FeatureIDE: An Extensible Framework for Feature-Oriented Software Development

Thomas Thüm[a], Christian Kästner[b], Fabian Benduhn[a], Jens Meinicke[a], Gunter Saake[a], Thomas Leich[c]

[a] *University of Magdeburg, Germany*
[b] *University of Marburg, Germany*
[c] *METOP GmbH, Magdeburg, Germany*

Abstract

FeatureIDE is an open-source framework for feature-oriented software development (FOSD) based on Eclipse. FOSD is a paradigm for construction, customization, and synthesis of software systems. Code artifacts are mapped to features and a customized software system can be generated given a selection of features. The set of software systems that can be generated is called a software product line (SPL). FeatureIDE supports several FOSD implementation techniques such as feature-oriented programming, aspect-oriented programming, delta-oriented programming, and preprocessors. All phases of FOSD are supported in FeatureIDE, namely domain analysis, requirements analysis, domain implementation, and software generation.

Keywords: Feature-oriented software development, software product lines, feature modeling, feature-oriented programming, aspect-oriented programming, delta-oriented programming, preprocessors, tool support

1. Introduction

Feature-oriented software development (FOSD) is a paradigm for the construction, customization, and synthesis of software systems (Apel and Kästner, 2009). A *feature* is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system (Kang et al., 1990). The basic idea of FOSD is to decompose software systems into features in order to provide configuration options and to facilitate the generation of software systems based on a selection of features. A *software product line* (SPL) de-

notes the set of software systems that can be generated from a given set of features (Czarnecki and Eisenecker, 2000).

FeatureIDE is an Eclipse-based framework to support FOSD. The main focus of FeatureIDE is to cover the whole development process and to incorporate tools for the implementation of SPLs into an integrated development environment (IDE). FeatureIDE's architecture eases the development of tool support for existing and new languages for FOSD and thus reducing the effort to tryout new languages and concepts.

Currently, the development of FeatureIDE focuses on teaching and research. FeatureIDE is used in software engineering lectures in Austin, Magdeburg, Marburg, Namur, Passau, Santa Cruz, and Torino. Before we implemented FeatureIDE, our students had to learn how to use several command-line tools each with different parameters and output, whereas our students are often used to work with modern IDEs such as Eclipse. With FeatureIDE, we provide a coherent user interface and automate tasks which previously required complex tool chains. We envision that FeatureIDE can also be used productively in future and serve as an open-source alternative to commercial product-line tools (pure::systems, 2010; Big Lever Software Inc., 2010).

FeatureIDE supports several implementation techniques for FOSD and others can be integrated with low costs. The user interface for different implementation techniques is almost identical. Hence, FeatureIDE is especially qualified for teaching and to compare SPL implementation techniques with respect to their applicability for the development of SPLs.

FeatureIDE underwent several changes since the initial development in 2004. In 2005, we presented a prototypical version of FeatureIDE (Leich et al., 2005). At that time, FeatureIDE was just a front-end for the programming language Jak of the AHEAD tool suite (Batory, 2006). The development of this tool support was costly for a research language, but we earned positive feedback from other universities using FeatureIDE for teaching. Hence, we made this effort reusable for the FOSD implementation tools FeatureHouse (Apel et al., 2009) and FeatureC++ (Apel et al., 2005), and presented FeatureIDE as a tool framework for FOSD (Kästner et al., 2009).

We present recent developments of FeatureIDE such as improved usability, new functionalities, and the newly integrated FOSD languages AspectJ (Kiczales et al., 2001), DeltaJ (Schaefer et al., 2010), Antenna (Pleumann et al., 2011), and Munge (Munge Development Team, 2011). Furthermore, we discuss the effort of extending FeatureIDE and describe how FeatureIDE is implemented and tested, while reporting pitfalls and oppor-

tunities interesting for other academic tool builders. Developers of Eclipse plug-ins get insights, as we share our lessons learned with FeatureIDE.

2. Feature-Oriented Software Development

FOSD can be used to plan and implement SPLs (referred to as domain engineering) as well as to select features and generate customized programs (application engineering). FeatureIDE supports the FOSD process and we distinguish between the following four phases.

1. *Domain analysis.* The aim is to capture the variabilities and commonalities of a software-system domain, which results in a feature model.
2. *Domain implementation.* Implementing all software-systems of the domain at the same time, while mapping code assets to features.
3. *Requirements analysis.* Requirements are mapped to the features of the domain and features needed for a customized software system are selected, resulting in a configuration.
4. *Software generation (or composition).* A software system is *automatically* built given a configuration and the domain implementation.

Domain implementation and software generation highly depend on each other. An *SPL implementation technique* describes how features are mapped to implementation artifacts and how to generate customized software systems. In this section, we introduce feature models, configurations, and SPL implementation techniques currently supported by FeatureIDE.

2.1. Feature Modeling and Configuration

In SPLs, not all combinations of features are considered valid and lead to useful software systems. A *feature model* defines the valid combinations of features in a domain (Kang et al., 1990). Feature models have a hierarchical structure, whereas each feature can have subfeatures (Czarnecki and Eisenecker, 2000). The graphical representation of a feature model is a *feature diagram* and an example is shown in Figure 1. Connections between a feature and its group of subfeatures are distinguished as *and*-, *or*-, and *alternative*-groups (Batory, 2005). The children of *and*-groups can be either *mandatory* or *optional*. A feature is *abstract*, if it is not mapped to implementation artifacts and *concrete* otherwise (Thüm et al., 2011). A feature model may also have cross-tree constraints to define dependencies which cannot be expressed

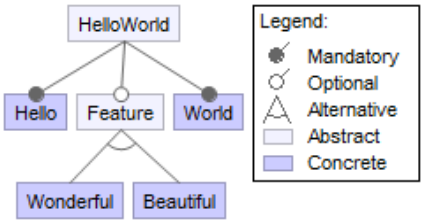


Figure 1: A simple feature model modeling an SPL of Hello World programs. The features *Hello* and *World* are mandatory and simply print the features name. The features *Wonderful* and *Beautiful* are alternatives, but not required. This SPL contains three valid Hello World programs.

otherwise. A *cross-tree constraint* is a propositional formula over the set of features and usually shown below the feature diagram.

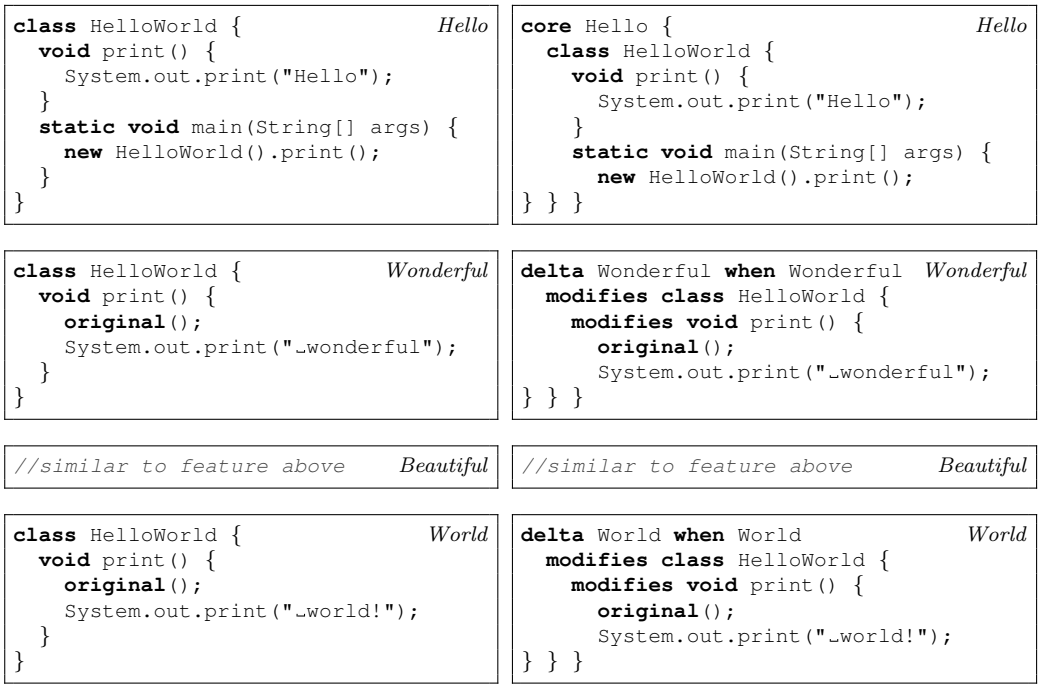
Feature models are a common notion for variability and their semantics is as follows: the selection of a feature implies the selection of its parent feature. Furthermore, if a feature is selected, all mandatory subfeatures of an *and*-group must be selected. In *or*-groups, at least one subfeature must be selected and in *alternative*-groups, exactly one subfeature has to be selected. Finally, all cross-tree constraints must be fulfilled.

