FeatureIDE: Empowering Third-Party Developers

Sebastian Krieter
Marcus Pinnecke
sebastian.krieter@ovgu.de
marcus.pinnecke@ovgu.de
Otto-von-Guericke-University
Magdeburg, Germany

Jacob Krüger
jacob.krueger@ovgu.de
Otto-von-Guericke-University
Magdeburg, Germany
Harz University of Applied Sciences
Wernigerode, Germany

Joshua Sprey
Christopher Sontag
Thomas Thüm
j.sprey@tu-braunschweig.de
c.sontag@tu-braunschweig.de
t.thuem@tu-braunschweig.de
Technische Universität
Braunschweig, Germany

Thomas Leich
tleich@hs-harz.de
Harz University of Applied Sciences
Wernigerode, Germany

Gunter Saake
gunter.saake@ovgu.de
Otto-von-Guericke-University
Magdeburg, Germany

ABSTRACT

FeatureIDE is a popular open-source tool for modeling, implementing, configuring, and analyzing software product lines. However, FeatureIDE's initial design was lacking mechanisms that facilitate extension and reuse of core implementations. In current releases, we improve these traits by providing a modular concept for core data structures and functionalities. As a result, we are facilitating the usage of external implementations for feature models and file formats within FeatureIDE. Additionally, we provide a Java library containing FeatureIDE's core functionalities, including feature modeling and configuration. This allows developers to use these functionalities in their own tools without relying on external dependencies, such as the Eclipse framework.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**;

KEYWORDS

Software product line, feature-oriented software development, feature modeling, configuration

ACM Reference format:
Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 4 pages.
https://doi.org/10.1145/3109729.3109751

SPLC '17, September 25-29, 2017, Sevilla, Spain

ACM ISBN 978-1-4503-5119-5/17/09…$15.00
https://doi.org/10.1145/3109729.3109751

1 INTRODUCTION

FeatureIDE [11, 16] is a collection of open-source plug-ins for the Eclipse IDE.[1] To this point, it supports several aspects for feature-oriented software development [3, 4] that are comparable to industrial tools, for example, feature modeling, type checking, or testing [2, 13, 14]. Overall, FeatureIDE provides competitive functionality and usability compared to other tools [6, 10]. With its open-source license, FeatureIDE became a popular tool for researchers and practitioners alike [1, 5, 6, 9].

Due to ongoing research, new feature-oriented approaches are developed regularly to support, for instance, cost models, feature-modeling techniques, and automated testing. However, rather than extending or reusing FeatureIDE, many developers still implement tools and techniques from scratch to address such new topics. Two main reasons for this seem to be FeatureIDE's complex structure, which hampers usability and extensions, and the dependency on Eclipse, which hampers the integration into other tools and IDEs. To address these points, we developed the following two extensions:

- We introduce an abstract factory pattern into FeatureIDE that facilitates extending FeatureIDE's core data structures. Hence, developers can integrate their own algorithms and extensions into FeatureIDE more easily.
- We restructured FeatureIDE's core architecture to provide a library that is independent of Eclipse. Thus, we enable developers to use FeatureIDE's functions for feature modeling and analyses in their own tools.

The source code of FeatureIDE, along with documentation for users and developers can be found at GitHub.[2]

2 FEATURE MODELING IN FEATUREIDE

The feature model [3, 7] is the core data structure within FeatureIDE to enable feature-oriented software development. It is essential for each product line in FeatureIDE and it

[1] https://www.eclipse.org/
[2] https://github.com/FeatureIDE/FeatureIDE

is required for most operations, for instance, configuration, analysis, and source-code traceability. The feature model primarily defines the *features* of a product line and their interdependencies. This includes a tree-based feature diagram and *cross-tree constraints*, which define dependencies outside of the hierarchical tree structure. Furthermore, the model also contains information about the feature order, feature descriptions, and additional properties.

FeatureIDE provides many functions, for instance, modeling, configuring, and testing [2, 13, 14]. A demonstration of most functionalities can be found online on our YouTube channel.[3] In the context of this paper, we are focusing on feature modeling. To this point, FeatureIDE's feature-modeling functionalities include:

- Creating and editing a feature model
- Analyzing a feature model and calculating statistics
- Converting a feature model to different file formats
- Configuring a feature model
- Automatically generating products

With the improvements presented in this paper, we facilitate the reuse of these functionalities within third-party tools. Furthermore, we encourage third-party developers to extend FeatureIDE with their own feature model implementations, for example to support non-functional properties.

