Automating Feature-Oriented Refactoring of Legacy Applications

Christian Kästner, Martin Kuhlemann
School of Computer Science
University of Magdeburg
{ckaestne,mkuhlema}@ovgu.de

Don Batory
Dept. of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu

Abstract

Creating a software product line from a legacy application is a difficult task. We propose a tool that helps automating tedious tasks of refactoring legacy applications into features and frees the developer from the burden of performing laborious routine implementations.

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 and decomposing one or more legacy applications into features [2]. In prior case studies, we detached optional features like transactions, statistics, or caches from database systems, and experienced that refactoring legacy applications manually is a complex and difficult task containing many routine operations [7].

When decomposing a legacy application into features, the developers focus is on identifying the feature code, i.e., classes, methods, fields, or statements associated with a certain feature. In contrast, the actual refactoring, consisting of removing code fragments and reintroducing them in feature modules, is a routine task that can be automated with a tool.

Previously, we built a tool called *ColoredIDE* to identify and mark feature code in a legacy Java application. Now, we use this marked code base to refactor a legacy application into a software product line with multiple features.

Features in an SPL can be implemented in different ways. Current research suggests to implement features as mixin layers [12] or aspects [5, 8], but other implementation approaches are possible. In the prototype of our tool we investigated refactorings into feature modules implemented with *Jak* [1] and *AspectJ* [8]. In this paper we focus on AspectJ as target language and assume a basic knowledge of it.

2. Refactoring

The input of our refactoring tool is a list of features and a marked version of the source code, where fragments are associated to these features. In Figure 1 we show an example class *Stack* with a feature *Locking* whose code is underlined. Technically, features are associated to elements in the *abstract syntax tree (AST)* of the source code, e.g., the AST node for the method *lock* (Line 8) and the statements in Lines 3 and 5 are marked with the *Locking* feature (underlined).

```
1 class Stack {
2   void push(Object o) {
3     Lock lock = lock(o);
4     elementData[size++] = o;
5     lock.unlock();
6   }
7
8   Lock lock(Object o) { /*...*/ }
9 }
```

Figure 1. Marked Legacy Code

Our tool now creates a new project for the SPL with directories for every feature and a directory for the base code. The base code contains the original program without any feature code. The feature directories contain aspects that reintroduce the feature code. In Figure 2 we show the resulting class of the base code and the aspect implementing the *Locking* feature for our example. The SPL can now be configured by selecting the directories to include in the compilation process.

For the implementation of the AspectJ refactoring we follow proposals for refactorings like *Extract Introduction* [6], *Move Field from Class to Inter-type* [10], or *Extract Advice* [6].

3. Advanced Topics

Generally, our tool uses more sophisticated rewrites than shown in the example above. For instance, when the feature

```
class Stack {
  void push(Object o) {
    elementData[size++] = o;
  }
}
```

```
aspect Synchronization {
  void around(Stack stack, Object o) :
    execution(void Stack.push(Object)) && args(o) &&
         this(stack) {
    Lock lock = stack.lock(o);
    proceed(stack);
    lock.unlock();
  }
  Lock Stack.lock(Object o) { /*...*/ }
}
```

Figure 2. Refactored SPL code

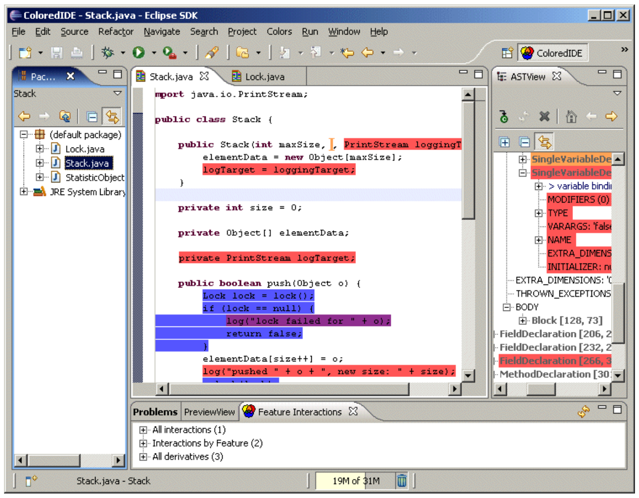
code is not placed at the beginning or end of the method. AspectJ does not support statement level join points [11]. In some cases it is possible to advise a method call that is located next to the feature code, in other cases we have to create artificial join points by preparing the base code. For example, we introduce calls to empty hook methods [11] or perform a preliminary *Extract Method* refactoring [4].

Furthermore, code can be associated with multiple features. Such code is usually a result of feature interactions, e.g., when one feature calls a method introduced by another feature. To refactor such cases we use the derivative model by Liu et al. [9] and create separate modules containing aspects for these code fragments. Again our tool automates the creation of the additional modules and the refactorings.

Finally, our tool initially refactored every marked code fragment individually. That means that advanced AspectJ mechanisms, e.g., pattern expressions for homogeneous pointcuts, were not employed. However, our tool combines pointcuts where advice statements have equal bodies with automated *Extract Pointcut* refactorings [3]. Thus, our refactoring tool takes advantage of AspectJ's capabilities and reduces code replication automatically.

4. Conclusion

Refactoring a legacy application into features to create a SPL is a difficult and laborious task. It consists of detecting features in the legacy code and of their refactoring. While detecting features is an interactive procedure the refactoring can be automated completely. We propose a refactoring tool which generates an SPL implemented in *Jak* or *AspectJ* based on marked legacy code.



```
mport java.io.PrintStream;

public class Stack {

    public Stack(int maxSize, |, PrintStream loggingT
        elementData = new Object[maxSize];
        logTarget = loggingTarget;
    }

    private int size = 0;

    private Object[] elementData;

    private PrintStream logTarget;

    public boolean push(Object o) {
        Lock lock = lock();
        if (lock == null) {
            log("lock failed for " + o);
            return false;
        }
        elementData[size++] = o;
        log("pushed " + o + ", new size: " + size);
```

Figure 3. ColoredIDE Screenshot

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] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2005.

[4] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] M. L. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference*. 2000.

[6] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of Aspect-Oriented Software. In *Proc. Net.ObjectDays*, 2003.

[7] C. Kästner. Aspect-Oriented Refactoring of Berkeley DB. Master's thesis, University of Magdeburg, Germany, 2007.

[8] G. Kiczales et al. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming*. 2001.

[9] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proc. Int'l Conf. on Software Engineering*, 2006.

[10] M. P. Monteiro and J. M. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2005.

[11] G. C. Murphy et al. Separating Features in Source Code: an Exploratory Study. In *Proc. Int'l Conf. on Software Engineering*. 2001.

[12] 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.