A *configuration* is a subset of all features defined in the feature model. A configuration is *valid* if the combination of features is allowed by the feature model (i.e., if it fulfills the semantics of groups and all cross-tree constraints). Otherwise, the configuration is called *invalid*.

2.2. SPL Implementation Techniques

There are several techniques to implement SPLs in FOSD. The main goal is to provide a mapping between features and source code, to enable *automatic* generation of software systems for a given configuration. SPL implementation techniques are very diverse, but they are usually based on a certain language (e.g., Java or C++) to which we refer to as *host language*.

Feature-oriented programming. Prehofer (1997) proposed *feature-oriented programming* as an extension to object-oriented programming. Classes are decomposed into feature modules each implementing a certain feature. A *feature module* may contain methods and fields of several classes. Feature modules can be composed to a program based on a given configuration and an order of the features.



(a) Feature-oriented programming (b) Delta-oriented programming

Figure 2: The SPL *HelloWorld* implemented with feature-oriented programming and delta-oriented programming.

In Figure 2(a), we present an example implementation of SPL *HelloWorld* modeled in Figure 1. Feature *Hello* consists of a standard Java program with a method printing `Hello`. All other features refine this method to print further words by specifying a class and method with the same name. The keyword `original()` is specific to feature-oriented programming and indicates a call to the previous, selected feature.

Feature-oriented programming was initially introduced for Java and intended for object-oriented programming. Batory (2006) and Apel et al. (2009) applied feature-oriented programming also to software artifacts which are not object-oriented according to the principle of uniformity. The principle states that all artifacts building up a software system should be composed in the same way including documentations, specifications, or models.

Delta-oriented programming. Similarly, Schaefer et al. (2010) proposed *delta-oriented programming*. In delta-oriented programming, there is a *core module*

which is a standard application written in the host language (such as Java or C++) and a set of delta modules. A *delta module* can add, but also remove fields, methods, and classes. Furthermore, it can change the super class of an existing class. Each delta module has an *application condition* (i.e., a propositional formula over the features). Given a configuration, a core module, and a set of delta modules, a composer identifies the delta modules with a fulfilled application condition and applies them to the core module to generate a software system.

In Figure 2(b), we exemplify a delta-oriented implementation of our running example showing the syntactical differences compared to feature-oriented programming. The core module contains the standard Java program to print `Hello`. Then, we have three delta modules modifying method `print()`. In this example, classes and methods are only modified, but could be added and removed using the keywords `adds` and `removes`. Application conditions are given for each delta module after the keyword `when` and can contain feature names or logical Java expressions built from feature names.

Aspect-oriented programming. Aspect-oriented programming provides a meta language to transform existing object-oriented programs (Kiczales et al., 1997). A *join point* is a specific position in the control flow of an object-oriented program. *Pointcuts* specify a set of join points and an *advice* is a piece of code that shall be injected at a pointcut. An *aspect* may define pointcuts and advices to inject code in programs of a certain host language. By applying a subset from a set of implemented aspects, different software systems can be generated. In FeatureIDE, we assume that each feature at the feature model corresponds to an aspect with the same name, but we are aware that more complicated mappings are reasonable (Apel et al., 2006).

In Figure 3(a), we give an aspect-oriented implementation of our running example. Again, feature *Hello* consists of a standard Java program and every further feature consists of an aspect altering the method `print()`. This is achieved by defining an advice and a pointcut. The definition of a pointcut requires several new keywords such as `after()` and `call`. Keyword `call` specifies that whenever a method is called with the given signature, then the advice is executed. Keyword `after()` indicates that the advice is executed after the method call.

Preprocessors. Another technique for SPL implementation are *preprocessors* (Kästner, 2010). Special *preprocessor directives* with an application condition

```
class HelloWorld {                          Hello
  void print() {
    System.out.print("Hello");
  }
  static void main(String[] args) {
    new HelloWorld().print();
} }
```

```
aspect Wonderful {                      Wonderful
  after(): call(* HelloWorld.print()) {
    System.out.print("␣wonderful");
} }
```

```
//similar to feature above              Beautiful
```

```
aspect World {                              World
  after(): call(* HelloWorld.print()) {
    System.out.print("␣world!");
} }
```

(a) Aspect-oriented programming

```
class HelloWorld {
  void print() {
    //#if Hello
    System.out.print("Hello");
    //#endif
    //#if Beautiful
    System.out.print("␣beautiful");
    //#endif
    //#if Wonderful
//@   System.out.print(" wonderful");
    //#endif
    //#if World
    System.out.print("␣world!");
    //#endif
  }
  static void main(String[] args) {
    new HelloWorld().print();
  }
}
```

(b) Preprocessor directives

Figure 3: The SPL *HelloWorld* implemented with aspect-oriented programming and preprocessor directives.

are inserted in comments of a program in an arbitrary host language. Given a certain configuration, the preprocessor will then remove or comment out parts annotated with a false application condition.

In Figure 3(b), we give the implementation of our running example. A block belonging to a feature is started with `#if` and ended with `#endif`. Given that the features *Hello*, *Beautiful*, and *World* are selected, a preprocessor can remove the code belonging to feature *Wonderful* or comment it out as shown in our example.

3. Problem Statement

As shown, FOSD provides several techniques for the implementation of SPLs. But, each technique comes with advantages and disadvantages, and there is no consensus on the best technique. SPL implementation techniques are usually proposed together with a tool enabling the development of SPLs with the respective technique.

Comparison of SPL implementation techniques is fundamental for research on FOSD and teaching of FOSD. For research, large case studies must be established with each technique to assess strengths and weaknesses.

For teaching, a low learning curve is necessary to teach several techniques and their basics. Both, large case studies and a low learning curve make sophisticated tool support indispensable.

SPL implementation techniques are often only supported by command-line tools or do not support all phases of FOSD. Command-line tools are hard-to-use for beginners, especially if several of them, with different parameters each, are needed to compare SPL implementation techniques. Furthermore, it is hard to develop large-scale case studies without helpful support of an IDE such as code completion, code navigation, or code visualizations. Usually, those tools only support software generation and further phases like domain analysis, requirements analysis, and domain implementation are not considered. Clearly, not every new implementation technique in FOSD research can be supported by a commercial-quality IDE built from scratch.

The objective of FeatureIDE is to reduce the tool building effort for new and existing SPL implementation techniques by providing domain analysis, requirements analysis, domain implementation, and software generation in an extensible framework. The idea is to reuse views, editors, and structures as much as possible between different SPL implementation techniques. Especially domain analysis and requirements analysis have good opportunities for reuse, as they are almost identical for all techniques.

4. Related Tools

Feature modeling and configuration. There are many tools for feature modeling with graphical or textual notations. Many of these tools also support developers in creating a valid configuration. The two main commercial products for this purpose are pure::variants (pure::systems, 2010) and Gears (Big Lever Software Inc., 2010). In open source development, especially KConfig (Zippel and contributors, 2011) and CDL (Veer and Dallaway, 2011) are well known. Both provide a textual modeling notation and a configuration editor that checks for valid configuration and, to some degree, helps resolving conflicts. KConfig is most prominently used for Linux, whereas CDL is primarily known from the operating system eCos (Berger et al., 2010).

In addition, several academic tools exist. In GUIDSL, a feature model is written as a grammar and the tool allows to create and save configurations using a generated form (Batory, 2005). The focus of GUIDSL is to provide explanations, why a certain feature cannot or has to be selected. The Software Product Line Online Tools (SPLOT) is a website providing a feature model

editor as a tree view, a configuration editor with decision propagation, automated analysis on feature models, and example feature models (Mendonca et al., 2009). In contrast, the FAMA framework focuses on the comparison of different solvers for the automated analysis of feature models (Benavides et al., 2007). The feature models in FAMA may also contain numerical constraints and optimal configurations can be derived. Numerical constraints are also allowed in the Feature Modeling Plug-In (FMP) (Antkiewicz and Czarnecki, 2004). FMP provides tree views for creating feature models, configurations, and partial configurations in the process of staged configuration. S^2T^2 Configurator is a tool supporting users in creating valid configurations by providing visual explanations derived from a satisfiability solver (Botterweck et al., 2009). FeatureIDE provides similar facilities.

Feature-oriented programming. The AHEAD tool suite provides several tools for feature-oriented programming (Batory, 2006). In AHEAD, the syntax of the host languages Java 1.4, XML, and JavaCC are extended by new keywords to support feature-oriented programming. AHEAD provides different composers as command-line tools. Similarly, FeatureC++ extends the syntax of C++ by new keywords and feature-oriented C++ files can be composed using command-line tools (Apel et al., 2005). FeatureHouse is a language-independent approach to create and compose feature-oriented files (Apel et al., 2009). Currently, FeatureHouse can compose Java 1.5, C#, C, Haskell, JavaCC, Alloy, and UML. All three tools take a list of features as a configuration and have no support to decide whether the configuration is valid. As we show later, FeatureIDE integrates all these tools in Eclipse and connects feature-oriented programming to feature model and configurations.

