Downgrading Java 5.0 Projects

An approach based on source-code transformations

Tamara Steijger
Abstract

The introduction of Java 5.0 came along with an extension of the language syntax. Several new language features as generic types and enumeration types were added to the language specification. These features cause downward-incompatibilities, code written in Java 5.0 will not work on older versions of the Java runtime environment. For some active projects, however, it is not possible to upgrade to higher Java versions, since some code might not be supported on Java 5.0. If one still wants to use components written in Java 5.0, these must be downgraded. Up to now this has been accomplished mostly by transforming the byte code of these programs.

In this thesis, we present a set of transformations which transform Java 5.0 source code to Java 1.4 compatible code. We successfully apply these transformations to two larger projects and compare our approach to the up to now common byte-code based tools.

Key-words: Source-Code Transformation, Java, Metaprogram, RECODER
# Contents

1 Introduction 1
   1.1 Problem Definition and Motivation 1
   1.2 Related Work 2
   1.3 Goal Criteria 3
   1.4 Contribution 3
   1.5 Overview of the Report 3

2 Background 4
   2.1 Introduction to J2SE(TM) 5.0 new Language Features 4
      2.1.1 Generic Types 4
      2.1.2 Enhanced for Loop 6
      2.1.3 Boxing / Unboxing Conversions 7
      2.1.4 Typesafe Enums 9
      2.1.5 Varargs 11
      2.1.6 Static Import 12
      2.1.7 Annotations (Metadata) 13
      2.1.8 Intersection Types in Conjunction with the Conditional Operator 14
      2.1.9 Covariant Return Types 15
   2.2 State of the Art 15
      2.2.1 Substituting Invocations of the Java 5.0 Runtime Library 15
      2.2.2 Source Code based Approach 16
      2.2.3 Byte Code based Approach 17
   2.3 RECODER 18
      2.3.1 Metaprogramming 18
      2.3.2 Features of RECODER 18

3 Processing 21
   3.1 Design Algorithms 21
   3.2 Providing Test Data 21
   3.3 Finding suitable Project for Testing Issues 21
   3.4 Implementation of Transformations 21
   3.5 Testing 22
      3.5.1 Compiling the transformed Program 22
      3.5.2 Running the transformed Testsuite 22

4 Transformations 23
   4.1 Introduction 23
   4.2 Resolving Generics 23
   4.3 Replacing Enhanced for Loops 32
   4.4 Resolving Boxing Conversions 35
   4.5 Replacing Enums 37
   4.6 Resolving Varargs 41
   4.7 Removing Static Imports 43
   4.8 Removing Annotations 44
   4.9 Making Conditionals compatible 45
   4.10 Removing Covariant Return Types 46
   4.11 Dependencies among the transformations 52
      4.11.1 EnhancedFor2For to MakeConditionalCompatible 52
## 4.11.2 ResolveGenerics to EnhancedFor2For

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Evaluation</td>
<td>54</td>
</tr>
<tr>
<td>5.1 Testing</td>
<td>54</td>
</tr>
<tr>
<td>5.2 Performance</td>
<td>54</td>
</tr>
<tr>
<td>5.3 Comparing our Results to related Work</td>
<td>56</td>
</tr>
<tr>
<td>5.4 Identified Bugs of RECODER</td>
<td>57</td>
</tr>
<tr>
<td>6 Conclusion and Future Work</td>
<td>59</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>59</td>
</tr>
<tr>
<td>6.2 Conclusion</td>
<td>59</td>
</tr>
<tr>
<td>6.3 Future Work</td>
<td>59</td>
</tr>
<tr>
<td>6.3.1 Testing</td>
<td>59</td>
</tr>
<tr>
<td>6.3.2 Performance</td>
<td>60</td>
</tr>
<tr>
<td>6.3.3 Detect and Resolve Dependencies</td>
<td>60</td>
</tr>
<tr>
<td>6.3.4 API Support</td>
<td>60</td>
</tr>
</tbody>
</table>

### A Details on Java Language Features

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Generic Types</td>
<td>63</td>
</tr>
<tr>
<td>A.2 Annotations</td>
<td>65</td>
</tr>
</tbody>
</table>

### B Glossary

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>61</td>
</tr>
<tr>
<td>A</td>
<td>63</td>
</tr>
<tr>
<td>B</td>
<td>68</td>
</tr>
</tbody>
</table>
1 Introduction

With Java 5.0, many new language features, e.g., generic types, were introduced to the Java programming language. While these features are intended to simplify the daily live of the programmers, they introduce downward-incompatibilities: Code written in Java 5.0 will not work on older versions of Java. Further downward-incompatibilities arise due to extension of the Java Runtime Library. Invocations of the newer runtime library will not work on older versions of the Java Runtime Library. For some running systems, however, it is not feasible to upgrade to higher Java versions. If those systems need to interact with applications written in newer versions of Java these need to be downgraded.

1.1 Problem Definition and Motivation

During maintenance the situation can occur where a legacy application (relying on an older Java runtime environment) needs to interact with a third party application written in a newer version of Java. To resolve that conflict either the legacy application needs to be updated to a newer Java version or the application it should interact with needs to be downgraded. The natural way would be to upgrade the legacy application. Still there may be several reasons why running systems cannot be upgraded to the current Java version without major problems:

- A project may rely on third party products that are not supported on 1.5 (persistence engines, UI frameworks, and application servers are common examples).
- Clients may not be able to install the latest version of the JVM, a typical scenario for companies that deploy applets or applications via WebStart or application servers not supporting 1.5, e.g., Websphere 5.x, which is still based on JDK 1.3(!).

If this is the case the applications written in newer Java versions need to be downgraded to enable the legacy application to interact with them.

In previous releases of Java it was possible to specify source and target release at compilation time. But for Java 5.0, javac does not allow you to go ahead with source=1.5 and target<1.5, (this will result in the following error message: "javac: source release 1.5 requires target release 1.5"). So a different way to has to be found how to make Java 5.0 applications Java 1.4.2 compatible.

There exist already some tools to make Java 5.0 code compatible for older versions of the Java runtime environment. Since these are mostly based on byte-code transformations, this thesis will focus on transforming Java 5.0 source code to Java 1.4 compatible source code. One advantage of transforming source code is that it simplifies debugging downgraded code. Debugging a program with transformed byte code can get annoying because the byte code might not correspond to the source code. So the debugger might point to a wrong position in the code while debugging. If the source code gets transformed debugging will work as usual and the debugger will always know "where it is".

Secondly, an important advantage of source code based transformations is the possibility of adapting the code afterwards. This enables developers to adapt the code to make it more suitable for their individual project. Thus, it enables the developers to maintain the transformed code together with its own project, which simplifies maintenance.

1.2 Related Work

Several projects trying to make Java 5.0 code compatible to previous releases exist. Since the problem of downgrading code can be divided into two parts, i.e., resolving new language features and handling calls to the Java 5.0 API, these projects differ in their scope.
Some are dealing with API changes only others focus entirely on the new language features and some tools combine both parts. Those handling new language features can be further divided by their approach, whether they work on byte-code or on source-code level. Most tools resolve the new language features by transforming the byte code of a program and, thus, require to first compile the code with a Java 5.0 compiler and then transform the resulting byte code. We know from one other tool, Declawer, that transforms source code directly, leaving out the compiling step in between. In the following a rough overview of the most important tools will be given.

Retroweaver and Declawer focus exclusively on resolving the new language features. But they differ in their general approach: While Retroweaver is based on byte-code transformations Declawer transforms source code directly. On the other hand, Retroweaver is a much more mature project, while Declawer is more a side product of another project that needed to resolve generic types. Since it gets a lot of attention, it has been extended step by step, but still it only supports some of the language features and also for those the support is limited, e.g., enhanced for-loops (described in Section 2.1.2) are supported but nested enhanced for-loops will probably cause troubles.

Backport-util-concurrent only focuses on simulating the Java 5.0 Runtime Library. It offers a Java 1.4 compatible library for the java.util.concurrent package of Java 5.0 Runtime Library.

Retrotranslator combines both parts. It handles the new Java language features by transforming byte-code and calls to the Java 5.0 Runtime Library by offering backports for most of the changes in the Java 5.0 Runtime Library. It also includes the backport-util-concurrent project.

These are not the only projects dealing with downgrading Java 5.0, but the most commonly used ones. More detailed information about those projects and some other projects can be found in Section 2.2.

1.3 Goal Criteria
This master’s thesis focuses on incompatibilities due to new Java language features. Therefore, the first goal of this thesis is to design and implement source code transformations resolving new language features using the RECODER metaprogramming framework. The transformations are supposed to be as independent of each other as possible, so that they can be used separately as well. The second goal is to test the transformations on RECODER itself and on the larger FindBugs project. After transformation, the projects have to be able to be compiled and run, and test suites belonging to the project must succeed. The third, but only secondary, goal is to test the RECODER framework. Because all transformations contain an extensive analysis phase and collect information from the input program, this thesis offers the possibility to check whether RECODER’s queries work properly. Bugs in the queries will result in wrong information for the transformations and thus in transformed code that will not compile. Therefore the transformations shall be added as an extensive TestSuite to RECODER. Testing the transformations on RECODER is already a quite advanced example and it is easy to check if the results are correct.

To meet the goals stated above, the following tasks need to be performed:

- Design algorithms to resolve the single language features and replace them with Java 1.4 compatible constructs.
- Implement these algorithms in a set of transformations, extending the RECODER framework.
• Implement a test case that applies the transformations separately or all together on a given input project.

• Apply the test case on the RECODER framework itself an one other project. The test case has to succeed and the transformed project must compile and its test suite has to succeed.

1.4 Contribution

We implemented nine transformations resolving the new Java 5.0 language features. To test our transformations we applied them to the RECODER framework itself. After transformation, we were able to compile the transformed code with a Java 1.4 compiler and to run the RECODER testsuite successfully.

Furthermore, we applied the transformations to the FindBugs project. FindBugs is a popular Java program, requiring Java 5.0 or higher, that uses static analyses to identify bugs in Java programs. After transformation we could compile FindBugs successfully and run the test cases.

During our work on the transformations, several bugs in the RECODER framework were discovered and resolved. These bugs were mainly related to generic type and type parameter handling. This resulted in extensive API changes and the new RECODER 0.90 release.

1.5 Overview of the Report

The report is structured as follows: In Section 2 some background information is given which is necessary to understand the problem and results proposed in this thesis. This includes information on the new Java language features, the RECODER framework, and some alternative tools downgrading Java. Section 3 provides information on the steps performed during this master’s thesis in order to achieve the results. In Section 4 for each single transformation the basic idea of the transformation algorithm is given. Furthermore problems that arose during implementation are discussed. For each transformation a pseudo-code algorithm is provided. In the end of the section an overview of dependencies between the transformations is given. In Section 5 the results of testing the transformations on RECODER and FindBugs are presented and our approach is compared to other projects. Finally, in Section 6 the main conclusions drawn in this study are reported, and future tasks are proposed.
2 Background

In this section, we give an overview of current projects downgrading Java 5.0. We will also give an introduction to RECODER, the framework used to implement our source-code transformations. But first we will begin with an introduction to the Java language features introduced in the Java 5.0 release.

2.1 Introduction to J2SE(TM) 5.0 new Language Features

The Java 2 Platform Standard Edition 5.0 implements some major changes including performance enhancements, extended base libraries, changes to the virtual machine as well as new Java language features, i.e., extension of the language syntax. These syntactical changes are those we are dealing with in this thesis and are described in this section.

2.1.1 Generic Types

Generic Types, also referred to as Generics, allow a type or method to work on objects of various types while still providing compile-time type safety. The types of objects a class or method can operate on are specified via type parameters passed at declaration time and evaluated at compile-time. One common use of generic types is in combination with collections. Therefore, generic types come along with an extension of the Java collection framework. Since generic types are quite complex, this introduction will only cover the very basics of all generic constructs. For a more detailed discussion, please confer to the generics tutorial [Bra04]. The examples in this section are taken from [Bra04].

Introduction

A typical usage of collections before Java 5.0 is shown in Listing 2.1.

```java
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

Listing 2.1: Collections without generics

Since the compiler can only guarantee that an Object will be returned by the iterator, the cast on line 3 is essential to ensure that the assignment to a variable of type Integer is typesafe. But it also introduces the possibility of runtime errors if the object returned is not of type Integer. One possible solution is to mark a list as being restricted to contain a particular data type which is the core idea behind generic types. Listing 2.2 shows the program fragment from above using generic types.

```java
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

Listing 2.2: Collections with generics

The type declaration for the variable myIntList now restricts the declared list myIntList to operate only on objects of type Integer. Thus myIntList is not just an arbitrary List, but a List of Integer, written List<Integer>. The type parameter - in this case, Integer - is passed at declaration time and can be evaluated at compile time. The compiler now has the information that objects myIntList can hold are of type Integer and can check all assignments. Therefore, the compiler knows that the invocation of next() will return an Integer and, thus, the cast on line 3 can be omitted.
Defining Generic Types and Methods

The following example in Listing 2.3 shows a small excerpt from the definitions of the interfaces List and Iterator in package java.util:

```java
public interface List<E> extends Collection<E> {
    void add(E x);
    Iterator<E> iterator<E>();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Listing 2.3: Excerpt from the definitions of the interfaces List and Iterator

This looks very similar to a normal interface declaration except for the *formal type parameters* specified in angle brackets. These type parameters can be used throughout the generic declaration anywhere ordinary types can be used. However, there are some restrictions, e.g., type parameters may not be used in annotation type declarations (see Section 2.1.7). The type `List<Integer>`, as we saw it in Listings 2.1 and 2.3, is an *invocation* of the generic type declaration `List`. Invocations of generic types are called *parameterized types*. In a parameterized type all occurrences of the formal type parameter (E in this case) are replaced by the *actual type argument* (in this case, `Integer`).

Beside generic types, Java 5.0 allows the definition of generic methods. Just like type declarations, method declarations can contain type parameters. A simple example for that is a method that takes an array of objects and a collection and puts all objects from the array into the collection as shown in Listing 2.4.

```java
public static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

Listing 2.4: A generic method copying an array into a collection

This method can be called with any kind of collection whose element type is a supertype of the element type of the array as shown in Listing 2.5.

```java
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co); // T inferred to be Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs); // T inferred to be String
fromArrayToCollection(sa, co); // T inferred to be Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn); // T inferred to be Number
fromArrayToCollection(fa, cn); // T inferred to be Number
fromArrayToCollection(na, cn); // T inferred to be Number
fromArrayToCollection(na, co); // T inferred to be Object
fromArrayToCollection(na, cs); // compile-time error
```

Listing 2.5: Calling generic methods
Notice that at no time the actual type argument is passed to a generic method. The compiler infers the type argument based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

**Bounded Type Parameters and Wildcards**

Type parameters as they were used in the previous section can be instantiated with any real type. Sometimes, however, it is required to restrict the type parameter of a generic type to special kinds of types. For example one might want to restrict the elements of a generic type `NumberList<E>` to be a subtype of `Number`. This can be achieved by using *bounded type parameters*, in this case bounding the type parameter `E` to be a subtype of `Number: NumberList<E extends Number>`. Also lower bounds of the type parameter can be specified using the keyword `super`.

Consider a method accepting a collection as input parameter that prints out the elements of the collection. Since the method should accept collections with all possible element types, we need to use the supertype of all kinds of collections as parameter, which is `Collection<?>` (called "collection of unknown"). The `?` is called wildcard and can be upper and lower bounded like type parameters. Listing 4.35 shows the method using a wildcard as parameter. For more information on generic types and subtyping, wildcards and their restrictions, please confer to appendix A.1.

```java
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Listing 2.6: Using wildcards

### 2.1.2 Enhanced for Loop

The enhanced for loop construct simplifies iterating collections or arrays. Using the new enhanced for loop construct, one can iterate through collections and arrays without the need for iterators or indices, minimizing the opportunity for error, e.g., misspelling the iterator variable name or the index name respectively (a typical copy-and-paste error).

Listing 2.7 will show a typical usage of an iterator before the enhanced loop construct has been introduced.

