BACHELOR THESIS

On Recognizing Design Patterns with Crocopat

Arend v. Reinersdorff

University of Freiburg
Department of Computer Science
Chair of Software Engineering

Reviewers:
Prof. Andreas Podelski
Prof. Peter Thiemann

February 9, 2009

Declaration

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, February 9th 2009
place, date

Signature

Acknowledgements

I am very grateful to Martin Schäf from the Chair of Software Engineering for his patience and good advice.
And I want to thank my wife Denisa for her support and encouragement.

Table of Contents

Abstract

The program Crocopat was created - among other things - to find design patterns. But it has its own input format and cannot read Java source code directly. So it is hard to determine how good the program really is in finding design patterns.
The goal of this paper is to create a tool, which converts Java projects into Crocopat's input format. Then different design patterns shall be tested to evaluate how well searching for design patterns with Crocopat really works.

Zusammenfassung auf Deutsch

Das Programm Crocopat von Dirk Beyer soll unter anderem Design Patterns erkennen. Das Programm arbeitet mit einem eigenen Eingabeformat und lässt sich daher nicht direkt auf Quellcode anwenden. Daher ist es schwer abzuschätzen, wie gut das Programm beim Suchen nach Design Patterns tatsächlich ist. Im Rahmen dieser These wurde daher ein Programm geschrieben (java2rsf), das Java Quelltext in das Eingabeformat von Crocopat umwandelt.

Das Durchsuchen eines Projekts nach Design Patterns kann Programmierern helfen, die Struktur des Projektes zu verstehen. Insbesondere, wenn das Projekt sehr umfangreich und nicht ausreichend dokumentiert ist. Außerdem kann man anhand von Anzahl und Art der gefundenen Design Patterns Aussagen über die Qualität des Quelltextes machen.

Es wurden vier Java Projekte nach Design Patterns durchsucht. Dabei zeigte sich, dass Crocopat für diese Aufgabe sehr gut geeignet ist. Eine große Anzahl von Design Patterns konnte gefunden werden. Und durch die Weiterentwicklung von java2rsf und den Crocopat-Suchalgorithmen könnte sich die Anzahl der gefundenen Design Patterns noch erhöhen.

Allerdings wird es nie möglich sein, sämtliche Design Patterns in sämtlichen Java Projekten zu finden, dafür sind die meisten Design Patterns zu variabel definiert. Durch kontinuierliche Verbesserung der Suche sollte es allerdings möglich sein, sich einer 90%igen Lösung anzunähren. Also einem Verfahren, das in 90% aller Java Projekte 90% aller Design Patterns finden kann.

Motivation

Definition and Uses of Design Patterns

A design pattern is a solution to an often-occurring problem in software design. A design pattern gives a description of the design problem and lays out how to implement a solution. The layout of the solution describes the classes needed to solve the design problem and the relationship between these classes.

A design pattern also has a name. It is important that the name of the pattern is well known and well defined. This makes it possible to find information about the pattern and to discuss it among programmers. The design patterns and their names defined by Gamma et al. in “Design Patterns : Elements of Reusable Object-Oriented Software” [Patterns] can be considered the standard body of patterns.

Design patterns work at a certain level of abstraction. A particular algorithm to solve a computational problem, for example the Euclidian algorithm to find the greatest common divisor, is not a design pattern. Instead, a design pattern describes the design of classes and the communication between them.

A design pattern describes the general solution to a general problem. For a specific program this solution has to be implemented in the context of program. While the idea of the design pattern is reused, the code is not. So a software library is not a design pattern.

Advantages of Design Patterns

If a design pattern is applicable to a problem, it should be used instead of a new solution created from scratch.
Firstly, it makes the solving of the underlying design problem easier. It removes the need to create a new solution from the ground up and leaves only the implementation to be done.
Secondly, a design pattern is likely to be of much better quality than a solution created from scratch. The standard design patterns have been used in thousands of programs. They have been reviewed by experts and their strengths and weakness are well known.
Additionally, a design pattern is easier to document. A new programmer only needs to know what design pattern was used and how it was implemented. There’s no need to explain the solution to the design problem in detail. If the new programmer doesn’t know the design pattern he can look it up easily.

Finding Design Patterns in Existing Code

Most work on design patterns is about using design patterns when writing software, most notably “Design Patterns” by Gamma and others, and for Java “Head First Design Patterns” by Freeman. But finding design patterns in existing source code is also very useful.

A very common task in software engineering is **program comprehension.** This is needed when a programmer has to start work on an existing software project that he is not familiar with. In order to work on the software project effectively, he must first understand its design. This is especially difficult for big projects with hundreds of classes. But even for relatively small software projects, experience has shown that it is much harder to read (and understand) code than to write it. [Spol]

Searching an existing software project for design pattern can help program comprehension by finding structures in the code and making the design of the software project more transparent. A mapping of design patterns can give an overview over the design of a software project and provide a basis for understanding it in detail. [Bey]

The relatively new research area of **architecture recovery** is the academic continuation of program comprehension. In “A Framework for Software Architecture Recovery” Wolfgang Eixelsberger and others defined it as “a process of identifying and extracting higher level abstractions from existing software systems”. Architecture recovery deals mainly with big legacy software. Here the understanding of the software can often be even more difficult because the original authors of the software are not available anymore.

Finding design patterns in existing software also helps in case of **bad or incomplete documentation.** Under ideal circumstances, software should have sufficient documentation for both the user and future maintainers of the software. In practise this is rarely the case. Due to time pressure, lack of resources or dislike of the programmers for writing it, software documentation is often insufficient or missing completely.
Extreme Programming and other lightweight programming methodologies often encourage programmers to keep documentation to a bare minimum. [Svet]
If good documentation exists, it usually explains how to use the software and not how it was designed. For example Java has built-in support for Javadoc documentation. But while Javadoc makes it easy to document single classes and methods, it doesn’t encourage documentation of the overall software design. This results in most Java software having at least basic documentation for classes and methods, but no documentation for the design.
Even when good software documentation exists, it might be outdated if it hasn’t been updated after the latest changes in the code. Searching the code for design patterns on the other hand can be done automatically and does not require manual updating by the programmer.
Notable work in this area includes Guo et al. “A Software Architecture Recovery Method” [Guo] which describes a semi-automatic process to recover the software architecture of a project with an emphasis on checking if the architecture conforms to the documentation. In “Linux as a Case Study: Its Extracted Software Architecture” [Bowm] Bowman and others use automated tools and manual input to recover the actual architecture of the Linux kernel and compare it to its expected architecture.