Delta-oriented programming. As delta-oriented programming is a young SPL implementation technique; there is only one tool namely DeltaJ (Schaefer et al., 2010). DeltaJ is an Eclipse plug-in based on the Xtext Framework and already provides an editor for core modules and delta modules, an outline view, a content assist and shows errors while typing. Currently, DeltaJ is based on a subset of Java (e.g., static methods are not included in the syntax). In DeltaJ, support for feature models is missing and configurations are given as a simple list of features directly in the core module. As we will show, we extended FeatureIDE for DeltaJ.

Aspect-oriented programming. AspectJ (Kiczales et al., 2001) is a widely used aspect-oriented language with mature tool support by the AspectJ Develop-

ment Tools (AJDT) (Colyer et al., 2004). AspectJ is compatible with Java 1.6 and AJDT comes with several views and a sophisticated AspectJ editor. AJDT can be used to develop SPLs, but the creation of customized software systems requires some effort, as the user needs to remove undesired aspects from the build path manually. With the AspectJ extension for FeatureIDE, we integrate aspects seamlessly into FOSD.

Preprocessors. Conditional compilation with preprocessors, such as #ifdef and #endif of the C preprocessor CPP (GCC Development Team, 2011), is a common means to implement compile-time variability. By mapping features to preprocessor macros, we can realize the vision of FOSD. Preprocessors exist for languages beyond C: Antenna (Pleumann et al., 2011) and Munge (Munge Development Team, 2011) are preprocessors for Java. Also the commercial product line tools pure::variants (pure::systems, 2010) and Gears (Big Lever Software Inc., 2010) provide configurable general purpose preprocessors that can be used with multiple languages. In addition to lexical preprocessors that process token streams, there are also several disciplined syntactic preprocessors, that process structures. Examples are CIDE (Kästner et al., 2008) and the tag-and-prune approach (Boucher et al., 2010), that map features to code structures, or fmp2rsm (Czarnecki and Antkiewicz, 2005) and FeatureMapper (Heidenreich et al., 2008) that map features to elements of graphical models. With FeatureIDE, we integrate Antenna and Munge into the FOSD process. Furthermore, CIDE reuses parts of the FeatureIDE infrastructure.

Integration of FOSD phases. Most tools discussed so far focus either on the feature modeling side or on the implementation side. However, some tools also bridge all phases similar to FeatureIDE. Pure::variants (pure::systems, 2010) and Gears (Big Lever Software Inc., 2010) allow a sophisticated mapping between features and code fragments and can be used to run preprocessors or specific compilers. In principle, they are open for extension, but their closed-source nature hinders experimentations in a research setting and limits it to predefined extension points.

Kbuild (Berger et al., 2010) and CDL (Veer and Dallaway, 2011) provide open-source infrastructures to map features to files and preprocessor macros. Based on script languages, they could also encode other mappings, but currently they are strongly associated with Linux and eCos and their implementation techniques.

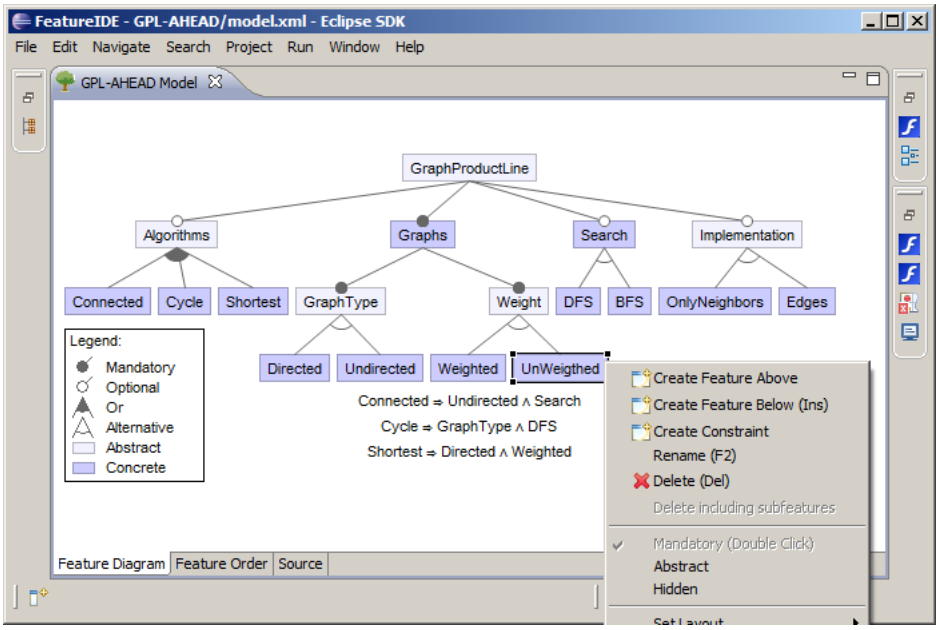


Figure 4: The graphical feature model editor of FeatureIDE. Features including possible children can be moved using drag-and-drop. Other editing functionality such as feature creation or renaming can be found in the context menu.

5. FeatureIDE: Tool Support for FOSD

FeatureIDE supports all phases of FOSD: domain analysis, requirements analysis, domain implementation, and software generation. The following sections describe how FeatureIDE supports each of these phases and the integration of all phases.

5.1. Domain Analysis in FeatureIDE

Typically, the domain analysis results in a feature model which documents the features of a domain and their dependencies. In FeatureIDE, a feature model can be constructed graphically by adding and removing features in a graphical editor (see Figure 4). In our experience, feature models can heavily change over time and thus, we also allow to move a feature including its subfeatures to a new parent feature.

The feature models created with FeatureIDE are stored in an XML format and the user can edit the feature model graphically and textually simultaneously. For integration with other feature modeling tools, we also provide

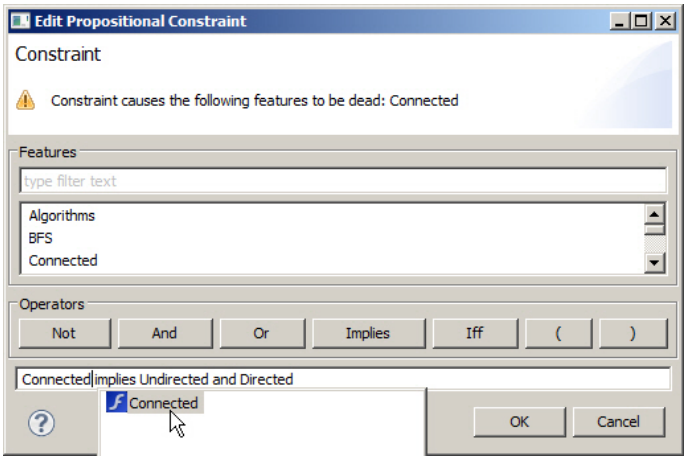


Figure 5: The editor for cross-tree constraints has a content assist and checks the syntax on the fly. If the syntax is correct, the semantics is validated. The editor can detect dead features caused by this constraint, as well as tautologies or contradictions.

export and import functionality for GUIDSL (Batory, 2005), Feature Modeling Plug-In (Antkiewicz and Czarnecki, 2004), S.P.L.O.T. (Mendonca et al., 2009), and SPLConqueror (Siegmund et al., 2008). FeatureIDE can be extended for new import and export formats, because only a reader and writer from our internal representation must be implemented. Feature models can also be stored in several graphics formats or printed to a PDF file.

A feature model may also contain cross-tree constraints. For this purpose, FeatureIDE provides a constraint editor in which constraints can be created or edited. The editor is enriched with a content assist for convenient handling as well as syntactic and semantic validity checks (see Figure 5). For example, these checks can detect mismatching brackets, dead features, false optional features, unsatisfiable constraints, and redundant constraints.

Feature models evolve over time (e.g., by adding new features or constraints) and can get unnecessary complex. Refactorings can be used to improve the readability by simplifying constraints, by removing features not occurring in any configuration, or by reorganizing the structure of the feature model (Thüm et al., 2009). FeatureIDE supports refactoring feature models using the feature model edit view. This view can classify whether the edit since the last saved version is a refactoring (the valid configurations do not change), a specialization (configurations became invalid), a generalization (configurations became valid), or an arbitrary edit.

FeatureIDE provides the first feature model editor supporting abstract features (i.e., features that do not belong to implementation artifacts). The distinction between abstract and concrete features is necessary for automated analysis and reasoning on the set of program variants which can be generated from a certain domain implementation and feature model (Thüm et al., 2011).

5.2. Requirements Analysis in FeatureIDE

Requirements analysis is supported within FeatureIDE by a configuration editor. The editor gets the feature model from domain analysis as input and offers configuration choices. The user can select required features and save the selection of features in a configuration file. Multiple configurations can be created and one configuration is marked to be the current configuration for which FeatureIDE composes and compiles source code.

