A Logic Based Approach to Locate Composite Refactoring Opportunities in Object-Oriented Code

Călin Jebelean[1], Ciprian-Bogdan Chirilă[1], Vladimir Creţu[1]
[1]Faculty of Automation and Computer Science
University Politehnica of Timişoara, E-mail: {calin.jebelean,ciprian.chirila,vladimir.cretu}@cs.upt.ro

Abstract—**In today's software engineering, more and more emphasis is put on the quality of object-oriented software design. It is commonly accepted that building a software system with maintainability and reusability issues in mind is far more important than just getting all the requirements fulfilled in one way or another. Design patterns are powerful means to obtain this goal. Tools have been built that automatically detect design patterns in object-oriented code and help in understanding the code. Other tools help in refactoring object-oriented code towards introducing design patterns, but human intelligence is needed to detect where these design patterns should be inserted. This paper proposes a logic approach to the automatic detection of places within object-oriented code where the Composite design pattern could have been used. Suspects identified by such a tool could very well be served as input data for other tools that automatically refactor the code as to introduce the missing design pattern.**

I. Introduction

In today's software engineering, more and more emphasis is put on the quality of object-oriented software design. In a world of rapidly changing requirements, it is vital for an object-oriented software system to be able to adapt quickly, saving time and, of course, saving money ([Bec99]). This kind of adaptation is not always easily performed, especially when the system's design ignores certain quality guidelines.

In his well-known book ([Fow00]), Martin Fowler captured a lot of potential design problems and called them *bad smells*. He then showed how these bad smells can be removed from the code by using refactorings. A very important issue here is automation. Since there are large software systems with more than 1 million lines of code, it is imperative for the detection and removal of bad smells to be done automatically. Important steps have been made towards the automatic detection of bad smells in object-oriented code.

Software metrics ([SSL01]) proved to be a very elegant and easy-to-use tool to detect object-oriented entities that fail to achieve certain goals of good design. For example, a method that accesses members of another class more often than it accesses members of its own class ought to be moved in the other class by means of refactoring. Such a situation could be easily detected if a distance metric between a method and a class was defined as the number of attributes of the class that the method accesses. In [Mar04], results from many measurements can be combined by means of logic operators to detect even more complex design problems. For example, a *god class* (a large class with many methods and very few attributes) is a design problem which can hardly be detected by only one measurement. Thus, a special language is defined where complex detection strategies can be expressed by filtering and combining the results of many measurements.

Logic metaprogramming is another well-known approach towards detecting design flaws ([Ciu99], [TM03]). The code is abstracted as a knowledge base of logic facts and an inference machine based on a logic language (like Prolog, for example) is fed with rules that describe potential design problems. By inspecting the knowledge base with respect to the given rule, the inference machine identifies suspects that verify the rule and therefore, become candidates for refactorings. Among others, such rules could detect classes that *know* about their subclasses ([Ciu99]), which is a common design problem, or, for example, methods that have unused parameters that can be removed ([TM03]).

All these approaches have one thing in common: they rely on models of the source code of the target system and these models contain little information from inside the method bodies. While package/class/method/attribute relationships are fully considered and analyzed, the actual method bodies are taboo zones. Only method/attribute accesses from within a method body are modeled, other kinds of information like control flow within the method body being massively disregarded. This situation is normal since a great deal of bad smells concern relations between software entities regardless of the control logic of the program. However, this situation changes when we want to discover places where certain design patterns ([GHJV95]) could have been fit. In [Ker04], Kerievsky extends the set of bad smells proposed by Fowler with a new set. These new bad smells all describe situations where some design pattern has been ignored and the author shows what refactorings are required in order to introduce the missing design pattern.

A very interesting idea would be to automatically discover situations presented by Kerievsky as problematic and to suggest corresponding refactorings. However, since design patterns usually describe more complex relationships between software entities, such a task is not easily done unless we also consider the full code of the software system under analysis, including the bodies for all methods. Were such information available, we could imagine detection rules for different situations described in [Ker04]. We named these situations *symptoms*, since there can be more than one symptom for each design pattern misuse. Thus, this paper is going to show how simple symptoms of ignored design patterns can be detected in Java programs by means of logic metaprogramming.

