Otto-von-Guericke University Magdeburg

School of Computer Science
Department of Technical & Operational Information Systems

Bachelor Thesis

Requirements and design for a language-independent IDE framework to support feature-oriented programming

Author:
Janet Feigenspan

January 30, 2009

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Thomas Leich

Otto-von-Guericke University Magdeburg
School of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Feigenspan, Janet
Requirements and design for a language-independent IDE framework to support feature-oriented programming
Bachelor Thesis, Otto-von-Guericke University Magdeburg, 2009.

Contents

List of Figures

List of Tables

List of Abbreviations

AST Abstract Syntax Tree

CDT C/C++ Development Tooling

DBMS Database Management System

FOP Feature-Oriented Programming

FST Feature Structure Tree

GUI Graphical User Interface

IDE Integrated Development Environment

JDT Java Development Tools

LOC Lines of Code

OOP Object-Oriented Programming

SPL Software Product Line

Chapter 1

Introduction

*Software Product Lines (SPL*s) constitute a relatively new approach to develop software systems. They are inspired by product lines in industry, like used in the production of a car or a meal at some fast food restaurants (POHL ET AL. [PBvdL05], p. 5f). When developing a product line, for a domain of interest, e.g. cars, a collection of assets is developed. Those assets are used to compose individual products without the disadvantage customized products are afflicted with: They usually take much effort to manufacture and hence are expensive. Compared to conventionally customized products (like a suite that is tailored for one single person), products built from a product line are not as expensive and time-consuming (CLEMENTS AND NORTHROP [CN01], p. 7f). While the former have to be designed and produced from scratch again and again, the latter are assembled from assets. Similar benefits can also be found, when software systems are developed as product lines:

1. The costs of development and time to market of a software product are reduced, since software artifacts can be reused in different products without large adjustments, once the software artifacts are built and ready to use (POHL ET AL. [PBvdL05], p. 9; CZARNECKI AND EISENECKER [CE00]; CLEMENTS AND NORTHROP [CN01], p. 226f).

2. SPLs allow configuring software products according to customers needs without building every product from scratch (mass customization) (EREÑO [ELC06]).

3. Further benefits include better quality, reduction of maintenance effort as well as the ability to better cope with complexity and evolution of software systems (POHL ET AL. [PBvdL05], p. 11f).

One way of implementing SPLs is *Feature-Oriented Programming (FOP)* (BATORY ET AL. [BSR04]). It allows modularizing a program into features. For example, in a simple calculator the basic operations plus and minus as well as the input of numbers can be regarded as features. In FOP, each of those features is implemented by one *feature module*. This allows cohesive source code, since everything that concerns the feature *adding* is contained in one feature module, everything that concerns the feature *subtraction* in another feature module and the input of numbers is implemented in a third feature module. When those feature modules are combined, different kinds of calculators can be produced, for example one that allows entering numbers and adding them or one that allows entering numbers and subtracting them.

Despite the benefits the development of SPLs is associated with, they are used reluctantly (POHL ET AL. [PBvdL05], p. 16). One reason is the lack of tool support. Well developed tool support is one criterion assumed to have an effect on the success of a programming paradigm (STEIMANN [Ste06]). However, there are only few tools supporting feature-oriented SPL development so far, e.g. pure::variants (PURE::SYSTEMS [pur06]), Gears (KRUEGER [Kru08]), or FeatureIDE (LEICH ET AL. [LAMS05]).

FeatureIDE was developed at the University of Magdeburg[1] (LEICH ET AL. [LAMS05]) and is a plug-in for Eclipse[2]. FeatureIDE constitutes a *Graphical User Interface (GUI)* for the AHEAD Tool Suite (BATORY [Bat03]), which is a command line based tool that allows composing software artifacts. The only programming language supported by FeatureIDE so far is *Jak*, an extension of the Java programming language (BATORY ET AL. [BLS98]).

In order to evaluate FeatureIDE, we conducted a user study. The results showed that FeatureIDE lacks typical functionalities of an *Integrated Development Environment (IDE)*, like auto-completion and refactoring of source code, which can be found in successful IDEs like Eclipse or Visual Studio[3] (GARVIN [Gar08]). For better acceptance of the user and thus to increase the success, FeatureIDE should provide the missing functionality.

A further shortcoming of FeatureIDE is that it is not designed for extensions. However, an efficient extension mechanism is one reason for the success of Eclipse (WOLFINGER ET AL. [WDPM06]). For example, Eclipse supports the integration of further programming languages by its plug-in mechanism (CLAYBERG AND RUBEL [CR06]). The effort for integration is rather low,

[1] http://www.fosd.de/fide
[2] http://www.eclipse.org
[3] http://msdn.microsoft.com/de-de/vstudio/products/bb933731.aspx

compared to building a new IDE from scratch for each new language.

Since there is already lots of work put into the development of FeatureIDE, the question arises if based on the current work, FeatureIDE can be improved. With this thesis, we want to provide a first step towards a widely accepted tool for the development of feature-oriented SPLs.

Goals

Our primary intent is to provide a base for FeatureIDE that increases its success. We therefore define the first goal of this thesis:

1. Develop a list of requirements that FeatureIDE must fulfill in order to become more successful.

Defining a list of requirements helps to evaluate the current state of FeatureIDE. Furthermore, steps for the development of FeatureIDE are defined. Thus, work can be focused on fulfilling the requirements, not on identifying next steps. In addition to the list of requirements, we want to provide a first step for fulfilling them. Thus, we define our second goal:

2. Provide a language-independent data model for the representation of feature projects.

A language-independent data model is a base for an efficient extension mechanism, because it allows implementing several functionalities language-independently. Thus, new languages, e.g. FeatureC++ and FeatureHouse, can be integrated and use the language-independent functionality without implementing it. Instead of defining a new data model, only a mapping of projects written with FeatureC++ or FeatureHouse has to be found. Then, all functionality that is based on this data model can be used by FeatureC++ and FeatureHouse without implementing it. Hence, the effort for extending FeatureIDE and customizing it is reduced compared to implementing a new IDE for a new language from scratch.

Structure

The rest of this thesis is structured as follows. In the next chapter, we give some theoretical background. We give an introduction to SPLs and domain analysis. We

present FOP as one way to implement SPLs. In Chapter 3, we analyze which requirements a tool supporting feature-oriented SPL must fulfill and evaluate, which requirements FeatureIDE fulfills and which not. We present a data model for the representation of feature projects in Chapter 4 and evaluate how it aids FeatureIDE to fulfill the requirements in Chapter 5. After that, we look at related work and future work in Chapter 6. In Chapter 7, we summarize this thesis.

Chapter 2

Background

In this chapter, we describe the theoretical background that is necessary for understanding the remaining part of this thesis.

Since it is necessary to understand SPLs and their development in order to understand the need for tool support, we first explain what an SPL is. Based on this explanation, we introduce domain engineering, which includes necessary steps for the development of an SPL. We continue with FOP, because we focus on it as one way to implement SPLs.

2.1 Software product lines

In this section, we explain the basic concepts of SPLs. We first define the term SPL and explain it with an example. In order to understand in which cases SPLs are useful and in which not, we discuss benefits and problems that result from using SPLs instead of conventional software engineering approaches.

2.1.1 What is an SPL?

The Software Engineering Institute defines an SPL as "[...] a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." (CLEMENTS AND NORTHROP [CN01], p. 5). Since this definition contains some phrases and terms that need further consideration, we explain this definition in the next paragraphs.

According to the definition, an SPL is designed for "a particular market segment or mission", also referred to as *domain*. It can be defined as "an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area." (BOOCH ET AL. [BRJ05], p. 461). A domain always has to be defined in consensus with the people interested in it. Those people are referred to as stakeholders of the domain, e.g. managers, developers, or investors. Since an SPL is only designed for one domain, it is designed for a fixed set of requirements. However, in contrast to a single software product, from one SPL different products can be assembled, each fulfilling a different set of requirements.

Assets contain everything that should be reused. From those set of assets, a product is composed (WITHEY [Wit96]). Core assets constitute the set of assets that are common for an SPL. In contrast to core assets, product-specific assets only occur in certain products. The complete software product for a customer is built from assets that are either product-specific or core assets (CLEMENTS AND NORTHROP [CN01], p. 8). From here on, this difference is only taken into account when it is relevant for the understanding, otherwise the term assets refers to both, product-specific and core assets.

In order to represent all possible products that can be composed of assets, feature models can be used. A *feature model* can be defined as hierarchically organized set of features (KANG ET AL. [KCH+90], p. 37). It describes commonalities and differences of products of an SPL as well as dependencies between features. A *feature* can be defined as a user-visible aspect or characteristic of a system (CLEMENTS AND NORTHROP [CN01], p. 114). Since features are usually understood by both, the customer for whom the SPL is developed and by the developer of the SPL, they are used to communicate between the customer and the developer (KANG ET AL. [KLD02]). Since visualization helps to understand issues (HABERLANDT [Hab97], p. 252), the definition of a feature model usually includes a *feature diagram* (KANG [KCH+90], p. 37), a graphical representation of the feature model. In Figure 2.1, the elements of a feature diagram are depicted. If features are linked with an *and*, it means that all have to be selected. From features that are linked with an *or*, at least one has to be selected. An *alternative* that links features means that exactly one has to be selected. *Optional* means that a feature can be selected, whereas *mandatory* means that a feature must be selected.

For better understanding SPLs, we present the feature diagram of an embedded *Database Management System (DBMS)* SPL in Figure 2.2. The domain of this SPL could be a network of sensors. One example for the application of an embedded DBMS is the sensor network on Great Duck Island, which was developed to observe the behavior of ducks without disturbing them in their natural

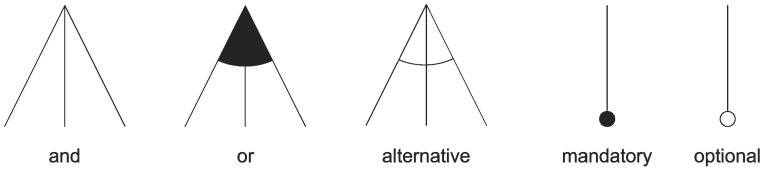


Figure 2.1: Elements of a feature diagram (BATORY [Bat05]).

behavior (MAINWARING ET AL. [MCP+02]). The root node, *EDBMS*, is called a conceptual node. All other nodes are feature nodes, whereas inner nodes, like *OS*, do not have a content, but are for structuring purposes only (THÜM ET AL. [TBK09]). If an inner node is selected, the selection applies for a certain number of its children (if a child node is an inner node again, the selection is propagated further until a leaf node is reached). In Figure 2.2, the child nodes of *OS*, *Windows* and *Linux*, are leaf nodes. Since they are linked by *alternative* and *OS* is mandatory, exactly one operating system has to be chosen. The node *Transactions* is marked with an empty circle, which means that not all products from this SPL have to provide them. Hence, *Transactions* are product-specific assets. The node *Basis* contains everything that is needed for an embedded DBMS of this SPL, like the requirements consistency and security (CODD [Cod82]) or an interface to get the data from the sensor it is saved on. Hence, *Basis* constitutes a core asset.

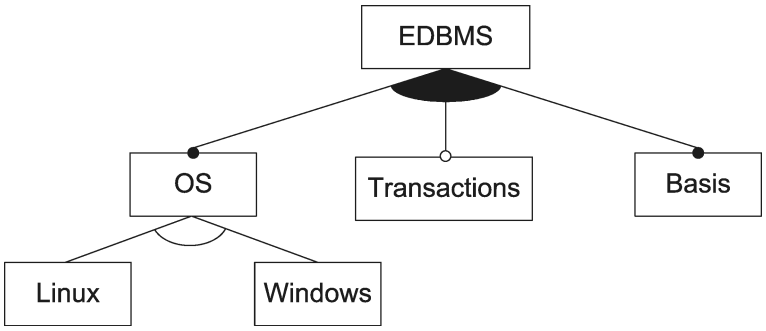


Figure 2.2: Feature model of the embedded DBMS SPL.

Based on the definition of an SPL, we can discuss advantages and disadvantages of SPLs in the next section.

2.1.2 Benefits and problems of SPLs

Having defined what an SPL is, we describe benefits and shortcomings of SPLs in this section. Compared to conventional application engineering, several benefits

have been identified:

1. Assembling a product from a set of assets saves time and reduces costs, because the assets do not need to be implemented again (POHL ET AL. [PBvdL05], p. 9f; APEL [Ape07]; CZARNECKI AND EISENECKER [CE00], p.13ff).

2. Since the same component is used in many products, it is well tested. The probability of bugs that have not been discovered yet is relatively small. This leads to better quality of products and satisfied customers (POHL ET AL. [PBvdL05], p. 11f; CLEMENTS AND NORTHROP [CN01], p. 17).

3. Since an SPL consists of several assets that can be plugged together, the complexity is reduced. Instead of having to know the complete software system at once, it suffices to understand the assets it is based on. Hence, it is easier to maintain the SPL and to cope with evolution, because it basically depends on how good the assets can be maintained and adjusted (POHL ET AL. [PBvdL05], p. 11f).

Hence, SPLs are useful, when several applications should be developed that share a common set of requirements. The number of applications for which an SPL begins to be profitable is estimated between two and three (CLEMENTS AND NORTHROP [CN01], p. 226). In Figure 2.3, this fact is visualized. The costs for developing products using conventional application engineering and using SPLs are shown. If one application is developed, the costs are significantly lower, when conventional techniques like the water fall model are applied in comparison to SPLs. However, the investment in SPLs pays off if more applications are developed. The number of applications where the costs of conventional application engineering and SPLs do not differ significantly is called the *break-even point*. Several case studies confirm the break-even point (CLEMENTS AND NORTHROP [CN01], p. 485ff; POHL ET AL. [PBvdL05], p. 413ff).

Despite the benefits the usage of SPLs is associated with, SPLs are stilled applied reluctantly (POHL ET AL. [PBvdL05], p. 16). We see two problems responsible:

1. There is still inefficient tool support for the development of SPLs. According to Steimann, a well developed infrastructure is crucial for the success of a programming paradigm. This is one potential reason why aspect-oriented programming is such a successful paradigm despite its shortcomings (STEIMANN [Ste06]).

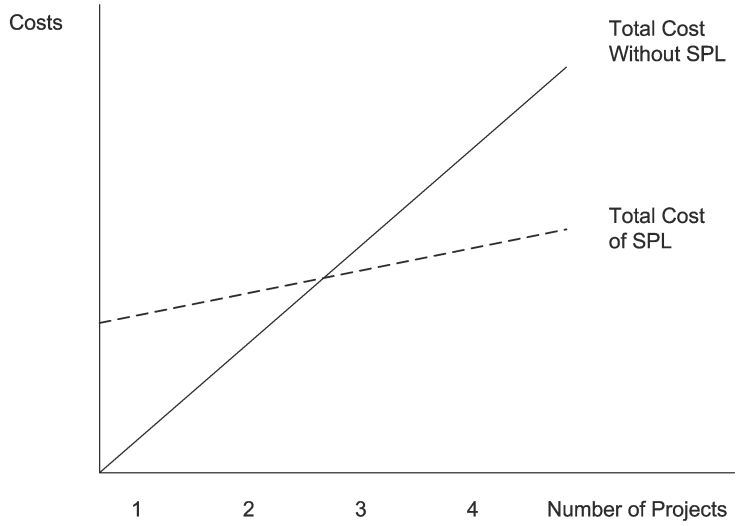


Figure 2.3: The break-even point (CLEMENTS AND NORTHROP [CN01], p. 226).

2. The above stated benefits apply only when the assets of an SPL are already implemented. Otherwise, they have to be identified, which is rather time consuming and costly compared to conventional software development.

In order to understand the effort and cost of developing an SPL and hence the need for tool support, we explain how SPLs are developed in the next section.

2.2 Developing SPLs

In this section, we first describe the water fall model, a widely used model for application engineering. This helps to understand the effort of SPL development, which we describe afterwards.

2.2.1 The water fall model

In Figure 2.4, the conventional water fall model for the development of software is depicted (PARNAS AND CLEMENTS [PC86]; ROYCE [Roy87]). This model is designed for the creation of one single software product. During the first stage of the model, the requirements that customers expect from the product are determined. Once the requirements are identified, a conceptual model of the software that should meet those requirements is developed. This model is then implemented

in the next stage. During verification, the implemented product is tested for bugs and against the requirements. If no bugs occur and all specified requirements are met, the product can be delivered to the customer and maintained. During development, one stage can be iterated several times, if the results of this stage are not satisfying. For example, if the product does not fulfill all requirements, its implementation is adapted. Interactions can only occur between two consecutive stages, e.g. implementation and verification. However, it is not intended that the requirements are adjusted, if a product does not meet them.

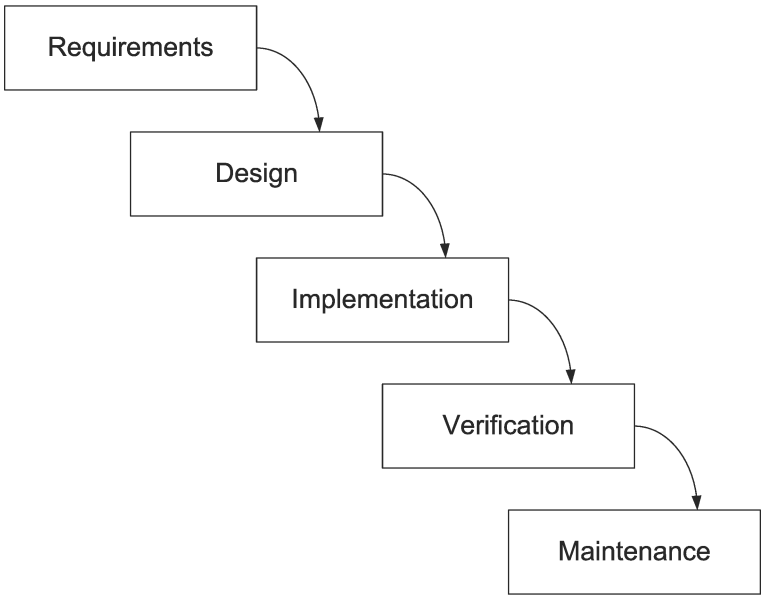


Figure 2.4: The water fall model (ROYCE [Roy87]).