Searching a software project for design patterns can also be used to assess the **quality of the code.** [Bey] On the most basic level the number of patterns found is an indicator for how well the code is structured. If very few patterns can be found in a large software project, the project is likely to be badly structured, hard to understand and hard to work with.
A more sophisticated approach would be to check if the patterns found are appropriate for the project. And if the patterns found coincide with the design specified in the documentation.
Last but not least the code can be searched for anti-patterns, which are software designs known to cause problems.

Crocopat

Crocopat [Croc] was written in 2003 by Dirk Beyer, who is now assistant professor of computing science at Simon Fraser University in Vancouver, Canada. It is freely available under the LGPL. Crocopat was created to analyse graph models of software projects and find patterns in these graphs. Crocopat can not only analyse and manipulate graphs but also relations of arbitrary arity. Searching for design patterns is one of the goals of Crocopat.

Internally Crocopat uses binary decision diagrams (BDD) to represent relations. This allows Crocopat to be fast in the use of time and efficient in the use of memory. In a comparison between Crocopat and other pattern recognition tools for C++, Crocopat came out ahead both in speed and the number of patterns found. [Fulo]

For both input and output Crocopat uses files in the Rigi Standard Format (RSF). RSF is a very simple text format that allows one tupel per line.
For example:

```
CLASS Class1
CLASS Class2
METHOD      method1
METHOD      method2
HAS   Class1      method1
HAS   Class2      method2
CALLS Class1      method1     Class2      method2
```

is an RSF file indicating that two classes, `Class1` and `Class2`, and two methods, `method1` and `method2`, exist, that `method1` is part of `Class1`, that `method2` is part of `Class2` and that `method1` in `Class1` calls `method2` in `Class2`.
The simplicity of RSF is a great advantage. It means that it is very easy to create input for Crocopat with other tools and to process Crocopat output from other tools.

To manipulate relations, Crocopat uses programs written in the Relation Manipulation Language (RML). RML is based on predicate calculus and features among other things forall ($\forall$), exists ($\exists$) and transitive closure operators.
In addition to operations on relations, RML also has elements of imperative programming such as WHILE, IF and FOR statements. There are also expressions for numeric calculations and regular expressions for dealing with strings.

For example:
```
Rec(c, m) := CALLS(c, m, c, m);
PRINT ["RECURSIVE_METHOD"] Rec(c, m);
```

is a program that searches for and outputs (directly) recursive methods. This program works with input as in the above RSF example.

For more information about RML see the Crocopat manual. [CrocM]

Crocopat works with relations and Java is an imperative programming language. This difference means that many parts of Java programs cannot be analysed conveniently by Crocopat. In particular Crocopat would be a very bad fit for analysing the code in Java methods. Statements in methods must be executed in order and to preserve the order of statements in the relational RSF input would be very difficult and verbose.

Fortunately, design patterns deal for the most part with the high-level design of classes and their relationships, not the concrete code in methods. And for this level of abstraction Crocopat is a perfect fit. A class having fields of a certain type or a method calling another method are relationships that are very easy to express with relations.

There are **other tools** that were created to find design patterns. Dietrich and Elgar created the “Web of Pattern Project”, an Eclipse plugin that searches Java source code for design patterns. Their pattern definitions are written in the Web Ontology Language (OWL) as described in their paper “A formal description of design patterns using OWL” [Diet].
In “Pattern-based Software Architecture Recovery” [Sart] Sartipi and Kontogiannis describe and test a program that finds design patterns in existing code. Like Crocopat they use relational graphs to represent the software components and their relations.

Parsing Java Code

Crocopat cannot read Java code. Crocopat's relational input format RSF is quite different from normal Java source code.

To use Crocopat on Java projects, a tool is needed that parses Java source code and creates Crocopat input format from it. More specifically, a tool is needed that creates an abstract syntax tree for existing Java source files.

There are a variety of tools that can do this. Some of them are more suited to this task than others. Common to all tools listed here is that they are freely available.

Eclipse JDT Core

JDT Core [JdtC] (package `org.eclipse.jdt.core`) is part of Eclipse, one of the most popular Java development environments. JDT Core is the best choice for parsing Java code and working with the resulting abstract syntax tree. Although for this paper JDT Internal was used.

JDT Core handles the Java-specific features of the Eclipse platform, such as syntax highlighting and as-you-type error reporting. As Eclipse is a very mature and widely used development environment, JDT Core is very reliable.
Also, there is very good documentation available in the form of tutorials and well-commented Javadocs. JDT Core is actively maintained, so any bugs can be expected to be fixed. And there is an active forum for users of the API (application programming interface) where questions are answered.

Its only drawback is that JDT Core can only be used as part of an Eclipse application. JDT Core relies on other features of the Eclipse platform, for example file system management, and does not work outside the context of the Eclipse runtime environment. So every program that uses JDT Core must bear the overhead of a full Eclipse runtime environment. It is not possible to use JDT Core as a library from normal Java code.
This is in principle not a problem for this paper, as Eclipse applications can be run as command-line programs.
The reason JDT Internal was used for this paper instead of JDT Core was a broken Eclipse installation on my computer that prevented me from creating working Eclipse applications. And I only discovered the problem, when my JDT Internal based program was nearly complete.

Eclipse JDT Internal

JDT Internal is also part of Eclipse (package `org.eclipse.jdt.internal`). But it is much more difficult to use than JDT Core.

While JDT Internal is as reliable and actively maintained as JDT Core, there is no documentation. Javadocs must be built from the sources and they are only sparsely commented.
Its only upside is that JDT Internal is not dependent on the Eclipse runtime environment. It can be used as a library from any Java code. Programs that use JDT Internal are for example the Eclipse Batch Compiler [Bat], the Spoon project mentioned below and Google GWT [Gwt], a Java-to-Javascript compiler.

JDT Internal should be avoided in favour of JDT Core whenever possible. For this paper JDT Internal was used only because of a faulty Eclipse installation.

Spoon

Spoon [Spoon] was written in 2007 at the *Institut National de Recherche en Informatique et en Automatique* in France. It's an open source program that creates an abstract syntax tree from Java code and it is fairly easy to use.

Internally, Spoon uses JDT Internal but hides this from the user behind a very clean API for accessing the abstract syntax tree. There is good documentation and even a forum for users of Spoon.

