2009 International Conference on Software Reuse

Refactoring Feature Modules*

Martin Kuhlemann[1], Don Batory[2], and Sven Apel[3]

[1] University of Magdeburg, Germany
kuhlemann@iti.cs.uni-magdeburg.de
[2] University of Texas at Austin, USA
batory@cs.utexas.edu
[3] University of Passau, Germany
apel@uni-passau.de

Abstract. In feature-oriented programming, a *feature* is an increment in program functionality and is implemented by a feature module. Programs are generated by composing feature modules. A generated program may be used by other client programs but occasionally must be transformed to match a particular legacy interface before it can be used. We call the mismatch of the interface of a generated program and a client-desired interface an *incompatibility*. We introduce the notion of *refactoring feature modules (RFMs)* that extend feature modules with refactorings. We explain how RFMs reduce incompatibilities and facilitate reuse, and report our experiences on five case studies.

1 Introduction

In feature-oriented programming, a *feature* is an increment in program functionality and is implemented by a *feature module* [1]. Feature modules can add new classes to a program, add new members, and extend members of existing classes. It is common for a program composed from feature modules to be used by another program [2], which we call an *environment*. An environment expects a composed program to have names of classes or methods that can be different from what was generated. We call the non-matching of expectations an *incompatibility* between the composed program and its environment. Incompatibilities occur frequently and hinder reuse [9, 20, 14].

In this paper, we concentrate on refactorings to eliminate incompatibilities. A *refactoring* alters the structure of a program but not its behavior [22, 6]. Existing approaches can be used to integrate a program – also with refactorings – but they have problems: To adapt a program composed from feature modules using contemporary refactoring engines like Eclipse [7], the program has to be composed first and then refactorings are applied to it. The key problem is, if there are n optional features in producing a program and m optional refactorings, then

* Martin Kuhlemann was supported and partially funded by the *DAAD Doktorandenstipendium* (No. D/07/45661). Batory's work was supported by NSF's Science of Design Project #CCF-0724979. Sven Apel's work was supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

2^{n+m} program variants are possible. Hence, brute force is not an option [10]. Wrappers (a.k.a. adapters), as a second approach, increase program complexity as they introduce additional methods and classes [9], and meta-programs require developers to guarantee the resulting program can be compiled. We later discuss these approaches and others in detail. In contrast to prior work, we aim at a unification of features and refactorings in order to establish a general model of configurable and reusable software based on transformations.

We propose that object-oriented refactorings be included in feature modules, called *refactoring feature modules (RFMs)*. We illustrate how RFMs automate recurring tasks that eliminate incompatibilities. When an off-the-shelf program is moved into a feature module then RFMs help automate its integration with other programs. We demonstrate the practicality of our approach with five case studies.

Feature Module *Base*

```
1  public class Container {
2    List _elements;
3    void insert_front(Element e){
4      _elements.add(e);
5    }
6  }
```

Feature Module *LimitedSize*

```
7  refines class Container {
8    int _depth;
9    void setElements(List newElems){
10     _elements=newElems;
11   }
12   void insert_front(Element e){
13     Super.insert_front(e);
14     _depth=_elements.size();
15   }
16 }
```

Feature Module *ContainerAsDeque*

```
17 public class Deque {
18   Container _c;
19   void add_front(Element e){
20     _c.insert_front(e);
21   }
22   void setElements(List newElems){
23     _c.setElements(newElems);
24   }
25 }
```

Fig. 1. Feature-oriented design of a container library.

2 Background

Feature-Oriented Design. In Figure 1, we show three feature modules implemented in Jak, a superset of Java that supports feature modularity and feature composition [1]. When feature modules are selected in a configuration process they add classes or class refinements to a given program. A *class refinement*, which is indicated by the keyword refines, adds members to and extends methods of existing classes.

Feature module *Base* defines class Container. Module *LimitedSize* refines class Container by adding field _depth and method setElements. Existing methods are extended by overriding, e.g., method insert_front (Lines 12-15) refines method insert_front of class Container (Lines 3-5) via an inheritance-like mechanism. This method refinement adds statements and calls the refined method using Jak's keyword Super (Line 13).