The configuration editor supports developers in creating valid configurations. The editor indicates whether a configuration is valid in the current configuration step. Moreover, only configuration choices are given that exist according to the feature model (i.e., it propagates decisions) (Mendonça, 2009). For instance, a mandatory feature in the feature model cannot be eliminated and two alternative features cannot be selected at the same time. Features for which the selection cannot be changed in the current state are grayed out. Additionally, features whose selection or elimination turns an invalid configuration into a valid one are marked with a different color (see left editor in Figure 6), to support the user in creating valid configurations.

FeatureIDE also provides an extended configuration editor (see right editor in Figure 6). In the simple configuration editor, each feature can have four states: manually selected, automatically selected, automatically eliminated, and undecided. When saving the editor, all undecided features are considered as eliminated. Contrary, the extended configuration editor allows an additional state: manually eliminated. Depending on the feature model, a manual elimination may cause automatic selection and elimination of many other features (Mendonça, 2009). For instance, eliminating a feature means that none of its subfeatures can be selected anymore.

5.3. Domain Implementation in FeatureIDE

Given the features specified during domain analysis, the user can implement the domain. Domain implementation comprises the implementation of the desired software systems with a mapping between implementation artifacts and features. The mapping is necessary to automatically

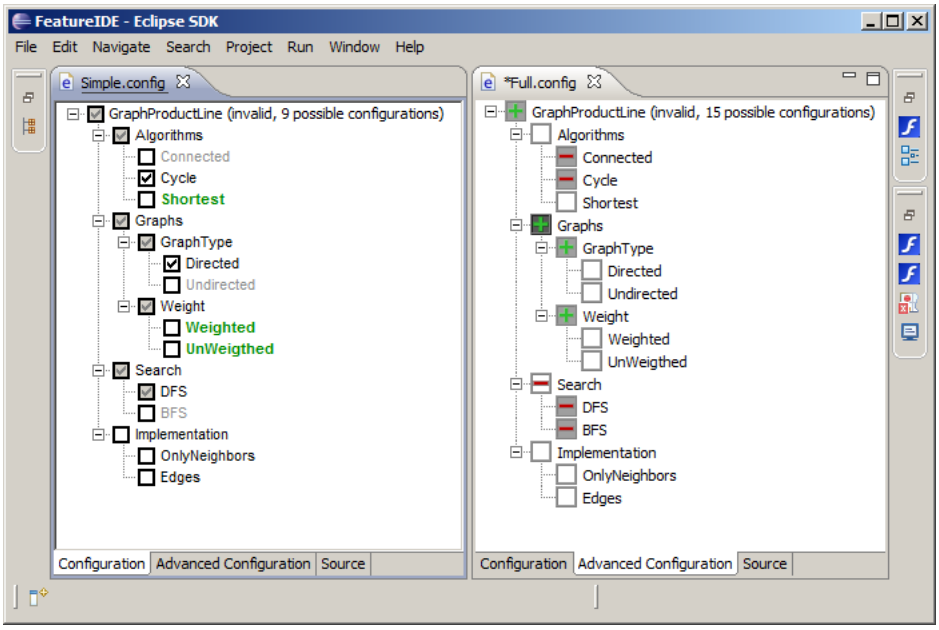


Figure 6: On the left side, the configuration editor contains an invalid configuration, in which the features *Cycle* and *Directed* are selected manually. Selecting one of the highlighted features *Shortest*, *Weighted*, or *UnWeighted* results in a valid configuration. On the right side, feature *Search* is eliminated manually in the advanced configuration editor to reduce the remaining configuration options (i.e., the features *Connected*, *Cycle*, *DFS*, and *BFS* are eliminated automatically).

generate a software system based on a selection of features. How this mapping is realized, depends on the specific SPL implementation technique and language. FeatureIDE currently supports seven SPL implementation tools: AHEAD (Batory, 2006), FeatureHouse (Apel et al., 2009), FeatureC++ (Apel et al., 2005), DeltaJ (Schaefer et al., 2010), AspectJ (Colyer et al., 2004), Munge (Munge Development Team, 2011), and Antenna (Pleumann et al., 2011).

The support for domain implementation differs between the tools. For aspect-oriented and delta-oriented programming external Eclipse plug-ins exist already, but with no support for domain analysis and requirements analysis. For feature-oriented programming and the preprocessors Antenna and Munge no external plug-ins exist and thus FeatureIDE needs to provide tool support for all stages. We give an overview of the tool support for the seven tools regarding domain implementation and software generation in Table 1.

	AHEAD *Java 1.4*	FeatureHouse *Java 1.5*	FeatureC++ *C++*	DeltaJ *Java Subset*	AspectJ *Java 1.6*	Munge *Java 1.6*	Antenna *Java 1.6*
Syntax Highlighting	PL	PL	PL	PL	PL	P	P
Content Assist	F	F	P	P	P	P	P
Outline View	F	F	F	F	F	P	P
Collaboration Outline	PL	PL	PL			PL	PL
Collaboration Diagram	PL	PL	PL			PL	PL
On-the-fly Error Checking			P	P	P		P
Error Propagation	P	P	P		P	P	P

Table 1: FeatureIDE supports SPL implementation tools with varying functionality. P indicates that a functionality is only provided for one product at a time, F for one a feature, and PL indicates that the functionality covers the entire SPL.

Editors with Syntax Highlighting and Content Assist. SPL implementation techniques usually rely on a host language such as Java or C, and build the new concepts on top of it. We were able to reuse editors of host languages for AHEAD, FeatureHouse, FeatureC++, Munge, and Antenna. For DeltaJ and AspectJ, editors are provided by external plug-ins. Syntax highlighting is available for all languages, whereas for Munge and Antenna there is no syntax highlighting for code not selected in the current configuration. For AHEAD, FeatureHouse, and FeatureC++ syntax highlighting is available for all features except the new keywords such as `original` (see Figure 7). A content assist is also available for all implementation techniques. For AHEAD and FeatureHouse the content assist only proposes methods introduced in the currently edited feature (see Figure 7), while for all other implementation techniques the whole code of the current configuration is available.

Outline View and Collaboration Outline. For Munge and Antenna, the outline view of Eclipse contains methods and fields of a class, but only those selected by the current configuration. For all remaining techniques, the Eclipse outline provided by host language plug-ins does only show the content of the currently edited feature (see Figure 7). FeatureIDE provides another view for the development of SPLs. The collaboration outline contains all members of all features, providing an easy navigation and overview (see Figure 7).

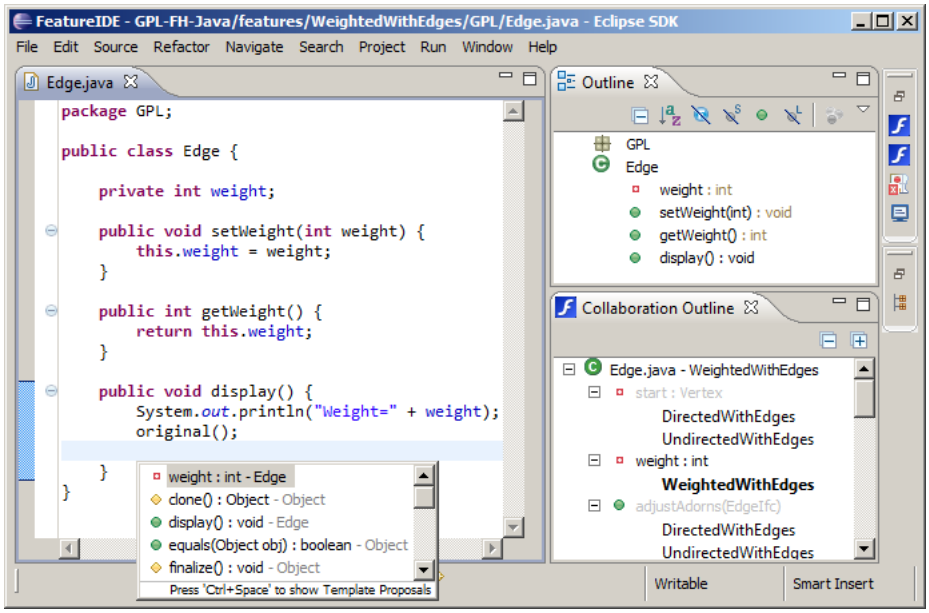


Figure 7: FeatureIDE reuses the standard Java editor (left editor) for the development of FeatureHouse source files. Content assist and the Eclipse outline (top view) provide data from the current feature only. Contrary, the collaboration outline (bottom view) shows also class members introduced in other features and enables easy navigation.

