FeatureIDE: Taming the Preprocessor Wilderness

Jens Meinicke,[1,2] Thomas Thüm,[3] Reimar Schröter,[2]
Sebastian Krieter,[2] Fabian Benduhn,[2] Gunter Saake,[2] Thomas Leich[1]
[1] METOP GmbH, Germany, [2] University of Magdeburg, Germany, [3] TU Braunschweig, Germany

ABSTRACT

Preprocessors are a common way to implement variability in software. They are used in numerous software systems, such as operating systems and databases. Due to the ability of preprocessors to enable and disable code fragments, not all parts of the program are active at the same time. Thus, programmers and tools need to handle the interactions resulting from annotations in the program. With our Eclipse-based tool FeatureIDE, we provide tool support to tackle multiple challenges with preprocessors, such as code comprehension, feature traceability, separation of concerns, and program analysis. With FeatureIDE, instead of focusing on one particular preprocessor, we provide tool support, which can easily be adopted for further preprocessors. Currently, we support development with CPP, Antenna, and Munge.

https://youtu.be/jVe7f32mLCQ

CCS Concepts

•Software and its engineering → Integrated and visual development environments;

Keywords

Preprocessor, Feature Traceability, Code Analysis

1. INTRODUCTION

Preprocessors are a powerful and widely used mechanism to implement variability in software. Using preprocessor directives (e.g., `#ifdef FEATURE` with the C preprocessor) code can be marked and is only part of the compiled program if the expression evaluates to true. Preprocessors are an accepted method in industry, especially because of the approach's simplicity. However, preprocessors are known to harm *code comprehension* due to the interleaving of source code and comments (i.e., preprocessor directives), hinder *feature traceability* due to missing modularization, and challenge existing *program analyses* as they are typically oblivious to preprocessor directives [12].

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4205-6/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889175

There have been several proposals to improve preprocessors in the last years. On the one hand, concepts that replace the annotative approach, such as preprocessing using abstract syntax trees [8] or modularization of annotated code using compositional approaches [7]. However, these all require replacing existing preprocessors and have not been applied in a large industrial setting. On the other hand, there are prototypical analyses for existing preprocessors [14, 16] and views on the system [6], which are not suitable for industrial-strength development. To promote current research and make it applicable for a wider audience, we integrated several of these approaches into our tool FeatureIDE [17] in a user-friendly manner. FeatureIDE is a variability management tool that suits well for the integration of preprocessors. It already supports domain modeling, configuration, implementation and testing of configurable systems.

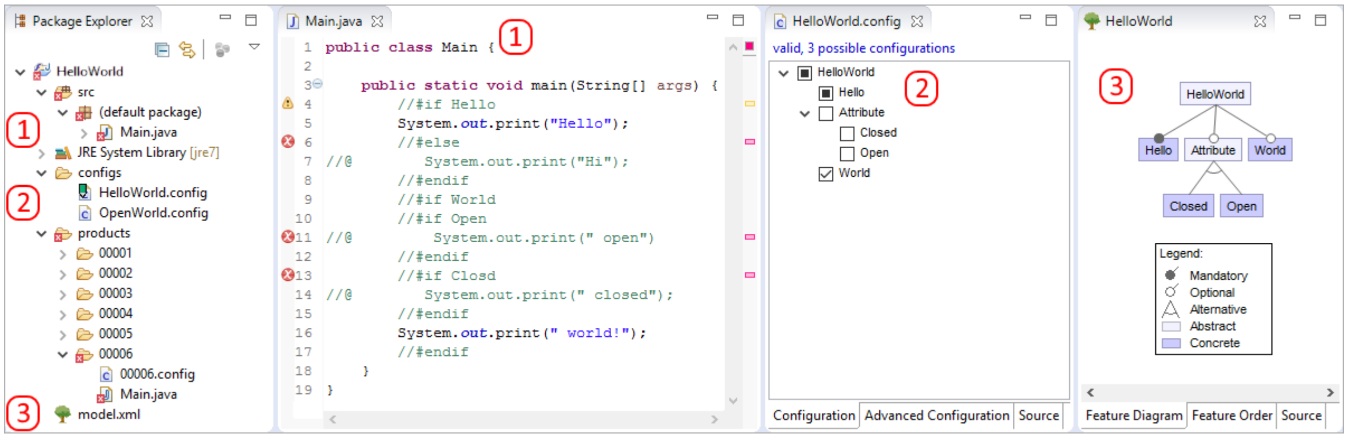
Contribution. Our tool support for preprocessors eases their application in the following ways:

- *Integrated development*: We improve the usability of preprocessors by integrating them into FeatureIDE. Therefore, we can reuse existing functionalities for feature modeling and product configuration.
- *Improvement of code quality*: By combining annotated source code with a feature model, we can provide specialized analyses that are aware of variability, such as detection of invalid annotations and code metrics to identify possible code smells. Furthermore, we can reuse the existing product derivation mechanism of FeatureIDE to provide automated product-based analyses for preprocessors, such as compiler warnings and testing.
- *Feature traceability*: In annotated code, features are often scattered over multiple files. We reuse existing views of FeatureIDE, such as outline and collaboration view, and adapt them to the needs of preprocessors. Furthermore, we integrate the concept of coloring features into FeatureIDE, to identify them in views and source code.

2. INTEGRATED DEVELOPMENT

Preprocessors come with several usability issues, especially for program configuration. In this section, we show how FeatureIDE eases preprocessor usage by integrating them into the process of feature-oriented software development [3].

At ① in Figure 1, we show our running example, a Java program with preprocessor annotations. In our example, we use the preprocessor Antenna, that comments out parts that should not be compiled using `//@` (shown in the source code editor of the file *Main.java* at the Lines 7, 11, and 14). At the left part of Figure 1, we show the standard explorer



```
HelloWorld
  src
    (default package)
      Main.java
  JRE System Library [jre7]
  configs
    HelloWorld.config
    OpenWorld.config
  products
    00001
    00002
    00003
    00004
    00005
    00006
      00006.config
      Main.java
  model.xml
```

```
 1 public class Main {
 2
 3     public static void main(String[] args) {
 4         //#if Hello
 5         System.out.print("Hello");
 6         //#else
 7 //@         System.out.print("Hi");
 8         //#endif
 9         //#if World
10         //#if Open
11 //@          System.out.print(" open")
12         //#endif
13         //#if Closd
14 //@         System.out.print(" closed");
15         //#endif
16         System.out.print(" world!");
17         //#endif
18     }
19 }
```

valid, 3 possible configurations

- ▣ HelloWorld
  - ▣ Hello
  - ☐ Attribute
    - ☐ Closed
    - ☐ Open
  - ☑ World

Figure 1: Integration of preprocessors in the process of feature-oriented software development with FeatureIDE. 1: Program implementation and automated preprocessing. 2: Product configuration using a configuration editor. 3: Domain analysis and modelling with a feature model.

view that depicts the project structure. To reuse Eclipse functionalities, such as automated and incremental compilation, FeatureIDE projects extend conventional Eclipse projects. Thus, the source files are automatically compiled after they are changed by the preprocessor. Furthermore, program execution can be done as usual.

The example program is a configurable application that prints different kinds of hello-world messages. Configuring the program without tool support is difficult and impractical. The programmer needs to be aware of features and their relations. Furthermore, build files need to be adjusted and configured manually. To ease the configuration process, required features can be selected at FeatureIDE's configuration editor ②. The result is a configuration that is used as input for the preprocessor.

Features can depend on each other. For example, the features *Open* and *Closed* cannot be selected together as they exclude each other. To define such dependencies, FeatureIDE provides an editor to create feature models ③. In the example feature model, the feature *Hello* is mandatory, *World* is optional, and *Open* and *Closed* exclude each other. These dependencies are used to ensure correct selections of features in the configuration editor, and implications are used to automatically propagate feature selections (e.g., if the feature *Closed* is selected, the parent feature *Attribute* is selected as well, and the feature *Closed* is deselected).

With the integration of preprocessors into FeatureIDE we ease their usage. Thus, a programmer can concentrate on implementing the software instead of handling the preprocessor. Currently, we provide support for the preprocessors CPP, Antenna,[1] and Munge.[2]

3. IMPROVEMENT OF CODE QUALITY

Ensuring good code quality is difficult, especially in configurable software, as defects may only appear in certain products. In FeatureIDE, we provide several analyses that improve code quality of preprocessor-annotated code with support for consistency checking, product-based analyses, and analysis of annotation usage to detect code smells.