Feature module *ContainerAsDeque* adds a new class Deque. The result of composing *Base*, *LimitedSize*, and *ContainerAsDeque* includes both classes Container and Deque. In this example, Deque is a wrapper class (a.k.a. adapter class) for Container, i.e., by delegating methods it makes Container objects accessible under the name Deque and makes method insert_front of Container accessible under the name add_front of Deque.

Refactoring. A *refactoring* is a transformation that alters the structure of a program without altering its observable behavior [22, 6]. One of the uses of

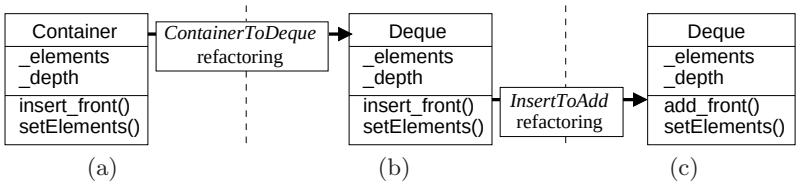


Fig. 2. Refactoring Container with 'Rename Class' and 'Rename Method'.

refactorings is to remove incompatibilities among programs in order to increase reuse [22]. Two common refactorings are 'Rename Method' and 'Rename Class'.[4] We use them as examples throughout the paper.

In Figure 2a, we depict class Container that has been composed from the feature modules *Base* and *LimitedSize* of Figure 1. Figure 2b shows the result of performing the refactoring *ContainerToDeque*. *ContainerToDeque* renames class Container into Deque and adjusts all references. Figure 2c shows the resulting class after the refactoring *InsertToAdd*. *InsertToAdd* renames method insert_front into add_front and adjusts all calls.

Refactorings have parameters that define the target program elements [19]. For example, the parameters of a 'Rename Method' refactoring are (1) the qualified name of the method to rename and (2) the new method name. We use the term *refactoring type* for the template that expects parameters. For example, 'Rename Class' and 'Rename Method' are refactoring types. Once its parameters are provided, the refactoring is fully specified and can be applied. Fully specified refactorings can have names, e.g. the 'Rename Class' refactoring that renames Container to Deque is called *ContainerToDeque* (see Fig. 2).

3 Refactoring Feature Modules (RFMs)

A *refactoring feature module (RFM)* integrates refactorings with feature modules. The basic idea is to define refactorings in refactoring units that become elements of feature modules. By packaging one refactoring per feature module, a particular sequence of refactorings can be applied to a program, just like feature module sequences are composed to build programs. That is, program generation and restructuring are integrated with RFMs.

Concept. Every refactoring type has an interface, which contains a getter method for each parameter of the refactoring type. A refactoring unit is a class-like module that implements a refactoring interface. It implements each getter method by returning the value for a designated parameter. Together the parameter values of a refactoring unit fully specify a particular refactoring. We choose to represent refactorings as class-like modules because this is similar to feature-oriented refinements, and technically it allows us to reuse tool support.

[4] 'Rename Method' changes the name of a method and 'Rename Class' changes the name of a class [6].

Figure 3 depicts a sample RFM, called *ContainerToDeque*, which encapsulates a refactoring unit MyRenameClass. MyRenameClass defines a 'Rename Class' refactoring, i.e., it implements the interface RenameClassRefactoring and defines the getter methods getOldClassId and getNewClassName. The getters of MyRenameClass return the qualified name of the class to rename and the new class name. Refactorings of other types than 'Rename Class' are defined analogously.

Feature Module *ContainerToDeque*

```
refactoring MyRenameClass implements                    1
                    RenameClassRefactoring {
  String getOldClassId(){return "Container";}           2
  String getNewClassName(){return "Deque";}             3
}                                                       4
```

Fig. 3. Refactoring unit that renames Container into Deque.