Collaboration Diagram. FeatureIDE supports domain implementation with an additional view. In FOSD, the standard means to visualize the mapping between features and code artifacts is a collaboration diagram (Apel, 2007). Basically, a collaboration diagram is a table where rows represent features and columns represent classes (see Figure 8). For feature-oriented programming, a cell represents a role (i.e., the fields and methods a feature introduces to a certain class). For other SPL implementation techniques, we provide similar mappings (e.g., for a preprocessor a cell contains all preprocessor directives containing a feature in a certain class). Especially useful for large projects is that the collaboration diagram can be filtered to show only subsets of classes or features.

On-the-fly Error Checking. On-the-fly error checking is the ability to check the source code for errors while the user is editing the code (i.e., the user does not need to save a file to get compiler errors). Checking errors on-the-fly strongly relies on incremental compilation, where only changed files or only parts of files are recompiled. On-the-fly error checking is supported for

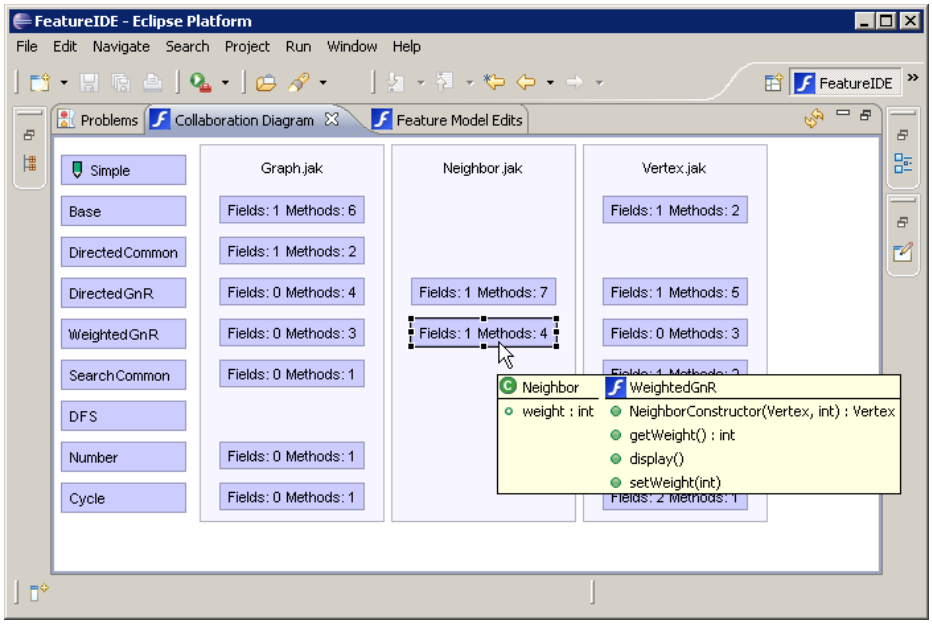


Figure 8: A collaboration diagram is a table where rows represent features and columns represent classes. Each cell is a *role* and may add certain fields and methods to a class, which are displayed in tool tips. The collaboration diagram can be filtered by features and classes.

FeatureC++, DeltaJ, AspectJ, and Antenna. This functionality is developed for DeltaJ and AspectJ in external plug-ins. For FeatureC++ and Antenna, FeatureIDE is able to reuse on-the-fly error checking from the host language plug-in, namely C/C++ Development Tooling (*CDT*) and Java Development Tools (*JDT*). The reuse is possible as Antenna works in-place and JDT does not need to know about the code generation and as C++ allows to specify the initial location of source code. But in all cases, errors are just reported for the current configuration and not the entire SPL.

5.4. Software Generation in FeatureIDE

We described how FeatureIDE supports domain analysis, requirements analysis, and domain implementation. Using the feature model, a valid configuration, and implementation artifacts, software systems can be automatically generated within FeatureIDE. The build process is realized as usual for Eclipse; the build can be triggered on demand or if the automatic build option is enabled, the source files are composed and compiled whenever one

of the source files changes. For compatibility towards host-language development tools, FeatureIDE only composes files to a special folder. This folder is then used as the input folder for the compiler of the host-language. Hence, all build options like build path or build parameters can be configured as known from host-language development tools (e.g., JDT or CDT).

The integration of existing composition tools in FeatureIDE is diverse. The AHEAD tool suite, FeatureHouse, and Munge are integrated as Java libraries. FeatureC++ is included using operating system specific executables. Contrary, AspectJ and DeltaJ exist as separate Eclipse plug-ins and are installed independently.

Error Propagation. In FeatureIDE, the reuse of existing Eclipse plug-ins for compilation has proven to be successful. For example, if an error in AspectJ code exists, AJDT will generate an error marker at the correct position (i.e., where it occurred). Usually, error markers from the host language compiler will appear at generated files, but this is not the position where the error should be fixed.

Error propagation is the ability to locate the source of an error for which only the position in the generated code is known. The preprocessors Munge and Antenna come with a useful property regarding error propagation: when preprocessing the code, the tools do not remove line breaks. Hence, the line numbers are identical and error markers can easily be relocated. FeatureC++ supports the error propagation by itself: the C++ compiler supports the C preprocessor and FeatureC++ generates preprocessor code helping the C++ compiler to find the origin of an error.

The error propagation for all other languages is more complicated. AHEAD provides annotations in the composed Java code helping to identify the origin of certain source code lines. Figure 9 illustrates a Java compiler error added as an error marker at the correct line of the Jak source file. We extended FeatureHouse to support error propagation within FeatureIDE. Error propagation for DeltaJ is not yet implemented and part of future work.

Execution of Software Systems. Generated software can be started within FeatureIDE as known from host-language plug-ins. Similarly as for compilation, FeatureIDE focuses on compatibility with other plug-ins and tools. Run configurations can be created and used as usual, with full support for parameters and environment configuration. In FeatureIDE, compiled programs can be executed for all integrated languages.

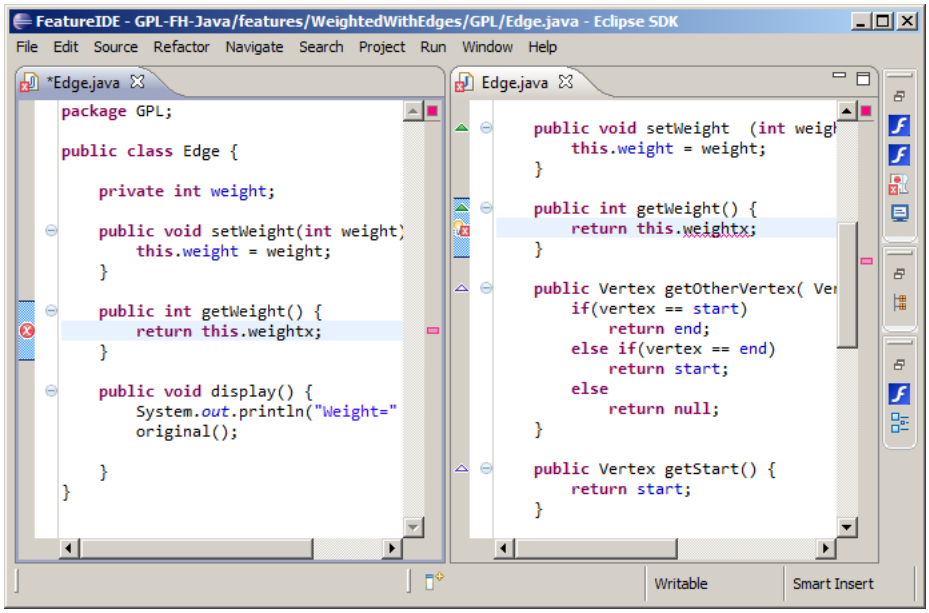


Figure 9: Compiler errors created by the Java Development Tools (right editor) are propagated to the original source file (left editor).

Generation and Compilation of all Products. In FeatureIDE, all sources are generated and compiled in background for one particular configuration only. But, any valid configuration may actually contain compiler errors such as dangling method references. FeatureIDE provides two options to uncover such errors. First, the user can trigger the automatic generation and compilation of all products (i.e., all valid configurations). The generation of all products is usually only applicable to smaller SPLs, because the number of products might be exponential in the number of features. Second, the user can trigger the automatic generation and compilation for all manually created configurations. The latter option can be used to detect all compiler errors in configurations delivered to customers.

5.5. Integration of All Phases in FeatureIDE

FeatureIDE not only provides support for each phase of FOSD, it also supports dependencies between particular phases. When changing the feature model, all configurations are checked for validity. Similarly, when features are renamed, the change is propagated to configurations and the domain implementation. Furthermore, when editing feature model and configura-

tions simultaneously, FeatureIDE synchronizes changes. When referencing features in the domain implementation, FeatureIDE checks that features do exist. Finally, for preprocessors, FeatureIDE analyses whether application conditions are tautologies or contradictions according to the feature model.