3 FEATURE-MODELING EXTENSIONS

FeatureIDE is one of more than 100 research prototypes and industrial tools that cover different phases and tasks of feature-oriented software development [5, 10, 12]. This large number is a result of many researchers and companies implementing their own tools, fitting to their specific requirements. However, most of those tools are specialized in one particular task or are implemented only as a proof of concept for particular research. Thus, they often lack useful functions, have to rely on other tools to support a full integrated development process, or become outdated due to a lack of maintenance. In addition, most tools internally use different implementations of feature models and features. This is often done to store specialized data within a specific structure to support the tools' task. Even file formats to store feature-model data vary, including, FAMA, GUIDSL, Velvet, Dimacs, and several XML formats such as SXFM. Thus, most tools are incompatible to one another.

Problem Statement. FeatureIDE's community approach is to offer a wide range of functionalities, to regularly incorporate current research, and to cooperate with institutions and companies to remain up-to-date. However, in order to use FeatureIDE's functionality, until now data must first be converted to its internal feature-modeling format. This might be unfeasible as it would be necessary to re-implement either FeatureIDE or the third-party tool or to switch between both.

Solution. To address the described diversity, our goal is to allow the usage of external implementations for feature models within FeatureIDE. In detail, we offer the following:

[3] https://www.youtube.com/channel/UC0xYesZDzhFUbq6GUKtr3uA

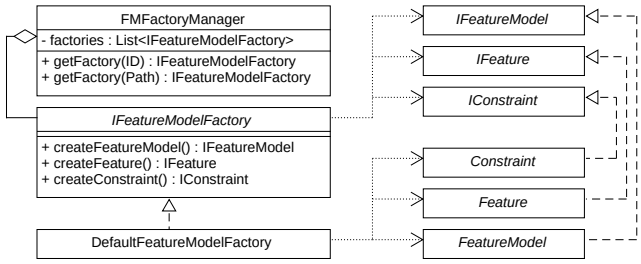

Figure 1: Classes of FeatureIDE's feature-model framework.

- We allow the usage of external implantations of feature models, single features, and constraints by providing interfaces to these classes.
- We enable dynamic instantiation of different implementations via the abstract factory pattern.
- We provide extensibility via Eclipse extension points.

We achieve these improvements by providing interfaces to FeatureIDE's classes for *feature model*, *feature*, and *constraint* and a factory framework to create concrete instances. In Java, interfaces can be added to an existing class (e.g., from other tools) by implementing their declared functions. Thus, interfaces provide the possibility to integrate external source code without changing the original implementation. Where possible, classes can also inherit from FeatureIDE's default interface implementations, which further eases integration.

A particular challenge regarding these modifications of FeatureIDE is that we had to modularize the feature-model class and its dependent source code. Furthermore, we had to enable the instantiation of different interface implementations during runtime.

Implementation. For the implementation, we use the well-known *abstract factory pattern* [8]. Thus, interfaces encapsulate all functions that are necessary to create a concrete factory for a specific kind of feature model and its corresponding elements (i.e., features and constraints). We depict this structure and the main classes of FeatureIDE in Figure 1. In detail, we use the three interfaces `IFeatureModel`, `IFeature`, and `IConstraint`, which abstract functions of the classes `FeatureModel`, `Feature`, and `Constraint`, respectively. Concrete instances of classes implementing these interfaces are derived with a factory that implements the interface `IFeatureModelFactory`. All methods inside FeatureIDE only use references to these interfaces, therefore every instance can be passed to the internal functions. Hence, developers can easily integrate other notations for feature models as they only have to develop classes that implement the interfaces. This allows them to rely on FeatureIDE while being able to adding own functions without changing the core implementation.

To create a particular instance, the corresponding factory must be called. However, often this must be done dynamically, as the usage depends on different aspects, such as the file format or user requirements. Therefore, factories are managed by the class `FMFactoryManager`. The factory manager

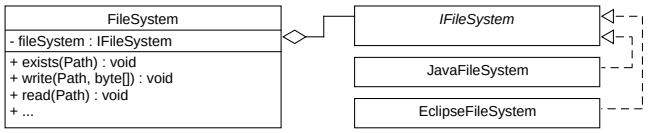

Figure 2: Classes of FeatureIDE's file system framework.

associates a factory with certain file paths, file formats, or the factory's identifier. For example, FeatureIDE uses the `ExtendedFeatureModel` implementation of `IFeatureModel` for all files that are loaded from the Velvet format [15], which can store additional feature properties. The different formats are managed via the interface `IFeatureModelFormat` that contains methods for saving and loading a feature model.