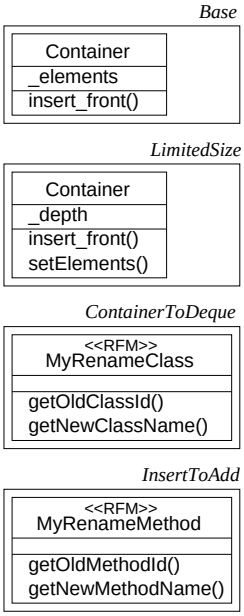


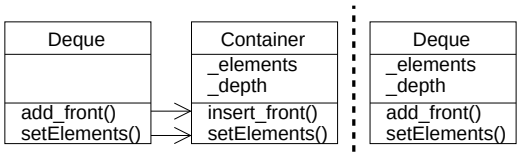
Fig. 4. Sequence of RFMs.

In Figure 4, we show a design in which RFMs are applied successively (in top-down order) to the composition of the two feature modules *Base* and *LimitedSize*. *InsertToAdd* is composed after *ContainerToDeque* and renames method Deque.insert_front into add_front. When all modules are selected in a configuration process the result is the same as composing the feature modules of Figure 1 (class Container is accessible under the name Deque; method Container.insert_front is accessible under the name add_front of class Deque).

Control the scope of RFMs. A transformation is applied when an RFM is composed [1]. That is, the program that is synthesized by composing modules prior to an RFM is transformed by that RFM.

In Figure 4, the classes refactored by the RFMs *ContainerToDeque* and *InsertToAdd* are limited to classes created by feature modules these RFMs follow. That is, *ContainerToDeque* refactors the code added by the feature modules *Base* and *LimitedSize* but not code added/changed by *InsertToAdd* as *InsertToAdd* is composed after *ContainerToDeque*. If an additional feature module *NewContainer* would apply after *ContainerToDeque* and introduce a second class Container, like feature module *Base*, this class would not be affected by *ContainerToDeque* because it would be added after *ContainerToDeque*.

With RFMs, program elements can be both added and deleted (e.g., renaming can be represented as a sequence of deleting and creating code elements). After an RFM renames a method, the old method no longer exists. If a subsequent feature module or RFM references the renamed method by its old name, an error is reported. To guarantee the absence of such errors in all feature compositions is possible with techniques of safe composition [12], another topic that we are investigating.



from Fig. 1 without RFMs

(a)

(b)

from Fig. 4 with RFMs

Fig. 5. Composition results.

RFMs in Action. In the introduction, we observed that generated programs often do not have the correct structure for them to be reused as-is in an environment (e.g., legacy code) [9]. Here is where RFMs can eliminate incompatibilities and promote reuse without altering the functionality of the generated program. RFMs allow us to avoid forwarding methods and classes (commonly used for integration), and thus simplify the resultant program. Figure 5 shows, the composed program in Figure 5b only encapsulates a class with the desired name Deque and no obsolete class Container as in Figure 5a.

Tool Support. We have implemented RFMs as an extension to the Jak language, which adds support for feature modules to Java [1]. We use the AHEAD tool suite [1] to compose feature modules. We extended AHEAD with a plugin mechanism that encapsulates a template program of one refactoring type, e.g., the 'Rename Method' refactoring type is implemented in its own plugin. Refactoring units refer to a plugin with their interface declaration and parameterize the refactoring template program with their getters. More details are given in a technical report [11].

4 Case Studies

We report on experiences with RFMs using two larger and three smaller case studies. We (1) transformed an off-the-shelf library in order to be able to reuse it in an incompatible database engine; (2) integrated variants of *configurable* libraries using RFMs with a legacy environment; (3) used uncommon refactorings to integrate a library of *graph data types (GDTs)*; (4) integrated a *large* Eclipse library using RFMs with minimal effort, and (5) integrated a library of *abstract data types (ADTs)* and thereby removed wrapper classes and methods that became obsolete with RFMs. In Table 1, we show the transformed programs and the refactorings applied to them.

Logging libraries. The off-the-shelf logging library Log4J[5] cannot be used as-is by the database SmallSQL[6] (∼20K lines of source code) due to incompatibilities. To standardize logging in SmallSQL we applied three RFMs to restructure Log4J

[5] http://logging.apache.org/log4j/
[6] http://www.smallsql.de/

Table 1. Information on case studies.