The rest of this paper is structured as follows: section II presents the metamodel we used to implement our detection rules, section III describes detection rules for one well-known design pattern (the Composite design pattern), section IV shows some practical results, section V studies some performance issues of the approach, section VI deals with related work and section VII concludes.

II. The JTransformer Framework

As said earlier, the detection rules we propose must be based on a model that covers the entire source code of the project under analysis, including the whole method bodies. JTransformer is a query and transformation engine for Java source code, available as an Eclipse plug-in. JTransformer creates an Abstract Syntax Tree (AST) representation of a Java project as a Prolog database consisting of Program Element Facts (PEFs) ([Kni06]). Once the Java code is translated into a Prolog database, one can use the Prolog inference machine to reason about the Java sources.

A Program Element Fact (PEF) is a statement (a Prolog fact) that tells something about a small aspect of the Java project under analysis. Each PEF represents a node in the abstract syntax tree of the source code. Since the Prolog database is a linear structure that describes an abstract syntax tree, each PEF is given an unique numeric identifier and the numeric identifier of its abstract syntax tree parent.

Because of space limitations, we can't provide a formal definition for PEFs. This is achieved more successfully in [Kni06]. We will rather present some examples that will ease the understanding of the rest of this paper. For example, the following PEF represents a class definition:

```
classDefT(10015, 10006, 'AbstractObject',
              [10024, 10039]).
```

The name of this class is *AbstractObject*, its numeric identifier is 10015 which was the next free identifier at the time the class definition was encountered by the Java parser and 10006 represents the numeric identifier of the abstract syntax tree parent for this class. There should be another PEF in the knowledge base having 10006 as its ID and this PEF should probably represent a package definition since classes are logic descendants of packages in Java programs. The last parameter of this PEF is a list of IDs of entities contained in the *AbstractObject* class. There are only two members in the *AbstractObject* class and by inspecting their respective PEFs (10024 and 10039) one can identify what they are (attributes, methods, internal classes, etc.).

The next PEF represents a member variable of a class:

```
fieldDefT(10024, 10015,
    type(basic, int, 0), 'value', null).
```

The name of the field is *value*, it is not initialized (null), its numeric ID is 10024, its parent ID is 10015 (that makes *value* a member variable in class *AbstractObject*) and its type is given by the compound term *type(basic, int, 0)*. Another possibility for the type would be *type(class, 10015, 2)* which would make *value* a bidimensional array of *AbstractObjects*, because of the 10015 ID which denotes the class *AbstractObject* and the final 2 which specifies the array dimension (0 is used for scalars).

Finally, the following PEF describes a while statement within some method body:

```
whileLoopT(11105, 10924, 10039,
                11106, 11125).
```

The first two parameters represent the ID of the while loop and the ID of its parent (which could be another control structure like a conditional statement or another loop statement that encloses this while), 10039 represents the ID of the method where this while loop is contained (and one can notice that 10039 is also a member of class *AbstractObject*), 11106 represents the ID of the conditional statement guarding the while body while 11125 represents the ID of the while body itself. By further inspecting the PEF with ID = 11125, one can find all the statements contained within the body of this while statement.

Although this short presentation lacks a certain formality, we believe it serves its purpose well, allowing a good understanding of the next section. Once a Prolog metamodel of the Java project is obtained, one can implement different Prolog rules to query the metamodel or even modify it.

III. The Detection Rules

We currently have the possibility to detect a couple of symptoms for five ignored design patterns in Java sources. In this section, because of space limitations, we will only present one of these symptoms together with its detection strategy. The design pattern we will study is the Composite design pattern. Other detection strategies are available for Abstract Factory, Strategy, State and Visitor ([GHJV95]).

A. The Composite Design Pattern

A Composite's intent is to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly. Figure 1 presents the structure of a Composite, as it is described in the original design patterns book ([GHJV95]).

