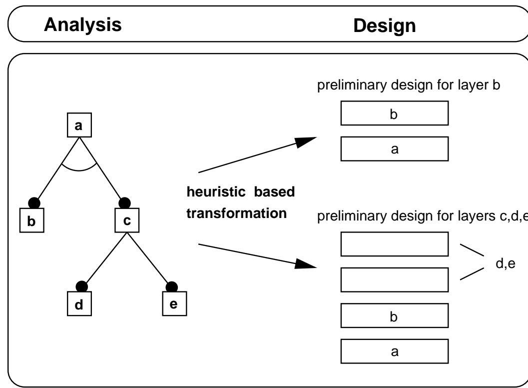

Figure 3: Mapping Functions

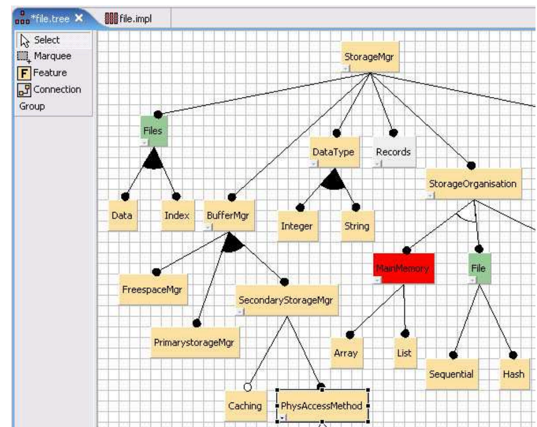

Figure 4: Feature Tree with DRC-Hints

However the development process is seldom straightforward. Commonly, new information is collected during the design an implementation phases, which is inconsistent to existing information from previous phases. It is very common to find that a feature has to be further differentiated. According to our model, this has to be back propagated, because it is not allowed to create new implementation files for non existing analysis features. Only this restricted model enforces consistency between all development phases and improves the quality of the resulting program family architecture.

4.1.2 *Supporting the Configuration Process*

Feature diagrams provide an abstract and intuitive representation of the variation points of program families. Therefore, these diagrams are perfect starting points to improve the configuration process. Due to achieving consistent and synchronized states between all phases and due to the one-to-one mapping of features to their counterparts in subsequent development phases, we are able to utilize the abstract feature tree to assist the configuration process. A drawback of this abstract feature model is that not all relationships are presented. Borrowed from [14] we improved our model with detailed information about additional relationships from the design and implementation phase. Avoiding an overkill of information the additional information is highlighted only on demand in the feature diagram. Figure 4 shows the configuration process. According to the chosen feature *PhysAcessMethod* additional relationships are displayed in the diagram.

4.2 Supporting FOP

The most challenging problem arises due to the fact that the refinement chains are highly variable. Code completion features are realized using an on-the-fly analysis of the collaboration stack. Context based views for analyses of dependencies are essential for programmers and designers.

4.2.1 *Handling the complexity of feature interactions*

There is a broad spectrum of different sources of information about feature interactions. To overcome this complexity we used different views on the implementation units. The created collaboration stack is the main view because it provides a good overview of the global structure. Additionally, this view is used for navigation through the implementation units. Several filters allow the user to adjust the complexity of the representation (file type, file name, relationship to other features, refinement level). Figure 5 shows the collaboration view. The representations of the class fragments are colored according to their type of the linked implementation file, e.g. blue for classes, white for documentation, etc.

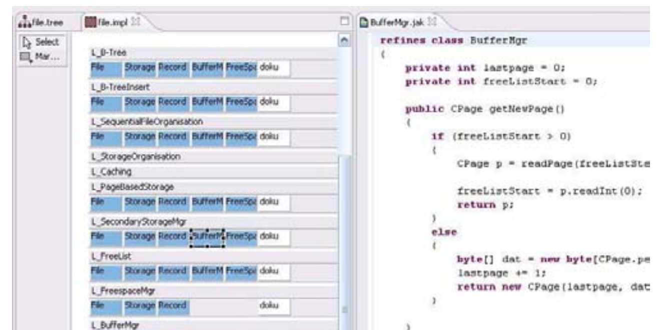

Figure 5: Collaboration Editor

With the collaboration view the programmer has a good opportunity to get an overview of the features and their refinement hierarchy. What is still missing is a possibility to get an overview of the refinements at implementation level. Files that implement the same class fragments can refine or add different code artifacts. For the programmer it is very difficult to extract these relationships from textual code. We used another representation to give the programmer a proper overview. Our visualization is based on an approach which is called *Seesoft* [6].