```java
void printAll(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}
```

Listing 2.7: Iterating over a collection without the enhanced for loop construct

The enhanced for loop hides the iterator and offers access to the elements by a local variable declared in the enhanced for statement. Listing 2.8 is equivalent to the example stated in Listing 2.7.

```java
void printAll(Collection<String> c) {
    for (String s : c) {
        System.out.println(s);
    }
}
```

Listing 2.8: Example for iterating over a collection with the enhanced for loop construct

6
Compared to the previous example, this is less code to write. To have a more intuitive understanding the enhanced for loop can be read as "for each String s in c".

The enhanced for loop construct even takes care of a common mistake when working with nested iteration over two collections as presented in Listing 2.9.

```java
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();

for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(newCard(i.next(), j.next()));
```

Listing 2.9: Common mistake when using nested iteration over two collections

The code in Listing 2.9 will throw a NoSuchElementException because the next() method of the “outer” collection (suits) is called too often. It is called every time when iterating through the “inner” collection. In order to fix this problem, it is inevitable to add a local variable to the scope of the outer loop in order to hold the value of the suit.

```java
for (Iterator i = suits.iterator(); i.hasNext(); ) {
    Suit suit = (Suit) i.next();
    for (Iterator j = ranks.iterator(); j.hasNext(); ) {
        sortedDeck.add(new Card(suit, j.next()));
    }
}
```

Listing 2.10: Nested iteration using iterators

Because the for-each construct is tailor-made for nested iteration and implicitly declares a local variable, it simplifies iterating collections using nested iterations as shown in Listing 2.11.

```java
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

Listing 2.11: Nested iterations using the for-each construct

Using the for-each loop construct beautifies the code and hence its readability. It should be used any time possible. Unfortunately, there are situations the for-each loop is not applicable. Since the enhanced for loop construct hides the iterator it is not possible to remove or to replace elements in a collection while traversing. Nor can it be used to iterate over several collections in parallel.

### 2.1.3 Boxing / Unboxing Conversions

Collections are a powerful tool for developers. But since collections can only hold object references and no primitive values, as, for instance, int, these have to be boxed into the corresponding wrapper class (which is Integer in the case of int). On the other hand, you have to unbox the object you get out of the collection using the appropriate method of the wrapper class (intValue in case of Integer and int), if you need the primitive type. Before Java 5.0 this had to be done by hand cluttering up the code. The boxing and unboxing conversion, also called Autoboxing, automates the boxing and the unboxing and eliminates the need for the additional code.

Listing 2.12 gives an example on the boxing and unboxing conversion.
import java.util.ArrayList;

public class Autoboxing {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            // instead of list.add(new Integer(i));
            list.add(i);
        }
        int sum = 0;
        for (Integer j : list) {
            // instead of sum += Integer.intValue(j);
            sum += j;
        }
    }
}

Listing 2.12: Example for autoboxing and unboxing

As can be seen in Listing 2.12 boxing and unboxing conversion enables you to add `int` values into collections and to add up `Integer` objects and to store them in an `int` variable.

One thing to note is the fact that `Integer` objects are cached in a pool for small integral values. If two different objects stand for example for the number 2, a single object will be referenced from different locations. In detail, autoboxing is guaranteed to return the same objects for integral values in the range [-128,127]. So, if two different objects are 128, two different objects will be referenced. See Listing 2.13 for better understanding.

import java.util.ArrayList;

public class BoxingEquality {
    public static void main(String[] args) {
        int i = 2;
        int j = 2;
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(i);
        list.add(j);
        System.out.println("It is " + (i == j) + " that i == j.");
        System.out.println("It is " + (list.get(0)==list.get(1)) + " that list.get(0) == list.get(1)." );
        System.out.println("It is " + (list.get(0).equals(list.get(1))) + " that list.get(0).equals(list.get(1)).");
    }
}

Listing 2.13: Example for caching of integer objects

The output of the program will be the following:

$ java BoxingEquality
It is true that i == j.
It is true that list.get(0) == list.get(1).
It is true that list.get(0).equals(list.get(1));

Re-running the program with i = 128 and j = 128 will show the difference.

Recompiling and re-running Boxing Equality will now result in the following:

$ java BoxingEquality
It is true that i == j.
It is false that list.get(0) == list.get(1).
It is true that list.get(0).equals(list.get(1));
Autoboxing and unboxing makes it easy to move back and forth between primitives and objects. But at the same time, it is easy to loose track whether you are dealing with objects or not. So when comparing two integral values special care is needed whether object references or primitive values are compared.

### 2.1.4 Typesafe Enums

The new Java language feature allows you to create enumerated types with arbitrary methods and fields. It implements all the benefits of the Typesafe Enum pattern reducing the needed effort for creating typesafe enumeration types.

A common way to represent an enumerated type used to be the \textit{int Enum pattern} as presented in Listing 2.14:

\begin{Verbatim}
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL = 3;
\end{Verbatim}

Listing 2.14: The int enum pattern

This pattern though has some major problems:

- **Not typesafe** - Since an enumeration constant is just an \textit{int} value, you can pass any other \textit{int} value where a season constant is required, or worse, add two seasons together or two enumeration constants from two different enumeration patterns (which both make no sense).

- **No namespace** - Because the enumeration type has no own namespace, it is common to prefix constant names with a string (in this case SEASON) to avoid duplicate names with other enumeration types.

- **Brittleness** - One problem arises out of the fact that the enumerations are compile-time constants. Consequently, they are compiled into clients that use them. If the code is adapted later on, and a new constant is added between two already existing constants or just the order of the constants gets changed, clients must be recompiled. The programmer then has to take care of this. The clients will still run, even without recompiling, but the behavior might change.

- **Printed values are uninformative** - Since the enumeration constants are just \textit{int} values, printing one of them gets you only a number, which provides you with no information about what it represents and what enumeration type it belongs to.

To get around all these problems, one could use the \textit{Typesafe Enum pattern} [Blo01]. But this requires quite some effort from the programmer as it is quite complex and hence error prone. Furthermore, enumeration constants from the typesafe enum pattern cannot be used in switch statements.

Listing 2.15 shows the typesafe enum pattern for the same enumeration as described above with the \textit{int Enum pattern}. As you can see, a lot of code has to be produced in order to obtain the same enumeration type.

\begin{Verbatim}
import java.util.*;
import java.io.*;

public final class Season implements Comparable, Serializable {
    private final String name;
    public String toString () { return name; }
}
\end{Verbatim}
Listing 2.15: The typesafe enum pattern

The new language features includes support for enumerated types. In their simplest form, they look the same as enumeration types in C, C++ or C#:

```java
class Season { WINTER, SPRING, SUMMER, FALL }
```

While the enumeration types of C, C++ and C# are not more than glorified integers, the `enum` declaration in Java defines a complete class. You may add arbitrary methods and fields to an enumeration type, implement interfaces, and even add individual behavior to the different enumeration constants. All enumeration types are automatically `Comparable` and `Serializable`. The serial form is especially designed to withstand changes in the enumeration type.

Listing 2.17 will show an enumeration type declaration adding special behavior to each enumeration constant.

```java
public enum Shape {
    CIRCLE("circle")
        { double circumference(double x) { return 3.14159 * 2 * x },
    SQUARE("square")
        { double circumference(double x) { return 4 * x },
    EQUILATERALTRIANGLE("triangle")
        { double circumference(double x) { return 3 * x };

    Shape(String name) {
        this.name = name;
    }
    private final String name;
    abstract double circumference(double x);

    double volume(double x) {
        switch(this) {
```
```java
case CIRCLE :
    return 3.14159 * Math.pow(x, 2);
case SQUARE :
    return pow(x, 2);
case EQUILATERALTRIANGLE:
    return (pow(x, 2) * pow(3, 0.5)) / 4;
}
throw new AssertionError("Unknown shape: " + this);
```

Listing 2.16: Enumeration type declaration

Listing 2.17 shows two different ways to specify individual behavior to enumeration constants. One way is to write a method declaration and switch on the enumeration constants. The drawbacks of this approach is the need for the `throw` statement and the fact that each time a new constant is added to `Shape`, the `switch` statement has to be adapted. If the programmer forgets about it, the `volume` method will fail, executing the aforementioned `throw` statement.

The second possibility, to add individual behavior to each enumeration constant for some method, avoids these problems. The method is declared as abstract in the enumeration type and each constant overrides it with a concrete method. Such methods are called `constant-specific` methods.

All enumeration types implement the static method `values()`, which returns an array containing all of the values of the enumeration type in the order they are declared, as well as the static method `valueOf(String name)`, which returns the corresponding enumeration type value. Furthermore, all enumeration types implement the methods of class `Enum`, which is the common base class of all Java language enumeration types.

### 2.1.5 Varargs

In Java 1.4 or prior releases, it was not possible to pass an arbitrary number of arguments to a method. Instead, it was required to create an array and put the values into the array before invoking the method. Consider a method adding up a list of integers as shown in Listing 2.17.

```
public class VarArgs {
    public static int max(int[] intArray) {
        int maxSoFar = Integer.MIN_VALUE;
        for (int i : intArray) {
            if (i > maxSoFar) maxSoFar = i;
        }
        return maxSoFar;
    }

    public static void main(String[] args) {
        int[] array = {5, 3, 7, 4, 9, 1};
        int max = max(array);
    }
}
```

Listing 2.17: Method declaration without varargs option

Using the varargs option the method declaration and corresponding invocation would be as presented in Listing 2.18.
Listing 2.18: Method declaration with varargs option

As the example shows, the array declaration can now be omitted. Another common example for varargs methods is formatting strings as shown in Listing 2.19.

```java
public class VarArgs {
    public static int max(int... intArray) {
        ...
    }

    public static void main(String[] args) {
        int max = max(5, 3, 7, 4, 9, 1);
    }
}
```

Listing 2.19: Formating strings without varargs option

The new declaration of the MessageFormat.format method in the Java 5.0 API accepts an arbitrary number of arguments. Listing 2.20 shows an example of how to use the new MessageFormat.format method.

```java
Object[] arguments = {
    new Integer(7),
    new Date(),
    "a disturbance in the Force"
};

String result = MessageFormat.format("At {1.time} on {1.date}, there was {2} on planet "+ "{0.number, integer ".", arguments);
```

Listing 2.20: Formatting strings with varargs option

2.1.6 Static Import

Static imports eliminate the need for using fully qualified names for static members of other classes. A common example are the static members of java.lang.Math as shown in Listing 2.21.

```java
double r = Math.cos(Math.PI * theta);
```

Listing 2.21: Usage of static members

In Java 1.4 or prior, a common way to get around this was to put static members into an interface and inherit from that interface. This is a bad idea and there is even a name for it: the Constant Interface Antipattern (see [Blo01] Item 17). The problem associated with this pattern is that the use of static members of another class belongs to the implementation detail. Putting the static member into an interface they become part of the class’ public API. But implementation details should never be contained in public APIs.

Using the static import construct enables the programmer to access static members without inheriting from the type containing the static members. Instead, the program imports the members, either individually or en masse as shown in Listing 2.22.
The syntax of the static import declaration is analogous to the normal import declaration. But in contrast to import classes from packages to enable the programmer to use them without package qualification, a static import declaration imports static members from classes, so that they can be accessed without class qualification.

2.1.7 Annotations (Metadata)
Annotations are a new language construct to include metadata in source code. Annotations are indicated by an @-sign followed by the name of the annotation type. They do not directly affect the program semantics, but they do affect the way programs are treated by tools and libraries. This in turn can affect the semantics of the running program. Depending on the annotation type, annotations can be read from source files, class files, or reflectively at run time.

The main idea for adding metadata to Java programs was the fact that many APIs require so called boilerplate code. The programmer is often forced to write a lot of additional code that needs to be maintained in parallel with the corresponding code. This code often has a regular structure depending on the code it is associated with. For example when implementing a JAX-RPC web service, one needs to provide a paired interface and implementation. It would be easy for a tool to generate this boilerplate code automatically if it knows which methods are remotely accessible. This information can be stored in annotations. The main uses of annotations are:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.

- **Compiler-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.

- **Runtime processing** — Some annotations are available to be examined at runtime.

### Defining own Annotation Types
An *annotation type declaration* is similar to an normal interface declaration. The keyword `interface` is preceded by an @-sign. An *element* of an annotation type is defined using method declarations. These method declarations must not have any parameters, type parameters or a `throws` clause. A further restriction is that return types have to be one of following: primitives, `String`, `Class`, `enum`, annotations, and arrays of the preceding types. Listing 2.23 is example for an annotation type declaration.

```
/**
 * Describes the Request-For-Enhancement (RFE) that led
to the presence of the annotated API element.
 */
public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}
```

Listing 2.23: Annotation type declaration
This newly defined annotation type can now be used to annotate declarations. Since an annotation is a special kind of modifier, it can be used anywhere that other modifiers (such as public, static, or final) can be used. It is common use that an annotation is the first modifier in the modifier list. An annotation consists of the annotation type name preceded by an @-sign and a parenthesized list of element-value pairs. All values must be compile-time constants. Listing 2.24 shows a method declaration annotated with the previously defined annotation type.

```java
@RequestForEnhancement(
    id = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date = "4/1/3007"
)
public static void travelThroughTime(Date destination) { ... }
```

Listing 2.24: Annotation use specification

An annotation type with no elements is called a marker annotation. If annotating a declaration with a marker annotation the parentheses may be omitted. In annotations with only one single element, this should be named value. If annotating a declaration with a single-element annotation whose element name is value you may omit the element name and the equals sign. Annotations that can be used to annotate annotation type declarations are called meta-annotations.

Predefined Annotation Types

The Java 5.0 API predefines the following annotation types in the packages java.lang:

- Deprecated
- Override
- SuppressWarnings

and in java.lang.annotation:

- Documented
- Inherited
- Retention
- Target

These predefined annotations types are listed and explained in Table A.1 in the appendix.

2.1.8 Intersection Types in Conjunction with the Conditional Operator

While in previous releases the two expressions of a conditional operator had to be in a superclass relationship and the return type of the conditional operator was well defined, now, due to capture conversion and type inference the return type of a conditional operator is allowed to be an intersection type which were introduced to Java 5.0. Listing 2.25 shows an example.
```java
public class ConditionalWithIntersectionTypes {
    public void testConditionals() {
        List<String> stringList = new ArrayList<String>();
        Set<String> stringSet = new TreeSet<String>();
        Queue<String> stringQueue = new PriorityQueue<String>();
        boolean list = false;
        boolean set = true;
        ...

        Iterable<String> coll = list ? stringList : set ? stringSet : stringQueue;
    }
}
```

Listing 2.25: Conditional operator with intersection type

### 2.1.9 Covariant Return Types
Covariant Return Types are an important enhancement to the previous JDK. In Java, it is not possible to have two methods in the same class with signatures that only differ by their return types. Until the J2SE 5.0 release it was not possible to change the return type of a method inherited from a superclass. J2SE 5.0 allows covariant return types. Thus, when overriding a method in a subclass, the return type may be changed to any subtype of the type returned by the method with the same signature in the superclass. Examples can be found on the Java home page [JSD04] and in Listing 2.26.

```java
public class SuperClass {
    public List<String> getElements() {
        ...
    }
    ...
}
public class SubClass extends SuperClass {
    public ArrayList<String> getElements() {
        ...
    }
    ...
}
```

Listing 2.26: Example for covariant return types

### 2.2 State of the Art
As already mentioned in the introduction, tools downgrading Java 5.0 can be differentiated by their scope, i.e., resolving language features or resolving invocations of the Java 5.0 Runtime Library. Those resolving language features can be even further divided according to their approach, i.e., byte-code based or source-code based. In this section we will first provide information on projects resolving invocations to the Java 5.0 Runtime Library. After that, we will give an overview of projects resolving the new language features. Table 2.1 gives an overview of the supported language features.