[1]http://antenna.sourceforge.net

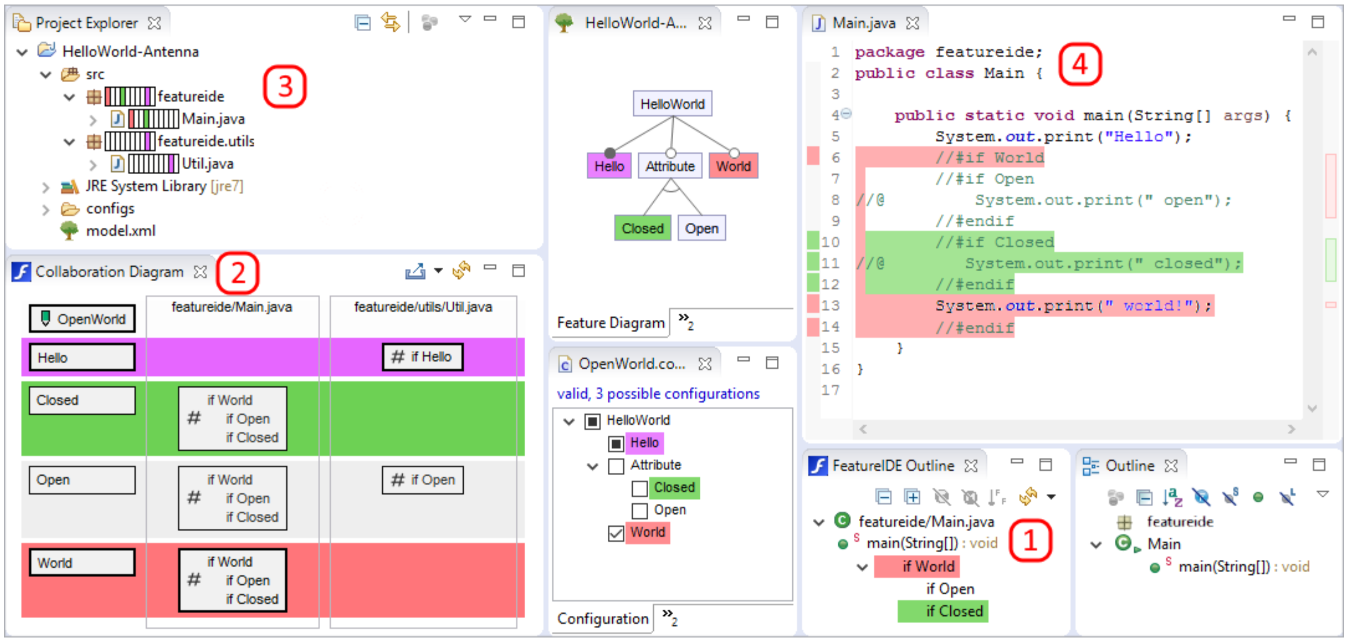
[2]https://github.com/sonatype/munge-maven-plugin

Consistency Checking. A first step to improve code quality is to ensure a consistent mapping between annotations and the feature model. Such inconsistencies either cause code to be dead (i.e., never included in any configuration) or annotations are defined redundantly. FeatureIDE therefore provides two types of checks. First, a check whether the features used in annotations are defined, and second, whether the combination of features in ifdef-expression is consistent to the dependencies in the feature model.

Inconsistencies can be caused by incomplete renaming. See that the feature *Closed* is typed incorrectly in the source code (Line 13). To prevent inconsistencies, FeatureIDE checks for all annotations whether their features are defined in the feature model. FeatureIDE also checks whether each concrete feature (i.e., features that should have implementations artifacts [18]) is used in at least one preprocessor annotation to detect unused features. FeatureIDE marks the defect in Line 13 with an error marker as this annotation causes the following code to be dead. To avoid inconsistencies in advance, FeatureIDE provides automated refactoring for renaming using the feature model editor. Thus, when a feature is renamed, all occurrences of the feature in annotations are renamed as well. For additional support to prevent wrong usage of features in annotations, FeatureIDE provides a content assist where the features can be selected.