To summarize, FeatureIDE supports domain analysis using a graphical feature model editor with inconsistency detection. Requirements analysis is supported by an configuration editor with decision propagation. FeatureIDE facilitates seven languages for domain implementation with syntax highlighting, content assist, and an outline view. For most languages, a collaboration diagram, a feature outline, on-the-fly error checking, and error propagation is available. Finally, FeatureIDE integrates all phases of FOSD.

6. Implementation and Usage

In this section, we share implementation details relevant to developers who want to extend FeatureIDE or integrate it in other tools. We describe the overall architecture of FeatureIDE and the connection to other academic tools. We briefly discuss how FeatureIDE can be extended. Finally, we share our experiences regarding development effort.

6.1. FeatureIDE's Architecture

FeatureIDE's architecture underwent several changes since its initial development as a front-end for AHEAD. We continuously improved the architecture of FeatureIDE to achieve a high reuse in the development of tool support for FOSD. High reuse was especially necessary to realize our vision of an IDE for several FOSD languages.

We achieved software reuse by implementing FeatureIDE itself using FOSD. FeatureIDE consists of several Eclipse features which can be chosen optionally to generate different IDEs (e.g., an IDE with support for feature-oriented programming and preprocessors). In Figure 10, we show a feature diagram specifying the dependencies between FeatureIDE's features.[1]

The FeatureIDE framework is split into feature *FeatureModeling* realizing feature modeling and configuration functionality and the remaining framework represented by feature *FeatureIDE* including creation wizards, collaboration diagram, collaboration outline, and an extensible FeatureIDE project

[1]In Eclipse terminology, a feature is a bundle of one or more Eclipse plug-ins that can be installed independently by users.

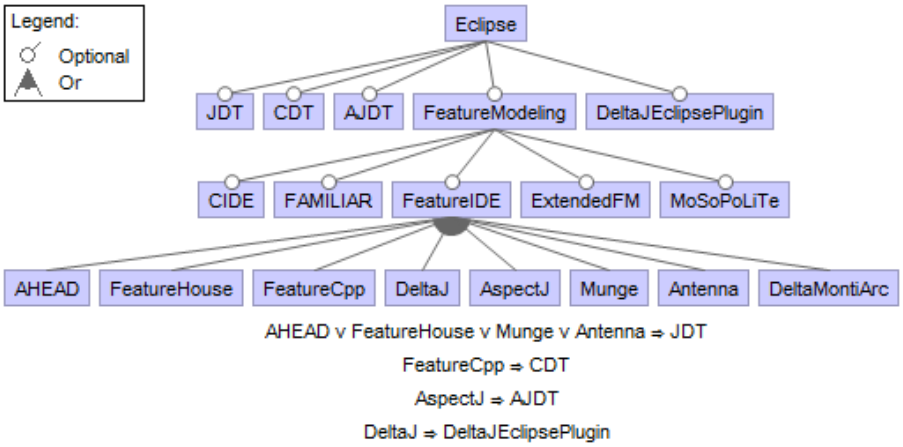


Figure 10: A feature model specifying the features of FeatureIDE and their valid combinations. FeatureIDE extensions require the FeatureIDE framework which itself requires the feature *FeatureModeling*. The latter was outsourced from FeatureIDE that it can be reused in other Eclipse plug-ins such as *CIDE*, *FAMILIAR*, *ExtendedFM*, and *MoSoPoLiTe*. FeatureIDE extensions also rely on external Eclipse plug-ins such as *JDT*, *CDT*, *AJDT*, and the *DeltaJEclipsePlugin*.

builder. Currently, there are seven FeatureIDE extensions namely *AHEAD*, *FeatureHouse*, *FeatureC++*, *DeltaJ*, *AspectJ*, *Munge*, and *Antenna*.

As in most product lines, there are dependencies between the features of FeatureIDE. First, feature *FeatureIDE* requires feature *FeatureModeling* as FeatureIDE relies on domain and requirements analysis. Second, every FeatureIDE extension requires the FeatureIDE framework. Third, FeatureIDE requires at least one extension to support all phases of FOSD. Otherwise, the IDE could not be used for domain implementation and software generation.

Furthermore, FeatureIDE has dependencies to external Eclipse features and plug-ins. These dependencies are described in the cross-tree constraints below the feature diagram in Figure 10. The FeatureIDE extensions *AHEAD*, *FeatureHouse*, *Munge*, and *Antenna* require JDT (e.g., the Java editor and builder). Equivalently, the feature *FeatureC++* requires CDT, *AspectJ* requires AJDT, and *DeltaJ* is based on the *DeltaJEclipsePlugin*.

Besides the FeatureIDE extensions presented in this paper, parts of FeatureIDE were reused in other tools for SPLs or are currently being integrated in other tools:

- *CIDE* is an IDE for the development of SPLs using virtual separation

of concerns (Kästner et al., 2008). Virtual separation of concerns is similar to preprocessors, but code fragments are annotated with colors and in a disciplined way.

- FAMILIAR is a domain-specific language that can be used to define, manipulate, and merge feature models (Acher et al., 2011). A tool for FAMILIAR provides support for feature model analysis based on FeatureIDE.

- Δ-*MontiArc* is a language for delta modeling of software architectures developed in Aachen and recently integrated into FeatureIDE as an extension (Haber et al., 2011).

- The feature modeling in FeatureIDE is extended in Passau to handle feature attributes and to provide reasoning based on pseudo-boolean satisfiability (Henneberg, 2011).

- The *MoSo-PoLiTe* framework for model-based and combinatorial testing of SPLs is currently based on a commercial tool and is going to use *FeatureModeling* in the future as a non-commercial alternative (Oster et al., 2010, 2011).

6.2. Extending FeatureIDE

FeatureIDE is open-source and published under General Public License. We encourage others to use and extend it. Currently, all sources are available via Apache Subversion (SVN), but we are planning to migrate to a distributed revision control system to ease the development for third-parties.

As Eclipse, FeatureIDE is built as a framework. That is, there are explicit extension points, where other plug-ins can contribute. Currently, we have extension points to integrate new composers and to provide extensions for feature model editor and configuration editor. We constantly improve extension points based on the needs of third-party extensions and our own extensions. But, FeatureIDE may also be reused by instantiating and using classes from our plug-ins as done in CIDE or FAMILIAR.

For example, if you want to extend FeatureIDE by a further SPL implementation technique, you have to create a new Eclipse plug-in extending a single extension point named `de.ovgu.featureide.core.composers`. We provide an abstract class `ComposerExtensionClass` as a default implementation of a composer. The abstract class provides default behavior for

extensions, but you should at least implement the method `performFull-Build(f,c)` generating or composing the source code for a given configuration `c` into the folder `f`. If you want to provide a collaboration diagram for your extension, you should implement method `buildFSTModel()` storing the data into a structure given by the FeatureIDE framework. A detailed description containing all technical steps can be found on our website.[2]

6.3. Development Effort

We describe the development effort required to implement FeatureIDE, so that others can better predict the effort needed to integrate new languages and tools into FeatureIDE. The initial development of FeatureIDE was very costly. From 2004 to 2008, we just worked on the integration of AHEAD into Eclipse. But it turned out that the integration of further languages was much faster, because we could reuse much of our effort. From 2009 to 2011, we integrated six further extensions into FeatureIDE with the same amount of developers. The reduced costs resulted from well-designed extension points within the FeatureIDE framework and high reuse potential between different FOSD tools. We give some measures in terms of code size and concrete time effort needed to develop FeatureIDE extensions.

Size of FeatureIDE Extensions. We argue that the integration of SPL implementation tools requires only a low development effort compared to the effort to develop FeatureIDE. A common measure for the development effort are the lines of code (LOC); we illustrate the LOC for each FeatureIDE extension and the FeatureIDE framework in Figure 11. The FeatureIDE framework is split into the parts realizing the feature modeling and configuration functionality and the remaining framework including creation wizards, collaboration diagram, and an extensible FeatureIDE project builder. The FeatureIDE framework has a total of 37,756 LOC.[3] In contrast, all seven extensions sum up to only 5,406 LOC. The smallest extension is DeltaJ with 1% and the largest FeatureHouse with 5% of the framework's size.

All FeatureIDE extensions require a similar effort to integrate the composer; between 372 LOC for Munge and 711 LOC for AHEAD. Error propagation required the most effort for FeatureHouse (529 LOC) and AHEAD

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

[3] All LOC measures were retrieved with the Eclipse plug-in Metrics.