#### 2.2.1 Substituting Invocations of the Java 5.0 Runtime Library
The package `backport-util-concurrent` [bac07] is a backport of the java.util.concurrent API, introduced in Java 5.0 and extended and refined in Java 6.0, to older Java platforms.
The intention of the project was to provide a concurrency library that works on all Java platforms currently in use and therefore allows the development of fully portable concurrent applications. The target scope is Java 1.3 and above but there is also some limited support for Java 1.2.

The backport is almost complete. Reasons for unsupported functionality are:

- Classes relying on explicit JVM support, that cannot be emulated at a satisfactory performance level on platforms before 5.0.
- Functions described in the original javadoc were "designed primarily for use in monitoring in system state, not for synchronization control".

The following list summarizes the functionality of `java.util.concurrent` that is supported in the backport:

- All JSR 166 executors, utilities, and everything related (thread pools, FutureTas, scheduled tasks and executors, etc.)
- Locks: ReentrantLock, Semaphore, ReentrantReadWriteLock, Conditions
- Queues: synchronous, array, linked, delay, and priority queues
- Deques: array, and linked deques
- Atomics: everything except reflection-based updators
- Other concurrency utils: CountDownLatch, CyclicBarrier, Exchanger
- Concurrent collections: ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet, ConcurrentLinkedqueue, ConcurrentSkipList\[Map,Set\]
- Retrofitted standard collections

Retrotranslator [ret08a] combines resolving invocations of the Java 5.0 Runtime Library and language features. To resolve invocations of the Java 5.0 API it includes backports for most changes of the API. Also the previous described backport-util-concurrent is included. Furthermore Retrotranslator gives the user the possibility to add backports by himself.

2.2.2 Source Code based Approach

Declawer [dec08] is the only tool we know about so far, that transforms Java 5.0 source code to Java 1.4 compatible code. Unfortunately it is badly documented and nowhere is stated what new language features now actually are supported. In the beginning declawer was only a small side product and intended to resolve generics. Because many got interested in declawer and requested new features it got extended with support for varargs, foreach and annotations. But still many bugs seem to be present. For example, the foreach transformation seems not to work for arrays and to introduce for each iterator the same name which will cause name clashes in nested foreach loops. Since there exists nothing like a project page it is hard to find information whether these bugs have been resolved or whether new features have been implemented. So Declawer can at the moment not yet be called mature tool for transforming Java 1.5 source code to Java 1.4 source code.
2.2.3 Byte Code based Approach

**Retrotranslator** [ret08a] is a tool based on byte code manipulation to make Java 1.5 applications compatible with Java 1.4, 1.3 or prior. It is based on the ASM byte code manipulation framework [asm08] and the backport of the Java 5.0 concurrency utilities [bac07]. It handles all new language features described in Section 2.1 and supports a significant part of the Java 5.0 API on both J2SE 1.4 and J2SE 1.3. In prior Java releases the new Java language features are resolved but no API support is given.

Beside support for all new language features (see Section 2.1) it also offers reflection on generics and annotations, formatted output, concurrent utilities and collection framework enhancements. A table showing what is supported from the new Java API can be found on the Retranslator webpage [ret08a]. Since the tool is open for extensions it is possible to search for or implement and include backports for missing API functionality. Additional backports have to be specified in the classpath and the backport option.

The tool can be used as standalone program from the command line, as Apache Ant script [ant08] or via a plugin for Maven [mav08] and the Java IDE IntelliJ [int08].

As the name lets suspect **Retroweaver** [ret08b] is a byte code weaver. It transforms Java class files compiled by a 1.5 compiler into version 1.4 class files which can then be run on any 1.4 virtual machine. Like Retrotranslator it supports all new Java language features. But in contrast to Retrotranslator Retroweaver does not have any support for the Java 5.0 Runtime Library. Instead it will warn the user if any of the code references a class, method, or field that cannot be found in the target VM so the user can locate it and correct the issue.

Retroweaver can be used from the command line but also a graphical user interface is provided.

**JBoss Retro** [jbo08] is supposed to convert Java byte code into byte code that can be run on any 1.4 virtual machine. As Retrotranslator it replaces invocations of the JDK5 API with equivalent code. In contrast to Retrotranslator JBoss Retro claims to support the whole JDK5 API. JBoss Retro can be run with Apache Ant [ant08] scripts.

**The EEL Project** [eel06] is a batch to enable Eclipse [ecl08] to produce JDK 1.4 (or
lower) compliant byte code, even when source level is set to 5.0. As already mentioned in Section 1.1 javac does not allow the user to use source = 1.5 and target < 1.5. Trying this will result in an error message: javac: source release 1.5 requires target release 1.5. EEL provides a trick to workaround this.

The EEL Project supports most of the new Java language features but not all. For example enumerations are not supported.

The author of the EEL Project is aware of the fact that this project is not a mature project but he sees it advantages in the simple integration with Eclipse. On the EEL Project webpage the author compares it to Retroweaver:

"Retroweaver is a sophisticated and great tool. This here is more a workaround (or better call it hack?). Retroweaver supports all JDK1.5 features while EEL is still missing important features as enumerations. On the other hand, when working with Eclipse, EEL allows you to more rapidly develop your applications, as you don’t need an additional step (retroweaving) to make you byte code JDK 1.4 ready. An Eclipse launch config will just be fine."

2.3 RECODER
RECODER is a framework for source code metaprogramming, a small introduction to metaprogramming is given in the next Section. Then an overview of the features of RECODER is given.

2.3.1 Metaprogramming
Metaprogramming describes the activity of writing computer programs that take other programs (or themselves) as input data and manipulate them. Sometimes metaprograms work at compile time and generate code at compile time, e.g., C++ metaprogramming using templates.

Metaprogramming simplifies the daily work of programmers since boilerplate code can be generated automatically. For example, defining a JAX-RPC Web Service requires both an interface and an implementation class. Using metadata, methods that are remotely accessible can be annotated and a tool can generate the corresponding interface and implementation class automatically. One further application area of metaprogramming is the maintenance of code. If a new component is added to a project or if an interface is changed, often changes all over the project have to be introduced. Metaprogramming allows the programmer to write programs that search for the places in the code that have to be adapted and that adapt the code accordingly.

Metaprograms most often parse the input code into a program metamodel representing the code structure. Analyses and transformations are carried out on the metamodel and later written back into source files. Figure 2.1, taken from [rec08], shows this approach.

2.3.2 Features of RECODER
RECODER supports static metaprogramming of Java sources. It allows the programmer to parse and analyze Java programs, transform the sources and write them back into source code form.

For an overview of RECODERs proceeding see Figure 2.1. RECODER derives a meta model of the input program provided in form of Java source code and class files. The derived model contains a detailed syntactic program model that can be unparsed with minimal losses. The syntactic model provides information about the containment relation between element, e.g., a method declaration contained in a type declaration. The complete
model adds further relations, such as type-of, or refers-to, as well as some implicitly defined elements, such as packages or primitive types.

The semantic information is derived running type- and name-analysis which resolve references to logical entities. The refers-to relation is made bidirectional for full cross referencing, which is necessary for efficient global transformations.

RECODER can be used both for pure *analyses* and for *metaprograms*. While for the analyses only a meta model is created and nothing is changed, static metaprograms directly transform the meta model. Those metaprogram applications use the RECODER pretty printer to write back the program model into source files afterwards. The pretty printer will try to retain the code formatting and to integrate new code fragments seamlessly.

The following list gives a short description of the different layers of RECODER features and the corresponding application perspective:

**Parsing and unparsing of Java Sources**
Since RECODER supports a highly detailed syntactic model without information loss, and also comments and formatting information is retained, it can be used for development of simple preprocessors, simple code generators as well as source code beautification tools. For the latter, the provided pretty printer is customizable and is able to reproduce the code or to improve it while retaining the given code structure as well as to embed new code seamlessly.

**Name and type analysis for Java programs**
RECODER offers a set of queries to derive types of expressions, evaluate compile-time constants, resolve all kinds of references, and maintain cross reference information. This offers the basis for software visualization tools, software metrics, Lint-like semantic problem detection tools, design problem detection tools that search for anti-patterns (patterns that are commonly known as bad coding style), and cross referencing tools.

**Transformation of Java sources**
RECODER contains a library of predefined analyses, code snippet generators, and frequently used transformations. Those can be used to build up preprocessors for
language extensions, semantic macros, aspect weavers, source code obfuscation tools, and compilers.

**Incremental analysis and transformation of Java sources**
Since transformations are executed on the program model and change it, the model has to be updated accordingly. Otherwise it can occur that a query is executed on an outdated model and returns a result conflicting with the updated, i.e., transformed model. Thus transformations have to take care of dependencies by updating their local data. RECODER will analyze change impacts for its model and perform updates automatically if a query is executed. This feature can be used for source code optimization, refactoring tools, software migration programs (Smart Patches), design pattern, clichés and idiom synthesis, architectural connector synthesis, adaptive programming environments, and invasive software composition.
3 Processing

The aim of the master’s thesis is to develop source code transformations that resolve the new language features. These transformations shall be used to extend the RECODER framework. The following section will describe the steps involved to fulfill this task and how to evaluate the results.

3.1 Design Algorithms

Information about the new features can be found in the Java documentation [J2S04]. For further information about, e.g., intersection types and capture conversion, the Java Language Specification [Gos05] was used as reference. This information was used as input for working out the main idea for the single transformations, one for each new language feature. This resulted in the basic ideas for the transformations stated in Section 4. For most transformations, the basic idea was quite straightforward. But since the advantages of the new typesafe enumerations were oriented on the advantages of the typesafe enum pattern from [Blo01] the basic idea for this transformation was based on this pattern and got only slightly extended to cope with the additional functionality of the Enum class, which is the base class for each enumeration type.

3.2 Providing Test Data

For each new language feature some simple classes were written. These were then used as simple input data for the transformations and evolved together with the transformations. If the transformation could handle a simple instance of the language feature a more complex example was added and the transformation was adapted until it could handle this situation as well. During the evolution of the test input data special complex constructs were identified which caused problems and resulted in adaptations of the algorithm. These problems faced during the implementation of the basic idea of each transformation can also be found in Section 4.

3.3 Finding suitable Project for Testing Issues

Additional to RECODER another project had to be found for testing the transformations. There were several requirements a suitable project had to fulfill.

- Written in Java 5.0 using preferably all Java 5.0 constructs.
- If possible, no or only few calls to new functionality from the Java 5.0 API, in order to reduce manual work afterwards.
- A complete test suite, that is supposed to succeed still after transformation.

We decided to take FindBugs [fin08] as second testing project (version 1.3.4). FindBugs is a program which uses static analysis to identify bugs in Java programs. It requires JRE 1.5.0 or later to run and provides a set of test cases. We chose FindBugs because it is a quite huge project and is very popular. We were interested if we were able to transform such an advanced project. Because if we can transform FindBugs we can probably transform many other projects as well.

3.4 Implementation of Transformations

The implementation of the transformations was the main task of the master’s thesis. For all transformations, a first version was available but most of them did not work. Therefore the main part of many transformations has been rewritten.
The existing transformations also had the problem that they all had different constructor signatures. Some accepted a whole list of compilation units while others only accepted a single invocation of a feature, e.g., a single enhanced for loop. First of all, this is confusing for the user and, secondly, has an impact on the performance. If only single compilation units or worse single instances of a certain feature are used to create a transformation, for each transformation instance the analyze and transform step are executed. Each time a transformation introduces some changes, the model has to be updated before the analysis phase of the next transformation instance can start, resulting in performance loss (see also Section 4.1). This has been adapted. All transformations now accept a list of compilation units, which optimizes performance. All compilation units are now analyzed together and then transformed at once saving all the model updates that otherwise would occur after each transformation of a single compilation unit that introduced some changes.

3.5 Testing

The implementation of the transformations was tightly connected to the testing activity. A new JUnit [Jun08] test containing test cases for each single transformation was implemented, and one for applying all transformations together. In the beginning of the project the transformations were tested separately, first using the small test classes described in Section 3.2. If the transformations worked for fairly complex test data, they were tested using RECODER as input. When finally all transformations succeeded on RECODER they were tested together. In that stage errors occurred due to dependencies between some of the transformations. If possible, these were resolved. For information on remaining dependencies between the transformations please confer to Section 4.11.

3.5.1 Compiling the transformed Program

Since RECODER assumes that the input program compiles correctly, it only performs a partial semantic analysis. This means even if all transformations could be executed without any problems, i.e., no model exceptions occur, this does not imply that the resulting transformed program is free of errors. For example, RECODER does not care if a left hand side expression has the same type as the corresponding right hand side expression. So, if for example ResolveGenerics forgets to introduce a cast, RECODER overlooks this fault. So first compiling the transformed program will show if the transformations really worked and resulted in a syntactically correct program. For compiling the transformed project, e.g., transformed RECODER, it was imported to Eclipse and the compiler settings were set to the Java 1.4 compiler. If the project contained invocations of the Java 5.0 Runtime Library, it was included to the project instead of the older runtime library. This is necessary, because during this work only the language features are resolved but not yet the invocations to the Java 5.0 API.

3.5.2 Running the transformed Testsuite

Since compiling the transformed code will only check for syntactically correctness it is desirable to have a complete test suite to check whether also the semantic has been retained. Otherwise it is not possible to rule out the possibility of unexpected behavior of a transformed program.

When all these steps succeeded for RECODER we repeated them for FindBugs. At this stage many bugs both in RECODER and the implementation of the transformation were identified and removed.
4 Transformations

In Section 2.1, we have presented the new language features of Java 5.0. In this section, we present our algorithms designed to resolve these language features. We designed one algorithm per feature. For each transformation the basic idea will be described first, supported by small examples. We discuss then problems related to each transformation and how we solved them. Furthermore for each transformation a draft of the algorithm in pseudo-code is given. In the end of this section we will give an overview of dependencies between the single transformations.

4.1 Introduction

Transformations implemented using RECODER are executed on the abstract syntax tree (AST), a syntactical program model (for detailed information see Glossary in Appendix B), of the input source code. All transformations described in the following section are so-called two-pass transformations. As the name implies, these transformations are performed in two steps. During the first step, the analyze-step, all information needed to perform the transformation, is collected and stored by traversing the AST. The second step, the transform-step is the actually transforming step. In this step all stored information is used to transform the program model, i.e., the AST. After all changes are executed, the program model is written back to source files.

The major advantage of this approach is performance. After each small change to the AST a query like `getType(Expression)` will cause a model update which is time consuming. This model update is needed since the change to the AST could have an impact to the result of the query, so it needs to be assured that the model is up to date each time a query is performed. RECODER is capable of incremental updates, but still there is a significant performance difference if the number of model updates is reduced as much as possible. This also due to the fact that the incremental update does not work for all situations. If the information gathering and the transformation would not be separated that strictly as it is in a two-pass transformation, many model updates would be required and accordingly the performance of the whole transformation would suffer.