Further inconsistencies are caused by invalid combinations of features in annotations. Such inconsistencies cause code to be dead or always included what makes the annotation redundant. In Figure 1, we show examples for both cases. In Line 4, the annotation uses the feature *Hello*. Since this feature is a core feature (i.e., included in any configuration), the expression *Hello* is always true. In this case the annotation should be removed. The next inconsistency is caused by the `else`-expression in Line 6. As *Hello* is always true, the else-branch can never be active. Thus, the corresponding code is never active in any configuration. FeatureIDE provides similar analyses for nested annotations and reasoning about more complicated expressions.

Product-Based Analyses. In FeatureIDE we provide automated product-based analyses that support generation of products as defects may only appear in certain products. In product-based analyses one configuration is



```
package featureide;
public class Main {

    public static void main(String[] args) {
        System.out.print("Hello");
        //#if World
        //#if Open
//@            System.out.print(" open");
        //#endif
        //#if Closed
//@          System.out.print(" closed");
        //#endif
        System.out.print(" world!");
        //#endif
    }
}
```

Figure 2: Improved feature traceability using specialized views, FeatureIDE outline and collaboration view, and colors to easily identify features.

analyzed at-a-time using common single-system analyses. FEATUREIDE provides several strategies for product-based analysis: All configurations, T-Wise configurations that cover all interactions among T features [1], and all manually defined configurations. To analyze these configurations FEATUREIDE generates the products according to the specified strategy in a separate folder using the preprocessor (see the folder "products" in the package explore in Figure 1). Then, FEATUREIDE compiles the product and applies the compiler errors and warnings to the preprocessed code. In the example of Figure 1, the preprocessed file *Main.java* in the folder 00006 contains a defect, because a semicolon is missing (see Line 11). As the compiled file in the folder 00006 is not the file of the source folder, FEATUREIDE propagates the error marker to the original file. Currently, we only integrate analyses provided by the Java compiler. However, because product generation and error propagation is a general approach, further analyses can be integrated with minor effort.

Dynamic analyses, such as unit testing, are often only applied to one single product as testing and reconfiguring the system are time consuming. In FEATUREIDE, we extended the automated program derivation to support unit testing (currently for JUnit). To identify the failing configurations, FEATUREIDE shows a hierarchical structure in the JUnit view that shows failed tests for each configuration.

A more sophisticated way to analyze a product line is variability-aware analysis. Such analyses are able to efficiently check all configurations of the product line [16]. The FEATUREIDE extension for the C preprocessor, Colligens [13], supports such analysis using TypeChef [9], a variability-aware type checker.

Code Metrics. Code metrics are useful to detect probably error prone code. In FEATUREIDE, we integrated several statistics on ifdef-usage presented in a statistics view. First, the number of ifdefs per file indicates files that are highly affected by variability. Second, the nesting depth of ifdef directives indicates complicated code due to interactions of multiple directives what makes the code harder to understand. Third, we measure the number of involved features in directives. A high number of features indicates also a high feature interaction. We currently integrated these three metrics as they are intuitive and useful. There exist more metrics on preprocessor usage that might be included in future [11].

4. FEATURE TRACEABILITY

The ability to identify features defined in the feature model, at code level is called feature traceability [2, 4, 5]. Preprocessor annotations may clearly specify certain features; however, as a feature may be scattered over multiple files, specialized traceability support is required [7]. To provide support for feature traceability, we adopted research results from virtual separation of concerns [8], code comprehension through background colors [5, 8], and collaborations from feature-oriented programming [3, 15]. First, we show how views can help to understand variability of the program. Second, we present how features can be mapped to colors to identify them in the program. We present our support for feature traceability based on the example program shown in Figure 2.

Specialized Views. The FEATUREIDE outline shown Figure 2 at ①, is an extension of the existing outline known from Eclipse. An outline usually shows the fields and methods of a source file. In addition, FEATUREIDE also shows the variability of those. The FEATUREIDE outline shows the ifdefs in which the elements exist and additionally the ifdefs inside each method. Thus, the FEATUREIDE outline is a compressed view on the variability of the file.