The resulting software product is only applicable for the specified set of requirements. For example, an embedded DBMS developed according to the waterfall model can only fulfill a fixed set of requirements, e.g. a Linux OS without transactions. If the DBMS should also be used for a Windows system or be able to do transactions, the complete development process has to be traversed again. To avoid this effort, domain engineering was introduced.

2.2.2 Domain engineering

Domain engineering aims at avoiding the effort of implementing software from scratch where it is not necessary. It can be defined as ”[...] the process of creating an infrastructure to support systematic reuse” (FRAKES ET AL. [FPDF98]).

It consists of two processes: Domain analysis and domain implementation (see Figure 2.5).

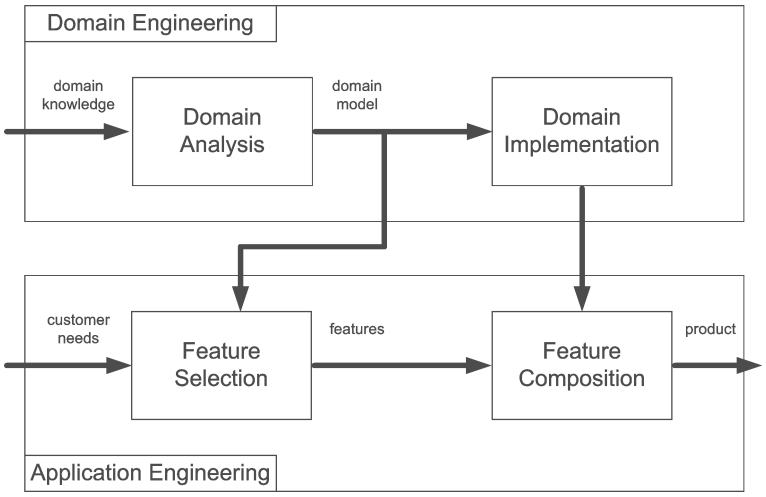


Figure 2.5: Domain & application engineering (adapted from CZARNECKI AND EISENECKER [CE00], p. 21).

During *domain analysis*, the domain of interest is identified. From this domain, the knowledge is collected and organized. The result of this process is a *domain model*, which shows the common and variable properties of the systems in a domain, the semantics of the properties and domain concepts, and the dependencies between the variable properties (CZARNECKI AND EISENECKER [CE00], p. 23). The domain model can be represented as feature diagram (cf. Figure 2.2). For example, *OS* and *Base* are common properties of an embedded DBMS, whereas *Transactions* is a variable property. In addition, a domain model can have a textual representation (BATORY ET AL. [BLHM02]). This representation allows an automatic validation of the selected features. For example, in an embedded DBMS product, the operating system *Linux* or *Windows* has to be chosen. That this is assured for an embedded DBMS product, can be checked automatically, if a logical representation of a domain model exists. For this small model, it can be easily checked by looking at the feature diagram. Usually, however, models are rather complex, so that an automatic validation is necessary.

The domain model serves as input for *domain implementation*, in which the assets are implemented. The result is a set of implemented assets that are used to compose specific products according to specific customer needs. In this context, application engineering is based on the result of domain engineering. Products are developed using the requirements defined in the domain model, which reduces the effort compared to deriving the requirements from scratch. Then, according to the

selected features, the product is composed.

In our example, when designing an SPL for embedded DBMSs, we would first determine a domain for which those DBMSs should be applicable, e.g. a network of sensors that collects information. Then, the requirements for this domain would be identified, e.g. small data size because of resource restriction, a way to input and output data, etc. Some of the products may need some transactions that allow a rudimentary data analysis. Some products should be compatible with Linux, others with Windows. All the requirements and their dependencies would be organized in a domain model and visualized, e.g. like in Figure 2.2. According to this model, the specified assets would be implemented, so that they can be combined as necessary. For example, an embedded DBMS with transactions should be composed with the same effort as an embedded DBMS without transactions. Only minor source code adjustments like changing the time intervals to which a backup of the database is created should be necessary.

To summarize, the result of domain engineering is a set of assets from which different products can be assembled, which fulfill different sets of requirements. The result of application engineering is a single product that only is applicable for a fixed set of requirements. If the requirements change, a new product needs to build, for which the complete process needs to be traversed again. This leads to a higher effort compared to just assembling the product from the set of available assets developed during domain engineering. However, this advantage only applies if the requirements have already been identified and integrated in the domain model. If not, they have to be identified and implemented first, which is more time consuming and costly than developing one single application.

From those explanations, it follows that domain engineering is a difficult process. Defining the domain too wide leads to much effort in modeling and implementing, probably more than necessary. On the other hand, if the domain is chosen too narrow, the products that can be built from the SPL are only suitable for a small set of requirements. If the domain is defined neither too wide nor too narrow, it still takes more time and effort to model the domain and implement the assets compared to application engineering, where only one single product is built. Hence, good tool support is necessary.

In the last sections, we described SPLs and how they are developed, which showed that good tool support is necessary in order to develop SPLs. In the next section, we describe FOP as a way of implementing SPLs. This is necessary to understand the requirements that a tool supporting SPLs should fulfill.

2.3 Feature-oriented programming

In this section, we give an overview of FOP. In order to aid understanding how FOP can be used, we explain tools that support FOP. We then describe how it can be used for implementing SPLs.

2.3.1 What is FOP?

FOP can be considered as an extension to *Object-Oriented Programming (OOP)* (PREHOFER [Pre97]). It is designed to modularize *crosscutting concerns*, a problem which OOP fails to solve (KICZALES ET AL. [KLM+97]). Crosscutting concerns cannot be modularized at the same time using OOP alone. No matter what criteria are used to decompose source code into modules, those concerns always crosscut several modules. However, separation of concerns is necessary for readable and maintainable source code (PARNAS [Par72]; DIJKSTRA [Dij76]). Crosscutting concerns do not allow a separation and lead to *scattered* and *tangled* code.

Scattered code means that the implementation of one concern can be found in several feature modules. For example, in an embedded DMBS product, transaction management should be integrated in order to read and write data and ensure consistency. Since writing data could violate consistency, after every writing access to the database, the consistency of the data has to be checked. An obvious idea using conventional OOP approaches would be to add a check for consistency after every piece of source code that accesses the DB. However, this implementation would scatter the transaction management over all modules that access the DB. If we want to check the implementation of the transaction management, we would have to look at nearly every module. This tracing of a concern of feature is referred to as feature traceability problem (ANTONIOL ET AL. [AMGS05]).

In order to avoid the feature traceability problem, we could decompose the source code according to the transaction management, so that everything that has to do with transactions is summarized in one module. But now, everything that accesses the DB is also in that module. We now have tangled code in this module, because it handles several concerns (KICZALES ET AL. [KLM+97]).

Hence, no matter what criteria we use for decomposing source code, there is either scattered or tangled code. This is referred to as tyranny of the dominant decomposition (TARR ET AL. [TOHS99]), because a concern that is encapsulated in one module forbids another crosscutting concern to be encapsulated in its own. Hence, OOP does not suffice for modular source code for all concerns.

This is where FOP brings further benefit compared to OOP alone: It encapsulates everything that has to do with transaction management in one unit, called feature module. Everything concerning transaction management is implemented in this feature module. There is no scattering, because everything that has to do with transactions is encapsulated in this feature module. There is also no tangling, because only source code concerning transaction management is in this feature module. Thus, FOP allows the separation of crosscutting concerns and hence supports the implementation of readable and maintainable source code.

In Figure 2.6, we visualize the modularization of concerns using FOP. The feature modules are denoted by the dashed lines, classes by solid lines. Since we only want to explain modularization of crosscutting concerns, two classes suffice. Usually, however, there are more classes in a DBMS implementation. The implementation of one class in one feature module is called *role* and depicted by a dotted line. One role encapsulates the implementation of one concern in one class. Considering transaction management, even though it is implemented in several classes, it is still modular, because it is encapsulated in one feature module. Furthermore, nothing that implements another concern is contained in this feature module. Hence, there is no scattering or tangling. The crosscutting concern transaction management is modularized in one feature module.

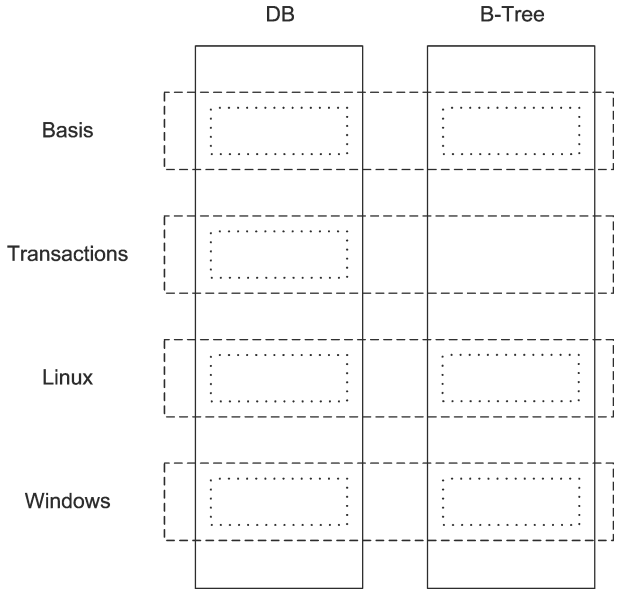


Figure 2.6: Collaboration diagram of an embedded DBMS.

How are feature modules implemented that they provide a modular structure? To answer this question, we explain several tools that support FOP in the next section.

2.3.2 Tools for feature-oriented programming

There are several mechanisms for the implementation. Since FeatureIDE provides a GUI for AHEAD, we first present the approach AHEAD uses (BATORY ET AL. [BSR04]). It extends the concept of classes established in OOP by splitting them. This way, modular source code can be created, because a class that would implement several concerns can be split, so that for every concern one fragment of the class is responsible. Furthermore, we describe how FeatureC++ (APEL ET AL. [ALRS05]) and FeatureHouse (APEL ET AL. [AKL09]) work, since we want to integrate both tools in FeatureIDE and we need some of the concepts of FeatureHouse.

Since the approaches of AHEAD, FeatureC++, and FeatureHouse can be best explained using an example, we first introduce a stack implementation which we use. We present a collaboration diagram of a stack in Figure 2.7. We choose another example, because the embedded DBMS example is too complex for showing source code examples. A lot of classes are necessary to implement a DBMS and the classes themselves usually contain so much source code that the are too complex to explain them. A stack suffices to show the concepts of AHEAD, FeatureC++, and FeatureHouse.

We defined the following features for the stack: (1) *Base*, which implements *push* and *pop*, (2) *Save*, which assures that no elements can be popped from an empty stack, (3) *Top*, which returns the top element of the stack without removing it[1], and (4) *Element*, which allows storing only elements on a stack that have a certain property, e.g. are integers. *Base*, *Save*, *Top*, and *Element* are the names of the features. `Stack` and `Element` are the classes. `Stack` implements push, pop and top, whereas `Elements` defines the elements that can be stored on the stack. The dashed lines denote feature modules, also called *collaborations*, which encapsulate the implementation of the features. The dotted lines denote the fragments of the split classes, which are called *roles*, indicating that the same class has different roles in different collaborations.

AHEAD and FeatureC++

In AHEAD, each feature module is stored in one folder, so that for every feature module a folder exists. This way, the modularization is also visible to the user. The stack class is now divided into three roles, which each implement a different

[1] Usually, top is part of a basic stack implementation. We choose to implement it as further feature to show how AHEAD and FeatureHouse work.

concern. Since each role is stored in different feature module, the concerns are separated.

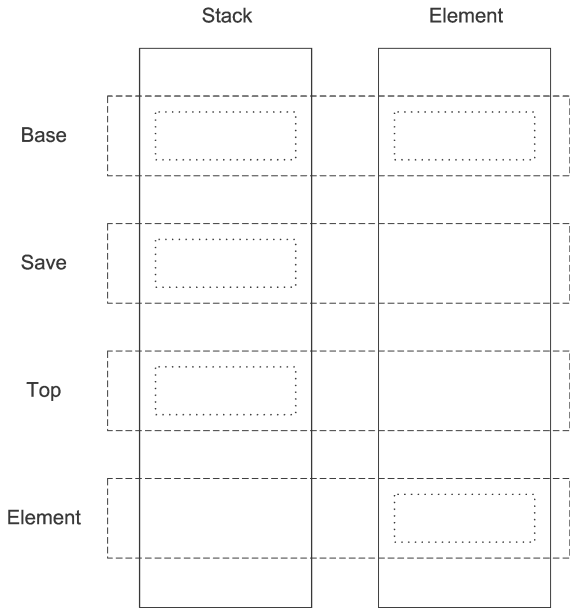


Figure 2.7: Collaboration diagram of a stack implementation.

Now that every feature is encapsulated in one feature module, they have to be composed for a software product. How does the composition work? In AHEAD, a base implementation has to exist that is part of every composed product and can be extended stepwise by further feature modules. To be able to express that one class extends another, AHEAD uses the Jak language, an extension of Java (BATORY ET AL. [BLS98]). Classes that extend other classes must be defined with the keyword `refines`. In a similar way, methods can be refined. In this case, the keyword `Super` is used, which refers to the method of the class that was processed before this class. In every refinement, new methods can be introduced. For better understandability, in Figure 2.8, an example with the class `Stack` is shown. In Figure 2.8 (a), the implementation of the stack in the feature *Base* is shown, in which the operations *push* and *pop* are implemented. In Figure 2.8 (b), the refinement of the class `Stack` in the feature *Save* is shown. In this feature, the method *pop* is extended so that no elements can be popped from an empty stack. In Figure 2.8 (c), the implementation of the feature *Top* is shown.

To achieve the composition of feature modules technically, AHEAD supports two ways: using inheritance or inlining. We first explain the inheritance mechanism. In Figure 2.9, this mechanism is shown with the stack example. During the first refinement, a new class `Stack$$Base` is defined that extends the base

```
1  class Stack {
2    LinkedList elements = new LinkedList();
3    void push (Element element) {
4      elements.addFirst(element);
5    }
6    Element pop () {
7      return elements.removeFirst();
8    }
9  }
```

(a) Implementation of the class `Stack` of the feature *Base*.

```
1  refines class Stack {
2    Element pop () {
3      if (elements.getSize() > 0) Super.pop();
4    }
5  }
```

(b) Refinement of the class `Stack` of the feature *Save*.

```
1  refines class Stack {
2    Element top () {
3      return elements.getFirst();
4    }
5  }
```

(c) Refinement of the class `Stack` of the feature *Top*.

Figure 2.8: Source code example in Jak of the stack implementation.

class with the feature *Save*. The second refinement extends the newly created class. In this example, the second refinement is also the last refinement. Hence, the class created last is named `Stack`, indicating that this class is complete. The former classes are named `Stack$$Base` and `Stack$$Save`, indicating that they are not complete yet and refined further. This approach is referred to as *mixin layers* (SMARAGDAKIS AND BATORY [SB98]). It automatically generates the described classes, when a configuration of feature is chosen. Every time a new set or order of feature is chosen, the final stack class is generated again. For example, if the features *Save* and *Top* had a different order, the first refinement would be named `Stack$$Top` and contain the method definition of *top*. *Save* would be added as last refinement and the generated class declaration would be `class Stack extends Stack$$Top`. The mixin layers allow an automatic generation of the final class according to an arbitrary set and order of features[2]. Hence, it allows modular source code and the composition of different products.

This is also possible using the inlining approach, called *jampacks* (BATORY ET AL. [BSR04]). However, the result of the composition is only one class, not

[2]If one feature is based on the implementation of another, e.g. the feature *Save* needs the feature *Base*, the order has to defined in the right way, i.e. *Base* has to be considered before *Save*.

```
1   class Stack$$Base {
2     LinkedList elements = new LinkedList();
3     void push (Element element) {
4       elements.addFirst(element);
5     }
6     Element pop () {
7       return elements.removeFirst();
8     }
9   }
10
11  class Stack$$Save extends Stack$$Base {
12    Element pop () {
13      if (elements.getSize() > 0) super.pop();
14    }
15  }
16
17  class Stack extends Stack$$Save{
18    Element top () {
19      return elements.getFirst();
20    }
21  }
```

Figure 2.9: The mixin approach of AHEAD.

an inheritance chain. In Figure 2.10, the final stack class with the features *Base*, *Save*, and *Top* is shown. The mixin approach is preferred, because it keeps the structure of the feature modules in the inheritance chain, whereas inlining destroys the structure (BATORY ET AL. [BSR04]). Thus, source code generated with the mixin approach is less complex, easier to understand and better to debug.