Spoon is a relatively new program and suffers from various small bugs. For example, it doesn't fully support Inner Classes ([Jls] Chapter 8) in Java. Unfortunately, Spoon is no longer actively maintained so these bugs will in all likelihood never be fixed.

With 5 more years of maintenance, Spoon could be a good, easy to use solution for parsing Java code. At the moment, it sadly looks like a dead end.

Uni Freiburg Internal Tool

The chair of software engineering at the University Freiburg has developed an internal tool that parses Java code for static analysis. [Stal]

This internal tool looks very promising so far. It is under active development and the developers are available to me for direct support.

Unfortunately, it is in an early stage of development. There is no public release nor any documentation and it has not been widely tested. So for the moment, it cannot be considered sufficiently reliable.

Sun's Java Compiler API

Starting with JDK 1.6, Sun's Java compiler comes with an API to access the abstract syntax tree of Java source files. [SunC]

The Java Compiler API is based on JSR-199 and is part of Sun's Java compiler. This means it is officially supported, standard based and eliminates the need for a third-party library.
The major disadvantage is that Sun's Java Compiler API doesn't resolve type bindings. For example in the statement

```
      File myFile = new File("test.txt");
```

`File` might be of type `java.io.File` or `com.example.File`, depending on the imports in the source code. But Sun's Java Compiler API cannot tell the difference. Also this is a fairly new feature and besides the Javadocs there's no documentation available.

JML Parser

The Java Modeling Language was developed to specify and check pre and post conditions in Java. JML is a very mature project, but the latest stable build has not been updated for Java 5.

The API for accessing the JML parser is part of a development build (JML3Core) [Jml] and not nearly as reliable as JML itself. To date (February 2009) there are several dependencies on a specific, outdated version of the Eclipse runtime. And as this is a development build, there is no documentation available and it can only be downloaded as source code.

Conclusion

Of these Java parsers, Eclipse JDT Core is clearly the best choice, despite the fact that it cannot be used as a simple library due to its need to run inside the Eclipse runtime.
Eclipse JDT Internal does not depend on the Eclipse runtime. But it is an obscure and little used API that is very difficult to use.
Sun’s Java Compiler API is usable for small tasks, but is too limited for more sophisticated needs.

Crocopat Input Format

To search Java projects for design patterns, the Java source code must first be converted to RSF. For this purpose I wrote as part of this thesis the java2rsf tool (on the attached CD). Java2rsf parses Java source code and outputs RSF data. See Appendix B for a user guide to java2rsf.

Because Java is an imperative programming language and Crocopat's input format RSF is relational, it is not possible to translate the whole meaning of Java source code to RSF. Nevertheless, as much information as possible should be included in the RSF output. At the moment not all information that can be extracted to RSF is used to search for design patterns. But in the future that might change.
And should performance be an issue in Crocopat processing, it is very easy to filter unneeded information from RSF files.

It is worth noting, that Crocopat includes various examples of RSF files. Among them is an RSF file representing the Java API version 1.4.2. However, this example is very basic and uses only three relations CALL, CONTAINMENT and INHERITANCE.

Obvious Relations

An obvious starting point is to create a relation for the existence of basic building blocks of Java projects. So for each package, type, method and field a line is added to the RSF output denoting its existence:

```
PACKAGE      packageName
CLASS ClassName
METHOD       methodName
FIELD fieldName
```

There are different relations for the different Java types, i.e. CLASS, INTERFACE, ENUM and ANNOTATION_TYPE. There is also a separate relation CONSTRUCTOR for constructors. For a complete reference of the java2rsf output format see Appendix A.

The basic building blocks of Java projects have a clear hierarchy. Packages can contain types, which can contain methods and fields. This hierarchy is expressed in relations:

```
CONTAINS     packageName typeName
CONTAINS     typeName    nestedTypeName
HAS    typeName    fieldName
HAS    typeName    methodName
```

The last category of obvious relations are the attributes of basic building blocks of Java. Such attributes are for example the (return-) type of fields and methods, modifiers (public, static, etc.), annotations and superclasses:

```
IS_OF_TYPE  fieldOrMethodName typeName
IS    fieldOrMethodOrTypeName public
IS    fieldOrMethodOrTypeName static
HAS_ANNOTATION    methodOrTypeName  annotationTypeName
```

Additional Relations

Four additional relations are included in the RSF output, for method calls, reading and writing of fields and general use of types:

```
CALLS methodName1 methodName2
READS methodName  fieldName
WRITES      methodName  fieldName
USES  methodOrTypeName  typeName
```

Method calls and field access can only take place in a method body. But use of a type can take place in a method body (variable type, parameter type, cast, etc.) as well as in a class body (annotation, type of field, etc.). The use of primitive types (int, char, boolean, etc.) is not included.

More information could probably be extracted, but these four relations have three advantages:

1. **Ease of extraction**. They are easy to extract from the abstract syntax tree.
2. **Completeness**. All instances of method calls, field accesses and type uses can be extracted and one can be confident that none is missed. The exception to this completeness is the use of reflection, which is not detected.
3. **Relevance**. These relations are very relevant to determining the relationship between different classes of a Java project and as such to the search for design patterns.
   For example, it would be easy to extract instances of nested loops from method bodies. But nested loops are not related to design patterns.

Notable Omissions

As stated above, large parts of Java code cannot be converted to RSF. For the most part, the omitted parts of the Java language are not very relevant to searching for design patterns or would be very hard to extract. Two notable exceptions are method parameters and generics, which are omitted at the moment but are likely to be included in a future version of java2rsf.

The main problem is that both method parameters and generics depend on the order of parameters:

```
void method1(int x, float y);
void method2(float y, int x);
void method3(int x, float y, int z);
Map<String, Integer> map1;
Map<Integer, String> map2;
```

It is not enough to state that a method has an int and a float parameter. The order of the parameters matters and so does the number of parameters of the same type. Similarly, it is not enough to state that a Map has type parameters String and Integer. The order of the parameters matters here as well. Describing order and multiple occurrences with relations is a difficult and verbose task, so for this first version of java2rsf method parameters and generics are not supported.