Program	#SLOC*	Refactorings
Log4J	∼12K	1x Move Class, 2x Rename Method
ZipMe	∼3K	2x Move Class, 1x Rename Class
Raroscope	∼250	2x Move Class, 2x Rename Class
TrueZip	∼13K	2x Move Class, 2x Rename Class, 1x Rename Method
GDT	∼1K	4x Move Class, 2x Rename Class, 2x Rename Method, 6x Encapsulate Field, 2x Extract Interface
Workbench.texteditor	∼16K	1x Rename Class, 2x Rename Field
ADT library	59	1x Rename Class, 1x Rename Method

*lines of source code

such that it can be used in SmallSQL. One RFM moves class org.apache.log4j.-Logger into the SmallSQL package smallsql.database and two RFMs rename methods into SmallSQL-compatible names. The RFMs transform single code elements and a number of references to these elements automatically. For example, to make class Logger compatible, we did not have to know and enumerate those 144 points in 38 Log4J classes (distributed over 10 packages), that reference the moved class and must be transformed; we also did not have to find the numerous members that needed to be qualified as public when we moved the class – the 'Move Class' RFM performs these transformations automatically. We observed that one incompatibility could not be eliminated by refactoring Log4J. We introduced with a feature-oriented refinement a single default constructor into the Logger class that calls setters. This way, RFMs do not replace refinements but complement them to integrate programs.

As a result, we can now select either the informal SmallSQL logging engine or the Log4J standard logging library for the SmallSQL database in a configuration process. RFMs allow Log4J (and future releases of it) to be reused in the formerly incompatible SmallSQL environment. We defined the adaptation changes *once*. We found the effort to define RFMs is small and found it comfortable that the code changes the selected RFMs must perform are applied automatically (hidden from us).

Configurable compression libraries. ZipMe[7] is a library to access ZIP archives, Raroscope[8] is a library to access RAR archives, and TrueZip[9] can access TAR archives. The used versions of ZipMe and Raroscope are *configurable*, i.e., different library variants can be composed for each of them from selectable features like *Checksum*. Furthermore, we developed a graphical tool that used an old library to analyze files inside ZIP archives (file names, last modification date, uncompressed footprint) and decompress them. We wanted to replace the old library with ZipMe, Raroscope, and TrueZip to also analyze RAR and TAR archives with our tool. But all variants of these libraries were incompatible with

[7] http://sourceforge.net/projects/zipme/
[8] http://code.google.com/p/raroscope/
[9] https://truezip.dev.java.net

our tool. We applied a number of RFMs to automatically restructure variants of the libraries such that they can be reused in our tool (e.g., in ZipMe we renamed class ZipArchive into ZipFile).

We observed that some incompatibilities cannot be eliminated by refactoring the library variants. This was the case for creating archive representations – our tool passes a File argument but all variants of the libraries take a FileInputStream or String argument. We added a feature module with a single factory method to each library and call the methods in order to bridge this gap. For TrueZip the feature module also encapsulates a method to access streams of single archive entries with certain parameters. Raroscope provides no such streams, so we disabled decompression here (still we can analyze archives). Again, RFMs do not replace refinements but complement them to integrate programs.

Technically, the version of ZipMe can be composed from 13 features to 2^6 different variants. The version of Raroscope can be composed from 5 features to 2^4 different variants. We composed different variants of the configurable ZipMe and Raroscope libraries and all were compatible with our tool automatically when we selected the refactoring features.[10] Interestingly, the variants were compatible although only the fully-fletched versions had been composed before.

After we applied RFMs to TrueZip, its implementation became compatible with our tool. We now also can analyze and decompress TAR archives with our tool. Beside of renaming and moving the TrueZip representations of archives and archive entries, we had to rename the archive method getArchiveEntries into entries because our tool expects this name. For TrueZip, RFMs automate adaptation changes when new versions of TrueZip are released.