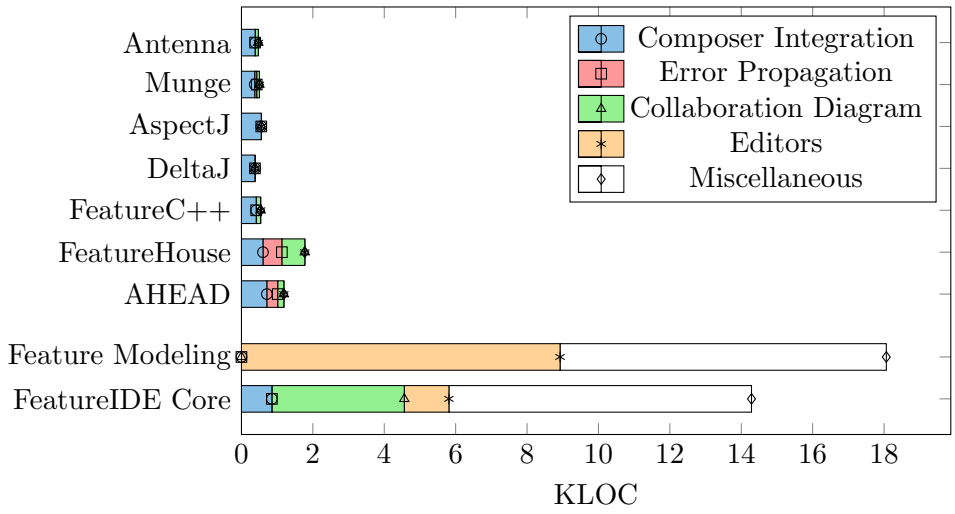


Figure 11: The code size of the FeatureIDE framework (the two lower bars) is much higher than that of the seven FeatureIDE extensions (i.e., a large part of the functionality can be reused by each FeatureIDE extension). For example, the support for feature modeling, deriving valid configurations, the synchronization between feature models and configurations, constraint creation, feature renaming, the visual components of the collaboration view, and generic file creation wizards can be reused.

(305 LOC); for other extension it was simple (Munge), not necessary (AspectJ, Antenna, FeatureC++), or is not yet implemented (DeltaJ) as explained in Section 5.4. Providing the data for the collaboration diagram required the most effort for FeatureHouse (642 LOC), is not yet implemented for (AspectJ and DeltaJ), and was simple for all other extensions.

These results show that the effort needed to extend FeatureIDE for a SPL implementation tool is low compared to the framework's size. Furthermore, other SPL implementation techniques than feature-oriented programming fit good into the FeatureIDE framework.

Development Time. In general, we did not measure the time needed for programming certain functionalities of FeatureIDE. But, we can share the concrete development time to integrate Antenna, because it was developed separately in a student project. Two undergraduate students integrated Antenna within four days (72 man hours). Both students had some experience with Java programming, worked with Eclipse as an end-user before, and participated in a two day tutorial about SVN, Eclipse programming, and

FeatureIDE in advance. Writing an IDE for Antenna from *scratch* would have taken months regarding to our experience with AHEAD.

7. Lessons Learned

The development of FeatureIDE was not always straightforward. We want to share our lessons learned that may be helpful for other academic tool builders especially for tools based on Eclipse. You might want to skip this section, if you are not interested in tool building issues.

Getting Started with Eclipse. Writing your first Eclipse plug-in is not a trivial task. Clearly, there are good examples for editors, views, and menu entries which provide a good starting point. But, when further extensions are needed it is hard to find the right extension point.

Writing Eclipse plug-ins is different from standard Java programming. The most obvious difference is probably the Hollywood principle stating 'don't call us, we'll call you'. In the first place it is not easy to understand when plug-ins are loaded and certain code is executed. Using IFile instead of Java's class File is unfamiliar, but really worth to use. If your choice is to write Eclipse plug-ins, plan some time for learning rudimentary concepts.

Frameworks in general and Eclipse in particular require a lot of work that is invisible for end-users. For example, we decided to base our feature model editor on the Graphical Editing Framework and it took almost 100 hours to get the first features drawn on screen. This can be frustrating and developers should try to make small steps to get fast results. In our example, we should have started with an editor that simply prints all feature names into one large box instead of designing and implementing all parts of the model first.

Teaching Eclipse Programming. Some of our early student projects were not successful in writing Eclipse plug-ins, because of the steep learning curve. Then, we decided to give more guidance in terms of tutorials and applied techniques from extreme programming. Students reported that pair programming especially helped dealing with the steep learning curve of Eclipse and many errors were prevented up-front. Furthermore, we made good experiences with daily stand-up meetings in which students were asked to report progress, problems they face, and their plan for the next day (see Figure 12a).

Motivation is crucial when learning to write Eclipse plug-ins. Our most successful strategy to motivate students was to use story cards splitting their

(a) Stand-up meetings (b) Story cards

Figure 12: Techniques from extreme programming that are useful when teaching the development of Eclipse plug-ins.

overall task into small tasks (see Figure 12b). Our experience is that extending an existing plug-in is easier if students start with simple tasks that can be resolved by adding or changing less than a dozen lines. When students solved a story card and we approved their implementation, they were allowed to work on the next story card which was usually harder to resolve. We pinned all story cards at a white board giving an overview on the progress and motivating each team to be faster than other teams.

Nightly Builds and Testing. Revision control systems as SVN are crucial for collaborative development of Eclipse plug-ins, but they do not prevent developers from improper commits. We decided to take advantage of nightly builds using the build server TeamCity. Build servers are useful for frequent releases on demand, since they fully automate building and publishing.

The development team changed several times since 2004. When we reached about 10 KLOC, we faced several problems with side-effects. Especially new developers had problems to understand all consequences of their changes and thus missed to manually test some of FeatureIDE's relevant functionality. In November 2010, we decided to write automated tests solely, which turned out to be inefficient. Developers can hardly write tests for code they are not familiar with. Even code that is written by the developer himself or herself months ago requires a high effort to identify and write valuable tests. Hence, tests should be written in parallel to code changes or even in advance, because writing tests for unfamiliar code is very unproductive.

In our experience, automated regression tests are crucial for the development of Eclipse plug-ins in particular and software projects with changing development teams in general. The introduction of automated functional tests turned out to be especially useful for developers new to FeatureIDE, but also when altering complex algorithms. Our experience is that automated tests save human resources and make FeatureIDE more stable. We recommend to run tests in a fixed schedule (e.g., on a build server) to detect side-effects immediately.

Integrating Libraries and Tools in Eclipse. The integration of Java libraries or command-line programs into Eclipse plug-ins is not always straight forward. We encountered problems regarding system exits in AHEAD and unclosed writers in FeatureHouse.

Including a Java library containing a method call `System.exit()` into an Eclipse plug-in is dangerous, because reaching this command terminates the Java virtual machine and thus kills Eclipse. Basically, there are two ways to deal with this problem. First, remove all system exits and replace them by Java runtime exceptions. This should be done automatically to ease the integration of library updates. We used feature-oriented programming to refine methods containing system exits and another option is aspect-oriented programming. Second, one can use the library as a command-line program. But then, Java classes of this library cannot be reused and the data has to transferred to the Eclipse plug-in somehow (e.g., by printing on the command-line). The first option is probably easier in most cases.

When a program accesses the file system, it usually uses readers and writers to process a byte stream. A common mistake is to not close a stream after it has been used. For programs used from command line this is often no problem as references to the files are removed when the program and the virtual machine terminate. But in Eclipse all writers need to be closed properly. Otherwise certain files are still blocked for modification. Besides correcting the tool to be integrated, it may also an option to run the tool using another virtual machine.

Tool Support for Code Generation. The builder concept in Eclipse is useful for code generation tools, because existing builders can be reused for compilation. Code generation means that certain input files are used to generate source code files, which in turn can be compiled. Eclipse builders are called for Eclipse projects based on Eclipse natures and several builders can be

used by adding several natures to a particular project. For code generation, only a new builder needs to be implemented that transforms certain input files into source files. The directory containing the generated source files can then be set as input for an existing builder compiling the sources. Thus, it is not necessary to re-implement features such as incremental compilation for languages already supported in Eclipse.

The compatibility towards host-language plug-ins was not always given within FeatureIDE. Our strategy in the first place was to compile source code after generation with a manual call of a compiler. The problem was that dialogs for run configurations had to be reimplemented with all support for libraries and execution parameters. Furthermore, we started implementing incremental compilation for AHEAD, but dismissed the plan as it involves the complex implementation of language-specific type systems. The idea to let host-language plug-ins compile the sources generated by FeatureIDE solved all these issues as the plug-ins' functionality can be fully reused.

8. Conclusion and Future Work

FeatureIDE is an open-source framework for FOSD. It integrates several tools for the development of SPLs and has shown good reuse opportunities. FeatureIDE supports all phases of FOSD; namely domain analysis, requirements analysis, domain implementation, and software generation. Currently, FeatureIDE supports feature-oriented programming with AHEAD/Jak, FeatureHouse, and FeatureC++, delta-oriented programming with DeltaJ, aspect-oriented programming with AspectJ, and the preprocessors Munge and Antenna. We showed that new SPL implementation tools can be integrated with low effort.

Since 2007, we and others utilize FeatureIDE in lectures for practical experiences with different SPL implementation techniques. The feedback from students and researches helped us to improve the usability of FeatureIDE throughout the years. In the last five years, we received support requests from more than 35 different cities from 17 countries, which motivates us to continue the development on FeatureIDE in terms of new functionality as well as improved usability.