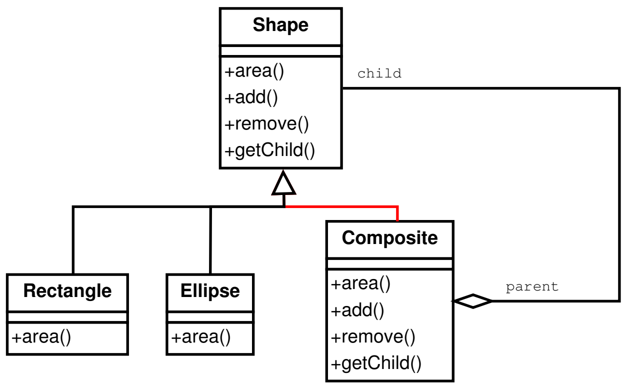

Fig. 1. Structure of a Composite

Ellipse, *Rectangle* and *Composite* are all subclasses of *Shape*, but *Composite* is special in that it contains an array (or another collection type) of objects of type *Shape* (or derived from *Shape*). Each *Shape* can be either an *Ellipse* or a *Rectangle* or a *Composite*, which means that this structure allows us to nest *Composites* and treat them as regular leaves of the structure. An operation applied on the *Composite* object (such as computing the area) is executed by executing it on all its components, regardless if they are *Ellipses*, *Rectangles* or other *Composites*. Thus, a Composite design pattern indeed provides uniform treatment for objects and compositions of objects.

Problems arise when *Composite* is not a subclass of *Shape* (if the red line in figure 1 is missing). Be that due to lack of attention or ignorance of the Composite design pattern, the structure loses the elegance of nesting complex composite objects into one another and treating simple and composite objects uniformly, although it is still usable for working with collections of simple objects (ellipses and rectangles). The flexibility provided by a Composite may not even be needed, but what if it is? Such a situation would present itself like in figure 2:

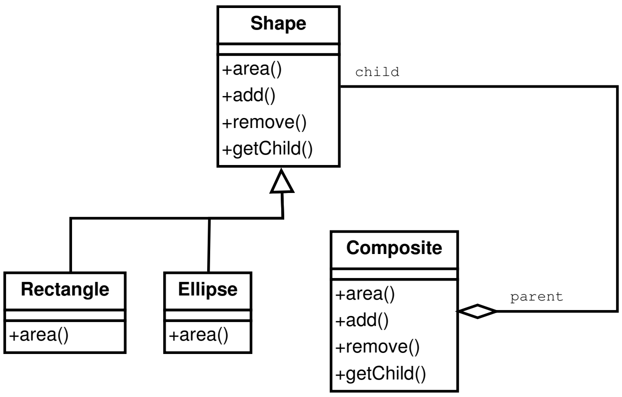

Fig. 2. A Composite Anti-Pattern

B. Detecting the Anti-Pattern

We could use the infrastructure provided by JTransformer to detect and pinpoint such situations in Java code. In case a Composite is needed, a human operator may thus be given the opportunity to modify the code (manually or automatically) such as to introduce the missing Composite. With only a small amount of code added to the system, flexibility is boosted significantly.

We need to detect two IDs in our Prolog knowledge base, one for the *Shape* class and the other for the *Composite* class. The *Composite* class must have a member variable which represents an array of *Shape* objects. The *Shape* class must have at least one descendant (*Ellipse*, or *Rectangle* in figure 2) and must share an overridden method with this descendant (method *area()* in figure 2). Finally, *Composite* must have a method that contains a loop where it calls *area()* on elements of its array of *Shapes*. If any of these conditions is not true, then probably the structure was not intended to be used as a Composite and it would probably be better to leave it as it is.

To find the two IDs for *Shape* and *Composite* we use (note that in Prolog, all identifiers that start with a capital letter are variables):

```
fieldDefT(FieldID, CompositeID,
  type(class, ShapeID, 1), FieldName, _).
```

Thus, *FieldID* and *FieldName* are the ID and the name of a field which is a member of class *CompositeID* (the ID of the *Composite* class), where the type of the field is *type(class, ShapeID, 1)*, an array of objects of class *ShapeID* (the ID of the *Shape* class). The '_' symbol in Prolog means *anything*. For example, the initial value of the field is not important for the query, that's why the last parameter of the *fieldDefT* PEF is symbolized with an '_'. All these are possible with a single query, because the JTransformer model contains (among others) all the fields in the entire project and they are accessible by using the *fieldDefT* PEF.

Next, the class with *ShapeID* (the *Shape* class) must have at least one descendant. This is accomplished by using the *extendsT* PEF:

```
extendsT(ChildID, ShapeID).
```

The *ChildID* variable can match either the ID of an ellipse or the ID of a rectangle.