Naming Conventions

For analysis Crocopat needs unique names for packages, types, fields and methods, the basic building blocks of Java. The previously mentioned RSF example for the Java API version 1.4.2 that is included in Crocopat uses the short name of a type and attaches a number to it, for example ASCII_61899 and Object_41710.
This ensures unique names, but they are hard to work with. For someone reading the RSF file it is not obvious where to find the source file of ASCII_61899, or whether Object_41710 refers to a field, a method parameter or to java.lang.Object.

Fortunately, Java makes it easy to give its basic building blocks unique names that are easily understood by someone reading the RSF file. For packages and types, their fully qualified name is used. For fields, the fully qualified name of the enclosing class plus the field's name is used. For methods, the fully qualified name of the enclosing class plus the method's name plus the list of method parameter types is used:

```
      PACKAGE     java.awt
      CLASS java.awt.Dimension
      FIELD java.awt.Dimension#height
      METHOD      java.awt.Dimension#setSize(java.awt.Dimension)
      METHOD      java.awt.Dimension#setSize(double,double)
```

The System.out problem

Consider this simple "Hello World" program in Java:

```
class Hworld{
      public void sayHello(){
            System.out.println("Hello, World!");
      }
}
```

Part of the output that java2rsf creates is:

```
      CALLS Hworld#sayHello() java.io.PrintStream#println(java.lang.String)
      READS Hworld#sayHello() java.lang.System#out
```

That the method reads `java.lang.System#out` is expected. Yet intuitively, one would expect a call to **`java.lang.System.out`**`#println(java.lang.String)`. The call to **`java.io.PrintStream`**`#println(java.lang.String)` is counter-intuitive.

java2rsf only outputs the type of the object on which a method is called. Sometimes it is known at compile-time that the object on which a method is called is a field in another object, like `out` which is a static field in the class `System`. But often it is not known whether or not an object on which a method is called is a field in another object or not. Consider this modified "Hello World" program:

```
class Hworld{
      public void sayHello(){
            PrintStream ps = System.out;
            ps.println("Hello, World!");
      }
}
```

It is still possible to deduce that ps is in fact the same object as System.out, but it is a lot more difficult. Finally, consider another modification:

```
class Hworld{
      public void sayHello(PrintStream ps){
            ps.println("Hello, World!");
      }
}
```

Now it is impossible to determine at compile-time, if `ps` references an object that is a field in another class or not. It depends on what value is used when calling `sayHello`. So noting down calls to `System.out#println` could never catch all those calls. It might still be useful in some cases to note only those calls that can be detected at compile time, but it would result in partial and unreliable data and might mislead someone using the RSF output.

It is easy to overlook that the identity of a concrete object can in some cases be deducted at compile-time, but in most cases it cannot. java2rsf's output models many relations as suggested in Andreas Haufler's paper "Erkennung von Entwurfsmustern". [Hauf] However he too makes the mistake of proposing that method calls to `this` and `super` objects should be noted. Consider:

```
class Hworld{
      public void saySomething(Object o){
            System.out.println("Something is: " + o.toString());
      }
}
```

At compile-time it is not possible to determine if `o` is a reference to `this` or `super`.

Finding Design Patterns

To test if this approach for finding design patterns is viable, it must be tested on real Java projects. For this purpose, three open-source Java projects were selected:

- The Java API, as included version 1.6, update 10, of the JDK for Windows
- Eclipse JDT Core package, as included in version 3.4.1 for Windows [JdtC]
- jEdit 4.2, a text editor for programmers [Jedit]
- Java Mail 1.4.1, an email and messaging framework [Jmail]

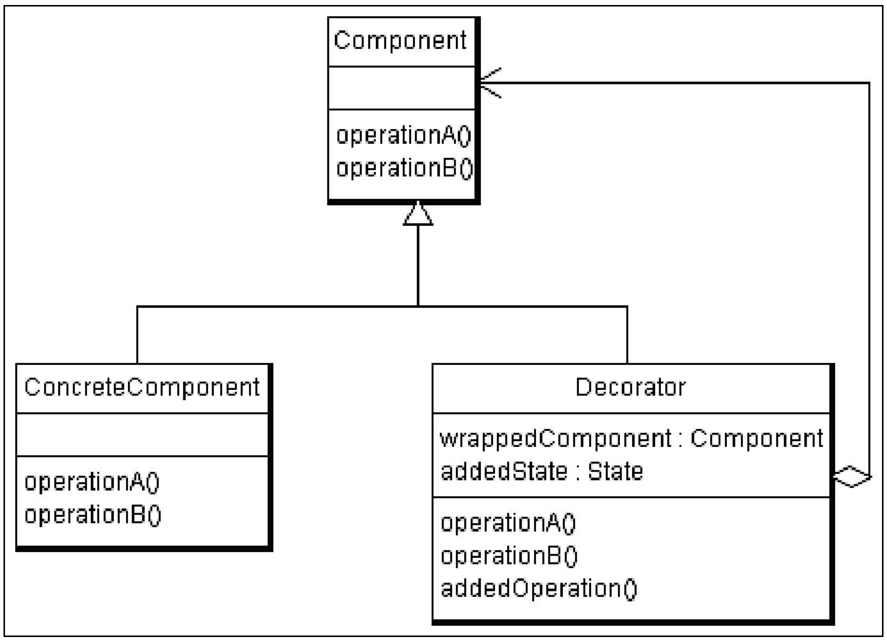
For more information about how to create the RSF output for these projects, see Appendix C.

The number of design patterns found is as follows:

Project	**Size of source**	**Size of RSF output**	**Number of decorator patterns found**	**Number of builder patterns found**	**Number of immutable classes found**
Java library	69.8 MB	165 MB	405	580	51
JDT Core	14.8 MB	54.8 MB	80	20	2
jEedit	5.4 MB	14.3 MB	3	4	2
Java Mail	1.4 MB	2.8 MB	17	2	0

The Decorator Pattern

The decorator pattern allows extending or modifying the behaviour of a class without subclassing it.



Decorator Pattern Class Diagram

The decorator pattern consists of three classes:

- A base class **Component** that defines an API.
- A concrete implementation of the Component's API, **ConcreteComponent**.
- And a **Decorator** class that extends the behaviour of Component objects.

To extend or modify the behaviour of a Component object, the decorator class must implement the same API. But instead of inheriting or re-implementing this API, the decorator class has a field **wrappedComponent** of type Component. The decorator now forwards method calls to the Component API to the object in its wrappedComponent field.

