FeatureIDE: A Tool Framework for Feature-Oriented Software Development

Christian Kästner, Thomas Thüm, Gunter Saake
School of Computer Science
University of Magdeburg, Germany
{*kaestner,thuem,saake*}*@iti.cs.uni-magdeburg.de*

Janet Feigenspan, Thomas Leich
Section Applied Computer Science
METOP GmbH, Magdeburg, Germany
{*janet.feigenspan,thomas.leich*}*@metop.de*

Fabian Wielgorz, Sven Apel
Dept. of Informatics and Mathematics
University of Passau, Germany
{*wielgorz,apel*}*@uni-passau.de*

Abstract

Tools support is crucial for the acceptance of a new programming language. However, providing such tool support is a huge investment that can usually not be provided for a research language. With FeatureIDE, we have built an IDE for AHEAD that integrates all phases of feature-oriented software development. To reuse this investment for other tools and languages, we refactored FeatureIDE into an open source framework that encapsulates the common ideas of feature-oriented software development and that can be reused and extended beyond AHEAD. Among others, we implemented extensions for FeatureC++ and FeatureHouse, but in general, FeatureIDE is open for everybody to showcase new research results and make them usable to a wide audience of students, researchers, and practitioners.

1 Introduction

Tool support, such as *integrated development environments (IDEs)*, is crucial for the acceptance and adoption of a new programming language or paradigm, both in academia and industry. Experience has shown us that programming languages (or language extensions) implemented as command line preprocessors are difficult to convey. When teaching such languages and the concepts behind them, students struggle with using the technology instead of learning the language concepts; when performing case studies, relying on a simple text editor is frustrating and limiting compared to modern IDEs; and finally, when working with industrial partners, languages without adequate IDE support do not stand a chance of being considered.

Some observers attribute at least some degree of AspectJ's success on the availability of the industrial-strength *AJDT* development environment for Eclipse, which makes aspects easier to use and lets users focus on the language instead of the compiler or tool infrastructure [15]. For example, when performing a major case study on aspect-oriented refactoring of the medium sized legacy application Berkeley DB (84 000 LOC) [8], we learned first-hand the value of such tool support, in that it helped us to understand the resulting program and the effects of aspects therein. As stated in [8], this case study would not have been possible at this scale without tool support.

However, developing industrial-strength tool support is a tedious task. The development of AJDT was sponsored by IBM and other companies. When developing tool support for proprietary language extensions [11, 9], we required major effort to release only a basic version, which is still far away from capabilities modern IDEs provide for mainstream programming languages. Even worse, this effort has to be repeated for every language or language extension.

In this paper, we present *FeatureIDE*[1], an open source framework of an IDE for software product line engineering based on *Feature-Oriented Software Development (FOSD)* [13, 5]. FeatureIDE supports the entire life-cycle of a product line in a coherent tool infrastructure, starting with domain analysis and feature modeling [6], but also covering design, implementation and maintenance with FOSD.

In contrast to earlier versions, FeatureIDE does not only cover a single language (e.g., *Jak* from the AHEAD tool suite [5]), but several languages based on the same foundation: the concept of FOSD. At the point of writing, FeatureIDE supports a multitude of different tools including *AHEAD* [5], *FeatureC++* [3], *FeatureHouse* [2], and *CIDE* [9]; this way FeatureIDE supports FOSD in many languages, including Java, C++, Haskell, C, C#, JavaCC,

---

[1]http://www.fosd.de/featureide/

and XML. As we will show, also other parts of FeatureIDE are opened up for extensions, which makes it possible to extend FeatureIDE further, either toward specific needs in an industrial setting or to showcase research results in a full IDE and make them quickly available to users in academia and industry (as in [16]).