4.2 Resolving Generics

The transformation resolving generics, implemented in class `ResolveGenerics`, is the most complex one of our nine transformations. To make it more readable and understandable, the task is divided in several subtasks implemented in separate methods. Figure 4.1 shows how the single subtasks collaborate with each other in order to resolve generics. Arrows indicate that a subtask might call another subtask. The tail of the arrow indicates the caller, the head points to the called subtask.

In the following, an overview of the subtasks is given. The descriptions are in order of their dependencies, i.e., subtasks depending on other subtasks, i.e., a subtask it might call during execution, are described after all subtasks it depends on have been described.

Resolving Method Return Types

This subtask, implemented in method `resolveMethodReturnType`, is responsible for resolving method return types of generic methods or methods with a return type containing type arguments. For the latter the type arguments are removed from the type declaration. Another subtask (resolving type parameters) takes care of the type parameters of generic methods so they can be ignored in this subtask. The main task of resolving method return types is to insert type casts where generic methods or methods with a parameterized
return type are referenced. Listing 4.1 shows a method declaration with a parameterized type as return type.

```java
public List<String> getNames() {
    ...
}
String s = getNames().get(i);
```

Listing 4.1: Reference to a method with parameterized return type

This corresponding transformed code is shown in Listing 4.2.

```java
public List getNames() {
    ...
}
String s = (String)getNames().get(i);
```

Listing 4.2: Reference to a method with parameterized return type after transformation

For a pseudo code representation of the transformation please confer to Algorithm 3.

**Resolve single Variable Declaration**

This subtask, implemented in method `resolveSingleVariableDeclaration`, is responsible for resolving variable declarations which type contains type arguments. The type arguments from the variable declaration are removed and for all variable specifications contained in this variable declaration, casts are introduced where these variables are referenced. The example in Listing 4.3 declares two variables of a parameterized type.
List<String> names, streets;
...
String s1 = names.get(i)
String s2 = streets.get(i);

Listing 4.3: Variable declaration with parameterized type

The corresponding transformed code is shown in Listing 4.4

List names, streets;
...
String s1 = (String)names.get(i);
String s2 = (String)streets.get(i);

Listing 4.4: Variable declaration with parameterized type after transformation

For a pseudo code representation of the transformation please confer to Algorithm 4.

Replace multiple Bounds

This task is a subtask of resolving type parameters explained next. It is implemented in makeReplacement and handles multiple bounds of type parameters, e.g., List<E extends Number & Comparable<E>>. It returns the type reference of the first bound and adds all further bounds as comment to the type reference. So in this case printing the resulting type reference would result in: Number /* Comparable */.

By replacing the type parameter with just one type reference, information is lost in case of multiple bounds. Therefore, the method checks whether methods provided by the types of the further bounds are used in the generic type or method declaration. For example, if the code would contain a reference to compareTo, a cast to Comparable needs to be inserted, since the information that E implements Comparable is lost. For a pseudo code representation of the transformation please confer to Algorithm 5.

Resolving Type Parameters

This subtask, implemented in resolveTypeParameters, replaces type parameters and inserts casts when a generic method or a method or variable from a generic type is referenced. It is a subtask of resolving generic type or method declarations. The type parameter is replaced with the result of makeReplacement. Listing 4.5 shows how the methods resolveTypeParameters and makeReplacement collaborate:

```java
public class NumberList<E extends Number & Comparable<E>>
    extends ArrayList<E> {
    public E max() {
        if (this.size() > 0) {
            E maxSoFar = this.get(0);
            for (int i = 1; i < this.size(); i++) {
                if (this.get(i).compareTo(maxSoFar) > 0) {
                    maxSoFar = this.get(i);
                }
            }
            return maxSoFar;
        } else return null;
    }
}
```

Listing 4.5: Generic type declaration with a multiple bound type parameter

Replacing the type parameter will result in the following type declaration (Listing 4.6):
```java
public class NumberList extends ArrayList {
    public Number /* & Comparable */ max() {
        if (this.size() > 0) {
            Number /* & Comparable */ maxSoFar = this.get(0);
            for (int i = 1; i < this.size(); i++) {
                if (((Comparable) this.get(i)).compareTo(maxSoFar > 0) {
                    maxSoFar = this.get(i);
                }
            }
            return maxSoFar;
        } else return null;
    }
}
```

Listing 4.6: Resolving type parameters with multiple bounds

For a pseudo code representation of the transformation please confer to Algorithm 6

### Resolving overridden Methods with Type Parameters in Parameter Declaration

This subtask, implemented in `resolveParameterDeclaration`, resolves methods that override a method with type parameters in their signature. Depending on whether the containing class type implements the raw type of the generic type or a special instantiation, the method declaration contains type parameters itself in its signature or the type argument specified in the inheritance specification. The former are resolved while resolving type parameters, but also the latter ones have to be taken care of. Assume the following example in Listing 4.7:

```java
public interface Comparable<T> {
    public int compareTo(T o);
}

public class Something implements Comparable<Something> {
    ...
    public int compareTo(Something o) {
        ...
    }
    ...
}
```

Listing 4.7: Overriding a method with type parameters in parameter declaration

In the Java 1.4 API the class `Comparator` is non generic and, thus, the method `compareTo` takes an `Object`. If the `compareTo` method of `Something` is not adapted the compiler will output an error, that `Something` needs to implement the method `compareTo(Object o)`. Therefore methods overriding methods with type parameters in signature have to be identified and the parameters have to be substituted by the upper bound of the type parameter. After transformation the example from Listing 4.7 will look as shown in Listing 4.8

```java
public interface Comparable {
    public int compareTo(Object o);
}

public class Something implements Comparable {
    ...
    public int compareTo(Object o) {
        ...
    }
    ...
}
```

26
Listing 4.8: After resolving parameter declarations

One special situation the transformation has to take care of is the declaration of two methods with different signature. A subclass could for example implement one method `compareTo` with `Object` as parameter and one second method `compareTo` with for example the containing class type as parameter. In this case nothing has to be transformed. By simply checking both method whether they override something they would both be transformed causing a duplicate declaration error after transformation. Thus, the transformation checks whether the containing class type declares a second method with a more general signature before it adapts the parameter type. For a pseudo code representation of the transformation please confer to Algorithm 7.

Resolving generic Method Declarations

This subtask, implemented in `resolveGenericMethodDeclaration`, is responsible for identifying generic method declarations, removing type parameters from the declaration and type arguments from corresponding method references. To replace the type parameters from the method body the subtask resolving type parameters is called. Listing 4.9 provides an example for a generic method declaration.

```java
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

Listing 4.9: Generic method declaration

Resolving the generic method declaration in Listing 4.9 will result in the code stated in Listing 4.10.

```java
static void fromArrayToCollection(Object[] a, Collection c) {
    for (Object o : a) {
        c.add(o);
    }
}
```

Listing 4.10: Resolving generic method declarations

For a pseudo code representation of the transformation please confer to Algorithm 8.

Resolving single generic Types

This subtask, implemented in `resolveSingleGenericType`, is equivalent to the subtask of resolving generic method declarations. It removes type parameters from the declaration, calls the subtask resolving type parameters to remove references to the type parameters from the type declaration. Furthermore, it calls the two subtasks resolving single variable declarations and method return types, if it contains a variable declaration or respectively a method declaration with the declared type as type reference, and removes type arguments from the type references. As example confer to Listing 4.5. For a pseudo code representation of the transformation please confer to Algorithm 9. During the analyze phase the AST is traversed and depending on the program element the corresponding subtask is called. For detailed information confer to Algorithm 1.

27
Algorithm 1 The ResolveGenerics algorithm: analyze Part 1

Require: Enhanced-For loops have been resolved earlier
Ensure: Generics have been resolved

1: function ANALYZE
2:     for all td instanceof TypeDeclaration
3:         ∧ !(td instanceof TypeParameterDeclaration) do
4:         RESOLVESINGLEGENERICTYPE(td)
5:     end for
6:     for all md instanceof MethodDeclaration do
7:         RESOLVEGENERICMETHOD(md)
8:         RESOLVEMETHODRETURNTYPE(md)
9:     end for
10:     for all tr instanceof TypeReference, parent = tr.getASTParent() do
11:         if parent instanceof VariableDeclaration then
12:             RESOLVESINGLEVARIABLEDECLARATION(parent)
13:         else if parent instanceof InheritanceSpecification
14:             ∧ !(tr instanceof TypeParameterDeclaration) then
15:             if tr contains type arguments then
16:                 for all m ∈ {methods declared in inherited type declaration} do
17:                     RESOLVEMETHODRETURNTYPE(md)
18:             end if
19:             Remove type arguments from tr
20:         end if
21:         RESOLVEPARAMETERDECLARATION(subtype, supertype)
22:     else if parent instanceof MethodReference
23:         ∧ parent.getASTParent() instanceof MethodReference then
24:         RESOLVEMETHODRETURNTYPE(m)
25:     end if
26:     end for
27:     for all n instanceof New ∨ n instanceof NewArray do
28:         Remove type arguments if present
29:     end for
30:     for all c instanceof TypeCast do
31:         Remove type argument if present
32:     end for
33:     for all i instanceof Import do
34:         Get corresponding class type c
35:         if c instanceof ClassFile then
36:             for all methods m declared in c with
37:                 m.getReturnType() instanceof TypeParameter do
38:     end function
39:     RESOLVEMETHODRETURNTYPE(m)
40:     end if
41:     end for
42: end function

Algorithm 2 The ResolveGenerics algorithm: analyze Part 2
Algorithm 3 The ResolveGenerics algorithm: resolveMethodReturnType

1: function RESOLVE METHOD RETURN TYPE (Method m)
2: 
3: returnType = m.getMethodReturnType()
4: 
5: if !(returnType instanceof ParameterizedType) 
6: \land !(returnType instanceof TypeParameter) 
7: \land !(m instanceof ParameterizedMethod) then
8: 
9: return
10: end if
11: 
12: if m instanceof MethodDeclaration and return type contains type arguments then
13: 
14: Remove type arguments
15: end if
16: 
17: for all mr \in \{mr | mr references m\} do
18: 
19: Introduce recursively casts to mr
20: end for
21: 
22: end function

Algorithm 4 The ResolveGenerics algorithm: resolveSingleVariableDeclaration

1: function RESOLVE SINGLE VARIABLE DECLARATION (VariableDeclaration vd)
2: 
3: if vd.getTypeReference() contains no type arguments then return
4: end if
5: 
6: typeArgs = vd.getTypeReference().getTypeArguments()
7: 
8: t = getType(vd.getTypeReference())
9: 
10: for all vr \in \{vr | vr references vd\} do
11: 
12: if vr is left hand side of an assignment then
13: 
14: continue
15: end if
16: 
17: parent = vr.getReferenceSuffix()
18: 
19: ty = getType(vr)
20: 
21: if t instanceof TypeParameter \land !(ty instanceof TypeParameter then
22: 
23: Insert casts to vr and recursively to parent
24: end if
25: 
26: else
27: 
28: Insert recursively casts to parent
29: end if
30: 
31: end for
32: 
33: end function
Algorithm 5 The ResolveGenerics algorithm: makeReplacement

1: \textbf{function} \textit{makeReplacement}(TypeParameterDeclaration \textit{tpd})
2: \hspace{1em} \textit{repl} = \textit{tpd}.getBounds().get(0)
3: \hspace{1em} \textbf{if} \:\textit{tpd}.getBounds().size() > 1 \textbf{then}
4: \hspace{2em} \textbf{for} \textit{i} ← 1, \textit{tpd}.getBoundCount() − 1 \textbf{do}
5: \hspace{3em} Add \textit{tpd}.getBounds().get(\textit{i}) as comment to \textit{repl}
6: \hspace{3em} \textbf{for all} \textit{mr} method reference that references a method of the bound type \textbf{do}
7: \hspace{4em} Cast reference prefix to the bound type
8: \hspace{3em} \textbf{end for}
9: \hspace{2em} \textbf{end if}
10: \hspace{1em} \textbf{return} \textit{repl}
11: \textbf{end function}

Algorithm 6 The ResolveGenerics algorithm: resolveTypeParameters

1: \textbf{function} \textit{resolveTypeParameters}(List<TypeParameterDeclaration> \textit{type-Params})
2: \hspace{1em} \textbf{for all} \textit{tpd} \in \textit{typeParams} \textbf{do}
3: \hspace{2em} \textbf{if} \:\textit{tpd} has no bounds \textbf{then}
4: \hspace{3em} Create replacement type reference \textit{repl} to \textit{Object}
5: \hspace{2em} \textbf{else}
6: \hspace{3em} Create replacement type reference \textit{repl} with \textit{makeReplacement}(\textit{tpd})
7: \hspace{2em} \textbf{end if}
8: \hspace{1em} \textit{rt} = \textit{tpd}
9: \hspace{1em} \textbf{repeat}
10: \hspace{2em} \textbf{for all} \textit{tr} \in \{\textit{tr}|\textit{tr} references \textit{rt}\} \textbf{do}
11: \hspace{3em} \textbf{if} \!:\textit{(tr}.getASTParent instanceof TypeArgumentDeclaration \textbf{then}
12: \hspace{4em} Replace \textit{tr} with \textit{repl}
13: \hspace{3em} \textbf{else if} \textbf{R then move} \textit{tr}.getASTParent()
14: \hspace{3em} \textbf{end if}
15: \hspace{2em} \textbf{end for}
16: \hspace{1em} \textit{rt} = \textit{getArrayType}(\textit{rt})
17: \hspace{1em} \textit{repl}.setDimension(\textit{repl}.getDimension() + 1)
18: \hspace{1em} \textbf{until} \:\textit{rt} == \textbf{null}
19: \hspace{1em} \textbf{end for}
20: \textbf{end function}
Algorithm 7 The ResolveGenerics algorithm: resolveParameterDeclaration

1: function RESOLVEPARAMETERDECLARATION(TypeDeclaration tDecl, ClassType superType)
2:     ml = superType.getAllMethods()
3:     for all mDecl declared in tDecl do
4:         for all m ∈ ml do
5:             if mDecl and m have same name and signature size and tDecl contains no other method with same name
6:                 and more general signature then
7:                     Get all methods that mDecl redefines and take the one highest in the type hierarchie mi
8:                         for all parameters p in mi and instance of TypeParameter do
9:                             replace type in corresponding parameter declaration pd of mDecl with the upper bound of p
10:                            for all vr ∈ {vr | vr references pd} do
11:                                Cast to previous type of pd
12:                             end for
13:                         end for
14:                     end if
15:                 end for
16:             end for
17:         end for
18:     end function

Algorithm 8 The ResolveGenerics algorithm: resolveGenericMethodDeclaration

1: function RESOLVEGENERICMETHODDECLARATION(MethodDeclaration md)
2:     if md contains no type parameters then return
3:     end if
4:     typeParams = md.getTypeParameters()
5:     RESOLVETYPEPARAMETERS(typeParams)
6:     Remove typeParams
7:     for all mr ∈ {mr | mr references md} do
8:         if mr contains type arguments typeArgs then
9:             Remove typeArgs
10:        end if
11:    end for
12: end function

Algorithm 9 The ResolveGenerics algorithm: resolveSingleGenericType Part 1

1: function RESOLVESINGLEGENERICTYPE(TypeDeclaration td)
2:     if td contains no type parameters then return
3:     end if
4:     typeParams = td.getTypeParameters()
5:     RESOLVETYPEPARAMETERS(typeParams)
6:     Remove typeParams
Algorithm 10 The ResolveGenerics algorithm: resolveSingleGenericType Part 2

7: for all \( tr \in \{ tr \mid tr \text{ references } td \} \) do 
8: \[ \text{if } tr.\text{getASTParent}() == vd \land vd \text{ instanceof VariableDeclaration} \]
9: \[ \land vd \in td \text{ then} \]
10: \[ \text{RESOLVE SINGLE VARIABLE DECLARATION(vd)} \]
11: \[ \text{else if } tr.\text{getASTParent}() == md \land md \text{ instanceof MethodDeclaration} \]
12: \[ \land md \in td \text{ then} \]
13: \[ \text{RESOLVE METHOD RETURN TYPE(md)} \]
14: end if 
15: \[ \text{if } tr \text{ contains type arguments typeArgs then} \]
16: Remove typeArgs 
17: end if 
18: end for 
19: end function

4.3 Replacing Enhanced for Loops

The enhanced for loop can be used to iterate arrays or collections, thus, the transformation EnhancedFor2For has to take different actions for both situations.

If iterating over an array, a temporary array variable is created. The new for loop declares and initializes an index variable and tests in the condition whether the index is greater then the length of the temporary array variable. The declaration of the element of the array, first expression in the enhanced for loop, is moved inside the for loop statement block and assigned with the array element at the current index. The enhanced for loop is then exchanged with an statement block containing the temporary variable declaration and the new for loop. For a better understanding the following example in Listing 4.11 shows an enhanced for loop.