FeatureC++ is very similar to AHEAD and was developed at the University of Magdeburg. It extends C++ in the same way as AHEAD provides an extension for Java. It defines the keywords `refines` and `super` that have the same meaning as in Jak. For composition, it uses the mixin approach, too. Hence, we do not need to explain in further. We chose FeatureC++ for our work because of the similarity

```
1   class Stack {
2     LinkedList elements = new LinkedList();
3     void push (Element element) {
4       elements.addFirst(element);
5     }
6     Element pop () {
7       if (elements.getSize() > 0)
8         return elements.removeFirst();
9     }
10
11    Element top () {
12      return elements.getFirst();
13  }
```

Figure 2.10: The jampack approach of AHEAD.

to AHEAD and because C++ is an important programming language besides Java.

FeatureHouse

FeatureHouse was developed at the University of Passau in cooperation with the University of Magdeburg (APEL ET AL. [AKL09]). FeatureHouse composes source code files by creating a *Feature Structure Tree (FST)* of those files and superimposing the FSTs. Since for our data model we need to understand what an FST is, we first explain its structure and give an example. We then explain superimposition, the process that composes FSTs.

An FST is built from an *Abstract Syntax Tree (AST)*, but contains only information that is necessary for the specification and composition components. It represents the hierarchical structure of a file. Structural elements of a file are stored in the nodes of an FST. For example, a class implemented in Java contains several structural elements: a package- and class-declaration, methods, and fields. Each of those structural elements is stored in a node of the according FST. In Figure 2.11, the Java class `Stack` of the feature *Base* is shown (upper part) as well as the according FST (lower part)[3]. The package declaration `stack` is set as root node with the type package, the class declaration `Stack` as its child with the type class. Both nodes are called *nonterminals*, because they both have at least one child. The methods `push` and `pop` (type: method) as well as the field `elements` (type: field) are set as children of the class node. Since they do not have any children, they are referred to as *terminals*. A further difference to nonterminals is that terminals have a content, e.g. a method definition, whereas nonterminals only contain a name and a type, e.g. the class name (APEL AND LENGAUER [AL08]).

Based on FSTs, we explain how two source code files can be composed using superimposition. This recursive process starts from the root nodes and combines two nodes if the following conditions apply:

1. The parents of the two nodes have been composed (except for the root nodes).
2. The nodes have the same type and name.

If both conditions apply, the two nodes are referred to as corresponding nodes.

[3] Since it suffices for understanding superimposition to explain it with one language, we restrict our explanations to Java. However, FeatureHouse currently supports C#, C, Haskell, JavaCC, and XML, in addition to Java.

```
1   package stack;
2   class Stack {
3     LinkedList elements
4       = new LinkedList();
5     void push (Element element) {
6       elements.addFirst(element);
7     }
8     Element pop () {
9       return elements.removeFirst();
10    }
11  }
```

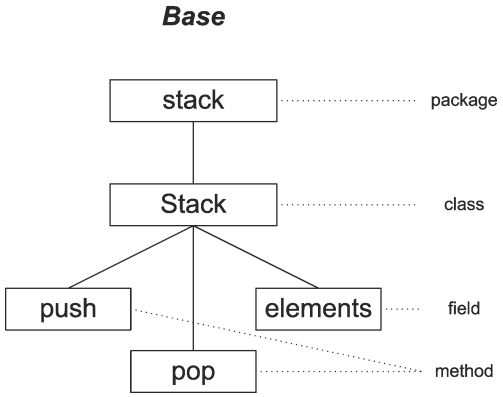


Figure 2.11: Basic stack class and the according FST.

If two nodes have been composed, the composition continues with their children. If a node has no corresponding node, it is added as child node of its composed parent node to the result FST. The process ends until all leaves are reached.

While the superimposition of nonterminals requires no special treatment, for the composition of corresponding terminals further rules are necessary. The problem here is that terminals have a content and it has to be specified what should happen with the content. If no further rules are defined, corresponding terminals cannot be composed. If terminals should be composed, the type of those nodes has to provide a rule for composition. An example rule could be: If two corresponding terminals are of the type field, they can only be composed if one node is initialized and the other is not. Another rule could be: Two corresponding nodes of the type `implements` are composed by concatenating the entries and then removing duplicates. Further rules can be found in APEL ET AL. [ALMK08] or HARRISON ET AL. [HOT06].

To make superimposition clearer, the composition of two Java-files represented as source code file (upper part) and as FST (lower part) is shown in Figure 2.12. First, the root nodes are superimposed, since they have the same name and type. Next, the nodes `Stack` are superimposed, since both rules apply. Then,

the children of `Stack` are regarded. Since `top` has no corresponding node in the *Base* FST, it is added as child of the node `Stack` to the result FST. None of the terminals of the *Base* FST has corresponding node in the *Top* FST, too. Hence, they are also added as children of `Stack` to the result FST.

```
1  package stack;
2  class Stack {
3    Element top () { return elements.getFirst(); }
4  }
```

●

```
1  package stack;
2  class Stack {
3    LinkedList elements = new LinkedList();
4    void push (Element element) { elements.addFirst(element); }
5    Element pop () { return elements.removeFirst(); }
6  }
```

═

```
1  package stack;
2  class Stack {
3    LinkedList elements = new LinkedList();
4    void push (Element element) { elements.addFirst(element); }
5    Element pop () { return elements.removeFirst(); }
6    Element top () { return elements.getFirst(); }
7  }
```

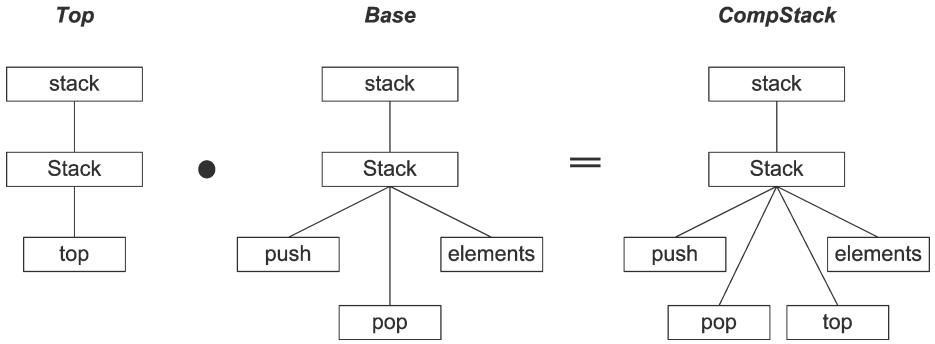


Figure 2.12: Superimposition of source code files and according FSTs.

Based on these explanations, we can describe how FOP can be used for implementing SPLs in the next section.

2.3.3 Feature-oriented SPLs

In the last section, we have introduced three tools that allow FOP: AHEAD, FeatureC++, and FeatureHouse. In this section, we explain how they can be used to

build SPLs.

The assets from which products are composed are stored in feature modules. For each feature, one feature module is created. Since in this work we mainly cover feature-oriented SPLs, from here on, we use the term feature module instead of asset. A product is composed of feature modules. At least one base feature module has to exist that is contained in every assembled product, it is mandatory. This base feature module is extended by the selected features, which can be *optional*, *mandatory*, linked by an *or*, *and*, or *alternative* and are summarized in a feature model.

In our example, a stack would be assembled of combinations of the feature modules *Base*, *Save*, *Top*, and *Element*. The composition of the feature modules would be done by the used tool. Using AHEAD or FeatureC++, an inheritance chain of the roles of the classes would be created. FeatureHouse would use superimposition of FSTs. In order to show how FOP concepts can be used for SPLs, a feature diagram for the stack is depicted in Figure 2.13. *Base* is defined as mandatory. The other features *Save*, *Top*, and *Element* are optional, hence they can be selected, but do not have to.

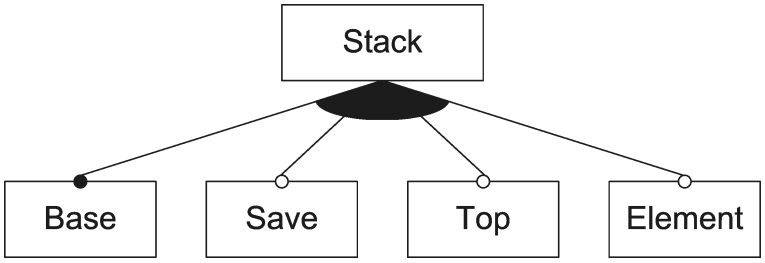


Figure 2.13: Feature diagram of the stack as SPL.

In this section, we explained tools supporting FOP. In order to understand how FeatureC++ and FeatureHouse can be integrated in FeatureIDE, which is based on the Eclipse platform, we explain the architecture of Eclipse in the next section.

2.4 Architecture of Eclipse

Since Eclipse is a successful IDE (GARVIN [Gar08]) and FeatureIDE is an Eclipse extension, we explain the architecture of Eclipse. This helps to understand the need for a language-independent framework of FeatureIDE.

Eclipse supports the integration of new languages by providing a language-independent platform. This platform already implements several functionalities

that can be used. To give an overview, the architecture of Eclipse is presented in Figure 2.14. The functionality of Eclipse is encapsulated in separated units, called *plug-ins*. The *Rich Client Platform* constitutes the base of Eclipse. It provides several functionalities, e.g. the help system of Eclipse. Based on the *Rich Client Platform*, further language-independent functionalities, like debugging or views, are provided.

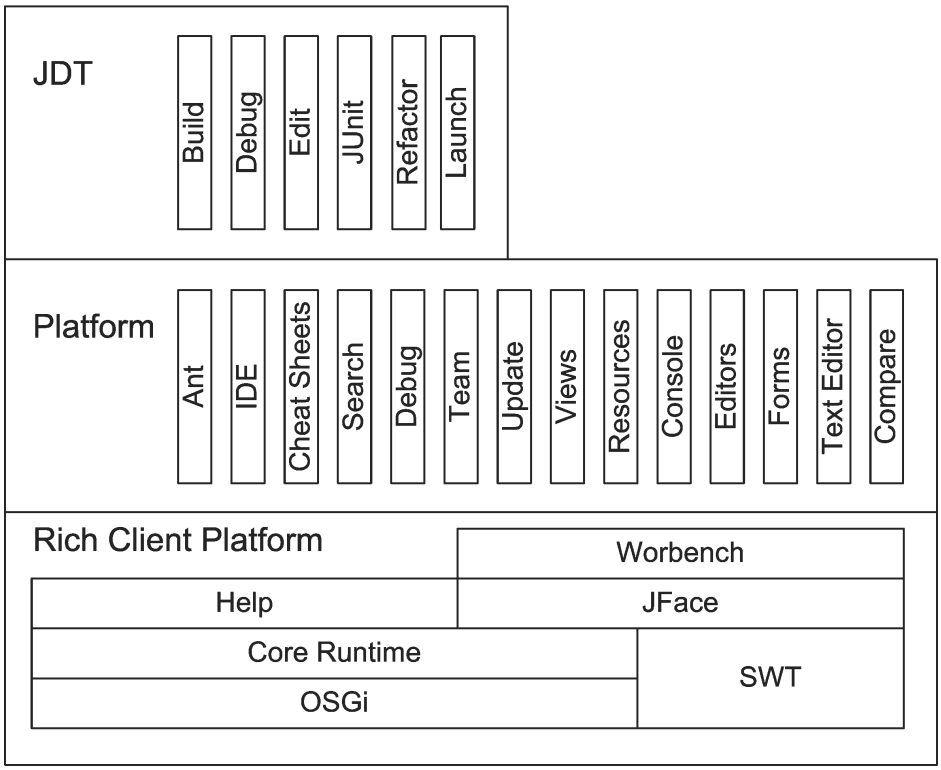


Figure 2.14: Architecture of Eclipse (KERPAN AND INTERSIMONE [KI05]).

Both, the *Rich Client Platform* and the *Platform*, provide several frameworks that can be extended. A *framework* can be defined as "[...] a reusable, 'semi-complete' application that can be specialized to produce custom applications." (FAYAD AND SCHMIDT [FS97]). Frameworks offer *hot spots* for extension, which define where and how frameworks can be extended. Extensions to frameworks are called *plug-ins*.

For example, the *Java Development Tools (JDT)* constitutes the Java extension for Eclipse. It provides Java specific functionalities, e.g. an editor with syntax highlighting. However, basic editor functionalities like *save* and *undo*, are already implemented in the framework of the editor. Since the framework is extended, *save* and *undo* do not have to be implemented again. Only Java-specific function-

ality like syntax highlighting has to be implemented.

Like JDT, further extensions of Eclipse are also based on the *Platform*, e.g. FeatureIDE and *C/C++ Development Tooling (CDT*). They also extend the frameworks they need for providing the support for FOP or programming in C and C++, respectively. Hence, Eclipse provides an efficient extension mechanism by encapsulating all language-independent functionality in frameworks and plug-ins. New functionality can be implemented by extending the provided frameworks and plug-ins.

2.5 Summary

In this chapter, we introduced SPLs and showed, in which cases they are useful. Then, we described domain engineering to show the need for tool support for SPLs. Next, we introduced FOP and showed how it can be used to implement SPLs. Since FeatureIDE integrates AHEAD, and we want to integrate FeatureC++ and FeatureHouse, we described these tools. Furthermore, we explained the architecture of Eclipse, in order to know how the integration of tools in FeatureIDE can be implemented.

However, the integration of further tools alone does not provide a good tool for feature-oriented SPLs. A tool must provide certain functionality in order to be usable. Hence, we present requirements that a tool supporting feature-oriented SPLs must fulfill in order to become successful in the next chapter.

Chapter 3

Requirements analysis

In this chapter, we derive requirements that a tool supporting feature-oriented SPLs should fulfill. We start with a language-independent framework as requirement for FeatureIDE, because this crosscuts all other requirements of this chapter. Then, we consider requirements to support the process of SPL development. To have a better overview, we present the steps of domain engineering again in Figure 3.1. This figure constitutes a simplified version of Figure 2.5. Based on our explanations in Section 2.2, we start with domain modeling. We continue with requirements for domain implementation. Since during this process a lot of functionality is useful for comfortable programming, we derive most of the requirements in this section. After the implementation, we discuss requirements for application engineering. Every section concludes with an evaluation of the fulfillment of the requirements. After deriving the list of requirements and evaluating their fulfillment, we give advice on how to fulfill the requirements.

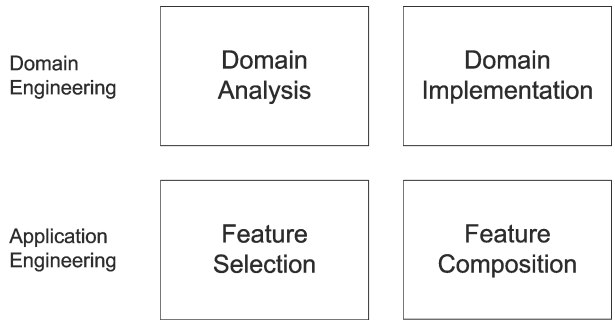


Figure 3.1: Domain & application engineering (simplified)

3.1 Language independence

We consider a language-independent framework as requirement because of two reasons: The Principle of Uniformity (BATORY ET AL. [BSR04]) and the constantly growing number of research prototypes supporting FOP (KÄSTNER ET AL. [KTS+09]).

The *Principle of Uniformity* states that software consists of source code and non-code artifacts. Those non-code artifacts need to be considered, too, when composing a product of feature modules. For example, a documentation or user manual of a product depends on the functionality of the product. An embedded DBMS without transaction management does not need to describe transactions in the documentation or user manual. On the other hand, if transaction management is part of the DBMS, it needs to be documented and described for the customer. A language-independent representation of non-code artifacts reduces the effort of composing a product, because the mechanism that is used to compose source code files can also be used to compose the non-code artifacts.

Furthermore, the number of tools that support FOP is growing. By integrating further tools in FeatureIDE, the offered functionality becomes wider and thus allows FeatureIDE to be used in more fields of applications. However, for an efficient integration of tools a language-independent framework is necessary, because otherwise, language-independent functionality that could be used by further tools would have to be implemented again.

Hence, if the success of FeatureIDE should be increased, it must provide an efficient mechanism for extensions. This would allow extending FeatureIDE by further languages without putting in all the work again that was necessary for AHEAD. Instead, if a core implementation existed that allowed to use the language-independent functionalities of FeatureIDE, extending it by further languages would become less time consuming. It also would allow integrating functionalities more easily.

For example, in Eclipse there is an extension point for the integration of refactoring. It handles adding the menu item to the menu bar and initiating the action when such an entry is clicked. Hence, the developer does not have to be concerned with the accessibility of the refactoring or to initiate the according action. Instead, the developer can concentrate on implementing the actual problem, a refactoring of source code. Thus, extending the functionality becomes more comfortable, which is one reason for the success of Eclipse (WOLFINGER ET AL. [WDPM06]). Based on these explanations, we can state our first requirement:

1. An IDE must provide an efficient mechanism for extensions.

The current version of FeatureIDE does not support extending it efficiently. When adding a new language, the existing plug-ins need to be adjusted. Furthermore, there is no language-independent representation of feature projects, which would be necessary for reusing the functionality that is already implemented, like the creation of an outline view. Hence, this requirement is not fulfilled.

This requirement crosscuts all remaining requirements that we explain in the next sections. Hence, all remaining requirements that can be fulfilled language-independently, have to be fulfilled language-independently. Otherwise, a requirement is not regarded as fulfilled.