Overall, we envision FeatureIDE as open source project that provides a broad foundation, but that can be used and extended by different parties to teach and productively use FOSD. In this demonstration, we give an overview of FeatureIDE's design and present recent developments on the background of current software product line projects.
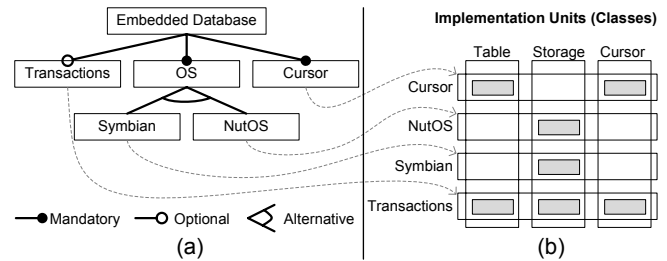
2 Feature-Oriented Software Development

Feature-oriented software development (FOSD) is a paradigm for designing and implementing applications based on features. A feature is an end-user visible characteristic or requirement in a software system. The basic idea of FOSD is to modularize software into *feature modules* which represent features [13, 5]. To create an application, feature modules are composed. As a side effect, this introduces flexibility to compose features in different combinations, e.g., omit certain features or implement alternative features. This way, FOSD can be used to develop software product lines.

A software product line is a family of related programs tailored to a domain, between which implementation artifacts are shared. To develop a software product line, a domain engineer analyzes the domain and identifies the differences and commonalities between programs in that domain [6]. Domain analysis results in a feature model as depicted in Figure 1a, which describes the features and their relationships. In a software product line, different programs can be produced on the basis of different feature selections.

There is a multitude of distinct concepts for implementing software product lines. With FOSD, each feature is implemented by a distinct feature module. Technically, a feature module contains classes or class fragments (also called 'refinements'). In Figure 1b, we depict a simple example of three classes (vertical boxes *Table*, *Storage*, and *Cursor*) and four feature modules (horizontal boxes). The classes are divided into class fragments (gray boxes) that can be associated to different features. A feature module contains all class fragments of its feature. There are different approaches to implement class fragments; most extend the syntax for class declarations with new keywords as *feature* [13], *refines* [5], or *partial* [in C#]. In order to generate a program, class fragments of the selected features are composed with a tool as AHEAD for Java [5], Xak for XML [1], or FeatureC++ for C++ [3]. This way, many different programs can be created from a set of feature modules.

Putting it all together, the typical process of feature-based software product line development consists of four phases as

Figure 1. A simple feature model and its implementation
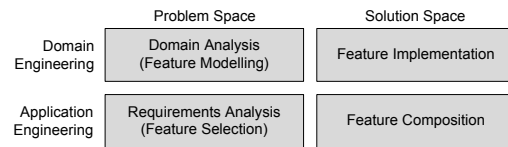
Figure 2. Phases of feature-based software product line development

shown in Figure 2. In the first phase, the domain is analyzed to identify features of the domain and their relationships. Next, but still as part of domain engineering, features are implemented as feature modules. To produce a specific program, the requirements of this program are analyzed and the according features are selected from the feature model. Finally, based on feature selection and feature implementation, specific programs can be composed.

3 History: AHEAD Support

FeatureIDE was originally designed in 2004/2005 as an IDE for the AHEAD tool suite, which integrates activities of all phases of software product line engineering [11]. It originated from the observation that all development phases are connected, and tool support could ensure consistency and automate certain steps. To give an overview, we outline FeatureIDE's support for development with AHEAD:

- Domain analysis and feature modeling are supported with a graphical feature model editor. Feature models are saved in a machine-readable form that can be used for reasoning or analysis in other phases.
- Feature implementation is supported by providing an editor (with syntax highlighting) for AHEAD's Java dialect *Jak*. Furthermore, feature modules are automatically created for and synchronized with features from the feature model to ensure consistency. For example, when a feature is renamed, so is the according feature module. Finally, views, as in Figure 1b, are provided to visualize the structure and facilitate navigation.