```java
String[] stringArray = { "this", "is", "a", "string", "array" };
for (String s : stringArray) {
    System.out.print(s + " ");
}
System.out.println();
```

Listing 4.11: Enhanced for loop over an array

Listing 4.12 shows the corresponding code after transformation.

```java
String[] stringArray = { "this", "is", "a", "string", "array" };
{
    java.lang.String[] a0 = stringArray;
    for (int i0 = 0; i0 < a0.length; i0++) {
        String s = a0[i0];
        System.out.print(s + " ");
    }
}
System.out.println();
```

Listing 4.12: Iterating over an array after transformation

In case of a collection, an iterator is used to iterate the collection. A new for loop is created, declaring an iterator of the corresponding type and initializing it with the invocation of iterator() on the variable specified in the enhanced for loop. The second expression of the for loop is set to the invocation of hasNext() on the newly created iterator. The third expression is left empty. As for arrays, the declaration of the element of the collection is moved into the for loop statement block. The enhanced for loop is exchanged by
the newly created for loop. An example is given in the following two listings, Listing 4.13 and Listing 4.14. The first listing shows the enhanced for loop before the transformation, the second the resulting transformed program.

```java
List<String> stringList = new ArrayList<String>();
...
for (String s : stringList) {
    System.out.print(s + " ");
} System.out.println();
```

Listing 4.13: Enhanced for loop over a collection

This will be transformed to:

```java
List<String> stringList = new ArrayList<String>();
...
for (java.util.Iterator<String> i0 = stringList.iterator();
    i0.hasNext(); ) {
    String s = i0.next();
    System.out.print(s + " ");
} System.out.println();
```

Listing 4.14: Iterating over a collection after transformation

One special situation the transformation has to take care of is the usage of conditional operators in enhanced for loops. An example is given in Listing 4.15.

```java
boolean list = true;
List<String> stringList = new ArrayList<String>();
Set<String> stringSet = new TreeSet<String>();
...
for (String s : list ? stringList : stringSet) {
    System.out.print(s + " ");
} System.out.println();
```

Listing 4.15: Enhanced for loop over a collection in combination with a conditional operator

In this case also for iterating over a collection a temporary collection variable is declared. It is initialized with the conditional operator. The for loop then iterates over the temporary collection variable. See the following example in Listing 4.16:

```java
boolean list = true;
List<String> stringList = new ArrayList<String>();
Set<String> stringSet = new TreeSet<String>();
...
{
    java.lang.Iterable<String> 10 = list ? stringList : stringSet;
    for (java.util.Iterator<String> i0 = 10.iterator(); i0.hasNext(); ) {
        String s = i0.next();
        System.out.print(s + " ");
    }
} System.out.println();
```

Listing 4.16: Iterating over a collection in combination with a conditional operator after transformation
Algorithm 11 provides a pseudo code representation of the EnhancedFor2For transformation.

Algorithm 11 The EnhancedFor2For algorithm

Require: Conditionals have been resolved earlier
Ensure: Enhanced-for-loops are resolved

1: for all enhancedFor instanceof EnhancedFor do
2:   guardType = getType(enhancedFor.getGuard())
3:   if enhancedFor.getGuard() instanceof ConditionalOperator then
4:     Set iteratorType to java.util.Iterator
5:     Copy type arguments from guardType
6:     conditional = true
7:   else if guardType instanceof ClassType then
8:     Create new MethodReference mr iterator() on guardType
9:     iteratorType = getType(mr)
10:    else if guardType instanceof ArrayType then
11:       iteratorType = null
12:   else
13:     throw new IllegalStateException
14:   end if
15: Create valid name for iteratorName
16: Create valid name for arrayName
17: if iteratorType == null then
18:   init = new local variable declaration of type int with name iteratorName
19:   guard = new expression checking iteratorName < arrayName.length
20:   update = new Increment on iteratorName
21:   firstStmt = local variable declaration storing arrayName[iteratorName]
22: else
23:   tmp = createTypeReference(iteratorType)
24:   if conditional then
25:     Create local variable tmp iteratorName = arrayName.iterator()
26:   else
27:     init = tmp iteratorName = collectionName.iterator()
28:   end if
29:   guard = method reference hasNext() on iteratorName
30:   firstStmt = local variable declaration storing iteratorName.next()
31: end if
32: stmt = statement block containing firstStmt and enhancedFor.getBody()
33: Create new for loop loop using stmt
34: if iteratorType == null || conditional == true then
35:   localVar = guardType arrayReferenceName = enhancedFor.getGuard()
36:   stmtBlock = statement block containing localVarDecl and loop
37:   Replace enhancedFor with stmtBlock
38: else
39:   Replace enhancedFor with loop
40: end if
41: end for
4.4 Resolving Boxing Conversions

The main idea of resolving Autoboxing, implemented in class `ResolveBoxing`, is to identify expressions that have class types where a primitive type would be expected, and the other way round respectively. The following example in Listing 4.17 shows some typical situations.

```java
Integer i = new Integer(3);
// trying to increment an Integer
i++;
// trying to put an int into an Integer
i = 3;
// trying to put 1 and an Integer on the same level
// and put it then into an Integer
i = (true ? 1 : i);
int k = 4;
// trying to put an Integer in an int variable
k = i;
// trying to call the getInt method with an int instead of Integer
k = getInt(k);
// trying to call the getInteger method with an Integer instead of int
i = getInteger(i);
// trying to use a switch statement with an Integer
switch (i) {
    case 1: { System.out.println("1"); break; }
    case 2: { System.out.println("2"); break; }
    default: { System.out.println("default"); break; }
}
```

Listing 4.17: Code using autoboxing

To explicitly box an expression the static `valueOf` method of the corresponding wrapper class is used. To explicitly unbox an expression the instance method `intValue()`, `byteValue` and respectively of the wrapper classes are used. Adding explicit boxing and unboxing will result in the code of Listing 4.18.

```java
Integer i = new Integer(3);
Integer.valueOf((i).intValue()++);
i = Integer.valueOf( 3);
i = Integer.valueOf( (true ? 1 : (i).intValue()));
int k = 4;
k = (i).intValue();
k = getInt(Integer.valueOf(k));
i = getInteger((i).intValue());
switch ((i).intValue()) {
    case 1: { System.out.println("1"); break; }
    case 2: { System.out.println("2"); break; }
    default: { System.out.println("default"); break; }
}
```

Listing 4.18: Code after transformation

As we can see in the example, there exist situations where it is needed to first unbox a class type and to box the result again. This is true for unary expressions. If an `Integer`
shall be incremented, it first has to be unboxed. The unboxed \texttt{int} is incremented and then again boxed into an \texttt{Integer}.

For some programmers it is furthermore convention, to cast primitive type variables to the type of the corresponding wrapper class. This will not work under Java 1.4. The transformation removes the cast and uses again explicit boxing and unboxing as described above.

A pseudo code representation of the transformation can be found in Algorithm 12.

\begin{algorithm}
\caption{The ResolveBoxing algorithm: Part 1}
\begin{tabular}{l}
1: \textbf{for all} \texttt{e instanceof Expression}, \texttt{parent = e.getASTParent() do} \\
2: \hspace{1em} \texttt{t = getType(e)} \\
3: \hspace{1em} \textbf{if} \texttt{parent instanceof MethodReference} \textbf{then} \\
4: \hspace{2em} \texttt{Set target type} \texttt{tt} \texttt{to the corresponding signature type} \\
5: \hspace{1em} \textbf{else if} \texttt{parent instanceof ConstructorReference} \textbf{then} \\
6: \hspace{2em} \texttt{Set} \texttt{tt} \texttt{to the corresponding signature type} \\
7: \hspace{1em} \textbf{else if} \texttt{parent instanceof ArrayInitializer} \textbf{then} \\
8: \hspace{2em} \texttt{Set} \texttt{tt} \texttt{to base type of array} \\
9: \hspace{1em} \textbf{else if} \texttt{parent instanceof Operator} \textbf{then} \\
10: \hspace{2em} \texttt{op = (Operator)parent} \\
11: \hspace{2em} \textbf{if} \texttt{op is a ternary, i.e., conditional operator} \& \texttt{e is not the condition} \textbf{then} \\
12: \hspace{3em} \texttt{tt = getType(op)} \\
13: \hspace{2em} \textbf{else if} \texttt{op is a binary operator} \textbf{then} \\
14: \hspace{3em} \texttt{target = getType(op)} \\
15: \hspace{3em} \textbf{if} \texttt{target instanceof PrimitiveType} \& \texttt{t instanceof ClassType} \\
16: \hspace{4em} \& \texttt{!(op instanceof Equals} \lor \texttt{op instanceof NotEquals}) \textbf{then} \\
17: \hspace{3em} \texttt{tt = target} \\
18: \hspace{2em} \textbf{else if} \texttt{target instanceof ClassType} \& \texttt{t instanceof PrimitiveType} \textbf{then} \\
19: \hspace{3em} \texttt{tt = target} \\
20: \hspace{2em} \textbf{end if} \\
21: \hspace{2em} \textbf{if} \texttt{!(op instanceof Assignment} \lor \texttt{op instanceof Equals} \\
22: \hspace{3em} \lor \texttt{op instanceof NotEquals}) \\
23: \hspace{3em} \& \texttt{getType(op.getExpressionAt(0)) instanceof ClassType} \\
24: \hspace{3em} \& \texttt{getType(op.getExpressionAt(1)) instanceof ClassType} \\
25: \hspace{3em} \& \texttt{no string concatenation} \textbf{then} \\
26: \hspace{3em} \texttt{Box both} \texttt{op.getExpressionAt(0)} \texttt{and} \texttt{op.getExpressionAt(1)} \\
27: \hspace{2em} \textbf{end if} \\
28: \hspace{2em} \textbf{else if} \texttt{op is unary operator} \& \texttt{!(op instanceof TypeCast} \\
29: \hspace{3em} \& \texttt{op instanceof Instanceof} \\
30: \hspace{3em} \& \texttt{op instanceof ParenthesizedExpression} \textbf{then} \\
31: \hspace{3em} \texttt{if} \texttt{t instanceof ClassType} \textbf{then} \\
32: \hspace{4em} \texttt{Unbox and then Box e} \\
33: \hspace{2em} \textbf{end if} \\
34: \hspace{2em} \textbf{else if} \texttt{op instanceof TypeCast} \textbf{then} \\
35: \hspace{3em} \texttt{if} \texttt{t instanceof ClassType} \\
36: \hspace{4em} \& \texttt{getType(op) instanceof PrimitiveType} \textbf{then} \\
37: \hspace{4em} \texttt{Remove type cast and unbox e} \\
\end{tabular}
\end{algorithm}
Algorithm 13 The ResolveBoxing algorithm: Part 2

38: \textbf{else if} $t$ \texttt{instanceof} \texttt{PrimitiveType} \\
39: \quad \wedge \texttt{getType(op)} \texttt{instanceof} \texttt{ClassType} \textbf{then} \\
40: \quad \text{Remove type cast and box} \ e \\
41: \quad \textbf{end if} \\
42: \textbf{end if} \\
43: \textbf{else if} parent \texttt{instanceof} \texttt{VariableSpecification} \textbf{then} \\
44: \quad \ tt = \texttt{parent.getType()} \\
45: \textbf{else if} parent \texttt{instanceof} \texttt{Switch} \textbf{then} \\
46: \quad \textbf{if} \ t \texttt{instanceof} \texttt{ClassType} \wedge \neg \neg (t \texttt{isEnumType}()) \textbf{then} \\
47: \quad \quad \text{Unbox} \ e \\
48: \quad \textbf{end if} \\
49: \quad \textbf{else if} parent \texttt{instanceof} \texttt{ArrayReference} \textbf{then} \\
50: \quad \quad \textbf{if} \ t \texttt{instanceof} \texttt{ClassType} \wedge \neg \texttt{t instanceof ArrayType} \textbf{then} \\
51: \quad \quad \quad \text{Unbox} \ e \\
52: \quad \textbf{end if} \\
53: \quad \textbf{else if} parent \texttt{instanceof} \texttt{Conditional} \lor \texttt{parent instanceof If} \textbf{then} \\
54: \quad \quad \textbf{if} \ e \text{ is first expression} \wedge \texttt{t instanceof ClassType} \textbf{then} \\
55: \quad \quad \quad \text{Unbox} \ e \\
56: \quad \textbf{end if} \\
57: \quad \textbf{end if} \\
58: \textbf{if} \ tt \neq \texttt{null} \textbf{then} \\
59: \quad \textbf{if} \ t \texttt{instanceof} \texttt{PrimitiveType} \wedge \ tt \texttt{instanceof} \texttt{ClassType} \textbf{then} \\
60: \quad \quad \text{Box} \ e \\
61: \quad \quad \textbf{else if} \ t \texttt{instanceof} \texttt{ClassType} \wedge \ tt \texttt{instanceof} \texttt{PrimitiveType} \textbf{then} \\
62: \quad \quad \quad \text{Unbox} \ e \\
63: \quad \textbf{end if} \\
64: \textbf{end if} \\
65: \textbf{end for}

4.5 Replacing Enums

The \texttt{ReplaceEnums} transformation replaces the enumeration types with classes according to the \texttt{Typesafe Enum} pattern \cite{Blo01}. But additionally to the methods \texttt{toString()}, \texttt{compareTo (Object o)} and \texttt{readResolve()}, the classes also implement the methods provided by the class \texttt{java.lang.Enum<E>} which is the common base class of all Java language enumeration types: \texttt{equals(Object other), getDeclaringClass(), hashCode(), name(), ordinal()} and \texttt{valueOf(Class<T> enumType, String name)}.

The \texttt{ReplaceEnums} transformation includes an inner class \texttt{ReplaceSingleEnum} that implements a two-pass transformation as well. The \texttt{ReplaceEnum} transformation creates for each enumeration type declaration an \texttt{ReplaceSingleEnum}, stores it and calls the \texttt{analyze()} method of it. In the transformation step accordingly the \texttt{transform()} method of the stored transformations is called. Therefore, the following section contains only a pseudo code representation of the \texttt{ReplaceSingleEnum} transformation.

Replacing Switch-Statements

The main idea of the transformation is simple: Replace the enum declaration with a new class declaration. The tricky part is to replace \texttt{switch} statements that are used in combination with enumeration types. Since no class types can be used in \texttt{switch} statements these have to be adapted. The first idea was to replace them by \texttt{if} statements. To simulate
the fall through behavior of a switch statement and the default statement two local help
variables were created. If not suppressed by a break, a switch statement will execute all
case statements after the first one, which condition has evaluated to true. If no condition
evaluates to true, the default statement will be executed. The variable fallThrough was
initialized with false. In a case statement it was set to true, if the last statement of
the statement block was not a jump statement, e.g., a break, otherwise to false. Each
condition for the nested if statement contained the original conditions of the case state-
ment connected with a logical AND to fallThrough. The second help variable doneAny
was initialized with false. In each if statement it was set to true. The last if state-
ment corresponding to the default statement was executed, if fallThrough == true
|| doneAny == false. An example is given in Listing 4.19.