3.2 Domain analysis

Domain analysis is a process that identifies relevant aspects of the domain for which the SPL should be developed (cf. Section 2.2.2). The result is a domain model, which captures the relevant aspects of the domain. Since the domain model is a conceptual model, it does not contain any implementation detail. Hence, domain analysis is inherently language-independent. The domain model is usually represented graphically, e.g. as feature diagram, because it is a good way to communicate it to customers. However, for a complex domain model, usually a textual representation exists, so that an automatic validation of the selected features can be done (e.g. that the dependencies between certain features are taken into account). Since both a graphical and a textual representation of a domain model is needed, an IDE must provide both representations. The graphical representation is needed for communication with a customer (KANG ET AL. [KLD02]), whereas the textual creation of a domain model is needed for an automatic validation of a selection (BATORY [Bat05]). Hence, we can state the requirement for domain analysis:

2. An IDE must support the graphical and textual creation of a domain model.

In Figure 3.2, the feature model editor of FeatureIDE is shown. It provides a graphical interface for the creation of feature models (left) as well as an editor for the textual creation (right). The textual representation uses a grammar which is mapped to feature diagrams (BATORY [Bat05]). An analysis of the implementation of the feature model editor indicated that the editor is implemented language-independently. As a test, we integrated a new language to FeatureIDE

and used the feature model editor for this new language. This test confirmed that the implementation is language-independent, because we did not need to adjust any source code to use the feature model editor. Since the support for the creation of feature diagrams is given, this requirement is fulfilled.

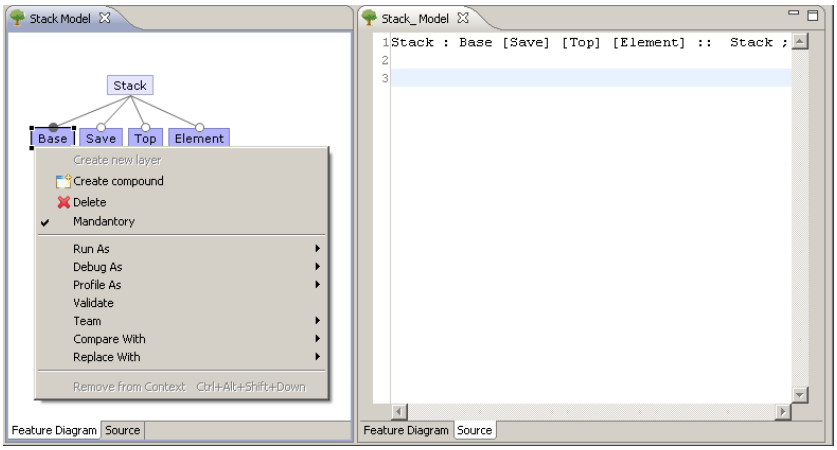


Figure 3.2: Screenshot of the feature model editor of FeatureIDE.

3.3 Domain implementation

The next step in developing SPLs is the domain implementation. This process is inherently language-dependent. A large set of functionality is helpful to increase the usability of an IDE. Since a discussion of all functionalities would exceed this thesis, we only consider a small subset of functionality. We proceed as follows: First, we describe how we select the functionalities and confirm with a user study that the selection is representative. Then, we continue with describing a functionality from the user study, stating the according requirement and evaluating, whether FeatureIDE fulfills the requirement.

3.3.1 User study

Eclipse and other IDEs like Visual Studio[1] provide a large set of functionality that aid the complete software development process (ECLIPSE [dRB06]; MICROSOFT [Mic08]). Discussing all functionality would exceed this thesis. Hence, we present a selection of the functionality that typical IDEs provide. To obtain the

[1] http://www.microsoft.com/germany/express/default.aspx

set of functionality, we proceed in two steps: We first examine the set of functionality of several IDEs. Then, we conduct a user study to confirm our selection of the functionalities.

First, we selected four IDEs of which we evaluate the functionality: Eclipse[2], IBM Rational Application Developer[3], Oracle JDeveloper[4], and Visual Studio[5]. We chose Eclipse, because FeatureIDE is an Eclipse plug-in and we want to improve FeatureIDE. Therefore, we can use some of the functionality Eclipse provides. The other three are rated the best IDEs according to a user survey conducted by the Evans Data Corporation (GARVIN [Gar08]). Eclipse was also part of this user survey, but according to Garvin another type of product, which is why it was not included in the ranking. We selected functionalities that are provided by all four IDEs. Then, we further looked for communities concerning the IDEs and the functionalities that the users discussed. The result was a list of functionalities we present in the first column of Table 3.1.

Second, to confirm that our selection of functionality is representative, we conducted a user study with seven subjects that completed a task using FeatureIDE. All subjects study at the School of Computer Science at the University of Magdeburg and attend the lecture *Erweiterte Programmierkonzepte für maßgeschneiderte Datenhaltung* (Advanced programming concepts for tailor-made data management). The task was to implement a simple chat application as SPL (APEL ET AL. [AKKL07]). Afterwards, we asked the subjects how important they regarded the functionalities of FeatureIDE. Moreover, we asked the subjects, how they liked the technical realization of the functionalities, because we can use it as indicator for determining whether the requirements are fulfilled. For importance and realization, we used a five-point Likert scale (LIKERT [Lik32]), where 1 denotes *not important/badly realized* and 5 *very important/well realized*. The results of the user study are presented in Table 3.1.

In order to be sure that we did not miss any functionality in our selection, we asked the subjects, which functionalities they wish FeatureIDE to have. None of the subjects mentioned any further functionality. Hence, we can assume that there is no important functionality missing and our list of functionality suffices to state requirements for FeatureIDE.

Since we present the functionality and according requirements in more detail in the next sections, we omit an explanation of the functionality at this point. The results of the user study can be summarized as follows: All functionalities we

[2]http://www.eclipse.org/

[3]http://www-01.ibm.com/software/awdtools/developer/application/

[4]http://www.oracle.com/technology/products/jdev/index.html

[5]http://www.microsoft.com/germany/express/default.aspx

Functionality	Importance	Realization
Compiling	4.6	3.3
Running	4.4	2.3
Syntax highlighting	4.1	2.5
Auto completion	4.1	1.7
Marking compiler problems in the source code	4.4	2.7
Outline view	3.3	2.9
Refactoring	3.4	2.6
Support for Debugging	3.9	1.6
Management of feature modules	3.7	3.6

Table 3.1: Results of the user study.

found during our search are regarded as important (they have a mean larger than 3), but only few are regarded as well realized. Only the management of feature modules and the compiling in FeatureIDE are evaluated as satisfying. To support our evaluation of the fulfillment of the requirements, we use the results of the user study. Therefore, we present the mean value of the evaluation of the realization of the functionalities in the according sections. However, since it is obvious that a requirement is not fulfilled when certain functionality is not implemented yet, we do not report the mean value in this cases again.

Based on the results of the user study, we explain the functionalities and the according requirements in the next sections. Since Eclipse is an open source project and FeatureIDE is a plug-in of Eclipse, we use Eclipse as example IDE to explain the functionality. Nevertheless, all functionality we discuss can also be found in IBM Rational Application Developer, Oracle JDeveloper, and Visual Studio.

3.3.2 Compiling and running source code

To give an overview of the functionality of Eclipse, we present a screenshot of the Eclipse workbench in Figure 3.3. As we explain the functionalities, we present fragments of this screenshot for better understandability.

Eclipse provides several features to support the development of Java projects. It integrates the Java compiler and runtime environment, so that a few clicks suffice to compile and run a program (tool bar button *Run*). The compiler is automatically called, when the user saves a file. For running a program, there is a shortcut *Run as*, which appears in the context menu, when a source code file

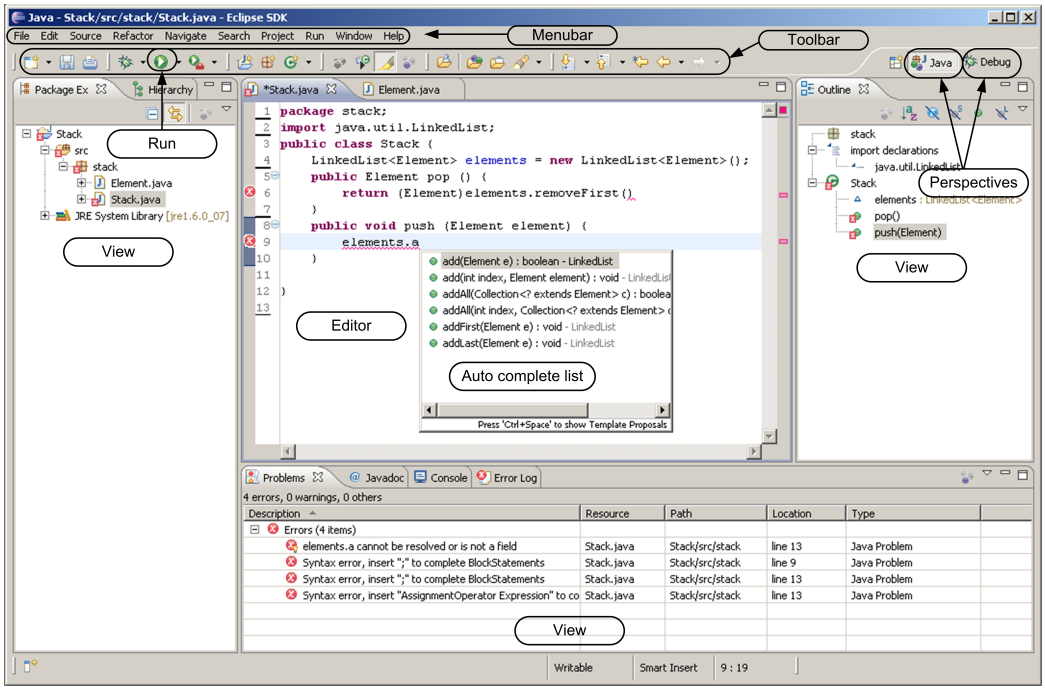


Figure 3.3: Screenshot of the Eclipse workbench.

is right clicked. This automatically defines a run configuration and executes the program, if the selected source code file contains a main type. When this run configuration already exists, the toolbar button *Run* can be used. Hence, the first requirement for domain implementation is:

3. An IDE must support comfortable compiling and running source code.

Compiling source code written in Jak is as comfortable as compiling source code in Java. When a file is saved, it is compiled automatically. The subjects of our user study regarded the compiling of source code as well realized (mean: 3.3).

However, running a Jak program takes more effort than a few clicks. In FeatureIDE, *Run as* is not supported. Hence, for Jak files, the run configuration has to be defined manually. The project name and the main class have to be specified by the user. For every combination of features, a new run configuration needs to be defined. Once this run configuration is defined, running the program is done by a click on the tool bar button *Run*. This evaluation is also reflected in our user study: The subjects evaluated the running of a program as badly realized (mean: 2.3).

Since running a program is less comfortable in Jak, this requirement is only partially fulfilled.

3.3.3 Editor support

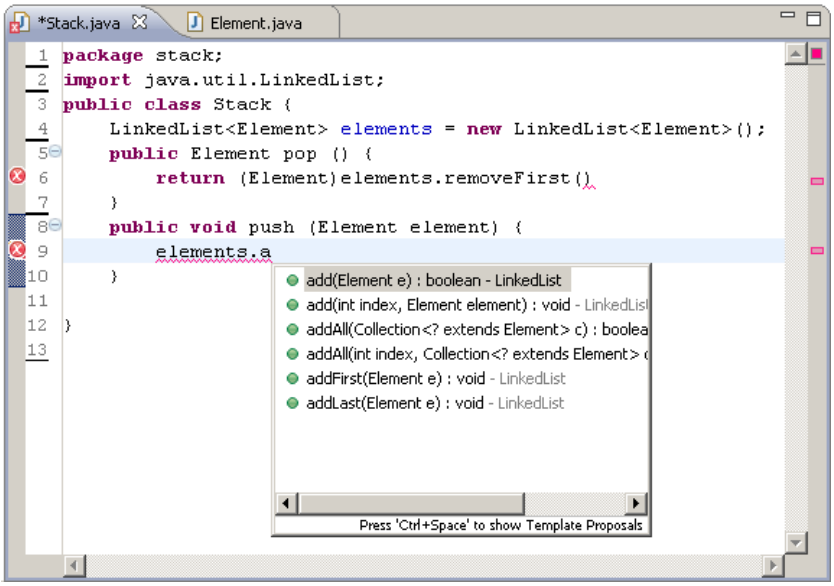
In Figure 3.4a, the Java editor of the Eclipse workbench is shown. The basic stack implementation in Java is depicted (cf. Section 2.3.2). The editor provides syntax highlighting and marks errors in the source code files, e.g. in line 6. While typing, options for auto completion are shown. For example, when starting to type the method `addFirst`, which should be invoked on the field `elements`, Eclipse displays a list of all methods that are applicable for the field `elements` and that start with an *a*. There are additional features that increase efficiency for the programmer, e.g. the automatically generation of the `get` and `set`-method for the field `element` or *quick fix*, that helps to correct errors, e.g. missing import statements or wrongly spelled variable names. Those errors are marked by a red line at the according position in the source code file, e.g. the missing semicolon at the end of line 6 in Figure 3.4a. Hovering with the mouse over the position in the source code, activates a pop up that explains the problem and provides suggestions for the solution. Since all those functionalities are part of the editor, we summarize them in one requirement:

4. An integrated editor must provide syntax highlighting, auto completion, and mark compiler errors.

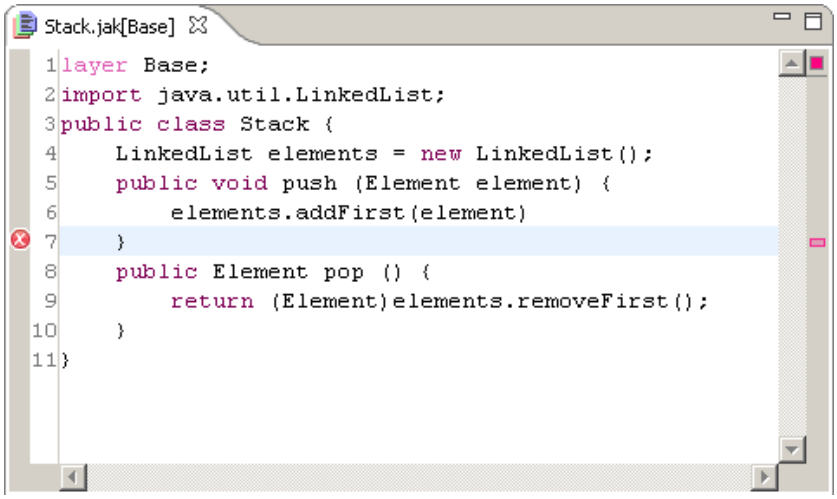
The Jak editor provides syntax highlighting (Figure 3.4b). However, subjects evaluate the syntax highlighting as rather badly realized (mean: 2.5). The reason is that the syntax highlighting for Jak is less sophisticated than for Java. The syntax highlighting for Jak only highlights keywords, comments, and String values in different colors. In Java, fields and constants are also highlighted. Furthermore, javadoc comments differ from non-javadoc comments. Hence, the syntax highlighting of FeatureIDE is improvable.

Auto completion of source code has not been implemented yet. Hence, it is not available.

Compiler errors are marked in the source code, but not as good as for Java. Firstly, the according position in the source is not underlined. Secondly, there is no pop up that describes the error and provides quick fix options for solving them. This is also confirmed by the subjects of our user study (mean: 2.7).



(a) Eclipse



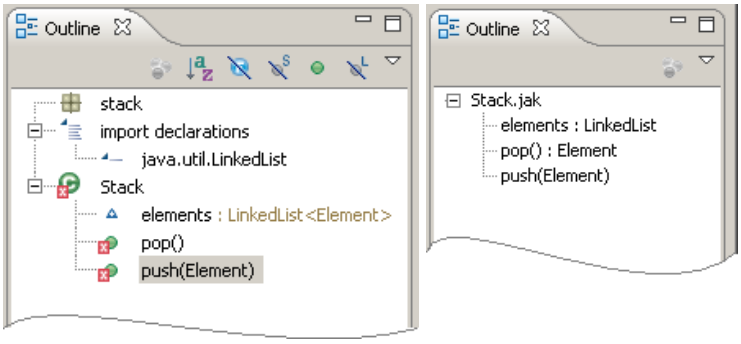
(b) FeatureIDE

Figure 3.4: Screenshot of the editor of Eclipse and FeatureIDE.

Hence, the editor only supports the implementation of Jak source code to a limited amount. Thus, this requirement is only partially fulfilled.

3.3.4 Visualizing the content of a project

In order to get an overview of the project, its content need to be visualized. Eclipse provides different views. One view is the outline view, which is depicted in Figure 3.5a. It gives an overview of the structure of the according source code file, e.g. the import declarations, methods, fields, or inner classes. The outline view has several customization options, e.g. an alphabetical ordering of its elements or hiding several elements, like the import declarations. Since in FOP, source code is stored in several feature modules, it is not sufficient to visualize the source code of single files like it is done in the outline view. Instead it is necessary to visualize feature modules and their content, in order to get an overview of the complete feature project. This is usually done by a collaboration diagram. Our according requirement is:



(a) Eclipse (b) FetureIDE

Figure 3.5: Screenshot of the outline view of Eclipse and FeatureIDE.

5. A structural overview of source code and feature modules must be provided.

Like Eclipse, FeatureIDE provides an outline view of source code files, which is depicted in Figure 3.5b. It provides information about fields and methods of a class. There are no further options to show or hide certain elements, like import declarations. Furthermore, there are no icons to visualize the different kinds of elements of a source code. Another problem is that it only works on compiled files and is not refreshed automatically. The subjects of the user study evaluated the outline view as rather badly realized (mean: 2.9).