To get an overview on the variability of the whole program, FEATUREIDE provides a collaboration diagram shown at ② in Figure 2. The diagram is a table where the files are shown as columns and each line represents a feature. The entries at the intersection show whether a file has implementations for a feature. For additional information, we show the nesting of the corresponding ifdef. Since the diagram grows with the number of features and files in the program, it can be filtered by the features and files of interest.

FeatureIDE is capable of creating variability-aware source-code documentation for annotated Java applications [10]. By using an extended Javadoc syntax, developers are able to generate documentation for individual products, single features, and meta documentation for the entire product line.

Colors. The identification of certain features using colors has been shown to be intuitive and useful [5]. Thereby, a feature corresponds to a color. As the feature model is used to define the features of the program, we use the feature model editor to apply colors to features, shown in Figure 2. These colors are then applied to several parts of FeatureIDE: the FeatureIDE outline, the collaboration view, and the configuration editor.

To easily find files that implement a certain feature, we extended the project explorer with color support, shown at ③ in Figure 2. In the box before the names we show which colors are used in the files. For example, if we are only interested in the feature *Closed*, annotated with green, we see that it only appears in the file *Main.java*. This functionality is applied to packages as well. The box for each package shows all colors of features that are implemented in the package and all sub-packages.

To ease the identification of features and their relations at source level, FeatureIDE provides highlighting via background colors, shown in the editor at ④ in Figure 2. Code that belongs to a certain feature is highlighted with the corresponding color (e.g., code that belongs to *Closed* is highlighted with green). As the annotations for *Open* and *Closed* are nested in the ifdef of *World*, we show this relation by nesting also the background colors.

5. CONCLUSION

Tool support in form of IDEs is crucial for efficient development of software. Preprocessors create several challenges, such as usability, code quality, program analysis, and feature traceability. With FeatureIDE, we support integrated development with preprocessors. We directly integrate the preprocessors CPP, Antenna, and Munge. With a direct connection to a feature model, features and their dependencies can be designed. Additionally, we support several analyses, such as consistency checking of annotations and product-based checks. Finally, we provide support for feature traceability using several specialized views and colors to identify features. With our complete integration we ease the use of preprocessors with a general approach that can be used in research and practice.

ACKNOWLEDGMENTS. This work is partially funded by the BMBF grant (01IS14017A, 01IS14017B).

6. REFERENCES

[1] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-Based Prioritization in Software Product-Line Testing. In *SPLC*, pp. 197–206. ACM, 2014.

[2] G. Antoniol, E. Merlo, Y.-G. Guéhéneuc, and H. Sahraoui. On Feature Traceability in Object Oriented Programs. In *TEFSE*, pp. 73–78. ACM, 2005.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013.

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

[5] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do Background Colors Improve Program Comprehension in the #Ifdef Hell? *EMSE*, 18(4):699–745, 2013.

[6] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *EASE*, pp. 66–75. IET, 2011.

[7] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *McGPLE*, pp. 35–40. Department of Informatics and Mathematics, University of Passau, 2008.

[8] C. Kästner and S. Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *JOT*, 8(6):59–78, 2009.

[9] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*, pp. 805–824. ACM, 2011.

[10] S. Krieter, R. Schröter, W. Fenske, and G. Saake. Use-Case-Specific Source-Code Documentation for Feature-Oriented Programming. In *VaMoS*, pp. 27:27–27:34. ACM, 2015.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE*, pp. 105–114. IEEE, 2010.

[12] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The Love/Hate Relationship with The C Preprocessor: An Interview Study. In *ECOOP*, pp. 495–518. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[13] F. Medeiros, T. Lima, F. Dalton, M. Ribeiro, R. Gheyi, and B. Fonseca. Colligens: A Tool to Support the Development of Preprocessor-based Software Product Lines in C. In *CBSOFT*, 2013.

[14] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, and G. Saake. An Overview on Analysis Tools for Software Product Lines. In *SPLat*, pp. 94–101. ACM, 2014.

[15] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pp. 419–443. Springer, 1997.

[16] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 47(1):6:1–6:45, 2014.

[17] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 79(0):70–85, 2014.

[18] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *SPLC*, pp. 191–200. IEEE, 2011.