An example of the decorator pattern can be found in the java.io package:

- java.io.InputStream is the base class, it defines the general behaviour of an input stream
- java.io.FileInputStream and java.io.ByteArrayInputStream are concrete implementations of input streams
- java.io.BufferedInputStream and java.io.PushbackInputStream are decorator classes. They support the InputStream API and extend it with additional behaviour.

The decorators, BufferedInputStream and PushbackInputStream do not implement the InputStream API themselves. Instead their constructors receive an InputStream object to which method calls are forwarded.

For more details concerning the decorator pattern see [Patterns] page 175 and [Head page] 91.

Finding the Decorator Pattern with Crocopat

Consider a pair of Java classes, BaseClass and DecoratorClass. For DecoratorClass to be a decorator of BaseClass, two conditions must be met:

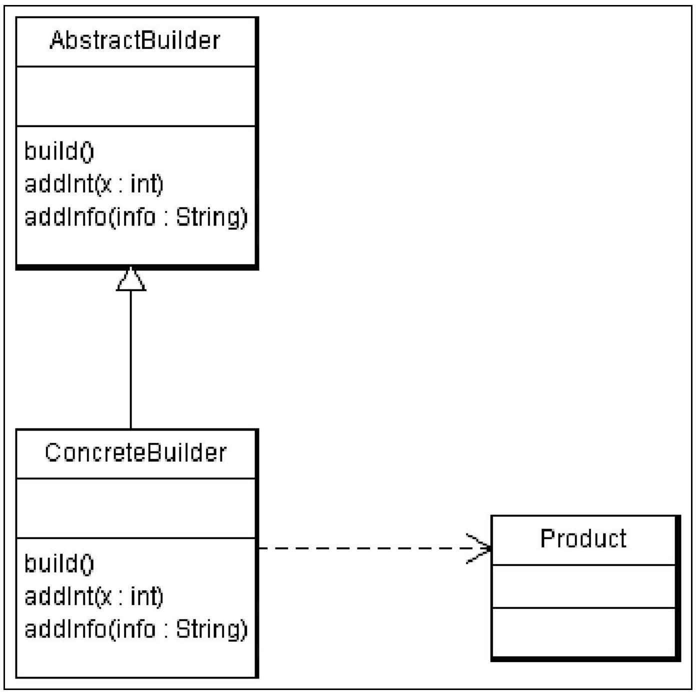
1. DecoratorClass must have a field which has a type that is a subtype of BaseClass
2. DecoratorClass must implement or inherit from BaseClass. Either directly or indirectly.

It is possible to think of additional restrictions. For example that the methods of DecoratorClass should call the methods of its BaseClass field. But the two fore mentioned conditions work very well.
The RML program for the decorator pattern is short and easy to write. The decorator pattern can be found easily and reliably because it is:

- **Unique.** The structure that the decorator pattern imposes on the Java classes is very specific and not very common. If a class has a field of the type of a base class, and it supports the same API as the base class, it is most likely a decorator.
- **Reliable.** The structure that the decorator pattern imposes on the Java classes is the same for all decorators. There is very little room for variation or optional components. The same structure can be expected in every decorator class.
- **Easy to find** with Crocopat. The structure that the decorator pattern imposes on the Java classes is easy to find with Crocopat. The complete information about inheritance and fields is part of the RSF output.

The Builder Pattern

The builder pattern defines an API for the construction of a complex object.
This separates the construction of the complex object from its representation. In this way, the complex object can be assembled step-by-step, as opposed to a single constructor call. With a clearly defined creation API it is also possible to easily change the type of object that is created. It is sufficient to use another concrete builder class that supports the abstract creation API and the same method calls can generate an object of a different type.



Builder Pattern Class Diagram

The builder pattern consists of three classes. The **Product** is the result of the creation process. It is a complex class that cannot easily be created in one step. Often the product is not a single class but a composition of different classes.
The **AbstractBuilder** is an abstract class that defines an API for creating product objects. In most cases the abstract builder defines methods for adding information about the product object that should be produced. And it defines a build() method that returns a product object after all necessary information has been added.
The **ConcreteBuilder** is a concrete class that implements the abstract builder API. There can be more than one concrete builder class. For example an AbstractDocumentBuilder class might have two implementations HtmlDocumentBuilder and RtfDocumentBuilder.

For more information about the builder pattern see [Patterns] page 97 and [Head] page 614.

Often a fourth class, the Director, is named as a part of the builder pattern. See for example [Patterns] page 97. The director is then placed between the client and the builder API and the client makes only a single call to the director. This approach has an advantage if the director has all the information it needs to create a product, for example when the product is created from a file. When the client has to provide the details on how the product is created, a director does not make sense.

Finding the Builder Pattern with Crocopat

Consider three Java classes AbstractBuilder, ConcreteBuilder and Product. For these three classes to be considered a builder pattern the following conditions must be met:

1. ConcreteBuilder must implement or inherit from AbstractBuilder, either directly or indirectly.
2. AbstractBuilder is an interface or an abstract class
3. AbstractBuilder defines a build method that has the return type Product.
4. The build method of ConcreteBuilder must create a new Product object.
5. AbstractBuilder must not be subtype of Product. That would break the separation of Product from its creation.
6. ConcreteBuilder must have at least one field to store its state.
7. The build method is *public* and not *static*.

These conditions search for the very basic structure of a builder pattern: An abstract builder defines a method that returns a product, and a concrete builder implements it. There are additional hints, for example that the concrete builder must have state, that the build method must be *public* but not *static*, and that the abstract builder must not be a subtype of product. But the results are not as good as for the decorator pattern.

The RML program is lengthy and complex, and it outputs many triples of classes that are not builder patterns. The reason for this being that the structure that the builder pattern imposes on the Java classes is:

- **Not unique enough**. A concrete class implementing an abstract one is very common. The restriction that a build method must be defined and that it must return a Product is not strong enough.
- **Not reliable enough**. There are too many variations in which the builder pattern can be implemented. This makes it very hard to find strong restrictions on the pattern structure. Some builders only have one build method and many set-methods that change the internal state and return nothing. Other builders also have get-methods that return default settings. A particularly useful implementation is for the set-methods to return *this* so that chained method calls like this are possible:
  ```
      builder.setX(12).setInfo(“Thirteen”).build();
  ```
  But this is only true of some builders and not others. Often products do not provide public constructors, but this too is not a reliable restriction.
- **Easy to find** with Crocopat. The structures of the builder pattern can be found in the RSF output.