- Requirements analysis is supported with a graphical editor for selecting features from the feature model. Information of the feature model is used to immediately validate the feature selection, e.g., to detect conflicting specifications. Furthermore, feature selections are synchronized with changes of the feature model.
- Feature composition is finally automated with a background thread that composes feature modules for the provided feature selections by invoking AHEAD's tools. Code structures gathered during this step are used for visualizations during implementation; errors are propagated back and marked on the according locations in the implementation, as compiler errors in modern IDEs. The composition is automatically repeated when implementation or feature selection are updated.

Since 2006, after the initial prototype presented in [11], we entirely reimplemented FeatureIDE from scratch to provide an even closer integration with AHEAD. The feature model editor builds directly on AHEAD's format. AHEAD's composition tools are included and shipped with FeatureIDE, and they were even extended to provide access to AHEAD's internal data structures for visualizations.

While this created a stable IDE that we could use for teaching, and for which we received positive feedback from users around the world who wanted to try FOSD with AHEAD, the close coupling with AHEAD turned out to be a limitation. In our research, we designed new languages and composition tools, e.g., FeatureC++ [3] and Feature-House [2], and we also developed other facilities such as automated refactorings [10] or analysis tools for feature models [16] which would be nice to use in an IDE but did not fit into the concept of FeatureIDE as an AHEAD front end. Therefore, we eventually decided to broaden the scope of FeatureIDE and to design it as an FOSD framework, in which AHEAD is one of many extensions.

4 Open Framework for FOSD

Eclipse is developed as a framework, that can be used to develop IDEs for various languages (e.g., JDT for Java, CDT for C++). Instead of adding FeatureIDE as another language-extension for AHEAD, we make FeatureIDE an extensible framework on top of Eclipse that implements only the part that is common to all feature-oriented languages.

The question remains: What is the common essence of FOSD? Let's revisit the four phases:
- Domain analysis and feature modeling is independent of the language and composition tool. Therefore, functionality to analyze the domain and to edit feature models can be placed in the common framework.
  Nevertheless, because of a large body of research on feature modeling and reasoning about features models, we also open up feature modeling for extensions. This

way, we extended FeatureIDE with a graphical frontend for our algorithm reasoning about edits to feature models [16], instead of providing only command line tools or writing a new model editor first. Furthermore, FeatureIDE can be extended to read and write other formats for feature models (currently AHEAD format and SXFM format used at University of Waterloo [12]), enabling it as frontend for other tools and even for conversions between formats.
- Feature implementation is language-specific to a high degree. Editors for feature-oriented languages (or language extensions) with syntax highlighting or code completion are provided as separate plugins, as long as existing editors in Eclipse cannot be reused (which is possible for FeatureC++ and many languages in FeatureHouse, but not for AHEAD). Still, synchronization and visualization can be reused.
  Specifically, visualizations as in Figure 1b are largely language-independent. Visualizations can be based on a common underlying data model based on the language-independent structures (called feature structure trees) found in research on the foundations of FOSD [2]. Only an adapter from each composition tool to this data model is needed. This way, all languages benefit from ongoing research on novel visualizations.
- Requirements analysis is independent of the language or composition tool. Nevertheless, to foster research on (semi-)automated product derivation as [17, 14], the implementation supports different file formats and is open for extensions.
- For feature composition, different composition tools can be plugged in. FeatureIDE currently integrates AHEAD [5], FeatureC++ [3], and FeatureHouse [2] as plugins, supporting FOSD with the following languages: Java, C#, C, C++, Haskell, JavaCC, and XML. Still, build automation and collecting underlying structures for visualizations are common to all extensions.
- Finally, advanced tools as type-checkers [7] or refactorings [10] for specific languages can only be generalized to a low degree and are plugged in separately. Still, the benefit remains of integrating in such research in an existing IDE, instead of offering only command line tools or having to write an entire IDE each.

This analysis shows that a large percentage of FeatureIDE's functionality is language-independent and can be generalized. A generalized framework allows us and others to extend FeatureIDE rapidly for new feature-oriented languages or composition tools. Similarly, new developments in the field of domain analysis, feature-modeling, type-checking, or visualizations can be incorporated quickly and are beneficial for users of all languages.