Graph library. We integrated a configurable library of GDTs [17] (15 features, 55 library variants) with RFMs into an incompatible environment that used originally the graph library OpenJGraph[11]. Beside renaming and moving Graph and Vertex classes, we had to encapsulate 6 fields in these classes with access methods using RFMs and had to extract interfaces for these classes. With refinements we added five methods. With RFMs, we can now configure multiple GDT variants to be compatible with the OpenJGraph client.

Eclipse library. Dig et al. reported on incompatible environments of the Eclipse library 'workbench.texteditor' (16K lines of source code) [5]. We applied three RFMs to automatically restructure 'workbench.texteditor' such that it can be reused in these environments. One RFM renames class Levenshtein into Levenstein because this name was expected in the environment and two RFMs rename fields from levenshtein into levenstein. In this study, three simple RFMs automatically integrate the large library (and its future releases) with aforesaid environments.

[10] Informally, we performed primitive performance tests and found that our tool decompressed ZIP archives ∼5% faster with (fully-fletched) ZipMe than with the replaced old library, i.e., integrating ZipMe was beneficial.

[11] http://sourceforge.net/projects/openjgraph/

Abstract data types. Our running example of Figure 1 (Container class with its wrapper class Deque) leans on a configurable library of ADTs (5 features, 7 library variants) [2]. We reimplemented this feature-oriented design with RFMs as we have shown in Figure 4. With RFMs, we can now automatically integrate differently configured library variants just by selecting refactoring features. In this study, we removed wrapper classes and methods, that provided access to classes and methods under a different name but became obsolete with RFMs.

Summary. RFMs integrate well with feature-oriented refinements. RFMs allow libraries to be reused in environments they were incompatible with before. Specifically, RFMs can apply (sequences of) pre-defined refactorings to hand-written or synthesized programs *automatically*. After defining RFMs, any number of variants of a configurable library can be configured to be compatible with an environment. While renaming appears most important, RFMs in our perspective may encapsulate any transformation which affects structure but not semantics, e.g., 'Extract Interface' refactoring [6] (cf. GDT study).

We observed that RFMs complicate debugging because the refactored classes of the debugged program differ from the developed classes inside the feature modules. Hence, we need advanced debugging tools that keep track of the performed refactorings such that changes to the program's classes are triggered back to the feature modules automatically.

5 Related Work

Different styles of *wrapper modules* (a.k.a. adapters) forward method calls to wrapped objects in order to integrate incompatible code and to increase reuse, e.g., [8, 14, 4]. Wrappers exist simultaneously with their wrapped objects and so a wrapper is a second way of accessing a wrapped object. RFMs transform bodies of classes such that there is no second way to access objects of a transformed class. However, RFMs avoid problems that wrappers have: Wrappers increase implementation and maintenance effort when they add methods and classes [18, 9]. The forwarding methods of wrappers impact negatively on performance and footprint of the resultant program [9]. Wrappers are complex because (a) wrapper objects have different identities than the wrapped object [5, 23, 9] and (b) they cause type problems as their location in a type hierarchy differs from the location of the wrapped class (redundant hierarchies emerge) [9, 23].

Meta-programming approaches like [20, 25, 3] restructure programs beyond refactoring and generally do not guarantee that generated programs are compilable. Refactoring units parameterize pre-defined meta-programs implemented globally in plugins of our composer. Therefore, developers of RFMs do not care whether generated programs are compilable (ensured by our composer). Some researchers propose refactoring meta-programs [26, 13] or refactorings as language concepts [15]. They all do not integrate refactoring with feature transformations and do not provide a general model of configurable and reusable software.

In order to adapt a program composed from feature modules using contemporary refactoring engines like Eclipse [7] every program variant has to be composed

first and then refactorings are applied to each variant. Since possibly many (up to millions) combinations of feature modules can be composed this approach is not feasible [10]. Re-applying a common set of refactorings with such engines on constantly updated incompatible programs is error-prone and laborious as well [18, 9]. RFMs automate refactorings and sequences of refactorings. RFMs are selected in a configuration process and thus apply at the time a program is composed (in case their feature is selected) – thus, RFMs make refactored programs available even if refactorings were not recorded for them individually.