Possible Improvements

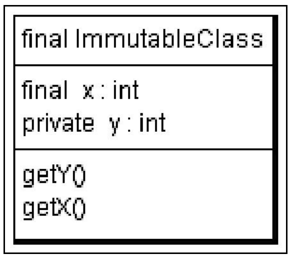
Although the builder pattern is not as easy to find as the decorator pattern, the results can be improved upon. For example by:

- Ensuring that ConcreteBuilder is not a subtype of Product
- Ensuring that Product has at least some fields to store internal state
- Looking for specific types of builders, such as those whose setter methods return *this*.
- Not listing classes that “build” more than one type of product. They are very likely not part of builder patterns.

Immutable Classes

Objects of immutable classes cannot be modified after they are created.

Immutable classes have many advantages. Having only one state makes them invulnerable against accidental modification, reducing the number of possible programming errors. Because immutable objects do not change their state, they can be safely shared between different threads. They are also good keys for hashed collections like java.util.HashMap. Generally, immutable objects can be shared freely and make good building blocks for the internals of other objects. For example the Flyweight Pattern (see [Patterns] page 195) relies on sharing a big number of small objects. This is greatly simplified if the small objects are guaranteed to never change their state.



An Immutable Class

The most prominent examples of immutable classes in Java are java.lang.String and the wrapper classes for primitive types java.lang.Integer, java.lang.Boolean, etc. Other examples include java.math.BigInteger and java.util.Currency.

For more information about immutable classes see [Effec] page 73.

Finding Immutable Classes with Crocopat

For a Java class to be immutable, the following conditions must be met:

1. The class must not allow subclassing. Otherwise a client could subclass it and make the subclass mutable. As subclasses are generally accepted instead of their superclasses, this could result in the immutability being broken.
2. The class must have at least one field to store some state. A class that doesn’t have any state is not considered immutable.
3. The non-static fields of the class must be immutable. They must either be declared “final” or they must be “private” and only be written to from constructors, static initializers or instance initializers of the class.
4. The same must be true for all inherited non-static fields.
5. All non-static fields must be of a primitive type (int, boolean, etc.) or they must be of an immutable type.

The fifth restriction is generally not necessary. Immutable classes may have fields of object or array type as long as these fields are never changed. But it is not possible to check this from the RSF output. In Java there are too many ways to modify object or array types that cannot be found with the current RSF output.
Consider for example java.lang.String, which stores its state internally in a `char[]` array. The String class could pass this array as a parameter to a method call. It could also assign it to

a local variable and then access the array through that variable. Both actions could modify the array and neither can be reliably detected with the current RSF output.

The RML program for finding immutable classes is rather complex compared to the program for finding decorators, but it does a good job of finding the subset of immutable classes, which have non-static fields that are only of primitive types or immutable classes. It does not find immutable classes, which have non-static fields of object or array types.
The reasons that only a subset of immutable classes can be found are:

1. **Unique**. The structure of immutable classes is unique. A class is immutable if and only if it has fields and these fields can never be modified after object creation.
2. **Reliable**. The structure of immutable classes must always be the same. There must not be any modification of the fields, nor any public non-final fields, nor any subclasses.
3. **Hard to find** with Crocopat. Classes with fields of object or array type cannot be verified to be immutable classes with the current RSF output. This is generally a hard problem, because there are many ways in which a Java object can be modified.

Possible Improvements

More immutable objects could be found if additions would be made to the RSF output:

- Note lazy initialisation. Lazily initialised fields are only written to once, but not at the time of object creation, instead they are written to once at the first time the field is used. For example, java.lang.String stores its hashcode in a field that is initialised at the first call of the hashCode() method.
- Track whether method parameters are modified.
- Track whether fields of array of object type used in a method are modified.

Conclusion

Converting Java projects to RSF and searching the RSF for design patterns with Crocopat is a very promising approach. Many design patterns can be found this way, though not all of them. Some patterns - like decorators - work better than others - like builders. But for those patterns that cannot reliably be found, at least a list of probable matches can be created.

If and how easily a design pattern can be found depends mainly on three factors.
A design pattern is easier to find if it is:

- **Unique**. Design patterns impose structures on the Java classes. If these structures appear in many patterns and classes, the design pattern is harder to find. If these structures are rarely used, the design pattern is easier to find.
  Immutable classes have a completely unique structure. A class is immutable if and only if its non-static fields can never be modified after creation.
  Decorators are mostly unique. Most but not all classes that have a field of a type T and are themselves a subtype of type T
  Builders are not sufficiently unique. Many classes that extend an abstract class and have a possible build() method that returns another type are not builders.

- **Reliable**. If the structures, which a design pattern imposes on the Java classes, are the same in every implementation of the pattern, the design pattern is easier to find. If these structures can be a bit different in each implementation, the design pattern is harder to find.
  Decorators and immutable classes are reliable. Every decorator must be a subtype of its decorated type and have a field of its decorated type. No immutable class may have a *public, non-final, non-static* field or allow inheritance.
  Builders are not reliable. A builder may have only set-methods, or it may also have some get-methods to inquire about default settings. It may have one build method or many. It might return *this* from every set-method or not.

- **Easy to find** with Crocopat. Some structures can be found in the current RSF output, but others cannot.
  Decorator structures like inheritance and the type of fields are present in the current RSF output and can be found by Crocopat.
  Builder structures like inheritance and object creation are mostly present in the current RSF output. But there's no easy way to check if a specific method in a class overrides another method in its superclass.
  Structures of immutable classes are only partially present in the current RSF output. While it is possible to check if a field of primitive type is immutable, it is not possible to check if a field of object or array type is immutable.

A 90% solution

Converting Java projects to RSF and searching the RSF output for design patterns with Crocopat is a 90% solution, meaning that this solution will never find 100% of all design patterns in all projects. Nevertheless the solution is good enough for practical use and with steady incremental improvements it might come close to finding 90% of the design patterns in 90% of all projects.

Crocopat cannot read Java input directly. But more importantly, because the RSF format is relational and code in Java methods is imperative, the RSF format is not well suited to represent code inside Java methods. This leaves a gap between what can be expressed in Java and what can be processed in Crocopat.
Nevertheless, this is only a technical problem. And of the three factors that determine how easily a design pattern can be found, "easy to find with Crocopat" is the simplest one to improve. Most in-method code that is relevant to design patterns, like lazy initialisation and passing of fields to other methods as parameters, could eventually be part of the RSF output.