Composite must not be a descendant of *Shape*. If it were, then we would detect an instance of the Composite design pattern, when in fact we want to detect a Composite anti-pattern, as shown in figure 2. We must define a *descendant* predicate which recursively uses *extendsT* to detect if there is an inheritance chain from *CompositeID* to *ShapeID*. The result is:

```
not(descendant(CompositeID, ShapeID)).
```

Below is the *descendant* predicate, which is defined in a recursive, simple way. A is a descendant of B if it inherits directly from B, or if its superclass is a descendant of B:

```
descendant(A, B) :-
          extendsT(A, B).
descendant(A, B) :-
          extendsT(A, C),
          descendant(C, B).
```

The next step is to find the name (or ID) of a method in *Shape* that the detected subclass of *Shape* overrides.

```
overrides(ShapeID, ChildID, MethodName) :-
  methodDefT(_, ShapeID,
    MethodName, ParamList1, Type, _, _),
  methodDefT(_, ChildID,
    MethodName, ParamList2, Type, _, _),
  params(ParamList1, ParamList2).
```

The first *methodDefT* finds a method *MethodName* in class *ShapeID* (the *Shape* class) having *Type* as the returned type. The second *methodDefT* then checks if there is a method with the same name *MethodName* in class *ChildID* (the child), and the same returned type. The parameter lists may differ between the two. For example, the first parameter in the superclass method may be named *x* and the first parameter in the subclass method may be named *y*. It is their types that matter, that's why we need a new predicate called *params* that checks the type compatibility between two lists of parameters:

```
params([], []).
params([Param1|Rest1], [Param2|Rest2]) :-
  paramDefT(Param1, _, Type, _),
  paramDefT(Param2, _, Type, _),
  params(Rest1, Rest2).
```

We should now return to the *Composite* class for the final step. This class contains an array of *Shapes*. The corresponding field in *Composite* is called *FieldName* and its ID is *FieldID*. The overridden method in the *Shape/Ellipse/Rectangle* hierarchy is called *MethodName*. We need to check if there is a method in *Composite* that calls method *MethodName* on an element of array *FieldName*. If this call is placed inside a loop structure it will become even more suspect. A method invocation in JTransformer is represented by means of the *applyT* PEF ([Kni06]).

```
applyT(InvocationID, _, _,
    ObjectID, MethodName, _, _).
```

InvocationID is the ID of the actual method call, *ObjectID* is the ID of the object on which the method is called and the name of the method should be *MethodName* which we obtained earlier. We have two new variables now: *InvocationID* and *ObjectID*. We should check that the method call (whose ID we have) is placed inside a loop. For that, we define a *descendsFromLoop* predicate:

```
descendsFromLoop(ID) :-
      forLoopT(ID, _, _, _, _, _, _),
      !.

descendsFromLoop(ID) :-
    getTerm(ID, Term),
    arg(2, Term, ParentID),
    descendsFromLoop(ParentID).
```

We used the following rule here: a PEF descends from a loop if it is a for loop or if its parent descends from a loop. To find the parent of a PEF, we used the known fact that the second argument of each PEF represents the ID of the parent of that PEF. Of course, there are also while loops and do ... while loops in Java, but that would be trivial yet space-consuming to add, so we decided to leave them out in this example. Normally, there would be *descendsFromLoop* clauses above that would treat also while loops and do ... while loops, but they are similar with the first *descendsFromLoop* clause, which treats for loops. The last thing to do is to check if *ObjectID* represents in fact an indexed version of the field *FieldID*. This is done in JTransformer by using the *indexedT* PEF ([Kni06]):

```
indexedT(ObjectID, _, _, _, FieldID).
```

When we put everything together, we come up with a detection rule that successfully detects the problematic situation described:

```
suspect(CompositeID, ShapeID) :-
  fieldDefT(FieldID, CompositeID,
    type(class, ShapeID, 1),
    FieldName, _),
  extendsT(ChildID, ShapeID),
  not(descendant(CompositeID, ShapeID)),
  overrides(ShapeID, ChildID, MethodName),
  applyT(InvocationID, _, _,
    ObjectID, MethodName, _, _),
  descendsFromLoop(InvocationID),
  indexedT(ObjectID, _, _, _, FieldID).
```

Fig. 3. The Detection Strategy

The *suspect* predicate returns the two IDs involved: the ID of the *Composite* class and the ID of the *Shape* class from figure 2. These are typically sufficient for a human operator to study if a Composite solution is indeed needed and act upon it by making Composite a subclass of Shape.