In future work, we intend to extend FeatureIDE with error propagation for DeltaJ and collaboration diagrams for DeltaJ and AspectJ. The domain knowledge from the feature model can lead to more specific and thus more

helpful content-assists, outline views, or error markers. For instance, an error message may not only indicate that a referenced method does not exist, but could also list features which provide this method. We are currently implementing product-line–aware type checking for FeatureHouse. In Passau, FeatureIDE is currently extended to support quality attributes in domain analysis and requirements analysis. Another interesting topic for investigation is the combination of several SPL implementation techniques. Finally, we hope that other researchers will continue extending FeatureIDE (e.g., for AspectC++, AspectC#, Hyper/J, or the C preprocessor).

Acknowledgments

We like to thank Marko Rosenmüller for comments on an earlier draft of this paper. FeatureIDE is supported by the METOP GmbH. We like to thank all current and former contributors: Constanze Adler, Sven Apel, Don Batory, Christian Becker, Stephan Besecke, David Broneske, Tom Brosch, Alexander Dreiling, Janet Feigenspan, Christoph Giesel, David Halm, Sebastian Henneberg, Marcus Kamieth, Stephan Kauschka, Dariusz Krolikowski, Maik Lampe, Laura Marnitz, Cyrill Meier, Marcus Leich, Melanie Pflaume, Eric Schubert, Hannes Smuracsky, Torsten Stöter, Patrick Sulkowski, Patrick Venohr, Jan Wedding, Fabian Wielgorz.

References

Acher, M., Collet, P., Lahire, P., France, R. B., 2011. A Domain-Specific Language for Managing Feature Models. In: Proc. ACM Symposium on Applied Computing (SAC). ACM, New York, NY, USA, pp. 1333–1340.

Antkiewicz, M., Czarnecki, K., 2004. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In: Proc. Workshop Eclipse Technology Exchange. ACM, New York, NY, USA, pp. 67–72.

Apel, S., 2007. The Role of Features and Aspects in Software Development. Ph.D. thesis, University of Magdeburg, Germany.

Apel, S., Kästner, C., 2009. An Overview of Feature-Oriented Software Development. J. Object Technology (JOT) 8 (5), 49–84.

Apel, S., Kästner, C., Lengauer, C., 2009. FeatureHouse: Language-Independent, Automated Software Composition. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE, Washington, DC, USA, pp. 221–231.

Apel, S., Leich, T., Rosenmüller, M., Saake, G., 2005. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). Springer, Berlin, Heidelberg, New York, London, pp. 125–140.

Apel, S., Leich, T., Saake, G., 2006. Aspectual Mixin Layers: Aspects and Features in Concert. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM, New York, NY, USA, pp. 122–131.

Batory, D., 2005. Feature Models, Grammars, and Propositional Formulas. In: Proc. Int'l Software Product Line Conference (SPLC). Springer, Berlin, Heidelberg, New York, London, pp. 7–20.

Batory, D., 2006. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In: Proc. Generative and Transformational Techniques in Software Engineering. Springer, Berlin, Heidelberg, New York, London, pp. 3–35.

Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A., 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In: Proc. Workshop Variability Modelling of Software-intensive Systems (VaMoS). pp. 129–134.

Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K., 2010. Variability Modeling in the Real: a Perspective from the Operating Systems Domain. In: Proc. Int'l Conf. Automated Software Engineering (ASE). ACM, New York, NY, USA, pp. 73–82.

Big Lever Software Inc., 2010. Gears: A Software Product Line Engineering Tool. Website, available online at `http://www.biglever.com/solution/product.html`; visited on November 4th, 2010.

Botterweck, G., Janota, M., Schneeweiss, D., 2009. A Design of a Configurable Feature Model Configurator. In: Proc. Workshop Variability Modelling of Software-intensive Systems (VaMoS). ICB Research Report. Universität Duisburg-Essen, pp. 165–168.

Boucher, Q., Classen, A., Heymans, P., Bourdoux, A., Demonceau, L., 2010. Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In: Proc. Int'l Conf. Automated Software Engineering (ASE). ACM, New York, NY, USA, pp. 333–336.

Colyer, A., Clement, A., Harley, G., Webster, M., 2004. Eclipse AspectJ: Aspect-Oriented Programming With AspectJ and the Eclipse AspectJ Development Tools, 1st Edition. Addison-Wesley Professional.

Czarnecki, K., Antkiewicz, M., 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). Springer, Berlin, Heidelberg, New York, London, pp. 422–437.

Czarnecki, K., Eisenecker, U. W., 2000. Generative Programming: Methods, Tools, and Applications. ACM/Addison-Wesley, New York, NY, USA.

GCC Development Team, 2011. The C Preprocessor. Website, available online at `http://gcc.gnu.org/onlinedocs/cpp/index.html`; visited on January 11th, 2011.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I., 2011. Delta Modeling for Software Architectures. In: Proc. Dagstuhl Workshop on Model-Based Development of Embedded Systems (MBEES). pp. 1–10.

Heidenreich, F., Kopcsek, J., Wende, C., 2008. FeatureMapper: Mapping Features to Models. In: Companion Int'l Conf. Software Engineering (ICSEC). ACM, New York, NY, USA, pp. 943–944.

Henneberg, S., 2011. Next-Generation Feature Models with Pseudo-Boolean SAT Solvers. Master's thesis, University of Passau, Germany.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., Peterson, A. S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute.

Kästner, C., 2010. Virtual Separation of Concerns: Toward Preprocessors 2.0. Ph.D. thesis, University of Magdeburg.

Kästner, C., Apel, S., Kuhlemann, M., 2008. Granularity in Software Product Lines. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM, New York, NY, USA, pp. 311–320.

Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S., 2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE, Washington, DC, USA, pp. 611–614.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., 2001. An Overview of AspectJ. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Springer, Berlin, Heidelberg, New York, London, pp. 327–354.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-Oriented Programming. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Springer, Berlin, Heidelberg, New York, London, pp. 220–242.

Leich, T., Apel, S., Marnitz, L., Saake, G., 2005. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In: Proc. Workshop Eclipse Technology Exchange. ACM, New York, NY, USA, pp. 55–59.

Mendonça, M., 2009. Efficient Reasoning Techniques for Large Scale Feature Models. Ph.D. thesis, University of Waterloo, Canada.

Mendonca, M., Branco, M., Cowan, D., 2009. S.P.L.O.T.: Software Product Lines Online Tools. In: Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). ACM, New York, NY, USA, pp. 761–762.

Munge Development Team, 2011. Munge: A Purposely-Simple Java Preprocessor. Website, available online at `http://github.com/sonatype/munge-maven-plugin`; visited on January 11th, 2011.

Oster, S., Markert, F., Ritter, P., 2010. Automated Incremental Pairwise Testing of Software Product Lines. In: Proc. Int'l Software Product Line Conference (SPLC). Springer, Berlin, Heidelberg, New York, London, pp. 196–210.

Oster, S., Zorcic, I., Markert, F., Lochau, M., 2011. MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In: Proc. Workshop Variability Modelling of Software-intensive Systems (VaMoS). ACM, New York, NY, USA, pp. 79–82.

Pleumann, J., Yadan, O., Wetterberg, E., 2011. Antenna: An Ant-to-End Solution For Wireless Java. Website, available online at `http://antenna.sourceforge.net/`; visited on November 22nd, 2011.

Prehofer, C., 1997. Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Springer, Berlin, Heidelberg, New York, London, pp. 419–443.

pure::systems, 2010. pure::variants. Website, available online at `http://www.pure-systems.com/pure_variants.49.0.html`; visited on November 4th, 2010.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-Oriented Programming of Software Product Lines. In: Proc. Int'l Software Product Line Conference (SPLC). Springer, Berlin, Heidelberg, New York, London, pp. 77–91.

Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Saake, G., 2008. Measuring Non-functional Properties in Software Product Lines for Product Derivation. In: Proc. Asia-Pacific Software Engineering Conference (APSEC). IEEE, Washington, DC, USA, pp. 187–194.

Thüm, T., Batory, D., Kästner, C., 2009. Reasoning about Edits to Feature Models. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE, Washington, DC, USA, pp. 254–264.

Thüm, T., Kästner, C., Erdweg, S., Siegmund, N., 2011. Abstract Features in Feature Modeling. In: Proc. Int'l Software Product Line Conference (SPLC). IEEE, Washington, DC, USA, pp. 191–200.

Veer, B., Dallaway, J., 2011. The eCos Component Writer's Guide. Manual, available online at `http://www.gaisler.com/doc/ecos-2.0-cdl-guide-a4.pdf`; visited on November 24th, 2011.

Zippel, R., contributors, 2011. KConfig Documentation. Website, available online at `http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`; visited on November 24th, 2011.