The visualization of feature modules and their content has not been implemented yet. Hence, the current version of FeatureIDE does not provide a collaboration diagram or any other visualization of feature modules.

Since the outline view has several problems and there is no visualization of feature modules, this requirement is not fulfilled.

3.3.5 Refactoring

In order to aid a well structured source code, Eclipse provides a large set of refactoring (LIPPERT AND HAVENSTEIN [LH05]). It can be accessed via the menu item *Refactor*. For example, *Rename* allows renaming the project, packages, classes, methods, and fields. If conflicts occur, e.g. a field with the new name already exists, Eclipse requests a user input before it continues. Another refactoring is the extraction of methods and classes. For example, when extracting a method, the source code that should be extracted has to be marked. Eclipse automatically adds necessary parameter to the declaration of the method. Hence, our next requirement is:

6. An IDE must provide a basic set of refactoring.

Despite first implementations (KLAPPROTH [Kla08]), the refactoring of feature projects has not been integrated yet. Thus, this requirement is only partially fulfilled.

3.3.6 Debugging

To support debugging, Eclipse provides a debugging perspective, which allows setting break points at nearly arbitrary positions in the source code. At those break points, the execution of a program stops. The state of the program can be examined and the execution only continues after an according user input. The according requirement is:

7. An IDE must support debugging of source code.

The debugging perspective of Eclipse has not been extended yet for debugging feature projects. Hence, there is no support for debugging feature projects yet. Thus, this requirement is not fulfilled.

3.3.7 Management of feature modules

Since in FOP the source code is divided into features, the according feature modules have to be managed by the IDE. Otherwise, they would have to be managed manually, which may be feasible for small projects, but is tedious and error-prone for large projects. Hence, our last requirement in this section is:

8. An IDE must manage the feature modules.

In Figure 3.6, the package explorer of FeatureIDE is depicted. The feature modules *Base*, *Element*, *Save*, and *Top* are shown. When a new feature in the feature model editor is created, a new folder is created automatically. The deleting of feature modules has to be done manually, but this way it is more user friendly: If a feature was deleted in the feature model editor, a user may not have in mind that this change was automatically propagated to the feature modules. Hence, it prevents accidentally deleting a feature module and source code. When a configuration is selected, FeatureIDE compiles it and creates a folder, in which the composed files are stored. For example, in the folder *build* the composed Java and Jak files are stored in the folder *Save*. The binary files of the Java files are stored in the folder bin (short for binary).

A test of the implementation reveals it as language-independent, because we could use the management of feature modules for a newly integrated language without adjusting the implementation. Since FeatureIDE manages the feature modules in a user friendly manner, this requirement is fulfilled, which is also confirmed by the user study (mean: 3.6).

This was the last requirement concerning domain implementation. According to our explanations in Section 2.2.2, the next step is assembling a product. We describe the according requirements in the next section.

3.4 Application engineering

After the implementation, a product can be generated according to a selection of a set of features. This process is language-independent, since it includes only the selection and composition of already implemented features. This process can be supported by a graphical interface for the selection of features. During the selection, the validation of the selected features can be done. This way, features that cannot be selected in a certain configuration can be disabled, preventing an invalid selection. An automatic composition of the selected features allows to test

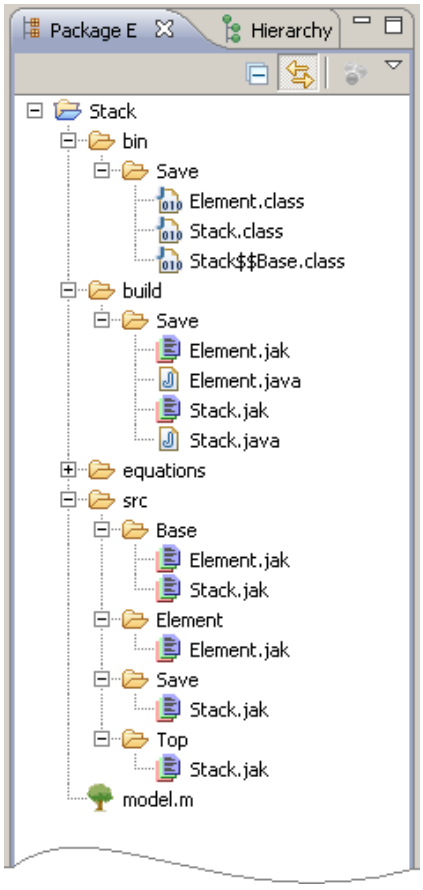


Figure 3.6: Screenshot of the package explorer of FeatureIDE.

that the program fulfills the specified requirement. The composition can even run in background and does not need to be visible to a user. Based on this explanation, the next requirement is:

9. An IDE must support the selection and composition of features.

In Figure 3.7, the configuration editor of FeatureIDE is shown. The feature *Base* is always selected, because it is mandatory. This is marked by the plus and its gray background. In this configuration, the feature *Save* is also selected. It also has a plus, but its background is white, which means that it does not have to be selected. The features *Top* and *Element* are not selected and hence not marked. If a feature was chosen that excludes another feature, this other feature would be marked with a minus and get a gray background, indicating that it cannot be selected. Since the configuration editor allows a relatively user friendly selection

Requirement	Fulfilled?
1. Efficient extending mechanism	no
2. Creation of feature model	yes
3. Compiling & Running	partially
4. Supporting editor	partially
5. Visualizing project	partially
6. Refactoring	partially
7. Debugging	no
8. Managing feature modules	yes
9. Feature selection & composition	yes

Table 3.2: Overview of the requirements and their fulfillment.

of features and the implementation is language-independently, which was revealed by the same test as for the management of feature modules, this requirement is also fulfilled.

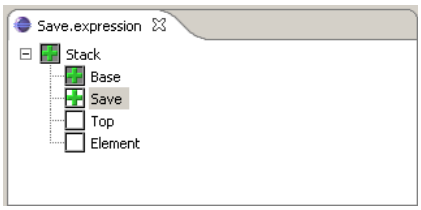


Figure 3.7: Screenshot of configuration editor of FeatureIDE.

In the last sections, we stated many requirements that FeatureIDE should fulfill. We summarize them in Table 3.2. We described for every requirement, if FeatureIDE fulfills it or not. In order to help future developers to know where to start, we give advice how FeatureIDE can fulfill the requirements in the next section.

3.5 Fulfilling the requirements

In the last sections, we presented several requirements and showed which of them FeatureIDE fulfills and which not. In this section, we present approaches that help fulfilling the above presented requirements. We therefore go through the list of requirements summarized in Table 3.2 and explain ideas to fulfill them.

As we go through the list, we omit requirements that are already fulfilled. We mark requirements that are not fulfilled with N, requirements that are partially fulfilled with P. Before a requirement can be regarded as fulfilled, the according implementation must be tested. We do not mention this explicitly from here on, but presume it as part of the solution for each requirement.

1. An IDE must provide an efficient mechanism to add new languages. (N)

Since FeatureIDE was specifically developed to support Jak, it is not language-independent. We suggest a similar approach as Eclipse uses, because it has proven successful. Hence, FeatureIDE should be structured as a set of plug-ins, whereas one set of plug-ins constitutes the language-independent core of FeatureIDE and defines extension points for further plug-ins. Those further plug-ins constitute the language-dependent part. Therefore, the first step is to separate those parts of FeatureIDE that support language-independent processes from those that support language-dependent processes.

Language-dependent processes are the implementation and the running of a program. Connected with the implementation are syntax highlighting and auto completion as well as refactoring. For example, in Java the extraction of a class is allowed, but not in Haskell or C. Or, when `name.` is typed, it depends on the language, which completions can be offered. Hence, refactoring and auto completion must be offered depending on the underlying language. The problem here is to define those set of refactoring and auto completion, that is language-independent, e.g. renaming or completion of variable names that are contained in the same file. Those can be included in a language-independent core implementation. Others, like extracting a class or syntax highlighting, have to be implemented as part of a plug-in for new languages.

After having identified the language-dependent functionality, we can continue with suggestions for the rest of the requirements. Since debugging is not supported in the current version of FeatureIDE, future implementation should implement this requirement language-independently from the beginning. If for a new language a new debugging functionality is needed, the language-independent debugger has to define an extension point at which the language-dependent functionality can be plugged in.

The creation of the domain model (graphically and textual), the management of feature modules, and the selection and composition of features are language-independent processes. However, they are included in FeatureIDE and not explicitly separated from the language-dependent part. Since the implementation of these functionalities is language-independent, the first step is already done. Now,

the language-independent implementation for those functionalities needs to be included in the language-independent core of FeatureIDE. Further languages like FeatureC++ can then use these functionalities without implementing it.

We discussed the solution of this requirement in more detail, because we show a base for fulfilling this requirement in the next chapters: a language-independent data structure for feature projects. Since the effort for fulfilling most of the requirements would itself be enough for one thesis, the solution of the remaining requirements is not pointed out in detail. We just present some ideas for future developers of FeatureIDE were to start.

3. An IDE must support comfortable compiling and running source code. (P)

Since the compiling step is already fulfilled, we only consider necessary steps to provide an easy execution of programs. Eclipse provides an extension point for different launch configuration types. FeatureIDE uses this extension point, which allows running a program. However, it fails to automatically determine the current variant of the program that should be run (e.g. the stack with the features *Base* and *Save* or just the stack with the feature *Base*) and the main class of the configuration. This leads to the manually definition of a launch configuration. In order to allow running a program with a few clicks, FeatureIDE must find the according variant of the program and main class automatically. When the according run configuration is selected, the file can be executed (if it contains a main type).

4. An integrated editor must provide syntax highlighting, auto completion, and mark compiler errors. (P)

For this requirement, extension points already exist. For syntax highlighting, the editor package was extended. Auto completion is one function of the content assist provided by Eclipse. It provides context sensitive suggestions to complete the input that is currently typed. To fulfill this requirement, the content assist package needs to be extended by an implementation of the auto complete function. For a better marking of compiler errors, the line number of an error has to be extracted. A tool tip has to be added that contains the error at the according mouse position.

5. A structural overview of source code and feature modules must be provided. (P)

Since a rudimentary outline view already exist, it needs to be improved for better usability. It should be automatically refreshed when the according source code is changed. Furthermore, filters should be added as well as icons for different structural elements of a source code file (e.g. class vs. field vs. method). For the visualization of feature modules, a collaboration diagram can be implemented. It must present feature modules, their content, and the according roles of a project. In the next chapter, we show how both the collaboration diagram and the outline view can be implemented language-independently.

6. An IDE must provide a basic set of refactoring. (P)

In order to support refactoring, Eclipse provides an extension point for refactoring. It provides support for the integration of refactoring, like change events that occur after something is changed or user interface support. The refactoring itself, e.g. renaming or extraction of source code have to be implemented and made accessible via the menu bar or context menu. A first implementation of refactoring feature projects already exists and can be used (KLAPPROTH [Kla08]).

7. An IDE must support debugging of source code. (N)

In Eclipse, there are different modes that a launch configuration can support. All supported modes have to be included in the definition of the extension of the launch configuration plug-in. Debug is such a mode. The first step for fulfilling this requirement is to add the mode *Debug* to the supported modes of the defined launch configuration. The second step is to extend the debug platform of Eclipse. It defines a debug model which has to be implemented in order to allow a comfortable debugging for the Jak language. The debugging user interface can be defined from scratch or the basic user interface that the debug platform provides can be used.

3.6 Summary

In this chapter, we derived a list of requirements that an IDE for feature-oriented SPL development should fulfill. We stated for each requirement, whether and how FeatureIDE fulfills it or fails to fulfill it. In order to give future developers help where to start, we presented some ideas for fulfilling the requirements.

In order to show that our suggestions are indeed a way to fulfill the list of requirements, we present a data model that allows a language-independent representation of feature projects in the next chapter. For evaluating the data model, we use it to provide a visualization of the feature modules and their content.

Chapter 4

Data model for feature projects

In the last chapter, we described requirements for a tool that supports feature-oriented SPLs. In order to provide a base for fulfilling the requirements, we implement a language-independent data model. To show that the data model is useful, we use it to build a collaboration diagram and an outline view (cf. Section 3.3.4). We chose those two requirements, because a visualization is a good way to communicate results (KANG ET AL. [KLD02]).

First, we specify requirements that our data model should fulfill. Then, we explain our data model in detail. Finally, we show how the creation of the data model is implemented.

4.1 Requirements for the data model

In order to be clear on what a model is, we start by explaining it. Then, we state requirements that the data model must fulfill. This allows us to evaluate if the data model is useful for creating a collaboration diagram and an outline view language-independently and thus fulfills its purpose.

4.1.1 Definition of a model

"A *model* is a simplified representation of a system (or process or theory) intended to enhance our ability to understand, predict, and possibly control the behavior of the system." (NEELAMKAVIL [Nee87]). Simplified representation of a system means that only those aspects of the system are considered that are relevant for

understanding. The purpose of the model determines which of the aspects are relevant. For example, a model that represents source code in a language-independent way, can neglect language-specific details, like certain keywords. Those language-specific details are not important to understand the source code. Hence, they can be omitted. Without irrelevant details, only aspects are left that are necessary for understanding the source code. Thus, a person who wants to understand the source code does not have to deal with irrelevant information.

Based on the definition of a model, we state requirements that the data model must fulfill. The requirements are necessary to evaluate if the data model can be used as language-independent representation of feature projects.

4.1.2 Sufficient information

In order to be able to build a collaboration diagram and an outline view from the data model, it must contain sufficient information. Sufficient means that the information contained in data model alone must suffice to create a collaboration diagram or an outline view. No information that is not contained in the data model should be needed.

For the collaboration diagram, the name of the feature project, the feature modules, and the content of the feature modules are necessary. The name of the feature project is necessary for identifying. The feature modules of a project are necessary to depict the collaborations, the files contained in a feature module are used to display the classes and roles in the collaboration diagram.

In the outline view, the content of a file is visualized (e.g. classes and methods). Since the purpose of the outline view is to provide an overview of a file, relevant structural elements of a file are needed. What the relevant structural elements are, depends on the language of the project. For example, for a Java file the class or interface declaration, import declarations, constructors, methods, fields, and inner classes and their content are needed to give an overview of a file. Furthermore, the access modifiers are helpful to understand the structure of a class. For a project in C, the include statements, define statements, and the definition of functions, structs, and unions are relevant for having and overview of the source code file.

To summarize, the data model must contain the name of the feature project, the feature modules, the files and folders contained in the feature modules and certain structural elements of the files. If all this information is contained in the data model, it contains sufficient information.

4.1.3 Efficient access of elements

In addition to sufficient information, our data model must support an efficient access of its contents, because we want to use it for different visualizations. Otherwise, if the creation of such a visualization lasts too long, a user of FeatureIDE would become frustrated. According to Spolsky (SPOLSKY [Spo01], p.107), to the user, seconds are hours. Hence, an efficient access of elements in our data model is crucial for the success of FeatureIDE. Otherwise, if a user has no opportunity to get a quick overview of the feature project or of the source code of a file, he probably becomes disappointed and discontinues using FeatureIDE.

4.1.4 Language-independent representation

In Chapter 3 we stated that FeatureIDE must provide an efficient mechanism for extensions. The base for fulfilling this requirement is a language-independent representation of feature projects. Furthermore, this requirement crosscuts all other requirements. This means that the creation of a collaboration diagram and an outline view must be implemented language-independently. Otherwise, both requirements were not regarded as fulfilled (cf. Section 3.1). Hence, the data model must allow a language-independent creation of a collaboration diagram and an outline view.

For a better overview, we summarize the requirements:

- The data model must provide sufficient information for visualizing feature modules and source code.
- The data model must support an efficient access to its elements.
- The data model must be language-independent.

The requirements help to understand the design of the data model, which is described in the next section.

4.2 The data model

Based on the requirements stated above, we explain the data model in this section. It describes feature projects as trees for two main reasons: Firstly, FeatureIDE represents a feature module as a folder, which contains all files belonging to that

feature module. Thus, a hierarchical organization is available already. Secondly, trees are efficient for accessing elements, which helps fulfilling the second requirement, an efficient access.

So how are feature projects represented as trees? There are two different kinds of nodes: Inner nodes and leaf nodes. *Inner nodes* represent the structure of feature projects, whereas *leaf nodes* represent the content of a file. We first explain, how inner nodes are created and then continue with leaf nodes.

4.2.1 Inner nodes

Inner nodes represent the structure of the feature project. They have a name and a type. The name is the name of the structural element they represent, e.g. the name of the feature project or the name of a feature. The type provides information about the kind of structural element. For example, the type of the node representing the project is *project*, the type of a node representing a feature is *feature*. Since in a feature module there can still be further folders for structuring purposes (in Java called packages), those would also be added to the data model. Their type would be *folder*. The root node of the data model contains the name of the feature project and the type *project*. We choose this as root node because the hierarchical project organization in FeatureIDE also starts with the name of the project.

In Figure 4.1, an excerpt of the model of the stack example is shown. Leaf nodes are omitted for simplicity. The name of the root node is *Stack* and its type is *project*. The name of the leftmost child of the root node is *Base*. The type is *feature*. This node has two children: The classes `Stack` and `Element`, which are stored in this feature module. The type of both nodes is *file*. We did not choose the type *class*, because this would limit the applicability to languages that have classes. Since we want to support other languages like Haskell or C, too, and files are used for modularizing source code for the languages we want to support, we choose the type to be `file`.