This is only one symptom of Composite misuse. There may be many others and detection rules could be written for them too. We believe this is the best way to handle the problem, because a general panacea for all the symptoms at once may be impossible to find.

IV. Practical Results

To evaluate our approach, we've chosen 2 Java projects freely available on the Internet: JHotDraw and BranchView.

JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed as a *design exercise*. Its design relies heavily on some well-known design patterns. JHotDraw's original authors have been Erich Gamma and Thomas Eggenschwiler ([jho]). Version 5.3 that we used for our evaluation contains about 150 classes and interfaces (we used an older version of this software because newer versions have less chances of containing the antipatterns we are searching).

On the other hand, BranchView is a 100% pure Java program that offers a graphical view of selected depot files with all available information about file revisions. It is written by Andrei Loskutov and contains about 60 classes. In both cases, the projects had to be opened in Eclipse since JTransformer works as an Eclipse plugin. The JTransformer engine then generated a knowledge base containing program element facts (PEFs) for each of the two Java projects. Figure 4 shows the size of each knowledge base and the number of seconds required to build it.

Even though BranchView is smaller than JHotDraw in terms of number of classes, the generated knowledge base is larger, which means the classes in BranchView are bigger than the ones in JHotDraw. Without anticipating, this looks like a bad

Projects	Classes	KB Build Time	KB Size
JHotDraw	~150	4-5 seconds	~5.1M
BranchView	~60	4-5 seconds	~5.3M

Fig. 4. Test Subjects

smell, a proof of an imperfect design, since classes should be as small as possible and should limit their implementation to their responsabilities. Next, we were interested in running our analysis on the knowledge bases generated by JTransformer. As we suspected, running the *suspect* rule on the JHotDraw sources ended up with no suspects at all. The Prolog inference machine responded with a NO to our queries, after it spent some time inspecting the knowledge base. However, the analysis of BranchView was more successful, and one suspect was found. After manually inspecting the code, the Composite anti-pattern we found turned out to be a positive suspect. Indeed, the design would have been better if a Composite design pattern had been used there. Running the detection strategy on JHotDraw took 345 seconds, and running it on BranchView took 232 seconds. Unfortunately, there are no similar approaches available to our knowledge to test these results against.

Even though the symptom presented in this paper seems unlikely to appear in code written by professionals, we believe it is helpful to have a tool capable of testing such a symptom. Even in code written by professionals it is still possible for the tested symptom to be found, since it might be the case that the versatility provided by the Composite design pattern was not desired in the first place, but later on, one could conclude that it was not bad to have, either.

V. Performance Issues

The results were obtained on a Pentium III notebook with 512 megabytes of memory running Windows XP. We used a slower machine on purpose, to see if the results are still obtained in reasonable time. Should the projects under analysis have been bigger, the amount of time required to complete the analyses would have grown. The purpose of this section is to find a theoretical relation between the size of the project under analysis and the time required to complete the analysis.

In order to perform the analysis, we must refer to the Prolog rule in figure 3. The great thing about Prolog rules is that the Prolog inference machine never walks past a predicate that fails. It always backtracks to find the closest point in the inference process where there have been alternatives to the chosen path. There, the next possible alternative is chosen and the process continues. Our strategy begins by finding a field *FieldName* which is member of a class with ID = *CompositeID* and whose type is an array of *ShapeID*. Prolog walks through all the facts in the knowledge base and only continues the analysis when it finds one that matches these criteria. Once such a field is found, a descendant of *ShapeID* (called *ChildID*) is located very fast because there can't be many direct descendants of *ShapeID*. *ChildID* has to override a method of *ShapeID* and detecting such a method is another straightforward process: for each method in *ShapeID*, we verify if a method with the same name also belongs to *ChildID* and if so, we check that the parameter lists for the two methods contain matching types. This process is linear with the number of methods in *ShapeID*. Next, we have to check all calls to this method localized in *CompositeID* and verify if one of them is enclosed within a loop (a for loop, a while loop, etc.). This process is linear with the number of calls of the target method in *CompositeID*.