Figure 6 shows an example: The boxes represent class fragments that compose the resulting class. Thus, the programmer gets insight which feature contributes to which class. The length of the boxes is determined by the size of the corresponding implementation file. The colored lines inside the boxes represent code fragments. Different code fragments are encoded by different colors. The width of the colored line encodes the size of the according code fragment. The colored lines are connected to the corresponding code and can be used to go quickly into the code to the right position. This functionality helps the programmer to quickly

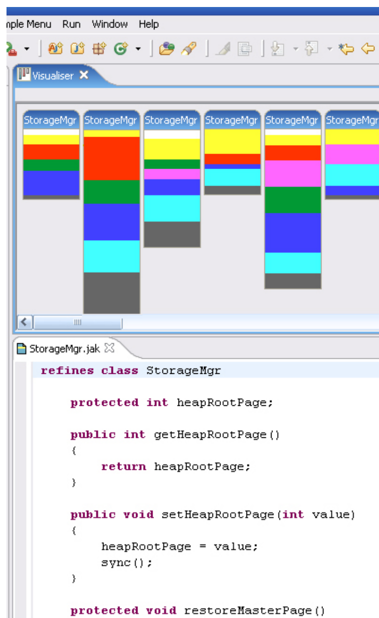

Figure 6: Feature Fragment Visualizer

make decisions that contribute to other class fragments.

4.2.2 Design Rules

AHEAD assumes that design rules are created during the implementation phase. We already start creating design rules during the analysis and design phase. The design rules are automatically extracted from the modelled relationships and constraints of these phases, e.g. encoded in feature diagrams, collaboration stacks, etc. Thereby the programmer just has to add the design rules for those relationships and constraints which are extracted during the implementation process. Due to this proceeding, the concept of DRCs becomes more practical and can be used for easily enforcing consistency. The design rules are a concentrated view on the relationships and constraints in all phases. Therefore some kind of inconsistency can be easily detected.

5. FURTHER RESEARCH

As in Section 3.3 mentioned AOP is suitable to enhance FOP in providing mechanisms to deal with certain crosscutting concerns. A problem of current AOP languages is that the binding of aspects is independent of the current development stages. That means that aspect may affect subsequent integrated features. This may lead to failures and unpredicted program behavior. Using aspect-enhanced FOP the power of aspects can be controlled. In [2] we present language support to solve this problem. Using this mechanism we discovered that in large-scale systems this language capabilities further has to be supported by visualization and generative techniques.

6. CONCLUSIONS

FOSD is important to build future program families. Current research on FOSD focuses mainly on language level support. Current tools are mostly command line based. We argue that tool support for FOSD is indispensable to increase the acceptance of feature-oriented techniques. In this contribution we featured a discussion on challenges of developing software in a feature-oriented style using common techniques and tools. Starting from identified weaknesses we present FEATUREIDE that solves certain problem of this field. Due to the integration into Eclipse we hope to increase the acceptance of FOSD.

7. REFERENCES

[1] S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *ASE SEM'04*, Springer, LNCS. 2005.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE'05*, Springer, LNCS, 2005.

[3] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1992.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.

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

[6] S. Eick, J. Steffen, and E. Summer. Seesoft – a tool for visualizing line oriented software statistics. In *IEEE TSE*, 1992.

[7] K. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[8] G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP*, Springer, 1997.

[9] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *ADBIS*, Springer, LNCS, 2005.

[10] R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. Technical report, 2005.

[11] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *ACM SIGSOFT FSE*, 2004.

[12] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions On Software Engineering*, SE-5(2), Mar. 1979.

[13] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 2002.

[14] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In *Object-Oriented and Internet-Based Technologies.* Springer, LNCS, 2004.

[15] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *OOPSLA '96 CA, USA*. ACM Press, 1996.