```java
int i;
...
switch (i) {
    case 1: System.out.println("one");
    case 2: System.out.println("two");
    case 3: System.out.println("three");
    default: System.out.println("many");
}
```

Listing 4.19: Simple-falling through switch-statement

The example will output all statements from the first one which condition evaluated to
ture. For example for \(i = 2\) the output would be:
two
three
\(\text{many}\)

The if statement in Listing 4.20 simulates the behavior of the switch statement of List-
ing 4.19.

```java
int i;
...
boolean doneAny = false;
boolean fallThrough = false;
if (i == 1 || fallThrough) {
    System.out.println("one");
    doneAny = true;
    fallThrough = true;
} else if (i == 2 || fallThrough) {
    System.out.println("two");
    doneAny = true;
    fallThrough = true;
} else if (i == 3 || fallThrough) {
    System.out.println("three");
    doneAny = true;
    fallThrough = true;
} else if (!doneAny || fallThrough) {
    System.out.println("many");
}
```

Listing 4.20: If-statement corresponding to the falling through switch-statement

The next example in Listing 4.21 will show a little program containing breaks, where
the Switch-statement cannot fall through all statements.
```
int i;
...
switch (i) {
    case 1: System.out.println("one");
    case 2: System.out.println("two"); break;
    case 3: System.out.println("three");
    default: System.out.println("many");
}
```

Listing 4.21: Simple falling-through switch statement

The corresponding transformed code is shown in Listing 4.22.

```
int i;
...
boolean doneAny = false;
boolean fallThrough = false;
if (i == 1 || fallThrough) {
    System.out.println("one");
    doneAny = true;
    fallThrough = true;
} else if (i == 2 || fallThrough) {
    System.out.println("two");
    doneAny = true;
    // fallThrough has to be set to false instead of true
    // to simulate the break
    fallThrough = false;
} else if (i == 3 || fallThrough) {
    System.out.println("three");
    doneAny = true;
    fallThrough = true;
} else if (!doneAny || fallThrough) {
    System.out.println("many");
}
```

Listing 4.22: If-statement corresponding to the falling through switch-statement

This approach seemed to work fine until a change in class TypeArgument of the RECODER framework. The code shown in Listing 4.23 caused trouble.

```
String res;
switch (ta.getWildcardMode()) {
    case None: res = ""; break;
    case Any: return "?"; // shortcut
    case Extends: res = "? extends "; break;
    case Super: res = "? super "; break;
    default: throw new Error();
}
res += ta.getTypeName();
```

Listing 4.23: Switch-statement with uninitialized variable

After transformation the compiler would report an error that the variable res might not have been initialized. While for this switch statement the compiler is able to detect that either the variable gets initialized in one of the case statements or the program terminates this is not true for the if statement. This is due to the fact, that the help variables doneAny and fallThrough need to be tested before executing the else statement corresponding to the default statement.

Since we did not want to check for not initialized variables and initialize them with default values, we decided to adapt the switch statements in another way. We switch
now on the ordinal of the enumeration type. But since the ordinal is not static it is not possible to access it in a case statement for comparison. So we needed to add for each enumeration type constant a static int field. As name for the fields we took the name of the enumeration constants and added a prefix ENUMCONST_. The transformed switch statement will hence look like shown in Listing 4.24.

```java
String res;
switch (ta.getWildcardMode().ordinal()) {
    case recorder.abstraction.TypeArgument.WildcardMode.ENUMCONST_None :
        res = ""; break;
    case recorder.abstraction.TypeArgument.WildcardMode.ENUMCONST_Any :
        return "?"; // shortcut
    case recorder.abstraction.TypeArgument.WildcardMode.ENUMCONST_Extends :
        res = "? extends "; break;
    case recorder.abstraction.TypeArgument.WildcardMode.ENUMCONST_Super :
        res = "? super "; break;
    default: throw new Error();
}
res += ta.getTypeName();
```

Listing 4.24: Switch-statement with uninitialized variable after transformation

Like this the compiler will be able again to detect that either the variable gets initialized or the program terminates.

Since a small example for the typesafe enum pattern was shown in Listing 2.15 an example with a full enum declaration transformation is omitted. Algorithm 14 provides a pseudo code representation of the transformation.

**Algorithm 14 The ReplaceEnums algorithm: Part 1**

1: for all etd instanceof Enum do
2:     Create a new class declaration cd
3:     Add declaration specifiers of etd to cd
4:     Set identifier of cd to identifier of etd
5:     Add inheritance specification for Comparable and Serializable to cd
6: for all m member of etd do
7:     if m enum constant declaration then
8:         Create field declaration including specification with according type
9:         and name
10:        Store references to switch statements in switchStmts
11:        Add a static int constant representing the ordinal of this enum constant
12: else if m instanceof ConstructorDeclaration then
13:     Add Constructor to cd
14: else if m instanceof MethodDeclaration then
15:     Add method declaration to cd
16:     if m is abstract method then
17:         Add abstract modifier to cd
18: end if
19: end if
20: end for
21: if cd contains no constructor then
22:     Add new constructor with a name parameter with valid variable name
23: end if
Algorithm 15 The ReplaceEnums algorithm: Part 2

25: Check for internal switch statements. If the class declaration
26: contains a switch statement including an enum constant, take
27: care that the stored switch statement refers to the one in the
28: newly created class declaration and not the enum declaration
29: that will be replace.
30: Create method declarations and method bodies for the method
31: provided by java.lang.Enum<E>
32: Create field declarations ordinal and CURRENT_ORDINAL
33: for all \( sw \in \text{switchStmts} \) do
34: Copy the \( sw \)
35: Add method reference ordinal() to switch expression
36: Replace enum constants from case statements with references to
37: the static int constants representing the ordinal of the corresponding
38: enum constant
39: Replace \( sw \) the new switch statement
40: end for
41: Replace etd with cd
42: end for

4.6 Resolving Varargs

The basic idea of resolving varargs implemented in class ResolveVarargs replaces the last parameter type with the corresponding array type. If a varargs method is referenced, it checks whether the varargs feature is used, or whether an array is passed to the method. If no array is passed, a temporary array is created out of the single parameters passed to the method. Listing 4.25 will illustrate this approach:

```
public class VarArgs {
    public static int max(int... intArray) {
        int maxSoFar = Integer.MIN_VALUE;
        for (int i : intArray) {
            if (i > maxSoFar) maxSoFar = i;
        }
        return maxSoFar;
    }

    public static void main(String[] args) {
        int[] array = {5, 3, 7, 4, 9, 1};
        int[] array = {5, 3, 7, 4, 9, 1};
        max = max(new int[] {5, 3, 7, 4, 9, 1});
        max = max(array);
    }
}
```

Listing 4.25: Varargs method

Applying the described approach on this method declaration will resolve in the following code of Listing 4.26.

```
public class VarArgs {
    // int array as parameter instead of varargs
    public static int max(int[] intArray) {
        int maxSoFar = Integer.MIN_VALUE;
        for (int i : intArray) {
            if (i > maxSoFar) maxSoFar = i;
        }
    }
}
```

41

return maxSoFar;
}

public static void main(String[] args) {
    // pass array instead of variable number of arguments
    int max = max(new int[] { 4, 9, 6, 27, 0, 3 });
    int[] array = { 5, 3, 7, 4, 9, 1 };
    max = max(new int[] { 5, 3, 7, 4, 9, 1 });
    max = max(array);
}

Listing 4.26: Transformed varargs method

The transformation also takes care of references to varargs methods of the Java 5.0 API. Since our test projects most probably contain invocations of Java 5.0 API we adapted the transformation in a way that references to these methods are transformed as well. This enables us to compile the transformed program with a Java 1.4 compiler if the Java 5.0 Runtime Library is included. For a pseudo code representation of the transformation, please confer to Algorithm[16]

Algorithm 16 The ResolveVarargs algorithm: Part 1
1: for all md instanceof MethodDeclaration ∧ md.isVarArgMethod() do
2:    Store md in varArgMeths
3:    Store type of last parameter in lastParamTypes
4:    for all mr ∈ {mr references md} do
5:        if mr getArguments().size() == md.getSignature().size() ∧ last argument instanceof ArrayType then
6:            continue
7:        else Store mr in refs and md.getSignature() in sigs
8:    end if
9: end for
10: end for
11: for all cd instanceof ConstructorDeclaration ∧ cd.isVarArgMethod() do
12:    Proceed analogously to method declaration
13: end for
14: for all mr instanceof MethodReference
15:    ∧ !(m = getMethod(mr) instanceof MethodDeclaration) do
16:       if m.isVarArgMethod() then
17:           if mr getArguments().size() == m.getSignature().size() then
18:               if !(last argument instanceof ArrayType) then
19:                   Store mr in refs and m.getSignature() in sigs
20:               end if
21:           else
22:               Store mr in refs and m.getSignature() in sigs
23:       end if
24: end for
25: end for
Algorithm 17 The ResolveVarargs algorithm: Part 2

27: \textbf{for all} \( n \) instanceof \textit{New} \n
28: \quad \& \! (c = \textit{getConstructor}(n)) \textbf{instanceof} \textit{ConstructorDeclaration}) \textbf{do} \\
29: \quad \textbf{end for} \\
31: \textbf{for all} \( mr \) stored in \textit{refs do} \\
32: \quad \textbf{Copy the varargs arguments stored in} \textit{sigs} \textbf{to a list of expressions} \\
33: \quad \textbf{Create an array initializer with the list of expressions} \\
34: \quad \textbf{Create a new method reference as copy of the old one} \\
35: \quad \textbf{Remove the varargs argument and add the array initializer as argument} \\
36: \quad \textbf{Replace} \( mr \) \textbf{with the newly created method reference} \\
37: \textbf{end for} \\
38: \textbf{for all} \( md \) stored in \textit{varArgMeths do} \\
39: \quad \textbf{Create a copy of the} \( md \) \\
40: \quad \textbf{Create a parameter declaration with the array type of the original type stored} \\
41: \quad \textbf{in} \textit{lastParamTypes} \textbf{as type reference} \\
42: \quad \textbf{Set} \textit{varargs} \textbf{for the new method to} \textbf{false} \\
43: \quad \textbf{Replace the last parameter declaration with the newly created} \\
44: \quad \textbf{parameter declaration} \\
45: \quad \textbf{Replace} \( md \) \textbf{with the new method declaration} \\
46: \textbf{end for} \\

4.7 Removing Static Imports

The transformation to remove static imports is implemented in \textit{RemoveStaticImports}. Static import statements get removed and the referenced static fields have to be exchanged by references with a fully qualified name. An example for static imports is given in Listing 4.27.

\begin{verbatim}
import static java.lang.Math.*;

public class StaticImports {
    public double radian(double angle) {
        return Math.cos(Math.PI * angle);
    }
}
\end{verbatim}

Listing 4.27: Example for static imports

The transformation replaces the references with fully qualified names, as shown in Listing 4.28.

\begin{verbatim}
public class StaticImports {
    public double radian(double angle) {
        return Math.cos(Math.PI * angle);
    }
}
\end{verbatim}

Listing 4.28: After removing static imports

A pseudo code presentation of the \textit{RemoveStaticImports} transformation is given in Algorithm 18.
Algorithm 18 The RemoveStaticImports algorithm

1: for all cu instanceof CompilationUnit do
2:     for all im ∈ cu.getImports() ∧ im.isStaticImport() do
3:         Store im in statics
4:     end for
5:     for all pe instanceof MemberReference ∧ pe instanceof ReferenceSuffix
6:         ∧ pe instanceof NameReference do
7:         if the reference suffix has a reference prefix then
8:             continue
9:         end if
10:         Get target type
11:         if target type instanceof TypeDeclaration and declared in cu then
12:             continue
13:         end if
14:         Store name reference and corresponding static import in hotSpots
15:     end for
16:     Sort name references, inner type references occur first
17:     Remove all static imports stored in statics
18:     for all name references stored in hotSpots do
19:         Replace the name reference with a fully qualified name reference
20:     end for
21: end for

4.8 Removing Annotations

Annotations cannot be simulated in previous releases of Java. To get a compiling Java program for previous releases, one has to remove all annotations and annotation type declarations. This is done in the transformation RemoveAnnotations. Annotations can be queried from runtime using the Java 5.0 API. If this is used in a program, it won’t work after the transformation. Since the program won’t work after the transformation anyway, it is okay to remove all import statements referring to annotations without further testing whether they are used or not.

There is only one situation where it is not sufficient to just remove an annotation type declaration. An annotation type declaration is a special kind of interface. This means that another class can implement an annotation type. In this case instead of simple removing the annotation type declaration, it has to be replaced by a normal interface. The annotation elements are replaced by method declarations. In this step information about the default values of annotation elements gets lost.

```java
public @interface A { ... }

public interface B extends A { ... }
```
Algorithm 19 The RemoveAnnotations algorithm

```java
for all aus instanceof AnnotationUseSpecification do
  Detach aus
end for
for all atd instanceof AnnotationTypeDeclaration do
  if atd is used outside of annotation use specifications or includes a type declaration then
    Create interface declaration id
    Copy member, identifier and declaration specifiers of atd to id
    Replace atd with id
  else
    Detach atd
  end if
end for
```

4.9 Making Conditionals compatible

The transformation MakeConditionalCompatible casts the expressions of conditional operators to their common supertype to make them compatible with prior Java releases. The following example in Listing 4.29 will show how the transformation handles nested conditional operators.

```java
public class ConditionalWithIntersectionTypes {
  public void testConditionals() {
    List<String> stringList = new ArrayList<String>();
    Set<String> stringSet = new TreeSet<String>();
    Queue<String> stringQueue = new PriorityQueue<String>();
    boolean list = false;
    boolean set = true;
    ...
    Iterable<String> coll = list ? stringList :
      set ? stringSet : stringQueue;
  }
}
```

Listing 4.29: Example for nested conditional operators with intersection types

What will be the expected result of the transformation. The transformation starts with inner conditional operators. So the expression set ? stringSet : stringQueue will be transformed first. The most special common supertype of stringSet and stringQueue fitting to the requested type is Iterator<String>. So the two expressions of the inner conditional operator will be casted to this type. The outer conditional operator will not get any further casts, since the type of stringList (List<String>) is a narrowing of Integer<String> and, thus, no further casting is needed. The result of the transformation will accordingly look like shown in Listing 4.30.