Therefore, the performance of the whole detection strategy depends on the following three factors:

- on the number F of fields in the whole project
- for each field that is an array of type *ShapeID*, on the number M of methods in class *ShapeID*;
- for each method, on the number C of times that method is called within the *CompositeID* class

Being so difficult to give an exact performance function, we choose to express the performance of the *suspect* detection strategy as a product: F * M * C. It certainly is not an exponential analysis, which is encouraging for extending it to bigger projects.

VI. Related Work

The field of detecting *design anti-patterns* is quite new in today's software engineering. This is probably due to the following reasons:

- such a detection typically requires deep analysis of code within method bodies, which is not easily achieved because of metamodel limitations - the solution we chose (the JTransformer framework) seems quite solid in this respect
- analyses are much more complicated than other analyses that check for simple bad smells in code, which may greatly affect scalability; this aspect is something that we, too, have to study in greater detail (see future work)
- the heuristic nature of these analyses is much more evident; results are often subject to interpretation and rejection by human operators

[JLB02] presents an approach where Java programs are analyzed to find places where the Abstract Factory design pattern could have been used. The approach is based on a Prolog metamodel (much simpler than the one offered by JTransformer) and the possibility to use Abstract Factory appears by analyzing at least two versions of the system. If a client creates a set of objects in one version of the system and it creates another set of objects in the second version such that each object in the first set is the *brother* of an object in the second set, an Abstract Factory pattern is suggested as a better solution. However, the results are not clearly pointed out and the work was not continued.

In [Jeb04], the same problem is tackled. However, the set of instantiated objects in a method is more accurately computed, by using the control graph of the method. If two objects are created along different branches of a conditional statement, they won't be part of the same set of instantiated objects.

VII. Conclusions and Future Work

In this paper, we presented how logic metaprogramming can help in detecting simple symptoms of ignored design patterns and developed a detection strategy to capture such a symptom in Java programs. The symptom is rather simple (a missing *extends* statement) but we believe the whole scenario and the underlying tests involved are by no means trivial. However, the elegant use of Prolog language simplifies much of the inherent complexity of dealing with program structures and allows a straightforward and logic specification of the problem.

Our tool is currently able to process about 1-2 symptoms for each one of the following five design patterns: Composite, Abstract Factory, Strategy, State and Visitor. We plan to study how it can be extended to deal with others, too. The analysis of symptoms works well on small to medium-scale projects and provides answers in decent amounts of time. However, it is the backtracking aspect of Prolog that could be a problem on large-scale projects. We plan to study this aspect as future work.

We also plan to extend analyses to a language-independent level. For that, we are currently working on language independent models for programs, using the same declarative language as a base language ([CJM08], [JCM08]).

References

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of the Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1999.

[CJM08] Ciprian-Bogdan Chirilă, Călin Jebelean, and Anca Măduţa. Towards automatic generation and regeneration of logic representation for object-oriented programming languages. In *Proceedings of the International Conference on Technical Informatics*, University Politehnica Timişoara, 2008.

[Fow00] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[JCM08] Călin Jebelean, Ciprian-Bogdan Chirilă, and Anca Măduţa. Generating logic based representations for programs. In *Proceedings of the 2008 IEEE 4th International Conference on Intelligent Computer Communication and Processing*, pages 145–151, Cluj-Napoca, Romania, 2008.

[Jeb04] Călin Jebelean. Automatic detection of missing abstract-factory design pattern in object-oriented code. In *Proceedings of the International Conference on Technical Informatics*, Politehnica University in Timişoara, 2004.

[jho] Jhotdraw as open source project. www.jhotdraw.org.

[JLB02] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, 2002.

[Ker04] Joshua Kerievsky. *Refactoring to patterns*. Addison-Wesley, 2004.

[Kni06] Günter Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006.

[Mar04] Radu Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society, 2004.

[SSL01] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Conference on Software Maintainance and Reengineering*, pages 30 – 38, 2001.

[TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintainance and Reengineering*, pages 91–100. IEEE Computer Society, 2003.