Usage. To use an external feature model, feature, or constraint within FeatureIDE, it must implement the corresponding interface. Furthermore, there must be a factory implementing `IFeatureModelFactory` that instantiates the corresponding classes. Additionally, FeatureIDE has to be made aware of the new classes before these can be used. Developers can do this in two ways: First, factories can be integrated by defining an extension via the Eclipse extension-point framework. In this case, the developers' implementation is loaded upon startup of FeatureIDE's `fm.core` plug-in. Alternatively, factories can also be added programmatically during runtime. To do this, the developers have to call the method `addExtension` of the factory manager and provide an instance of the implemented factory. Additionally, associations of factories to specific file formats or file paths can be created and modified during runtime.

To use a specific factory, a developer requests an instance from the factory manger by specifying the factory's identifier or the path of the feature model on the file system. For convenience, FeatureIDE provides methods for loading feature models that determine the corresponding factory automatically (e.g., `readFromFile` in class `FeatureModelManager`).

4 FEATURE-MODELING LIBRARY

Previously, we explained FeatureIDE's new extensibility using externally developed feature-model classes. This might not be suitable for developers that only want to include certain parts of FeatureIDE's functionality in their own tools.

Problem Statement. When using FeatureIDE as a library, dependencies to Eclipse may become a liability as in most cases they are unnecessary, memory inefficient, and might cause problems when used with older Eclipse versions.

Solution. We provide a library for FeatureIDE's feature-modeling functionality that does not require the Eclipse framework, but instead relies on native Java. Most of FeatureIDE's functions already use solely native Java. However, there is a certain number of classes that rely on Eclipse's functionality, which have to be resolved. In particular, these include: Interaction with the underlying file system (i.e., Eclipse I/O framework), management of external extensions

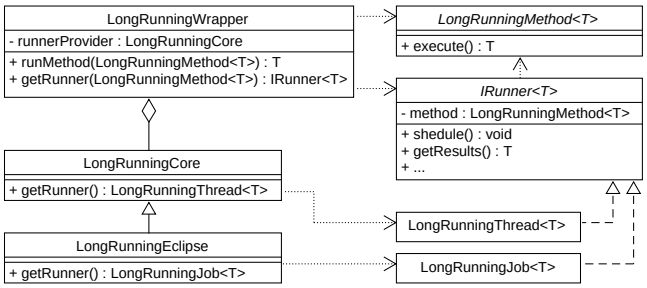

Figure 3: Classes for parallel execution in FeatureIDE.

(i.e., Eclipse extension point framework), and parallel computing (i.e., Eclipse job framework). All of these functions are crucial parts of FeatureIDE. Furthermore, when employing FeatureIDE as an Eclipse plug-in, using the Eclipse functionalities offers some advantages over native Java regarding UI interactions. For instance, an Eclipse job enables parallel computing similar to a Java thread, but is also capable of displaying its progress and being canceled by the user.

To create an Eclipse-independent library, we first have to replace Eclipse functionality with native Java code. In addition, this native Java code should only be used in the library but not while running FeatureIDE as an Eclipse plug-in, requiring two different variants. To reduce the effort of developing the FeatureIDE plug-in and its corresponding library, we aim to generate both from the same source code. Hence, we have to adapt our conceptual development approach to avoid developing both in parallel.

Implementation. Considering our prior mentioned requirements, we employ the well-known *bridge pattern* [8] to exchange the implementation of important functionalities. This means, we use two implementations either based on an Eclipse or native Java API.

In detail, for I/O operations we use the abstract class `FileSystem` and adapted classes that fulfill a corresponding interface. We display the structure of this source code part in Figure 2. Every class within FeatureIDE that accesses the file system calls the corresponding method in an object of the abstract type `FileSystem`. Internally, this concrete object forwards calls to one of the respective implementations of the interface `IFileSystem`. While the class `JavaFileSystem` uses only native Java classes (e.g., java.nio.file.Path, java.nio.file.Files), the class `EclipseFileSystem` employs the respective I/O operations from Eclipse (e.g., org.eclipse.core.runtime.IPath, org.eclipse.core.resources.IResource).

Loading extensions works similar to accessing the file system. When a sub class of the abstract class `ExtensionManager` (e.g., `FMFactoryManager`) is instantiated, it calls an instance of `IExtensionLoader` to get all extensions (e.g., factories) specified by the developer. While the Eclipse implementation (i.e., `EclipseExtensionLoader`) asks the platform's registry for all extensions that implement the given extension point, the Java implementation (i.e., `JavaExtensionLoader`) simply refers to a specified list of extensions. In both cases,

the class `ExtensionManager` allows adding extensions during runtime, making it easier for developers to use lazy loading of their classes. This enables developers to directly integrate their own approaches into FeatureIDE by creating a corresponding extension, without adapting their base code.