By refactoring FeatureIDE into a framework, we found a way to coordinate development efforts by different develop-

ers and researchers and can use the same tool in a broader scope. By making it available as open source, we encourage others to participate in the project and leverage from existing implementations. This framework makes it easier to evaluate new ideas in academic and industrial settings, as users – students and professionals alike – are not discouraged by preprocessors and command line tools. We hope that others pick up the framework and build their own extensions, instead of reimplementing common parts as domain analysis, builders, or visualizations over and over again. In our research, we are furthermore planning to use FeatureIDE for own research on aspectual feature modules [4], novel visualizations, and virtual separations of concerns [9].

5   Related Work

While FeatureIDE focuses on integrating all phases of FOSD, there are several IDEs or graphical editors for individual phases of FOSD. For example, there are dozens of implementations of a feature model editor available, e.g., *Captain Feature*[2]. Furthermore, commercial closed-source software product line solutions as *pure::variants* or *Gears* provide IDE support, with focus on domain analysis and less on implementation, so that many integration advantages of FeatureIDE do not apply. Though they are extensible as well, their closed-source nature makes extensions difficult, especially in a research context.

Furthermore, many Eclipse projects are developed in an open fashion and the Eclipse community often focuses on building reusable open frameworks (e.g., data tools platform, open healthcare framework). Closest to our work is *openArchitectureWare*[3], that provides an Eclipse based open framework for developing model-driven development solutions and domain specific languages. In this project, also a common core is defined that can be extended by different modeling languages. Its goal to integrate several languages and to provide a reusable tool infrastructure is similar, but FeatureIDE focuses more specifically on FOSD and software product lines.

6   Conclusion

Nowadays, tool support, as in the form of an IDE, is crucial for the acceptance of a new language. However, providing such tool support is a huge investment that can usually not be provided for a research language. With FeatureIDE, we have invested several years in providing IDE support for the AHEAD tool suite, to make research results easier to adopt for academic projects and also for industry. To reuse this investment, internally for own recent research

results and externally by other researchers, we restructured this project as a framework to support multiple languages and composition tools. FeatureIDE can be extended quickly to showcase new research results and make them usable to a wide audience of students, researchers, and practitioners.

References

[1] F. I. Anfurrutia, O. Diaz, and S. Trujillo. On the refinement of XML. In *Int'l Conf. Web Engineering*, 2007.

[2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automatic software composition. In *Proc. Int'l Conf. on Software Engineering*. 2009.

[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering*. 2005.

[4] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng.*, 34(2), 2008.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Trans. Softw. Eng.*, 30(6), 2004.

[6] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press, 2000.

[7] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proc. Int'l Conf. Automated Software Engineering*. 2008.

[8] C. Kästner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *Proc. Int'l Software Product Line Conference*, 2007.

[9] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. on Software Engineering*, 2008.

[10] S. Klapproth. Analysis of feature interactions in modular designs. Master's thesis, University of Magdeburg, 2008. (German).

[11] T. Leich, S. Apel, and L. Marnitz. Tool support for feature-oriented software development: FeatureIDE: an eclipse-based approach. In *OOPSLA workshop on eclipse technology eXchange*, 2005.

[12] M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan. Efficient compilation techniques for large scale feature models. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, 2008.

[13] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming*. 1997.

[14] N. Siegmund et al. Measuring non-functional properties in software product lines for product derivation. In *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*. 2008.

[15] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2006.

[16] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *Proc. Int'l Conf. on Software Engineering*. 2009.

[17] J. White, A. Nechypurenko, E. Wuchner, and D. Schmidt. Optimizing and automating product-line variant selection for mobile devices. In *Proc. Int'l Software Product Line Conference*, 2007.

---

[2]http://captainfeature.sf.net
[3]http://openarchitectureware.org