4.2.2 Leaf nodes

After the structure of the feature project, i.e. its name and feature modules and the contained files, is represented, the content of the files has to be added to the data model. Therefore, leaf nodes are created. They also have a name and a type, and, in addition to inner nodes, some content. Name and type have the same meaning as for the inner nodes. The content of the leaf nodes is the content of the files.

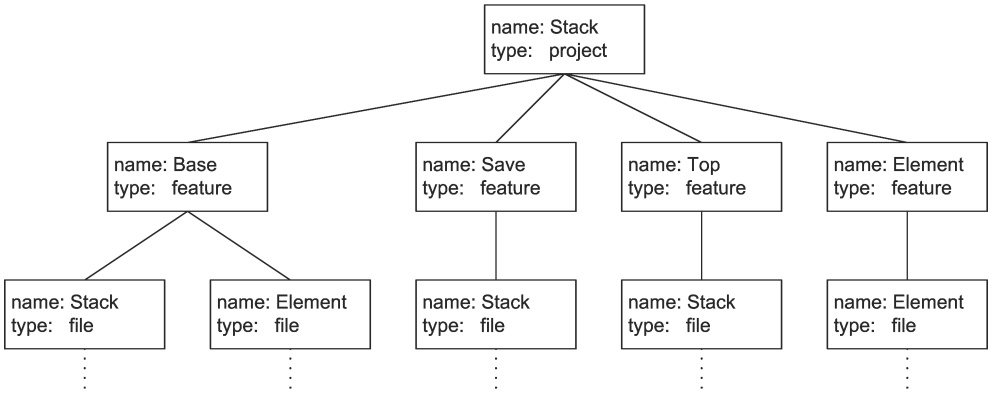


Figure 4.1: Inner nodes of the data model of the stack example.

A leaf node contains a tree. Hence, the data model is a nested tree. This structure may seem unnecessary, but we chose it to have a better performance when accessing the elements. We store all leaf nodes of a project tree in a linked list. Thus, when only the content of the files is relevant, the complete project tree does not have to be traversed. Only the list of leaf nodes has to be traversed. If the content of a certain file is relevant (like for the outline view), it is still sufficient to only traverse the leaf nodes and then check, whether the parent denotes the according file. Furthermore, the separation of the project structure from the content of files leads to a clearer overview of a feature project.

The content of a leaf node is created from an AST. We choose ASTs, because in order to compile source code, they are generated for every file. Hence, we can use a structure that is generated anyway. Furthermore, ASTs contain structural elements of a file, which we need in the data model in order to be able to generate an outline view. We do not need all the information contained in the AST, so we make a selection of relevant structural elements. As we stated above, this selection is language-dependent. For Java, we only extract the class declaration, import declarations, methods, fields, and inner classes, including the access modifiers. For other languages, we need to extract other information.

Furthermore, we choose a start node in the AST. This is also language-dependent, because source code in different languages is structured differently. For Java, we begin with the class declaration. In Haskell there are no classes, and hence the type of start node is *module*.

For the data model of the projects that are implemented in a language supported by FeatureHouse, we choose to use FSTs. Since FSTs are stripped down ASTs, it already contains less irrelevant information (APEL AND LENGAUER [AL08]). However, because FeatureHouse creates additional nodes that are only

important for the technical realization of the composition, we still filter out some irrelevant nodes.

A leaf node for the class `Stack` in the feature *Base* is shown in Figure 4.2. The name of this leaf node is the name of the class, *Stack*. The type is *leaf tree node*. This indicates that not the structure of the project, but the content of the file is represented.

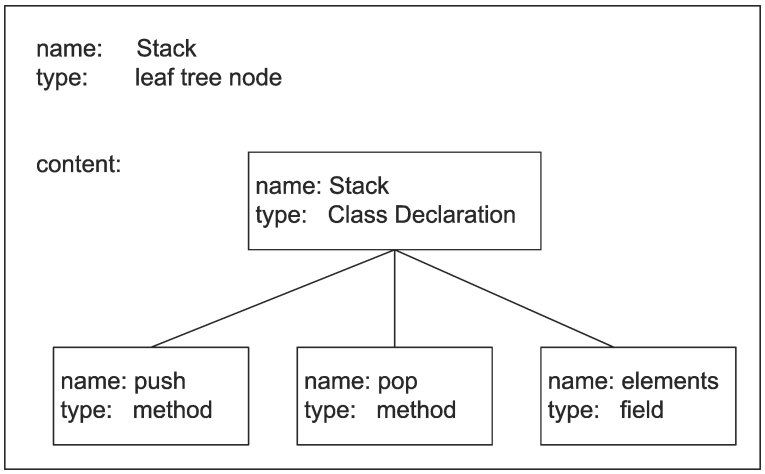


Figure 4.2: A leaf node of the data model of the stack example.

The content of the leaf node is extracted from the FST. The type of the first node is *class declaration*, the type of its children are *method* and *field*. The name and content (if it exists) of the first node is `Stack`, the name of its children are `push`, `pop`, and `elements`.

4.2.3 The complete data model

After the creation of the inner nodes and leaf nodes, we can present the complete data model in Figure 4.3. It provides an overview and helps to understand the structure of the data model. The solid lines denote the inner nodes, the dashed lines surround the leaf nodes, which contain trees. If a class contains an inner class, then there is one level more in the leaf node (left most leaf node in Figure 4.3). If a file contains more than one class, all classes are added to the leaf node as siblings (right most leaf node in Figure 4.3). For a better overview, we added those nodes only to the left most and right most leaf node. To improve understandability, we also present the model of the stack example in Figure 4.4.

At first sight, this model looks rather complex, but it is indeed simple. Firstly, programming languages are structured hierarchically and thus source code files

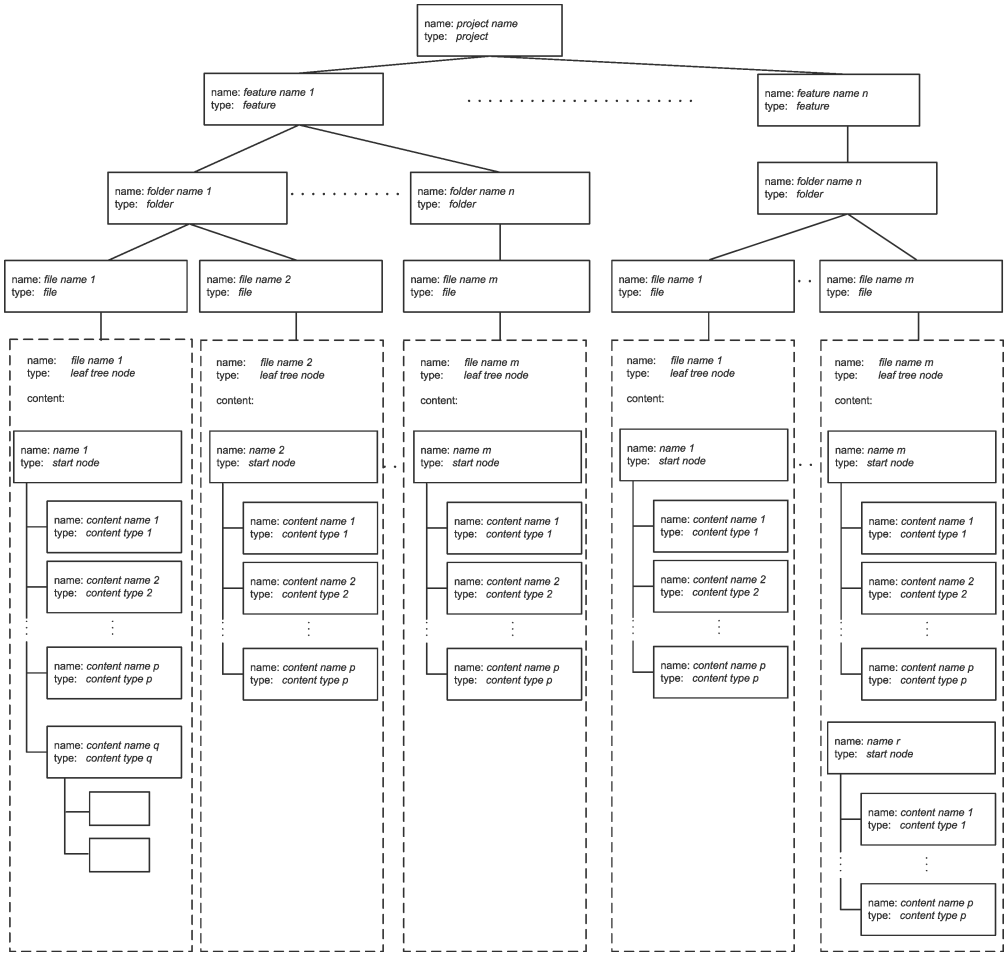


Figure 4.3: The complete data model.

are structured hierarchically, too. Secondly, a feature project itself is represented as tree: There is a root folder that contains the complete project and every feature module is stored in a folder, which itself contains a set of files. Hence, there is a hierarchical organization of feature projects, which is reflected by the data model.

In this section, we showed the requirements for our data model and explained its structure. In order to know how it can be used for other languages, we explain the implementation of the data model in the next section.

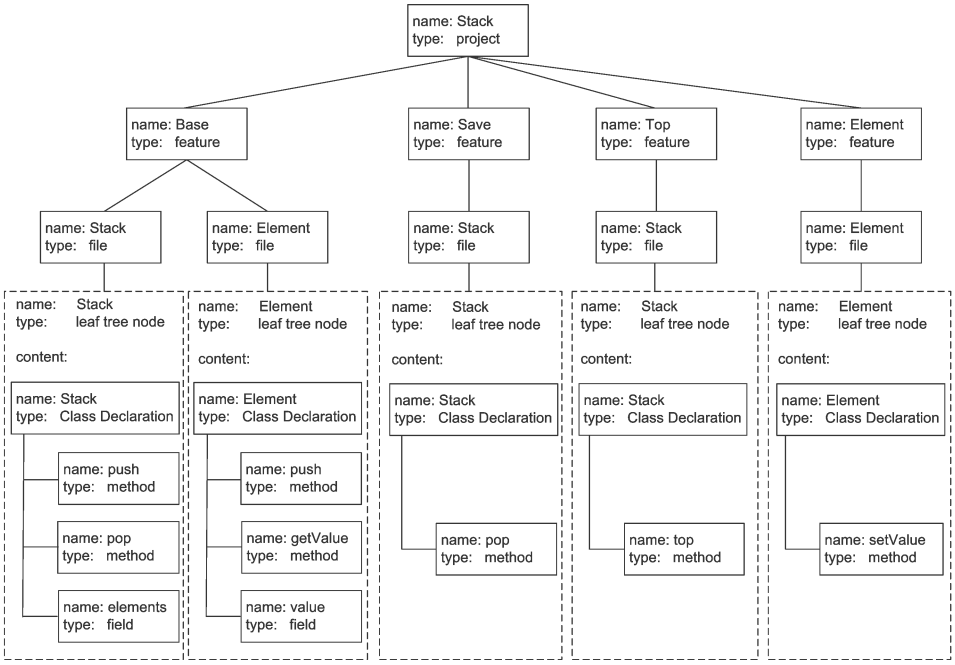


Figure 4.4: The complete data model of the stack example.

4.3 Implementation details

In the last section, we explained the data model and its components. In order to understand how our data model is created and what information we need for different programming languages, we present some implementation details. First, we give general information on our implementation. Next, we explain how the ASTs of different languages are used for the creation of the data model. Then, we describe how we extracted information from the ASTs of those languages that are supported by FeatureHouse. Finally, we describe how we build the data model from AHEAD and how further languages can be integrated.

4.3.1 Building the data model

In order to build the data model, we start by creating the inner nodes. If an inner node with the type *file* was created, we create a leaf node and start to traverse the AST. Depending on the language, we need to choose a node with which we start. This means, we start extracting information beginning with the start node. All preceding nodes are not extracted. For Java, we choose the start node to be the class declaration. We then check for every node if it has a name. Nodes

without names are not extracted, because they are not relevant for understanding the source code. We also check the type of the node in order to choose which node to extract. In Java, we only extract nodes that contain relevant structural elements, like nodes that denote a method-, field-, or class-declaration (cf. 4.1.2). For every node we extract, we create a `NonTerminalNode`, if a node has children, or a `TerminalNode`, if the node has no children. If we are finished with the content of a file, we continue with the next element of a folder, or the next feature module.

For the data model, we implemented an iterator that is used for traversing the data model. We use it for the creation of the collaboration diagram and outline view. For both, we implemented a visitor that visits the nodes of the model. For the creation of the collaboration diagram, we only traverse the inner nodes. For the outline view, we only traverse the leaf nodes and check, if the parent denotes the currently displayed file of the editor. Then, we add the content of the according node to the outline view.

4.3.2 Building the data model from FSTs

In the last section, we said that leaf nodes of our data model are build from ASTs. However, an AST contains more information than we actually need. Since an FST is a stripped-down AST, we decided to use an FST where it is provided. This is the case for all languages that are supported by FeatureHouse. However, the granularity of FSTs that FeatureHouse creates is still too fine for our data model. For the implementation of the superimposition of FSTs, FeatureHouse creates additional nodes. Those nodes do not contribute to understanding the structure of a file and thus are not necessary to be displayed in the collaboration diagram or outline view. Hence, we omitted them.

For different languages, we need to extract different nodes from the FST for the outline view. We present them in Table 4.1. We chose these elements, because in the IDEs we examined (cf. Section 3.3.1), those elements are also displayed in an outline of the source code.

4.3.3 Building the data model from AHEAD

Since FeatureIDE integrates AHEAD, a data model for Jak projects is developed already (Jak model). Instead of working directly on the AST that AHEAD generates, we decided to use the Jak model to generate the data model. This way, we do not have to traverse the AST and filter irrelevant information. All necessary information is already contained in the Jak model.

Language	Structural elements
Java	class, interface import constructor, method, field
C#	class, struct, interface constructor, destructor method, field
C	include, define struct, union function
Haskell	module data type, monad, class, type function
XML	attributes elements

Table 4.1: Relevant structural elements for different languages.

The Jak model defines import declarations, classes, features, fields, and methods. We take these elements from the Jak model and use it to build our data model. Since Jak extends Java only by a few keywords, we extract for the data model the same structural elements as for Java. Since AHEAD only supports one language, we do not need to determine a start node.

4.3.4 Integrating further languages

So what has to be done when further languages should be integrated in FeatureHouse? Firstly, the AST needs to be obtained. Then, a builder that builds the data model from the AST needs to be defined. It has to be decided, which elements are to be taken from the AST. This depends on the new language. After the data model is build, the collaboration diagram and outline view can be created.

For example, the next tool we plan to integrate in FeatureIDE is FeatureC++. It also generates an AST, from which a part of is stored in an XML file. This AST needs to be traversed. Relevant structural elements in FeatureC++ are classes,

interfaces, include, define, constructor, destructor, struct, union, and function. Those need to be added to our data model. Once the data model is defined, the collaboration diagram and outline view can be generated without further implementation. We already finished the conceptual design of integrating FeatureC++. The only thing missing is the implementation, which we plan as one of our next steps for improving FeatureIDE.

For languages that do not provide ASTs, another mapping to the data model must be found. Since the data model is a tree and thus hierarchically organized, a hierarchically representation of the language should be available. If no such representation can be found, it must be tested whether another mapping to the data model or an overlaying module structure for those languages can be found (ANFURRUTIA ET AL. [ADT07]).

In this chapter, we presented our data model. We described how it is designed and how further languages can use it. We also stated requirements, which the data model must fulfill. In order to know if our data model indeed constitutes a language-independent base for the representation of feature projects and can be used for the collaboration diagram and outline view, we evaluate if it fulfills the requirements in the next chapter.

Chapter 5

Evaluation of the data model

In the last chapter, we presented our data model and specified the requirements it should fulfill. In this chapter, we evaluate whether our data model indeed fulfills these requirements. We proceed by considering each requirement in an own section. First, we check if the data model contains sufficient information. Second, we evaluate the performance of the data model. Then, we evaluate whether the data model is language-independent. In every section, we first describe how we want to evaluate whether the requirement is fulfilled and then we present the necessary tests and results. Finally, we summarize the results of this chapter.

5.1 Does the data model contain sufficient information?

In this section, we evaluate whether our data model contains sufficient information for a visualization of the feature modules of a feature project. In order to show that this requirement is fulfilled, we implement a collaboration diagram and an outline view.

5.1.1 Collaboration diagram

In order to comprehend our evaluation, we explain how we build the collaboration diagram from the data model.

For the implementation of a collaboration diagram, we define several classes. We define the class `Role`, which contains the information of a *role* in a collabo-

ration diagram, e.g. the definition of `Element` in our stack example. The content of a role is collected in a list, where each element of the list contains the content of one terminal node, e.g. the definition of a method. We define the classes `Collaboration` and `Class`, which correspond to the elements *collaboration* and *class* of the collaboration diagram. Both contain a list of roles, which are contained in the according class or collaboration. For example, the class *Stack* would contain all roles that implement the class `Stack`. The collaboration *Base* would contain all roles that implement the base variant of the stack. We implement a visitor that traverses the data model. We first create the collaborations, then the classes that are contained in the collaboration, and then the roles of the classes.

For the Stack example (cf. Section 2.3.2), we present the generated collaboration diagram in Figure 5.1. We use a simple visualization, since we only show that our data model contains sufficient information rather than providing a high quality visualization. For the same reason, we omitted the contents of the roles. For the creation of the collaboration diagram, we only needed to traverse the data model.