ReBA [5] helps to integrate a library into environments that use the library but rely on an outdated interface of that library. ReBA uses a trace of edits and refactorings, which lead from the old to the evolved library version. Using this trace, ReBA adds code which allows using the evolved instead of the old version, e.g., elements, that are deleted in the evolved version, are added back when referenced. RFMs can bridge incompatibilities that occur when a library A is replaced by a completely *different* library B. Thereby, in general no helpful trace is available, which maps all code of A to all code of B.

In KIDS, users select correctness-preserving code transformations that improve performance and footprint, e.g., partial evaluation [24]. Refactoring transformations keep correctness too. When selected, RFMs restructure a program to simplify its reuse. Note, by inlining methods and classes with refactorings, RFMs may also improve performance and footprint of a composed program.

Feature-oriented refactoring [16] and aspect-oriented refactoring [21] decompose a program into feature modules of a feature-oriented design and aspects respectively. In contrast, RFMs perform object-oriented refactorings on a program which is composed from features.

6 Conclusion

The structure *and* features (increments in functionality) of a program are important for the program to be reused by an environment. When the interface of a generated program and a client-desired interface mismatch, the generated program cannot be reused by this client. In current technology, transformations to alter the structure of programs (e.g., refactorings) and to alter the features of programs (e.g., feature modules) are still treated as disjoint concepts. In this paper, we have introduced *refactoring feature modules (RFMs)*, which integrate feature modules with refactorings. An RFM automatically alters the structure of programs, which are composed from feature modules. We have implemented support for RFMs and demonstrated in a number of case studies that RFMs can help to reuse programs. Specifically, we showed that with RFMs the studied programs can be integrated automatically and reused in environments they were incompatible with before, i.e., RFMs simplify the reuse of code.

References

1. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *TSE*, 30(6):355–371, 2004.

2. D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *FSE*, pages 191–199, 1993.
3. T. J. Biggerstaff. A new architecture for transformation-based generators. *TSE*, 30(12):1036–1054, 2004.
4. J. Bosch. Design patterns as language constructs. *JOOP*, 11(2):18–32, 1998.
5. D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *ICSE*, pages 441–450, 2008.
6. M. Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.
7. R. M. Fuhrer, M. Keller, and A. Kieżun. Advanced refactoring in the Eclipse JDT: Past, present, and future. In *WRT*, 2007.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley, 1995.
9. U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP*, pages 36–56, 1993.
10. C. W. Krueger. New methods in software product line practice. *CACM*, 49(12):37–40, 2006.
11. M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. Technical Report 15, Faculty of Computer Science, University of Magdeburg, 2008.
12. M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *GPCE*, 2009.
13. R. Lämmel. Towards generic refactoring. In *Workshop on Rule-Based Programming*, pages 15–28, 2002.
14. K.-K. Lau, L. Ling, V. Ukis, and P. V. Elizondo. Composite connectors for composing software components. In *SC*, pages 266–280, 2007.
15. J. R. Lewis, M. B. Shields, E. Meijert, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118, 2000.
16. J. Liu, D. Batory, and C. Lengauer. Feature-oriented refactoring of legacy applications. In *ICSE*, pages 112–121, 2006.
17. R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *GCSE*, pages 10–24, 2001.
18. M. Mattsson and J. Bosch. Framework composition: Problems, causes and solutions. In *TOOLS*, pages 203–214, 1997.
19. T. Mens, N. V. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
20. M. Mezini, L. Seiter, and K. Lieberherr. *Component integration with pluggable composite adapters.* Kluwer, 2000.
21. M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD*, pages 111–122, 2005.
22. W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
23. K. C. Sekaraiah and D. J. Ram. Object schizophrenia problem in modeling Is-Role-Of inheritance. In *Inheritance Workshop*, 2002.
24. D. R. Smith. KIDS: A knowledge-based software development system. In *Automating Software Design*, pages 483–514, 1991.
25. M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Workshop on Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133, 2000.
26. M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A scripting language for refactoring. In *ICSE*, pages 172–181, 2006.