The bigger problem lies with the fact that some design patterns are not unique and reliable enough, because this is the very nature of design patterns.
Design patterns themselves are unique. They are descriptions of solutions for common design problems. These problem descriptions and solutions are different for each pattern and one pattern is clearly distinct from every other, for example the builder pattern is clearly distinguishable from the abstract factory pattern.
But the structures that design patterns impose on the code are not as unique. The implementation of a design pattern depends not only on the structure of the code, but also on how and in what context the code is used. For example the interface java.awt.Shape and its implementations Line, CubicCurve and Ellipse have a method getBounds() that returns a Rectangle which encloses the shape. These classes could certainly be used to create a number of different rectangles. But instead they are mostly used for representing the various shapes, not for creating rectangles. While the structure of these classes is similar to the builder pattern, they are used in a different context, which is very hard to measure.

The structure that design patterns impose on the code is often not reliable. Design patterns don't have formal mathematical definitions, they only have descriptions. Of course it would be easy to redefine the builder pattern more restrictively so that every implementation may only have one build method, a number of set-methods that must return *this* and no get-methods. But this redefinition would not be a useful in practice, as many builder implementations exist that do not conform to it.
Different implementations of design patterns must fulfil different design requirements. So there will always be variations in the implementation.

The fact that many design patterns are not sufficiently unique and reliable is a theoretical problem with its root in the very nature of design patterns. So although it is possible to improve the search process to find more design patterns, there can never be a way to find them all.

The RSF output format

The chosen RSF output format generally works well. There are still obvious parts missing, like method parameters and lazy initialisation. But it is already very useful for finding design patterns.
Two decisions in particular paid off: Completeness and readable names. Even minor information like field modifiers (*pubic*, *static*, etc.) were useful for some patterns. And long readable names in the form `hworld.HelloWorld.main(java.lang.String[])`are a very useful in testing. For every relation that is created, it is immediately clear which class, field or method in which source file is meant. This makes it easy for example to verify if a class in the output really implements the searched design pattern.

Crocopat

Crocopat is very well suited for searching for design patterns. Its simple RSF input and output format and expressive RML programming language lived up to its promises. And it handled big input files (165 MB) well.

Future Work

Directions for future work on this approach in the order of importance:

1. Finding more design patterns. Three patterns can give insight into the viability of this approach. But for further work, RML programs for a greater number of patterns are needed.
2. Making use of the results. The patterns found should be used in programming or design tools. This would give practical feedback about how good the results really are and where improvements are needed.
3. Improving the RSF output. While the current RSF output gives good information, it is clear that it could contain more information.

Appendix A: Output Format of java2rsf

For a Java project or class processed by java2rsf the following RSF output is created:

For Basic Building Blocks

```
PACKAGE     packageName
CLASS typeName
MEMBER_CLASS      typeName
LOCAL_CLASS topLevelTypeName$Index$SimpleName
ANONYMOUS_CLASS   topLevelTypeName$Index
ENUM        typeName
INTERFACE   typeName
ANNOTATION_TYPE   typeName
METHOD      typeName#methodName(typeName1,typeName2, …)
CONSTRUCTOR typeName#<init>(typeName1,typeName2, …)
ANNOTATION_METHOD typeName#methodName()
FIELD typeName#fieldName
ENUM_CONSTANT     typeName#fieldName
STATIC_INITIALIZER      typeName#<staticInit>
INSTANCE_INITIALIZER    typeName#<instanceInit>
```

`CLASS` marks a top-level class and `MEMBER_CLASS` a member class. For enums, interfaces and annotation types no such distinction is made.

For local and anonymous classes, Index is a number greater than zero. Even if local or anonymous classes are declared inside a nested class, only the top level enclosing class is mentioned in their name.

If no constructor is declared, a standard empty argument constructor is provided. This standard constructor is visible in the RSF output.

Initializers are present not only for initializers but also for classes, which have fields with field initilizers.

Names

A `packageName` is the fully qualified name of a package. If no package is declared, `<defaultPackage>` is used.

A `typeName` has the form

```
packageName.EnclosingType1$EnclosingType2$SimpleTypeName
```

where package name is the name of the package. If no package is declared, `packageName.` is omitted. There may be zero or more enclosing types.

For primitive types, the name is the identifier of the primitive type, e.g. `int` or `boolean`. For array types, the name has the format

```
arrayElementType[]
```

for example `int[]` or `java.lang.String[]`

Attributes
```
      IS    typeExecutalbeOrFieldName     modifier
      IS_OF_TYPE  methodOrFieldName typeName
      HAS_ANNOTATION    typeExecutalbeOrFieldName     typeName
```

Where modifier is a java modifier like `private, public, final`, etc.

For methods, `IS_OF_TYPE` is the return type of the method.

Relations
```
      CONTAINS    packageName topLevelTypeName
      CONTAINS    enclosingTypeName enclosedTypeName
      HAS   typeName    executableOrFieldName
      EXTENDS     typeName    typeName
      IMPLEMENTS  typeName    typeName
      THROWS      executableName    typeName
```

Others
```
      READS executableName    fieldName
      WRITES      executableName    fieldName
      CALLS executableName    executableName
      USES  typeOrExecutableName    typeName
```

Appendix B: User Guide for java2rsf

```
Usage: java -jar rsfparser.jar <inputFile> [-r] [-c <classpath>]
[-l <javaVersion>] [-e <encoding>] [-v]
  <inputFile>
        Set the .java file or directory to create Rsf output from.
        For a directory, all .java files in the directory will be added, but
        files in its subdirectories will not be added.

  [-r]
        If the input file is a directory, adds all .java files in its
        sudirectories.

  [-c <classpath>]
        Name a directory or .jar file that contains Java classes that are
        referenced in the input files.
        Can be used multiple times

  [-l <javaVersion>]
        Set the java language version of the .java files, can be 1.1, 1.2, 1.3,
        1.4, 1.5 or 1.6, defaults to 1.6

  [-e <encoding>]
        Set the encoding of the .java files, defaults to the system default
        encoding

  [-v]
        Give feedback about which file is parsed at the moment.
```

Appendix C: Creating the example RSF output

Java API
Sources as included in JDK 1.6, update 10, for Windows.

```
java -Xmx512m -jar java2rsf.jar “c:\src\java api” –r -v -c "C:\Program
Files\Java\jre6\lib\jce.jar" -c "C:\Program Files\Java\jre6\lib\jsse.jar" >
java-api.rsf
```

Where C:\Program Files\Java is the directory into which Java is installed.