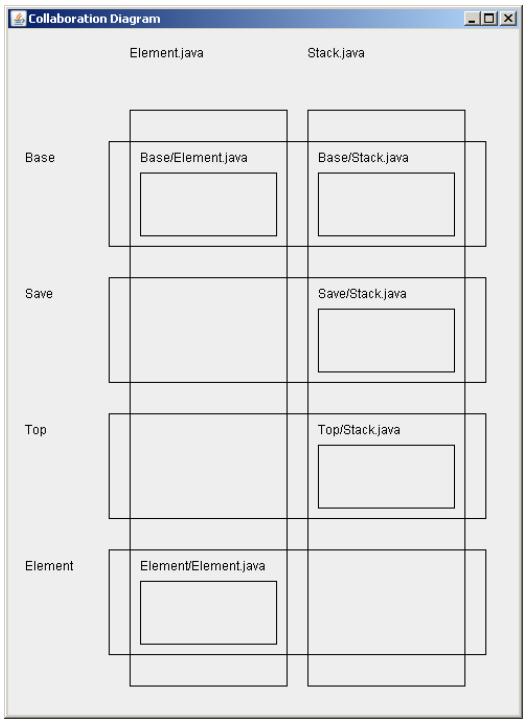


Figure 5.1: Generated collaboration diagram of the stack example.

All information displayed in the collaboration diagram is contained in the data model. Hence, our data model is suitable for the creation of a collaboration di-

agram. In the next section, we describe the implementation of the outline view, which constitutes a further test of the contained information in the data model.

5.1.2 Outline view

For the outline view, we obtain the file that is currently opened and displayed from the editor of Eclipse. We then search the according leaf node by traversing all leaf nodes and checking the parents (cf. Section 4.3.1). When we found the leaf node, we traverse its content and add for every node the type and the name to the outline view.

The result of our implementation is shown in Figure 5.2. It shows the structure of the class `Stack` of the feature *Base* (cf. Section 2.3.2). The class declaration *Stack: ClassDeclaration*, the constructor {<IDENTIFIER>}: ConstructorDecl, the field *LinkedList<Element>-elements, elements: FieldDecl*, and the methods *pop({FormalParametersInternal}): MethodDecl* as well as *push(Element-Element): MethodDecl* are contained.

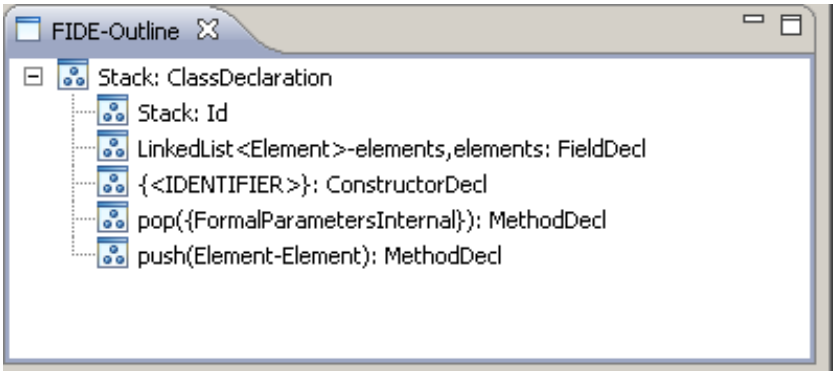


Figure 5.2: Outline view of the stack example.

However, compared to the outline view of the JDT plug-in of Eclipse (cf. Figure 3.5a in Section 3.3.4), it displays more information than necessary. The reason is that we display the complete type and name of a node from our data model. FeatureHouse adds more information to the node of an FST than we actually need for an outline view. In order to obtain a better outline view, we just need to filter out the unnecessary information. However, since it is not necessary to filter the additional information in order to show that our data model contains sufficient information, we kept the information in the outline view. Furthermore, the collaboration diagram and outline view are just examples for applying the data model. There may occur further usage of the data model, where information that

is irrelevant for the outline view may be necessary. Hence, we kept it in the data model.

Thus, although the information is not presented as nicely as in the outline view of the JDT plug-in, it still contains sufficient information to get an overview of the contents of a source code file. In order to obtain a user-friendly outline view, the additional information just needs to be filtered out.

Hence, our data model contains sufficient information for the creation of a collaboration diagram and an outline view. Thus, the data model fulfills the first requirement. In the next section, we evaluate the performance of our data model and state, whether it fulfills the second requirement.

5.2 Is the data model efficient?

In this section, we test the performance of our data model and thus evaluate, whether the data model fulfills the second requirement. Therefore, we measure the time it takes to build the data model and create a collaboration diagram and an outline view, respectively.

As test project, we use the implementation of the stack example. Since we already presented source code examples in Section 2.3.2 and used it as example for the collaboration diagram and outline view, we do not need to explain it again. The project consists of 280 *Lines of Code (LOC*), including comments. The number of nodes in the FST is 62, the number of nodes in the data model is 36. The number of nodes is relevant, because we traverse the trees for the creation of the visualizations. The project consists of four features and five files. This information gives an impression of the size of the project and helps to compare it with other projects. The project is small, but in order to know whether the data model is usable at all, it suffices for performance tests.

In order to determine the time for the visualizations, we used the Java method `currentTimeMillis` of the class `java.lang.System`. It returns the current system time in milliseconds. We measured the time before and after the data model and the visualizations were created. The difference between those numbers is the actual time of creation. We included the duration of creation of the data model in our measurement, because when the content of a file is changed, both visualizations need to be updated. For this, the data model needs to be rebuild. Hence, the time for creating the data model is also important for evaluating the performance.

To minimize errors of measurement, we took twenty measurements and cal-

culated the mean value. The creation of a collaboration diagram took 178 ms with a standard deviation of 20 ms, the creation of the outline view took 27 ms with a standard deviation of 7 ms. The time for the outline view is shorter than for the collaboration diagram, because in the outline view, one file is represented, whereas in the collaboration diagram the complete feature project is represented.

How can those numbers be interpreted? Guides for the development of GUIs suggest that waiting time should be kept small (SPOLSKY [Spo01], p.107). According to Nielsen, there are three time limits for the user perception of the execution time of a GUI (NIELSEN [Nie94], p. 135): 10 seconds, 1 second, and 100 ms. We explain their meaning next:

- Everything that lasts longer than 10 seconds is regarded as failure by the user. Up until 10 seconds, however, the user will wait for a result, although it is perceived as long waiting time (SPOLSKY [Spo01], p. 107).
- If the execution time for a command lasts up to 1 second, the user will notice the delay, but still have an uninterrupted flow of thought (NIELSEN [Nie94], p. 135). Hence, an execution time up to 1 second is regarded as acceptable and thus fulfills this requirement.
- However, a good execution time must be smaller than 100 ms (MILLER [Mil68]) or 200 ms (SCHNEIDER [Sch08]). Since according to Donders (DONDERS [Don68]), it takes about 200 ms to perceive and react to a visual stimulus, we choose 200 ms as upper bound for a good execution time.

We summarize the limits in Figure 5.3. An execution time that is greater than 1 second does not fulfill the requirement. On the other hand, an execution time that is below 200 ms is regarded as a good value.

Now we can compare our measured time with the limit we defined. Both values are less then 200 ms, which means that the duration of the creation of a collaboration diagram and an outline view can be regarded as good. Hence, the performance of the data model fulfills this requirement.

The last requirement the data model must fulfill is language-independence. We evaluate this in the next section.

5.3 Is the data model language-independent?

In the last sections we showed that our data model is suitable for Java projects. In this section, we test if our data model can be used for other languages, including

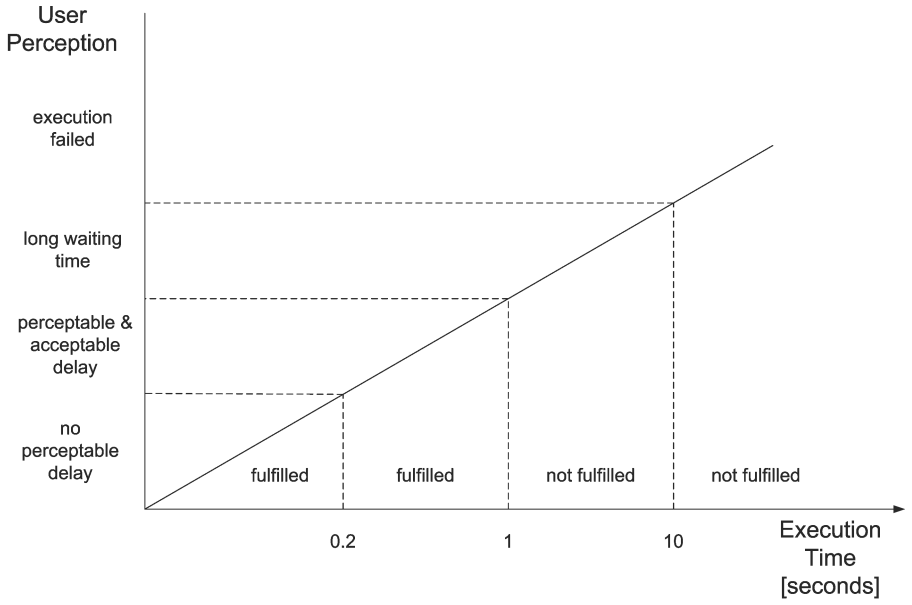


Figure 5.3: Classification of execution times.

C#, C, Haskell, XML, and Jak.

First, we describe the examples we use. Then, we present collaboration diagrams and outline views generated from our data model. Next, we present execution times for creating the collaboration diagrams and outline views. Finally, we summarize the results of the evaluation.

5.3.1 Test projects

The projects we evaluate are written in C#, C, Haskell, XML, and Jak. The project in C# and C is a graph product line with several features. The Haskell project concerns arithmetic expressions and their evaluation. The XML project specifies a phone call. The Jak project has the same features and files as the stack example in Java and only differs in a few key words.

We chose the projects, because they have about the same size as the Java project or, if not, several features could be omitted to better match the size. In Table 5.1, we present LOC and the number of nodes in the FST and in the data model. Furthermore, we present the number of features and files of the projects to better understand the size of the project. Since we build the data model for the Jak project not from the AST, but from the existing Jak model, we omit the number of nodes.

Language	LOC	Nodes (FST)	Nodes (Data model)	Feature	Files
Java	280	62	36	4	5
C#	270	142	82	2	6
C	598	75	73	2	4
Haskell	300	146	134	3	3
XML	414	507	351	2	2
Jak	272	-	72	4	5

Table 5.1: Size of the test projects.

The description of the projects should help to interpret the results of our evaluation. Except for the stack example in Java and Jak, the source code of the projects can be found on the FeatureHouse homepage[1].

5.3.2 Collaboration diagrams

Based on the description of the projects, we present the collaboration diagram of those projects in Figure 5.4.

The collaboration diagram of every feature project is displayed correctly. Furthermore, we did not have to adjust any source code that creates the collaboration diagram from the data model. Hence, the data model allows a language-independent creation of a collaboration diagram.

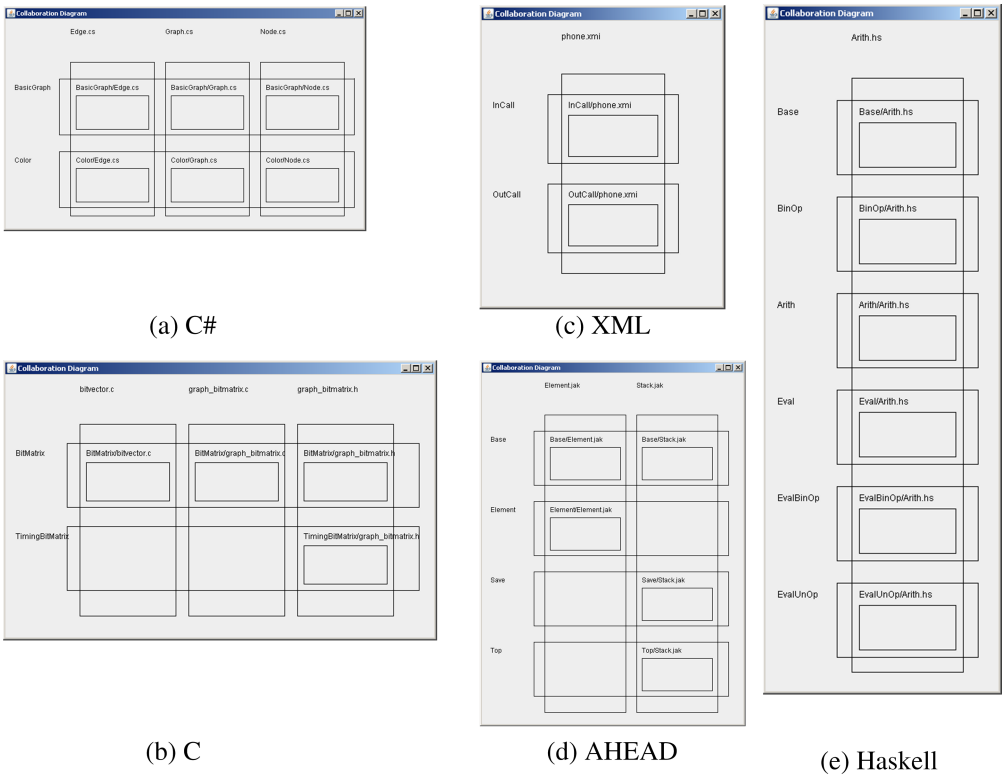
5.3.3 Outline views

In Figure 5.5, the outline views of the same examples as in the last section are shown. For all projects, all relevant structural elements are contained.

As for the outline view of the Java stack example (cf. Section 5.1.2), the outline views of all projects contain more information than necessary, except for the Jak stack example. The reason is that we create the data model not from an AST, but from the Jak model, which only contains information that is relevant for the outline view. For the outline views of all other projects, the same arguments as for the Java stack example apply here.

In order to filter the irrelevant information, there are two ways: Firstly, the

[1] http://www.infosun.fim.uni-passau.de/cl/apel/fh/



(a) C# (c) XML

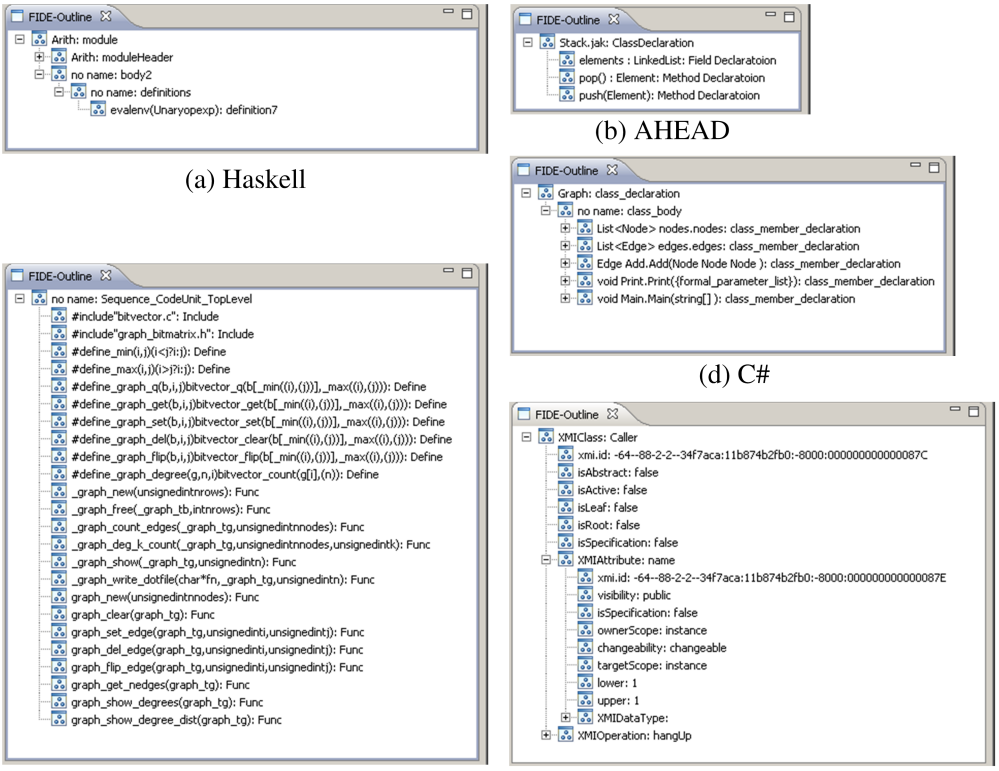
(b) C (d) AHEAD (e) Haskell

Figure 5.4: Generated collaboration diagrams.

information could be filtered during the creation of the data model. This is only applicable if all cases where the data model is used are identified upfront. Otherwise, we could lose relevant information for one of the cases. Secondly, the information could be filtered during the creation of the outline view. However, this makes the creation of the outline view from the data model language-dependent, because for every language, the filtering needs to be implemented.

The preferable of both ways can be determined, when all use cases of the data model are defined. Only then the irrelevant information can be identified and filtered. If the data model still contains more information than necessary for the outline view, they need to be filtered during the creation of the outline view.

Since the data model contains sufficient information to create a collaboration diagram and an outline view for different programming languages, we showed that the data model is language-independent so far. In order to confirm the language-independence, we evaluate the performance of the data model with the same projects in the next section.



(a) Haskell

(b) AHEAD

(c) C

(d) C#

(e) XML

Figure 5.5: Outline views.

