

CIDE: Decomposing Legacy Applications into Features

Christian Kästner
School of Computer Science
University of Magdeburg
39106 Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

Abstract

Taking an extractive approach to decompose a legacy application into features is difficult and laborious with current approaches and tools. We present a prototype of a tool-driven approach that largely hides the complexity of the task.

```
1 class Stack {  
2     boolean push(Object o) {  
3         Lock lock=lock();  
4         if (lock==null) {  
5             log("lock failed for: "+o);  
6             return false;  
7         }  
8         elementData[size++] = o;  
9         ...  
10    }  
11  
12    void log(String msg) { /*...*/ }  
13 }
```

1 Introduction

A software product line (SPL) aims at creating highly configurable programs from a set of features. To reduce costs and risks, developers often take an extractive approach for creating the SPL by refactoring one or more legacy applications into features [2]. In prior case studies we experienced that refactoring legacy applications is a complex and difficult tasks [4]. When implementing features with mixin layers [6], aspects [5, 3], or other language-based approaches, an severe overhead is required to implement features.

During development and discussions we communicated with code examples where we underlined code, or used other text styles and colors, to mark features in the original code. An example of a *Stack* with two features *Locking* and *Logging* is shown in Figure 1.¹ This way to present features turned out conveniently, because it hides all language constructs that are usually necessary to implement these features. Similarly, in our refactoring process we first deleted all feature code, i.e., all underlined or colored code, and afterward reintroduced this code with an aspect or mixin layer [4].

Finally, we took this idea and created a tool called *Colored IDE (CIDE)* where refactoring a legacy application is as simple as underlining or coloring code in the example above. In the demonstration we show the basic concepts and benefits of *CIDE*.

¹In the printed version we use **bold** and *italic* fonts to mark features.

Figure 1. Colored Example Code

2 CIDE

CIDE is an Eclipse plug-in that replaces the Java editor in SPL projects. Developers start with a standard Java legacy application, then they select code fragment and associate them with features from the context menu. The marked code is then permanently highlighted in the editor using a background color associated with the feature. It is also possible to associate a code fragment with more than one feature, as Line 5 in our example above. In this case we currently mix the colors and show the full list of associated features in a tool tip.

Internally, *CIDE* uses an abstract syntax tree (AST) representation of the source code. Thus, the users can only mark AST nodes and their children – e.g., classes, methods, statements, or parameters – but not arbitrary text. The focus however is on hiding as much SPL related complexity as possible from the user.

To create a configuration, the user invokes a dialog to select the features of the configuration. *CIDE* copies the project and removes all code that is associated with features not included in the configuration. This resembles preprocessors, but additional preprocessor statements are not required in the base code. Technically, the removal of feature code is implemented as AST transformations.

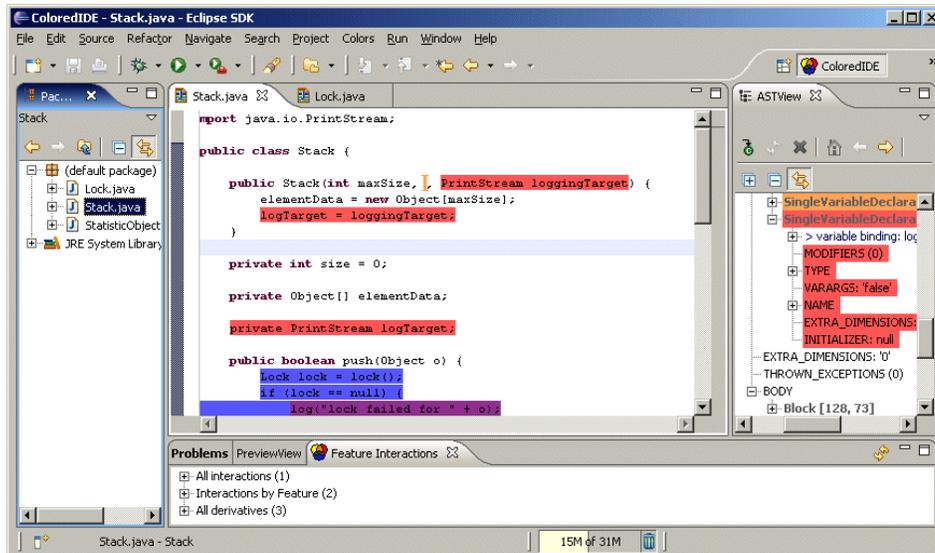


Figure 2. CIDE Screenshot

3 Advanced Topics

CIDE provides a *virtual desktop* on the project, where certain features can be hidden from the editor. Thus, it is possible to view and edit the simplified source code without feature code. Markers that indicate hidden code are still present to allow safe changes.

CIDE checks syntactical correctness of the decomposed application. For example a method call must be associated with all features the method's definition is associated with, otherwise some configurations result in missing references. *CIDE* provides several such checks that *mirror compiler checks* and can ensure syntactical correctness of the source code in all configurations.

CIDE provides a *granularity* that we have not found in any language-based approach. For example, we can associate single statements inside a method, or even method parameters with features and thus remove them in some configurations.

CIDE supports analyzing *interactions* between feature implementations. During decomposition each feature can be marked individually, but interactions are found naturally each time features overlap. Tools and statistics support the developer in finding and analyzing interactions.

4 Future Work

Our current focus is on providing a tool-based approach on decomposing legacy applications to hide as much complexity as possible from the user. In future work we intend to expand our tool as a general purpose SPL tool that can

also be used to design new SPLs and support other code fragments than Java code.

Furthermore, we intend to provide a language-based foundation, so that the tool can be used as a *front end* to a language-based approach. For example, we could store features according to the AHEAD model [1] but still present the composed marked front end to the user that hides all internal complexity.

Acknowledgments

We thank Don Batory, Sven Apel, Martin Kuhlemann, and Christian Lengauer for their helpful discussions and comments on earlier versions of the tool.

References

- [1] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6), 2004.
- [2] P. Clements and C. Kreuger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4), 2002.
- [3] M. L. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference*. 2000.
- [4] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference*, 2007.
- [5] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming*. 1997.
- [6] Y. Smaragdakis and D. Batory. Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2), 2002.