Regarding parallel computing, we use the interface `LongRunningMethod` to define methods that can be executed in a separate thread. This includes analyses, checking configurations, and long lasting file operations. In Figure 3, we depict the involved classes. Most of the shown classes are parametrized with the type parameter `T`, which refers to the return value of `LongRunningMethod`. With the class `LongRunningWrapper` every instance of `LongRunningMethod` can be executed either in-place, in a separated thread, or as an Eclipse job. We encapsulate the functionality of both, threads and jobs, within the interface `IRunner`, including methods for starting and canceling, requesting the current status, adding listeners, and retrieving the final results. The implementation of this interface is either `LongRunningThread`, inheriting the Java class `java.lang.Thread` or `LongRunningJob`, inheriting the Eclipse class `org.eclipse.core.runtime.jobs.Job`. A concrete instance of `IRunner` is returned by the method `getRunner` of `LongRunningWrapper`. The concrete instance returned depends on the field `runnerProvider` of the class `LongRunningWrapper`. This field is either an instance of `LongRunningCore`, which returns `LongRunningThread` or an instance of `LongRunningEclipse`, which returns `LongRunningJob`.

Usage. In practice, for all mentioned interfaces the Java implementation is used as default when loading the respective classes. In an Eclipse environment, during the startup routine of FeatureIDE's `fm.core` plug-in, the instances are replaced by their corresponding counterparts. This means that the user does not need to worry about using the correct API, but it is handled automatically.

When building the library, only classes that do not use Eclipse dependencies are included. The result is the JAR file *de.ovgu.featureide.lib.fm.jar*, which is not executable (i.e., does not include a user interface), but can be used as library in any Java application. Besides containing the compiled classes, it also includes the corresponding source files, as well as a text file specifying the version number and checksum of the current build.

5 CONCLUSIONS

In this paper, we described two improvements of FeatureIDE: An abstract factory for the central feature-modeling classes and a separated library containing FeatureIDE's core functionalities. Both allow developers to extend FeatureIDE or to use its functionalities in their own tools. Hence, we hope to empower third-party developers to reuse existing functionality rather than starting from scratch when implementing a research or industrial tool.

Still, despite being a complex tool and depending on the Eclipse framework, we are aware of several academic and industrial developers that already rely on FeatureIDE, especially for feature modeling. Different surveys and case studies confirm this and illustrate the popularity and usability of FeatureIDE in both areas [1, 5, 6, 9]. For this reason, our new contributions seem useful to empower especially companies in using feature-oriented software development. In cooperation with an industrial partner, we currently integrate their feature-modeling technique into FeatureIDE to customize the tool to their requirements. Thus, we tailor existing tooling to their needs instead of forcing them to rely on our implementation.

ACKNOWLEDGMENTS

This research is supported by DFG grants LE 3382/2-1, SA 465/49-1, and Volkswagen Financial Services AG.

REFERENCES

[1] Mathieu Acher, Roberto E. Lopez-Herrejon, and Rick Rabiser. 2013. A Survey on Teaching of Software Product Lines. In *VaMoS*. ACM, 3:1–3:8.
[2] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Tool Demo: Testing Configurable Systems with FeatureIDE. In *GPCE*. ACM, 173–177.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
[4] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. 8, 5 (????), 49–84.
[5] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*. ACM, 7:1–7:8.
[6] Kattiana Constantino, Juliana Alves Pereira, Juliana Padilha, Priscilla Vasconcelos, and Eduardo Figueiredo. 2016. An Empirical Study of Two Software Product Line Tools. In *ENASE*. SciTePress, 164–171.
[7] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*. ACM, 173–182.
[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.
[9] Daniela Lettner, Klaus Eder, Paul Grünbacher, and Herbert Prähofer. 2015. Feature Modeling of Two Large-Scale Industrial Software Systems: Experiences and Lessons Learned. In *MODELS*. IEEE, 386–395.
[10] Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Daniel Lucrédio, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Renata Pontin de Mattos Fortes. 2010. A Systematic Review of Domain Analysis Tools. *Information and Software Technology* 52, 1 (2010), 1–13.
[11] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer. To appear.
[12] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. 2014. An Overview on Analysis Tools for Software Product Lines. In *SPLC*. ACM, 94–101.
[13] Jens Meinicke, Thomas Thüm, Reimar Schröter, Sebastian Krieter, Fabian Benduhn, Gunter Saake, and Thomas Leich. 2016. FeatureIDE: Taming the Preprocessor Wilderness. In *ICSE*. ACM, 629–632.
[14] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. 2015. A Systematic Literature Review of Software Product Line Management Tools. In *ICSR*. Springer, 73–89.
[15] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-Dimensional Variability Modeling. In *VaMoS*. ACM, 11–22.
[16] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85.