5.3.4 Execution times

In order to test the execution times, we used the projects and their configuration as described in both previous sections. We also took 20 measurements of the time it took to create the data model as well as the collaboration diagram and outline view.

As presented in Table 5.1, the projects still differ in their size despite our effort to match them. Either the LOCs differ, or the number of nodes in the FST or the data model. Since we have no comparable projects, we omit statistical tests. If we could match the projects in LOC and number of nodes, features, and files, we could compare the performance of all projects with the performance of the Java project. Statistical tests that could be used in our case are the t-Test for dependent samples (STUDENT [Stu08]) or a one-way analysis of variances for repeated measures (ANDERSON AND FINN [AF96], p. 597ff).

However, since we cannot be sure about the comparability of the projects, we proceed as we did for the Java stack example. If a mean value for the execution

is smaller than 1 second, the performance is evaluated as acceptable. A duration smaller than 200 ms is rated as good performance (cf. Figure 5.3). For the data model to be language-independent, the execution time for all projects must fulfill the requirement. Otherwise, we assume that the language of the project affects the performance of the data model, which would mean, that the data model is language-dependent.

In Table 5.2, the results of the creation of the collaboration diagrams and the outline views are shown. For better comparison, the time for the Java project is also presented. The column *StdDev* denotes the standard deviation of the mean values.

Language	Time	StdDev
Java	179	20
C#	178	13
C	181	10
Haskell	184	11
XML	194	13
AHEAD	165	12

(a) Collaboration diagram

Language	Time	StdDev
Java	27	7
C#	26	9
C	25	9
Haskell	31	10
XML	41	13
AHEAD	23	11

(b) Outline view

Table 5.2: Performance results.

All execution times are smaller than 200 ms. Furthermore, the execution times for the collaboration diagram as well as the outline view are for all projects similar to the execution times for the Java projects. Hence, we can assume that the programming language of the project does not affect the performance of the data model. Since the creation of the collaboration diagram and outline view is also not affected by the underlying programming language of the project (cf. Section 5.3.2 and 5.3.3), we can assume that the data model is language-independent. Hence, this requirement is also fulfilled.

The next step to measure the performance would be to test the scalability of the data model. However, since AHEAD and FeatureHouse are relatively new tools (BATORY ET AL. [BSR04]; APEL AND LENGAUER [AL08]), the number of projects big enough to test the scalability of the data model is very limited. We discuss this problem in Section 6.2.

5.4 Summary

In this chapter, we evaluated our data model. First, we showed that it contains enough information for the creation of a collaboration diagram and an outline view for Java. Then, we showed that the data model provides an efficient access to its elements. Next, we showed that the data model is language-independent by conducting the same tests for C#, C, Haskell, XML, and Jak. We created collaboration diagrams, outline views, and measured the performance of the creation. We did not obtain any results that would indicate an influence of the language of a project on the results.

Since the data model fulfills all three requirements, it can be used as language-independent representation of feature projects. Hence, we provided the first step for an efficient extension mechanism for FeatureIDE and thus improving it. In order to relate our results to other work, we discuss some related issues in the next chapter. In addition, we give suggestions for future work.

Chapter 6

Related and future work

After having presented our work in the previous chapters, we discuss some related work. This helps to understand the scope of our work and to relate it to other work. Then, we present some suggestions how the work presented in this thesis can be continued.

6.1 Related work

Since one goal was the development of a data model, we first discuss work that is related to the data model. Then, we present other tools that visualize software, because we used the data model for visualizations and visualizing helps understanding aspects (HABERLANDT [Hab97]). Finally, we describe other extensible IDEs in order to understand the benefit of FeatureIDE.

Data model

For the plug-in structure of Eclipse, language-specific data models exist. For example, the JDT plug-in defines a Java model, the CDT plug-in a C model. Both are used for several functionalities, like the outline view and class- or call-hierarchies. However, for every new language that should be integrated, a new data model has to be defined. The advantage of a language-independent data model like the one we developed is that it can be used for several languages and does not need to be defined for every new language that should be integrated.

Strein et al. (STREIN ET AL. [SLLL07]) have developed an extensible meta

model for representing relevant properties of projects. They extended this approach so that their meta model can represent a project or a file in which more than one programming language occurs (STREIN ET AL. [SKL06]). This allows dealing with a multiple language project language-independently. For our purpose, the data model suffices for the representation of feature projects.

Visualizing software

For visualizing software, several tools and visualizations are available. Diehl provides an overview of available tools (DIEHL [Die07]). Since there are so many tools and we only want to explain why visualization of source code is useful, we selected three tools and shortly explain what they do. The selection is not meant to be representative, but only roughly points out the use cases of software visualization.

StackAnalyzer aids programmers during development by supporting a static program analysis (EVSTIOUGOV-BABAEV [EB02]). It visualizes the user stack and the system stack. In order to support dynamic program analysis, X-Tango was developed (STASKO [Sta90]). It visualizes algorithms. Both could be used to support the debugging of programs. Similar functionalities in FeatureIDE would increase the usability of FeatureIDE and thus help to make it more successful. For the visualization of the contents of a project, SeeSoft was developed (SEESOFT [ESEES92]). It represents files as rectangles and source code lines as colored pixel lines. The color is an indicator of the age of the according source code line. This concept can be used for a collaboration diagram. All tools constitute single application and are not part of an IDE.

Extensible IDEs

Further tools that support feature-oriented SPL development are pure::variants (PURE::VARIANTS [pur06]) or Gears (KRUEGER [Kru08]). Both are commercial systems. Unlike FeatureIDE, they are closed-source software and thus make extension difficult, although it is possible. Furthermore, they focus on domain analysis and provide less support for the implementation. FeatureIDE, on the other hand, aims at supporting all stages of domain engineering.

Another extensible IDE is Visual Studio (NAYYERI [Nay08]). It is a commercial closed-source Microsoft product. However, due to the closed-source nature, extensions are more difficult than with Eclipse. Furthermore, extending Visual Studio requires purchasing it, whereas Eclipse can be downloaded free of charge.

6.2 Future work

In this section, we present further steps for improving FeatureIDE. Of course, the most obvious step is starting to fulfill the requirements. The following requirements remain to be considered. The letters in braces indicate the current status of the requirements. N means not fulfilled and P partially fulfilled. Since our data model is a first step towards an efficient extension mechanism, this requirement is now partially fulfilled. For the remaining, the status is the same.

1. An IDE must provide an efficient mechanism for extensions. (P)
2. An IDE must support comfortable compiling and running source code. (P)
3. An integrated editor must provide syntax highlighting, auto completion, and mark compiler errors. (P)
4. A structural overview of source code and feature modules must be provided. (P)
5. An IDE must provide a basic set of refactoring. (P)
6. An IDE must support debugging of source code. (N)

Since a language-independent structure of FeatureIDE crosscuts all other requirements, this requirement should be fulfilled at first.

With our data model, we took the first step to provide an efficient mechanism for extension. One unanswered question concerning our data model is its scalability. In our evaluation of the performance, we only considered small projects, since we wanted to evaluate if our data model is suitable for the representation of feature project at all. However, the performance test of the data model was necessary (cf. Section 5.2 and 5.3.4), because if the execution time of the data model was unacceptable for small projects, we would not need to consider it further. However, since our evaluation yielded that the performance of our data model is good, a test for scalability constitutes a next step for evaluating the data model. We did not test the scalability of the data model, because AHEAD and FeatureHouse are relatively new tools (BATORY ET AL. [BSR04]; APEL AND LENGAUER [AL08]), thus, there are too few example projects we could use to test the scalability and implementing them ourselves would exceed this thesis. Furthermore, we used a rudimentary visualization 5.1.1. However, a test for scalability must also take into account the performance for a high quality visualization, because such a visualization may need longer to be build.

An important approach to test a GUI is a usability test (NIELSEN [Nie94]). Hence, to evaluate the fulfillment of the requirements, a usability test can be conducted. This does not necessarily require a large set of users. Often, it suffices to let the developers observe, how two or three subjects fulfill a task with their GUI (HIRSCH [Hir81]; MILLS ET AL. [MBR+86]). This reveals flaws and good realization of the design. To evaluate the data model, we could observe how users deal with our collaboration diagram and outline view or how developers solve a programming task with our data model (e.g. generating a collaboration diagram).

Chapter 7

Conclusion

SPLs constitute a powerful approach for the development of large software systems. In order to increase the acceptance and use of SPLs, good tool support is necessary. One tool focusing on feature-oriented SPL development is FeatureIDE. However, our user study revealed that FeatureIDE supports the development of SPLs only to a limited amount. In order to improve FeatureIDE and thus its success, we defined two goals:

1. Develop a list of requirements that FeatureIDE must fulfill in order to become more successful.
2. Provide a language-independent data model for the representation of feature projects.

For the list of requirements, we examined several IDEs and identified a list of functionality that is common to all of them. In a user study, we revealed that all of the identified functionality is important, yet only few are regarded as well realized in FeatureIDE. Based on the process of SPL development, domain engineering, we developed a list of requirements and stated that only few are fulfilled: The support for the creation of a domain model, the management of feature modules, and the composition of a product. For the remaining requirements, further effort is needed, as we outlined. In order to help future developers to start, we presented information for first steps, e.g. the extension points of Eclipse and how to use them.

Regarding our second goal, a language-independent data model, we provided a first step towards an efficient extension mechanism. The data model represents a feature project as a tree in two steps: First, the structure of the feature project

given by folders and files is mapped to the tree. Then, the content of files is added as leaves to the tree. Since the content of files is organized hierarchically, the leaves of the data model are trees, too. Hence, the data model is a nested tree (cf. Figure 4.3 in Section 4.2.3). In an evaluation, we showed that the data model is indeed suitable for our purpose:

- First, we showed that the data model contains sufficient information for the creation of a collaboration diagram and an outline view.
- Second, a performance test of the data model for the creation of a collaboration diagram and an outline view showed that the data model provides an efficient access to its elements.
- Third, we showed that the data model is language-independent by creating collaboration diagrams and outline views as well as performance tests for several projects in different programming languages. We did not find any evidence that the programming language has an effect on the results of the tests.

Hence, we have fulfilled our goals. The next steps for improving FeatureIDE should address the requirements we stated. Since an efficient mechanism for extending FeatureIDE crosscuts all other requirements, this should be the first problem to solve. The result should be a structure similar to the Eclipse architecture: A language-independent platform that provides all language-independent functionality and plug-ins that encapsulate language-dependent functionality based on the language-independent platform. The other requirements, which are not fulfilled at the moment, can then be implemented based on this architecture.

Bibliography

[ADT07] Felipe I. Anfurrutia, Oscar Díaz, and Salvador Trujillo. On Refining XML Artifacts. In *ICWE '07: Proceedings of the 7th International Conference on Web Engineering*, pages 473–478. Springer, 2007.

[AF96] T. W. Anderson and Jeremy D. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[AKKL07] Sven Apel, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Pointcuts, Advice, Refinements, and Collaborations: Similarities, Differences, and Synergies. *Innovations in Systems and Software Engineering (ISSE) – A NASA Journal*, 3(4):281–289, 2007.

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*. IEEE Computer Society, 2009. To appear.

[AL08] Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *SC '08: Proceedings of the 7th International Symposium on Software Composition*, pages 20–35. Springer, 2008.

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An Algebra for Features and Feature Composition. In *AMAST '08: Proceedings of the 12th international conference on Algebraic Methodology and Software Technology*, pages 36–50. Springer, 2008.

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE '05: Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[AMGS05] Giuliano Antoniol, Ettore Merlo, Yann-Gaël Guéhéneuc, and Houari Sahraoui. On feature traceability in object oriented programs. In *TEFSE '05: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 73–78. ACM, 2005.

[Ape07] Sven Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, University of Magdeburg, Germany, 2007.

[Bat03] Don Batory. A tutorial on feature oriented programming and product-lines. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 753–754. IEEE Computer Society, 2003.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC '05: Proceedings of the 9th Conference on Software Product Line Conference*, pages 7–20. Springer, 2005.

[BLHM02] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating Product-Lines of Product-Families. In *ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 81–92. IEEE Computer Society, 2002.

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2005.

[BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley Longman, 2000.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practice and Patterns*. Addison Wesley, 2001.

[Cod82] Edgar Frank Codd. Relational database: A practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982.

[CR06] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley Longman, 2006.

[Die07] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

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

[Don68] Franciscus Cornelis Donders. On the speed of mental processes. *Pflügers Archiv für Anatomie und Physiologie*, pages 657–681, 1868.

[dRB06] Jim des Rivieres and Wayne Beaton. Eclipse Platform Technical Overview. Technical report, IBM, 2006.

[EB02] Alexander A. Evstiougov-Babaev. Call Graph and Control Flow Graph Visualization for Developers of Embedded Applications. In *Revised Lectures on Software Visualization, International Seminar*, pages 337–346. Springer, 2002.

[ELC06] Montse Ereño, Uxue Landa, and Dra. Rebeca Cortazar. Software product lines structuring based upon market demands. *ACM SIGSOFT Software Engineering Notes*, 31(2):13–16, 2006.

[ESEES92] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. SeeSoft - A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[FPDF98] William Frakes, Ruben Prieto-Diaz, and Christopher Fox. DARE: Domain analysis and reuse environment. *Annals of Software Engineering*, 5:125–141, 1998.

[FS97] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[Gar08] Janel Garvin. Users' Choice IDEs - 2008 A comprehensive study of over 1,200 software developers. Technical report, Evans Data Corporation, 2008.

[Hab97] Karl Haberlandt. *Cognitive psychology*. Allyn and Bacon, 1997.

[Hir81] Richard S. Hirsch. Procedures of the Human Factors Center at San Jose. *IBM Systems Journal*, 20(2):123–171, 1981.

[HOT06] William H. Harrison, Harold Ossher, and Peri L. Tarr. General Composition of Software Artifacts. In *SC '06: Proceedings of the 5th International Symposium on Software Composition*, pages 194–210. Springer, 2006.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, 1990.

[KI05] Patrick Kerpan and David Intersimone. The IDE is Dead, Long Live the IDE. In *Proceedings of the EclipseCon*, 2005.

[Kla08] Stephan Klapproth. Analyse von Merkmalsinteraktionen in Modularen Designs. Master's thesis, School of Computer Science, University Magdeburg, 2008.

[KLD02] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Project Line Engineering. *IEEE Transactions on Software Engineering*, 19(4):58–65, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopez, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[Kru08] Charles W. Krueger. The Software Product Line Lifecycle Framework. Technical report, BigLever Software Inc., 2008. White Paper.

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*. IEEE Computer Society, 2009. To appear.

[LAMS05] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In *ETX '05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, pages 55–59. ACM, 2005.

[LH05] Martin Lippert and Andreas Havenstein. Basic skills. Refactoring with Eclipse step by step, efficiently and easily. *Eclipse Magazin*, 4:52–56, 2005.

[Lik32] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[MBR+86] Carol Bergfeld Mills, Kevin F. Bury, Teresa Lynn Roberts, Bruce Tognazzini, Anna Marie Wichansky, and Paul Reed. Usability testing in the real world. In *CHI '86: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 212–215. ACM, 1986.

[MCP+02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, pages 88–97. ACM, 2002.

[Mic08] Microsoft. An Overview of Microsoft® Visual Studio® 2008. Technical report, Microsoft, 2008.

[Mil68] Robert B. Miller. Response time in man-computer conversational transactions. In *AFIPS '68: Proceedings of the Joint Computer Conference*, pages 267–277. ACM, 1968.

[Nay08] Keyvan Nayyeri. *Professional Visual Studio Extensibility*. Wiley Publishing, Inc., 2008.

[Nee87] Francis Neelamkavil. *Computer Simulation and Modelling*. John Wiley & Sons, Inc., 1987.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[Par72] David Lorge Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[PC86] David Lorge Parnas and Paul Charles Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[pur06] pure::systems. Variantenmanagement mit pure::variants. White Paper, 2006.

[Roy87] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society, 1987.

[SB98] Yannis Smaragdakis and Don S. Batory. Implementing Layered Designs with Mixin Layers. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570. Springer, 1998.

[Sch08] Thomas Schneider. *SAP-Performanceoptimierung: Analyse und Tuning von SAP-Systemen*. Galileo Press, 2008.

[SKL06] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-Language Program Analysis and Refactoring. In *SCAM '06: Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 207–216. IEEE Computer Society, 2006.

[SLLL07] Dennis Strein, Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. An Extensible Meta-Model for Program Analysis. *IEEE Transactions on Software Engineering*, 33(9):592–607, 2007.

[Spo01] Joel Spolsky. *User Interface Design for Programmers*. Springer, 2001.

[Sta90] John T. Stasko. TANGO: A Framework and System for Algorithm Animation. *ACM SIGCHI Bulletin*, 21(3):59–60, 1990.

[Ste06] Friedrich Steimann. The paradoxical success of Aspect-Oriented Programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 481 – 497. ACM, 2006.

[Stu08] Student. The probable error of a mean. *Biometrika*, 6(1):1–25, 1908.

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*. IEEE Computer Society, 2009. To appear.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.

[WDPM06] Reinhard Wolfinger, Deepak Dhungana, Herbert Prähofer, and Hanspeter Mössenböck. A component plug-in architecture for the .NET platform. In *JMCL '06: Joint Modular Languages Conference*, pages 287–305. Springer, 2006.

[Wit96] James Withey. Investment Analysis of Software Assets for Product Lines. Technical report, Software Engineering Institute, 1996.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den January 30, 2009

Janet Feigenspan