The Eclipse JDT parser is a little more restrictive than Sun’s javac (this is probably a bug in javac). So two manual changes must be made to the source files where the Eclipse JDT parser throws an error but javac only throws a warning:

- com\sun\jmx\mbeanserver\OpenConverter.java, line 1169
  add wildcard parameter Constructor<**?**> in line 1167, or cast to ConstructorProperties in line 1169
- java\beans\MetaData.java, line 1389
  add wildcard parameter Constructor<**?**> in line 1388, orcast to ConstructorProperties in line 1389

JDT Core
Sources and libraries as included in Eclipse 3.4.1 for Windows.

```
java -Xmx512m -jar java2rsf.jar "c:\src\jdt core" -r -v -c
org.eclipse.core.contenttype_3.3.0.v20080604-1400.jar -c
org.eclipse.core.filesystem_1.2.0.v20080604-1400.jar -c
org.eclipse.core.jobs_3.4.0.v20080512.jar -c
org.eclipse.core.resources_3.4.1.R34x_v20080902.jar -c
org.eclipse.core.runtime_3.4.0.v20080512.jar -c
org.eclipse.equinox.app_1.1.0.v20080421-2006.jar -c
org.eclipse.equinox.common_3.4.0.v20080421-2006.jar -c
org.eclipse.equinox.preferences_3.2.201.R34x_v20080709.jar -c
org.eclipse.equinox.registry_3.4.0.v20080516-0950.jar -c
org.eclipse.jface.text_3.4.1.r341_v20080827-1100.jar -c
org.eclipse.jface_3.4.1.M20080827-2000.jar -c
org.eclipse.osgi_3.4.2.R34x_v20080826-1230.jar -c
org.eclipse.text_3.4.0.v20080605-1800.jar > jdt-core.rsf
```

Java Mail 1.4.1

```
java -jar java2rsf.jar "c:\src\java mail" -r –v -c "C:\Program
Files\Java\jre6\lib\jsse.jar" > java-mail.rsf
```

jEdit 4.2

```
java -Xmx512m -jar java2rsf.jar c:\src\jedit -r –v -c c:\src\jedit\ant.jar
-c "C:\Program Files\Java\jdk1.6.0_10\lib\tools.jar" -c
c:\src\jedit\xercesImpl.jar > jedit.rsf
```

The ant.jar and xercesImpl.jar libraries needed are from Ant 1.7.

The jedit\jars directory needs to be deleted to create the RSF output.

References

[Bat]	Eclipse JDT Core Batch Compiler http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_api_compile.htm
[Bey]	D. Beyer, C. Lewerentz Crocopat: Efficient Pattern Analysis in Object-Oriented Programs International Workshop on Program Comprehension (IWPC), 2003
[Bowm]	I. Bowman, R. Hold, N. Brewster *Linux as a Case Study: Its Extracted Software Architecture* Proceedings of the International Conference on Software Engineering, 1999
[Croc]	Crocopat Homepage http://www.cs.sfu.ca/~dbeyer/CrocoPat/
[CrocM]	D. Beyer, A. Noack *Crocopat Introduction and Reference Manual* 2004
[Diet]	J. Dietrich, C. Elgar *A formal description of design patterns using OWL* Proceedings of the Australian Software Engineering Conference (ASWEC), 2005 Web of Patterns http://www-ist.massey.ac.nz/wop/
[Effec]	J. Bloch *Effective Java : Second Edition* Addison-Wesley, 2008
[Eixe]	W. Eixelsberger, L. Warholm, R. Klösch, H. Gall, B. Bellay *A Framework for Software Architecture Recovery* 1996
[Fulo]	L. Fülöp, T.Gyovai, R. Ferenc *Evaluating C++ Design Pattern Miner Tools* Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006,
[Guo]	G. Guo, J. Atlee, R. Kazman. *A software architecture reconstruction method* Proceedings of the first Working IFIP Conference on Software Architecture, 1999
[Gwt]	GWT’s use of JDT Internal http://www.google.com/codesearch?hl=en&q=org.eclipse.jdt.internal&exact_package= http%3A%2F%2Fgoogle-web-toolkit.googlecode.com%2Fsvn
[Hauf]	Haufler Erkennung von Entwurfsmustern University of Stuttgart, 2006
[Head]	Eric Freeman, Elisabeth Freeman *Head First Design Patterns* O’Reilly, 2004
[JdtC]	Eclipse JDT main page http://www.eclipse.org/jdt/ JDT Core Newsgroup http://www.eclipse.org/newsportal/thread.php?group=eclipse.tools.jdt Eclipse help about JDT Core http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_int_core.htm
[Jedit]	JEdit Homepage http://www.jedit.org/

[Jls]	J. Gosling, B. Joy, G. Steele, G. Bracha *The Java Language Specification, Third Edition* Addison-Wesley, 2005 http://java.sun.com/docs/books/jls/
[Jmail]	Java Mail Homepage http://java.sun.com/products/javamail/
[Jml]	Java Modeling Language (JML) homepage http://www.eecs.ucf.edu/~leavens/JML/ JML3Core Repository http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/JML3/trunk/JML3Core/
[Patterns]	E. Gamma, R. Helm, R. Johnson, J. Vlissides *Design Patterns : Elements of Reusable Object-Oriented Software* Addison-Wesley, 1995
[Sart]	K. Sartipi, K. Kontogiannis Pattern-based Software Architecture Recovery Proceedings of the Second ASERC Workshop on Software Architecture, 2003
[Spol]	J. Spolsky *Things You Should Never Do, Part I* http://www.joelonsoftware.com/articles/fog0000000069.html 2000
[Spoon]	Homepage of the Spoon project http://spoon.gforge.inria.fr/
[Stal]	D. Dietsch *STALIN: A plugin-based modular framework for program analysis* University of Freiburg, 2008
[SunC]	T. Ball *Hacking javac* http://weblogs.java.net/blog/tball/archive/2006/09/hacking_javac.html 2006 Compiler Tree API http://java.sun.com/javase/6/docs/jdk/api/javac/tree/index.html JSR 199: The Java Compiler API http://jcp.org/en/jsr/detail?id=199 2006
[Svet]	D. Svetinovic, M. Godfrey A Lightweight Architecture Recovery Process 2003