```java
public class ConditionalWithIntersectionTypes {
  public void testConditionals() {
    List<String> stringList = new ArrayList<String>();
    Set<String> stringSet = new TreeSet<String>();
    Queue<String> stringQueue = new PriorityQueue<String>();
    boolean list = false;
    boolean set = true;
    ...
    Iterable<String> coll = list ? stringList :
      set ? stringSet : stringQueue;
  }
}
```

45
Iterable<String> coll = list ? stringList : set ?
  (java.lang.Iterable<String>) stringSet :
  (java.lang.Iterable<String>) stringQueue;
}
}

Listing 4.30: Conditional operator with intersection types after transformation

A pseudo code representation of the MakeConditionalCompatible transformation is
given in Algorithm 20.

Algorithm 20

The MakeConditionalCompatible algorithm

1: for all \( c \) instanceof \( \text{Conditional} \) do
2: if \( c \) is a nested conditional then
3: Set \( c \) to innermost conditional
4: end if
5: \( t = \text{getType}(c) \)
6: \( e1 = c\.getExpressionAt(1) \)
7: \( e2 = c\.getExpressionAt(2) \)
8: \( t1 = \text{getType}(e1) \)
9: \( t2 = \text{getType}(e2) \)
10: if \( \text{t instanceof Intersectio}nType \lor (t1 != t2 \land t1 \text{ and } t2 \text{ have no supertype} \)
11: or widenin relation) then
12: Calculate target type of conditional
13: Cast \( e1 \) and \( e2 \) to target type
14: end if
15: if \( c \) is a nested conditional then
16: Repeat till outermost conditional
17: end if
18: end for

4.10 Removing Covariant Return Types

The basic idea of removing covariant return types, implemented in class RemoveCovariantReturnTypes, is to exchange all covariant return types with the original return type of the superclass and to insert explicit type casts if a method of a subclass is called.

Given a method declaration that overrides a method, traversing up the AST and comparing the return types of the superclass’ method will identify covariant return types. This includes information present in source code as well as in byte code. Listing 4.31 shows an example for covariant return types.

```java
public class SuperClass {
    public List<String> getElements() {
        ...
    }
    ...
}

public class SubClass extends SuperClass {
    public ArrayList<String> getElements() {
        ...
    }
    ...
}
```

46
Listing 4.31: Example with covariant return types

Listing 4.32 shows how the code from Listing 4.31 gets transformed.

Listing 4.32: Same program after the transformation

But this transformation does not hold for all situations. In the following we will give a small overview of some problems that have to be taken in consideration. They are all associated with generics. The easiest solution would be to resolve generics before applying this transformation. Because the transformations are supposed to be as flexible as possible, we implemented the transformation in such a way that it can handle generics as well.
Inserting type casts containing type arguments
As we could see in the example above (Listing 4.32), the transformation inserts explicit type casts containing type arguments. These type arguments could refer to a class type not known to the current class. There are two alternatives to solve that problem. Either add missing import statements or exchange the simple type names in the type arguments with fully specified type names. We implemented the second approach.

Inheriting Wildcards
Since ClassType<SuperType> is not a supertype of ClassType<SubType> and, thus, not applicable for covariant return types, one might think that this situation that a type argument’s type is unknown could only occur in the explicit type cast and not in the substitution of the return type itself. But in fact there is a situation where it does occur. ClassType<? extends SuperType> is a supertype of ClassType<SubClass> and so they can be used as covariant return types and the return type will be set to ClassType<? extends SuperType>. SuperType might not be known to the current class. Again we have the two possible solutions: inserting missing import statements or using fully specified names for type argument. Again the second approach was implemented. In Listing 4.33, an example taking from RECODER itself shows the described situation.

```java
public interface ClassType {
    ...
    public java.util.List<? extends TypeParameter> getTypeParameters() {
        ...
    }
    ...
}

public class ClassDeclaration implements ClassType {
    ...
    public java.util.List<TypeParameterDeclaration> getTypeParameters() {
        ...
    }
    ...
}
```

Listing 4.33: Inheriting wildcards

Applying the transformation will result in the following code, shown in Listing 4.34.

```java
public interface ClassType {
    ...
    public java.util.List<? extends TypeParameter> getTypeParameters() {
        ...
    }
    ...
}

public class ClassDeclaration implements ClassType {
    ...
    public java.util.List<? extends reconder.abstraction.TypeParameter> getTypeParameters() {
        ...
    }
    ...
}
```

Listing 4.34: Same program after transformation
Inheriting from generic Classes

Sometimes it occurs that generic classes are inherited as non-generic classes. You cannot exchange the return types of the subclass with the type parameters of the superclass but you must stick to the new return types. An example can be found in RECODER itself: In class AbstractIndex, the non-generic internal class Enumerator extends the generic java.util.Enumeration interface. Listing 4.35 shows the example.

```
public interface Enumeration<E> {
    ...
    public E nextElement() {
        ...
    }
    ...
}

public class Enumerator implements Enumeration {
    ...
    public Object nextElement() {
        ...
    }
    ...
}
```

Listing 4.35: Inheriting from a generic classtype

In this example the transformation should leave the code as it is. It makes no sense to change the return type in the subclass to java.util.Enumeration.E. This means that one cannot just go through all method declarations, get their redefined methods and check whether the return types are different but you also have to check whether the return type of the superclass is a type parameter and find the uppermost return type in the class hierarchy that is not.

Another Problem occurs in association with generics. If covariant return types occur between a hierarchy of generic classes, the real instance of the type parameter must be known before inserting an explicit type cast. So the real type of the expression, e.g., VariableReference, MethodReference or ParenthesizedExpression, must be known before creating the new type cast expression.

Multiple Inheritance

Another problem occurs if covariant return types are used on conjunction with multiple inheritance. Listing 4.36 shows an example from the FindBugs project.

```
public interface INullnessAnnotationDatabase {
    public abstract NullnessAnnotation getResolvedAnnotation(
        final Object o, boolean getMinimal);
    ...
}

public class AnnotationDatabase {
    public AnnotationEnumeration getResolvedAnnotation(
        Object o, boolean getMinimal) {
    }
    ...
}

public class NullnessAnnotationDatabase extends AnnotationDatabase implements INullnessAnnotationDatabase {
    public NullnessAnnotation getResolvedAnnotation(
        final Object o, boolean getMinimal) {
```
The types `INullnessAnnotationDatabase` and `AnnotationDatabase` declare both the method `getResolvedAnnotation(Object o, boolean getMinimal)`. But the return types are different. The return type in interface `INullnessAnnotationDatabase` is `NullnessAnnotation` which is a subtype of `AnnotationEnumeration` used in class `AnnotationDatabase`. The class `NullnessAnnotationDatabase` extends or respectively implements those two types. Since the return type `NullnessAnnotation` is compatible with both return types declared in its supertypes this is a legal covariant return type. The transformation will detect this covariant return type and replace the return type of the methods in `INullnessAnnotationDatabase` and `NullnessAnnotationDatabase` with `AnnotationEnumeration` since this is the most general type which is consistent for all three type declarations. On the other hand in class `TypeQualifierNullnessAnnotationDatabase` no covariant return type is identified and nothing is changed. Because at this stage it is not known, that the return type of `INullnessAnnotationDatabase` will change. This will cause an incompatible return type error after transformation. So when adapting the return types in the supertypes all methods redefining those have to be identified and transformed as well. Thus, the transformed example would look like shown in Listing 4.37. For sakes of simplicity the normal type name `AnnotationEnumeration` has been used. The transformation would insert fully qualified names.

```java
public class TypeQualifierNullnessAnnotationDatabase implements INullnessAnnotationDatabase {
    public AnnotationEnumeration getResolvedAnnotation(//
        Object o, boolean getMinimal) {
        ...
    }
    ...
}
```

Listing 4.36: Example for covariant return type in multiple inheritance.
Object o, boolean getMinimal) {
  ...
}
...

Listing 4.37: After resolving covariant return types in multiple inheritance

Algorithm 21 gives a pseudo code presentation of the RemoveCoVariantReturnTypes transformation.

Algorithm 21 The RemoveCoVariantReturnTypes algorithm
1: for all md ∈ { md instanceof MethodDeclaration ∧
2: !(md.getMethodType() instanceof PrimitiveType) } do
3:   Get all redefined methods
4:   Get recursive original return type
5:   if original return type != present return type then
6:     Create type references for present return type and original return type
7:     if original return type instanceof ParameterizedType then
8:       Use fully qualified type names for type arguments
9:     end if
10:    Replace the present return type reference with the original return type reference
11:   end if
12: for all mr ∈ { mr references md } do
13:   Cast mr to the present return type, use qualified name
14: end for
15: end if
16: end for
4.11 Dependencies among the transformations

Figure 4.2 displays dependencies between transformations. As we can see, most transformations work independent of each other, thus, can be used to resolve single language features. There are only two strong dependencies between resolving enhanced-for loops and conditionals, and resolving generics and enhanced-for loops. These are described in the following.

![Diagram showing dependencies between transformations](image)

Figure 4.2: dependencies between the transformations

4.11.1 EnhancedFor2For to MakeConditionalCompatible

As shown in Listing 4.15, the guard of an enhanced for can be a conditional operator. The transformation EnhancedFor2For cannot handle input code containing conditional operators with intersection types. The problem with intersection types is that they cannot be declared explicitly. So when creating a temporary variable for storing the array or the collection in, it cannot be declared as an intersection type. It would probably be possible to resolve this dependency by taking the most special supertype of the intersection type. But, since this is basically what MakeConditionalCompatible does, it is not desirable to do. This would result in redundant code that has to be maintained in parallel. Listing 4.38 shows an example of an enhanced for loop including a conditional with intersection type.

```java
Set<String> stringSet = new TreeSet<String>();
Queue<String> stringQueue = new PriorityQueue<String>();
boolean set = true;
...```

52
for(String : string Set : string Queue) {
    ...
}

Listing 4.38: Enhanced for loop including conditional with intersection type

4.11.2 ResolveGenerics to EnhancedFor2For

Since the enhanced for loop is, besides arrays, mainly developed for collections, it is easy to see that there exists a dependency between ResolveGenerics and EnhancedFor2For. Since during resolving generics all type arguments are removed, the compiler can only guarantee that an object retrieved from a collection is of type Object. So in an enhanced for loop, the type of the collection is unknown and cannot be used in the first expression of an enhanced for loop. The following example declares a collection without type parameter as it would be after resolving generic types. Then it tries to iterate through the collection using an enhanced for loop, as shown in Listing 4.39.

```java
List strings = new ArrayList();
strings.add("hallo");
for(String s : strings) {
    System.out.println(s);
}
```

Listing 4.39: Dependency between EnhancedFor2For and ResolveGenerics

Line two would cause a warning (Type safety: The method add(Object) belongs to the raw type List. References to generic type List<E> should be parameterized). Line three would result in an error (Type mismatch: cannot convert from element type Object to String). The main problem with the enhanced for loop is that it hides the iterator. So it is not possible to cast references to the iterator to the right type. Only after resolving enhanced for loops is the iterator explicit present. Its type arguments can be removed and references like next() can be assigned with a cast.

According to that, the ResolveGenerics transformation would have to implement its own approach to handle enhanced for statements to avoid this dependency. This would, again, result in redundant code and is, thus, not desirable.
5 Evaluation

In this section, we evaluate the implementation of the transformations. We will first state the results of transforming RECODER and the FindBugs project. This includes information about runtime and size of the projects. Furthermore, we will give a brief outline of bugs identified in the RECODER framework during the implementation of the transformations. In the end of this section, we will compare our project to the downgrading tools introduced in Section 2.2.

5.1 Testing

The current implementation of the transformations transforms the RECODER framework successfully. The transformed project is able to compile and the testsuite of the transformed project, consisting of 68 test cases, returns the same results as the testsuite on the original project.

Also the FindBugs project can be transformed successfully. The transformed version of FindBugs is able to compile and the test cases succeed.

For testing, the transformed projects were imported into Eclipse. To compile the projects the compiler was set to the Java 1.4 Compiler. Nevertheless, the runtime library included in the projects was the Java 5.0 Runtime Library. This was inevitable because right now invocations to the new Java API are ignored.

Since both projects are quite large we can assume that in most cases the transformations work. But unfortunately, not all language constructs are used in RECODER and FindBugs. The conditional operator with intersection time is neither present in the REDCODER framework nor in the FindBugs project. We could only test the transformation on our small test instances. Thus, it is desirable to apply the transformations on more projects to make sure all language constructs are tested exhaustively.

5.2 Performance

Performance was only secondary in this project, since usually the transformations have to be applied only once. But still runtime should be feasible. Since performance was not a primary issue it can probably be improved. Table 5.1 provides information on the runtime of the transformations. Runtime includes parsing the input project and building up the program model as well as writing back the program model to source code. Tests were performed on an 1.67 GHz PowerPC G4, Mac OS X 10.4.11. Please note, that the running time of the single transformations not solely depends on the size of the input program but also on the number of items to be transformed, e.g., the number of enhanced-for loops in an input program. Therefore, Table 5.1 gives also an overview of the number of items performed during the transformation of RECODER and FindBugs.

An explanation for the single abbreviations used in Table 5.1 is given in the following:

- The transformation resolving covariant return types (CoVar) replaces decl many return types and introduces casts many type casts.
- The transformation resolving enhanced for loops (EnFor) replaces enFor enhanced for loops.
- The transformation that removes static imports (StImp) removes impSt static import statements and adapts refs references.
- The transformation responsible for resolving enumeration types ( Enums) replaces decl many enumeration type declaration with a class type declaration.
<table>
<thead>
<tr>
<th></th>
<th>Recoder</th>
<th>FindBugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes time (min:sec)</td>
<td>408:1:56</td>
<td>999:5:44</td>
</tr>
<tr>
<td>CoVar decl</td>
<td>170</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>casts</td>
<td>179</td>
</tr>
<tr>
<td>EnFor enFor</td>
<td>157</td>
<td>785</td>
</tr>
<tr>
<td>StImp impSt refs</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>Enums decl</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Vararg decl refs</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Annot annotUseSpec</td>
<td>133</td>
<td>2050</td>
</tr>
<tr>
<td></td>
<td>usedAnnotTypeDecl</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>unusedAnnotTypeDecl</td>
<td>0</td>
</tr>
<tr>
<td>Generics casts</td>
<td>5515</td>
<td>25086</td>
</tr>
<tr>
<td></td>
<td>param</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>typeParam</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>TAD / TPD</td>
<td>3382</td>
</tr>
<tr>
<td>Boxing boxed</td>
<td>0</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>unboxed</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>replacedCasts</td>
<td>0</td>
</tr>
<tr>
<td>Cond cond</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Transformations performed

- The transformation resolving vararg methods (Vararg) adapts decl many method declarations and refs many method references referring to a vararg method.

- During the execution of the transformation removing annotations (Annot), annotUseSpec annotation references are removed. Furthermore, usedAnnotTypeDecl + unusedAnnotTypeDecl annotation type declarations are found. Used annotation type declarations are those that include further type declarations or, are used in an inheritance specification. These are replaced with an interface declaring methods for all members of the annotation type specification but without the default values. All unused annotation type declarations are removed.

- Resolving Generics (Generics) introduced casts many type casts. Furthermore, it adapts param many type references in parameter declarations when a method overrides a method with type parameters in its signature. The transformation replaces typeParam many type parameters by their upper bound and removes TAD / TPD many type argument or type parameters.

- In our input projects, boxed expressions are boxed to their wrapper class type, and unboxed expressions are unboxed to their corresponding primitive type. Furthermore, the transformation resolving boxing conversions (Boxing) replaces replacedCasts with the according boxing or unboxing statement.

- The transformation resolving conditional operators with intersection types (Cond) replaces cond many conditional statements.
5.3 Comparing our Results to related Work

We will compare our work to the projects described in Section 2.2. We do not consider the following projects for the following reasons:

**backport-util-concurrent**
Focuses on API support which is not part of this master’s thesis.

**EEL-Project**
Follows the same approach as for example Retroweaver but supports less features. The only advantage is that it is embedded in the Eclipse IDE. So, comparing our project to Retroweaver is sufficient.

**Declawer**
Follows the same approach as we do but supports significantly less features than we do.

One disadvantage of our project is that invocations to the Java 5.0 API are not supported so far. (To run the transformed projects we have to include the Java 5.0 runtime library as explained in Section 3.5.) Furthermore, we cannot guarantee that code using reflection on generics or annotations will compile or will have the right behavior after transformation.

Our advantages are the fact that the resulting transformed code can be handled as normal code. Debugging is simplified and the user can maintain the transformed code together with his own code.

If we compare these facts to the other projects we can see that our project can easily compete with Retroweaver. This tool has also no support for the Java 5.0 API and reflection on generics and annotations. Furthermore, it does not support covariant return types which we do. If we take a look at JBossRetro we have the problem that no exact information on supported features can be found. Thus, the only issue JBossRetro surely supports which is not supported by our project is full Java 5.0 API support. But at the same time JBossRetro cannot handle anonymous implementation of abstract methods in enumeration types which we can. In contrast to JBossRetro Retrotranslator has no full API support. But it can be extended easily if something is missing. It supports all Java language features as well as reflection on generics and annotations. Furthermore, it provides an enhanced version of the Java 1.4 collection framework. Especially, the support of reflection makes Retrotranslator special.

So it seems that the main drawback of our approach is the lack of API support. But it should be no problem to add it in the future. It should be possible to partly downgrade the Java 5.0 API. Furthermore backports from Retrotranslator can be used and be adapted for our purposes. In contrast to API support reflection support on a source code based approach is probably infeasible. Resolving generics ports generic types back to their raw types and introduces casts where needed. This means that there is no information about the actual types available at runtime. The same holds for annotations. Our transformation just remove all annotations and annotation use specifications. So there is no information about annotations available after transformation.

We conclude that for projects that include no reflection our transformations can compete with the other projects. If API support is added it is a really alternative to the bytecode based tools. For projects containing reflection Retrotranslator is probably the only solution up to now.
5.4 Identified Bugs of RECODER
As already mentioned in Section 1.4, several bugs in the RECODER framework have been identified and resolved, resulting in a totally revised RECODER API and the new release of RECODER 0.90. The following section outlines those bugs that have been identified while implementing the transformations. For more information confer to the bug tracker of the RECODER webpage [rec08].

[1965975] Endless Loop when querying Methods of a Parameterized Type
The method getMethods(ClassType ct) of class DefaultSourceInfo would cause an endless loop if queried for a parameterized Type, which is declared in byte code.

```java
DefaultCrossReferenceSourceInfo ci = getCrossReferenceInfo();
TypeReference tr;
...
ClassType ct = (ClassType) ci.getType(tr);
List<Method> mList = ci.getMethods(ct);
```

If `ct` is instance of `ParameterizedType` and declared in byte code the method `getMethods(ClassType ct)` of class `DefaultSourceInfo` will call result = `ct.getProgramModelInfo().getMethods(ct)`; Since `ct` is declared in byte code `getProgramModelInfo()` will return an instance of `DefaultByteCodeInfo`. Because `ct` is instance of `ParameterizedType` and not instance of `ClassFile` the `getMethods(ClassType ct)` method of class `DefaultByteCodeInfo` will again call `ct.getProgramModelInfo().getMethods(ct)`; resulting in an endless loop until the stack overflows. Figure 5.1 visualizes the situation more intuitively.

![Figure 5.1: Endless loop when querying getMethods(ct)](image)

[1996819] Covariant Return Type of clone() Method for Arrays
While in previous Java releases the `clone()` method for arrays returned `Object`, in Java 5.0 the return type of `clone()` method for arrays with element type `T` is `T[]`. 

57
String[] s = {"string", "array"};
String[] s2 = s.clone();

This is legal in Java 5.0 but illegal in Java 1.4.2 without a cast to String[]. Be mr
the method reference corresponding to s.clone(), then method getType(mr) should
return String[]. Unfortunately RECODER would return Object, despite it is supposed
to support Java 5.0.

[2000780] Problems resolving Return Type of generic Methods
The method getSourceInfo().getType(methodReference) returned null in case of a
generic method, e.g., the toArray() method of the collection framework.
An example can be found in class DefaultProjectFileIO of the RECODER frame-
work.
res = v.toArray(new String[v.size()]);
This caused trouble while resolving generic methods. The method itself could be trans-
formed but it was not possible to add the needed casts where the generic method was
referenced. It was not possible to query the real type instance of the generic method.

Assigning an Integer to a float Variable
Assigning an Integer to a variable of type float could not be parsed by RECODER. In the
following piece of code of FindBugs line 4 resulted in a parsing error. This assignment is
legal in Java 5.0 since the Integer can be unboxed to int and an int can be assigned to an
float variable without any problems.
float num = 0;
try {
    i++;
    num = Integer.valueOf(args[i]);
}
catch (NumberFormatException exc) {
    num = fontSize;
}

RECODER could not parse this because instead of unboxing the right-hand-side of the
expression, the left-hand-side got boxed, thus trying to assign a Integer to a Float.

Duplicate Declaration due to Caching Error
If a statement block contains a variable declaration and after the statement block another
variable with the same name and type is declared RECODER will parse this the first time.
But as soon as something is changed in this compilation unit and the program model gets
updated this will cause an AmbiguousDeclarationException, independent of whether
actually something changed or just got replaced by a clone of itself. This was due to a
caching error. After the first time parsing the later variable declaration is still in the cache.
If the parser then finds another variable declaration it outputs an error.
6 Conclusion and Future Work

In this section, we summarize the work performed. We then draw conclusions from it and present possible tasks and ideas for future work.

6.1 Summary
The problem addressed by this thesis were incompatibilities between Java 5.0 and older Java versions due to the extended language syntax, and the fact that some legacy applications that depend on older Java runtime environments and cannot be updated need to interact with applications written in newer Java versions. To enable the interaction the applications written in newer Java versions need to be downgraded. This thesis implemented an source code based approach to resolve incompatibilities due to the extended language syntax of Java 5.0. Therefore algorithms to resolve the nine new language features had to be designed and implemented. A test case had to be provided to apply the transformations on a given input project. To verify the correctness of the transformations they needed to be tested extensively. In order to do that at least two larger projects, which include test suites, had to be selected. The test suites were important to be able to test whether the behavior of a transformed project had been preserved.

6.2 Conclusion
We designed and implemented 9 source code transformations making Java 5.0 source code Java 1.4 compatible. These transformations are implemented in the java5to4 package of the RECODER kit. We implemented a test case that applies the single transformations to a given input project or to completely downgrade it. Tests on RECODER and FindBugs were successful, both could be transformed, compiled and included test cases succeeded. Thus, we were able to show that it is possible on a certain level to downgrade Java 5.0 source code to Java 1.4 compatible code. It is possible to downgrade single features, e.g., only resolve static imports, as well as to apply all transformations together, respecting dependencies.

There remain, however, some restrictions: It is not possible to guarantee that the transformations will work on every input code and result in compilable code. One example for that is the usage of reflection in combination with annotations. By simple transforming the source code it is not possible to simulate annotations, and, thus, it is not possible to query them during runtime. Programs containing this sort of code will not compile after transformation. Also programs reflecting on generic types can cause problems. These programs probably compile after transformation but the behavior can be changed. According to that our transformations are not applicable for these kind of programs. But reflections are a topic of its own and were not part of this work. Beside Retrotranslator no tool for downgrading Java 5.0 supports reflection on annotations or generic types. So, our transformations can compete with the other tools. For ordinary Java programs without reflections our transformations work fine and help the developer take advantage of new components written in Java 5.0 using the new language features.

6.3 Future Work
There are several obvious possibilities of future work in this project, some mentioned below.

6.3.1 Testing
Since the transformations have only been tested on two larger projects it is desirable to test it on further projects to minimize the possibility of remaining bugs. This holds
especially for those transformations that resolve language features that are not present or not extensively used in RECODER and FindBugs. Conditional operators in conjunction with intersection types, for example, neither occur in RECODER nor in FindBugs.

6.3.2 Performance
Since the performance of the transformations was just a secondary goal of this work, the implementation can probably be improved under this aspect.

6.3.3 Detect and Resolve Dependencies
The implementation of the transformations contains two strong dependencies. But maybe further testing will detect more dependencies. These should be resolved if possible. Also the dependency between the transformations MakeConditionalCompatible and EnhancedFor2For could probably be resolved.

6.3.4 API Support
The present implementation ignores invocations of the Java 5.0 Runtime Library and leaves it to the developer to resolve them. It would be nice to develop an automated approach for replacing Java 5.0 API invocations with code that is Java 1.4 compatible but achieves the same result. A possible solution would be to offer a framework that substitutes for the newer functionality and then replaces an invocation to the newer API with an invocations of a method of the substitute framework.
References


A Details on Java Language Features

A.1 Generic Types

Generics and Subtyping

The following section will outline some peculiarities of class hierarchies involving generic types. Examples will be used to explain what has to be taken into account when dealing with class hierarchies of generic types. Consider the following code:

```java
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

Line 1 is certainly legal. The question is if line 2 is legal, too. What is the supertype of `List<String>`? Is a `List` of `String` a `List` of `Object`? One might think so, but take a look at the following lines of code:

```java
lo.add(new Object());
String s = ls.get(0);
```

Line 4 attempts to assign an `Object` to a `String`. To prevent this from happening, line 2 will cause a compile time error. In general, if `T2` is a subtype (subclass or subinterface) of `T1` and `GenericType` is some generic type declaration, it is not true that `GenericType<T2>` is a subtype of `GenericType<T1>`. On the other hand if `T1` and `T2` are both generic type declarations and `T2` is still a subtype of `T1` then `T2<SomeType>` is subtype of `T1<SomeType>`.

Wildcards

The previous section rises the question what the supertype of all kinds of collections is. It is written `Collection<?>` (called "collection of unknown"), that is, a collection whose element type matches anything. So when replacing a method `printCollection(Collection c)` with a method using generics, it is wrong to replace it with `printCollection(Collection<Object> c)` since `Collection<Object>` is not a supertype of all kinds of collections. The correct replacement would be `printCollection(Collection<?> c)` which could look like this:

```java
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Listing 1.1: Using wildcards

This looks pretty promising but there are still restrictions. It is always safe to retrieve objects from `c`, since whatever the actual type of the collection is, it has to be a subtype of `Object`. Unfortunately it is not safe to add arbitrary objects to it:

```java
Collection<?> c = new ArrayList<String>();
c.add(new Object());
```

Line 2 will cause an compile time error. Since it is unknown what the element type of `c` actually is, it is not possible to add objects to it.

Bounded Wildcards

To have a closer look at this, consider a simple drawing application that can draw shapes such as rectangles and circles. These shapes could be represented by the following class hierarchy:
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { ... }
}

public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }
}

Listing 1.2: Simple drawing application

A normal drawing will typically contain a number of shapes. Assuming that they are stored in a list, it would be convenient to have a method in Canvas that could draw them all:

public void drawAll(List<Shape> shapes) {
    for (Shape s : shapes) {
        s.draw(this);
    }
}

Listing 1.3: Drawing the Shapes without wildcard

Because of the type rules explained above, the drawAll() method can only be called on lists of exact shapes, i.e., List<Shape>, but not on a List<Circle>. The intended idea was to have a method that accepts a list of any kind of shape:

public void drawAll(List<? extends Shape> shapes) { ... }

Now drawAll() will accept lists of any subclass of Shape. List<? extends Shape> is called a bounded wildcard. Again, the ? stands for an unknown type, but this time we know that this unknown type is a subtype of Shape. Shape is said to be the upper bound of the wildcard. Note that bounded wildcards still do not solve the problem of adding objects to a collection of unknown type. Thus, the following results in a compile time error.

public void addRectangle(List<? extends Shape> shapes) {
    shapes.add(0, new Rectangle());
}

Listing 1.4: Trying to add objects to a list of unknown type

Since the actual type of the list could also be a supertype of Rectangle, it is not safe to pass a Rectangle there. There are lots more things to learn about wildcards, e.g., type parameters can be bounded too, type arguments and parameters can have lower bounds using the keyword super. For more details, please confer to the generics tutorial [Bra04].
### A.2 Annotations

#### Predefined Annotation Types

<table>
<thead>
<tr>
<th>Annotation Type</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
</table>
| `@Deprecated`   | Used to designate classes, attributes or methods that should not be used anymore. The compiler will prompt a warning, if a so marked element is used. It is helpful to add furthermore a javadoc comment, explaining how to substitute the corresponding element. | ```
``` |
| `@Override`     | The Override annotation type is used to mark methods that override methods from a supertype. The compiler will check whether the supertype actually contains the method and if not prompt an error. | ```
``` |
| `@SuppressWarnings` | This annotation type can be used to suppress specified warnings. | ```
``` |
<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Documented</td>
<td>This is a meta-annotation. It is used to specify that a new created annotation type is considered by javadoc when creating the documentation.</td>
</tr>
</tbody>
</table>
| @Documented | ```java
@interface ClassPreamble {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String lastModifiedBy() default "N/A";
    String[] reviewers();
}
``` |
| @Inherited | This meta-annotation indicates that a newly created annotation type is automatically inherited. An annotation used for a class will also be valid for all the subclasses of this class. If an annotation type, with an Inherited meta-annotation on its annotation type declaration, is queried on a class type and the class declaration has no annotation for this type, the class’s superclass will automatically queried for the annotation type. This continues until an annotation for the specified annotation type is found, or the top of the class hierarchy is reached. |
| @Retention | This meta-annotation indicates in what stages the newly created annotation type can be accessed. There are three possible values for an annotation of this type, listed in the enumeration java.lang.annotation.RetentionPolicy. |
| **SOURCE** | Annotation types annotated with an Retention annotation with value RetentionPolicy.SOURCE are to be discarded by the compiler. |
| **CLASS** | If annotated with the value RetentionPolicy.CLASS annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time. |
| **RUNTIME** | Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively. |
@Target

This meta-annotation indicates on what program elements a newly created annotation types is applicable. The possible values for this annotation are listed in the enumeration `java.lang.annotation.ElementType`.

<table>
<thead>
<tr>
<th>Annotation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANNOTATION_TYPE</td>
<td>Annotation type declaration</td>
</tr>
<tr>
<td>CONSTRUCTOR</td>
<td>Constructor declaration</td>
</tr>
<tr>
<td>FIELD</td>
<td>Field declaration (includes enumeration constants)</td>
</tr>
<tr>
<td>LOCAL_VARIABLE</td>
<td>Local variable declaration</td>
</tr>
<tr>
<td>METHOD</td>
<td>Method declaration</td>
</tr>
<tr>
<td>PACKAGE</td>
<td>Package declaration</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>Parameter declaration</td>
</tr>
<tr>
<td>TYPE</td>
<td>Class, interface (including annotation type), or enum declaration</td>
</tr>
</tbody>
</table>

Table A.1: Predefined annotation types
B Glossary

Concrete Syntax
The common sequential representation of a program source.

Abstract Syntax Tree (AST)
The abstract syntax tree is a representation of the syntax and hierarchical structure of the source code. Each node of the syntax tree denotes a construct contained in the source code. The tree is called abstract since not all constructs that appear in the code might be represented. Some AST representations for example might omit grouping parentheses, since in an AST the grouping of operands is implicit in the tree structure.

Program Model
A model of a program. An AST is a syntactic program model.

Metamodel
A model of a model; the metamodel describes entities of a given base model.

Program Analysis
An algorithm that derives information about a program, usually by inspection of its sources.

Semantic Analysis
Program analysis that checks if a program source conforms to the language specification.

Two-pass transformation
A transformation containing of a distinct analyze phase and an transform phase.

Program Transformation
Syntactic change of a program, usually described in terms of the abstract syntax tree.

Metaprogram
A program changing another program; an implementation of a program transformation using an analysis to drive the transformation. Static metaprograms operate offline, while dynamic metaprograms change programs during runtime. For more information about metaprogramming, please confer to Section 2.3.1.