Implementing Traits in Java

Bart Groot
bmroot@cs.uu.nl
Student no. 9908277

MSc Thesis
INF SCR-04-50

Center for Software Technology,
Institute of Information and Computing Sciences,
Utrecht University,
P.O. Box 80089, 3508 TB,
Utrecht, The Netherlands.

23 December 2004
Preface

You are reading my doctoral thesis about adding ‘Traits’ to the programming language Java. Traits are a relatively new and promising concept in OO programming. It was in the end of 2003 that I chose this subject for my thesis. The subject sounded appealing, because I had an interest in Java programming since the beginning of my study of computer science. When I read the paper that introduced Traits [SDNB02], I found it a good idea to try and add it to Java. This idea was of Dave Clarke, who became my first thesis advisor. After he went to work elsewhere, Merijn de Jonge became my thesis advisor. Just before he went to work elsewhere, I finished my thesis.

Acknowledgements

In the first place, I would like to thank both my thesis advisors, Dave Clarke and Merijn de Jonge, for providing me with fresh views and critique on my work. Dave provided me with the idea, and with a lot of advice of how to implement it. Merijn helped me on more than one occasion to put me back on the proverbial track, when I had lost inspiration. He also provided me with much help at the actual writing of this thesis.

Although they probably do not know me, I would like to thank the creators of Polyglot, and especially Nate Nystrom, who helped me via the Polyglot mailing list. Next I would like to thank Arthur van Dam for creating the \LaTeX{} style, in which this thesis is written.

Of course, big thanks goes out to my parents for supporting me throughout my study. I would also like to thank my grandparents for the same reason. Special thanks goes out to my girlfriend, drs. Neeltje Schamp, who always believed in me and took me along on two holidays she had won, during this project. Also, thanks to her parents, who provided me with a change of scenery, by taking me along to study a week in the south of France. Finally, I would like to thank all the teachers and professors from which I took lectures at the University of Utrecht.
Contents

1 Introduction

2 Inheritance structures in OO
  2.1 Single Inheritance .............................................. 11
  2.2 Multiple Inheritance ............................................ 13
  2.3 Delegation ...................................................... 16
  2.4 Interface delegation ........................................... 17
  2.5 Mixins ........................................................ 18
  2.6 Traits .......................................................... 20
  2.7 Overview ....................................................... 21
  2.8 Other reuse structures ........................................ 22
    2.8.1 Aspect oriented programming ............................ 22
    2.8.2 Templates ................................................ 23
    2.8.3 Generics ................................................... 24
    2.8.4 Reuse variables .......................................... 24

3 Traits
  3.1 Language and graphical conventions .......................... 27
  3.2 Trait specification ............................................. 28
    3.2.1 Composing classes with Traits .......................... 28
    3.2.2 Composing Traits ......................................... 30
    3.2.3 Exclusion .................................................. 31
    3.2.4 Aliasing .................................................... 33
  3.3 Traits and state ............................................... 35
  3.4 Typing Traits .................................................. 37
  3.5 Summary of features .......................................... 39

4 Implementation
  4.1 Framework ..................................................... 41
  4.2 The JTL extension ............................................. 42
  4.3 Syntax ........................................................ 43
  4.4 Compilation scheme .......................................... 44
    4.4.1 Barrier passes ........................................... 45
    4.4.2 Modified passes ......................................... 45
    4.4.3 New passes ............................................... 46
Chapter 1

Introduction

In the Object Oriented (OO) programming paradigm, objects are used to package data and functionality. Objects are the basis of modularity and structure in an OO computer program. The content of objects is defined by classes. By using a class, objects can be constructed, which is called instantiation. Classes serve as abstract data types (ADTs) and define the functionality (i.e. methods and fields) of the instantiated objects.

An object has a interface, which consists of the methods and fields that are accessible to other objects. This allows a programmer to hide the information that is only used within the objects themselves, thus ensuring that other objects cannot change the internal state of an object in unexpected ways. This concept is called encapsulation. Classes are used to define the interface of objects.

Multiple objects can provide the same interface, even if they are instantiated with different classes. The objects are then said to conform to the same interface. The behavior of the objects may be different, even though the interfaces are the same. This allows multiple things to be interchangeable with each other, and is called polymorphism. For example, a Cow object and a Dog object may conform to the same interface, which allows them to make a sound. Obviously, their sounds will be different (lowing and barking), but it allows other objects that use objects that can make a sound to use both objects interchangeably.

Figure 1.1: An example of classes and objects
The use of objects also allows for abstraction, which allows programmers to focus on the essential parts of the information they are manipulating. Each object in a system serves as a model of an abstract ‘actor’ that can perform work, report on and change its state, and ‘communicate’ with other objects in the system, without revealing how these features are implemented.

In Figure 1.1 the classes Person and Student specify what functionality is present in instances of that type. The object John is an instance of Person, and therefore has a field name. The type of an object corresponds to the class of which it is an instance.

Classes are also used for inheritance, which is another important feature of OO programming. A class can inherit another class (called the superclass), by which it gains the functionality of the superclass. In Figure 1.1, Student has Person as its superclass. Inheritance so allows objects to be created that are specialized instances of the superclass, which share (and extend) their behavior, without having to reimplement that behavior. This means that the name field does not have to be defined again in Student, and that instances of Student are of type Student and Person. The object Bart is an instance of Student and therefore has fields name and student number. Not having to reimplement existing behavior is called code reuse. This is only possible by extending an existing class. One cannot for example reuse just one method of another class, and another method of another class. Classes are therefore also used as units of reuse.

Classes are used as ADTs and as units of reuse. These are two roles that are contradictory. As ADTs, classes should provide complete functionality to objects. Because of this, classes combine a number of different features into a class. We cannot reuse these features separately in other classes, because the unit of reuse is a class. With Multiple Inheritance, it is possible to split out the features over separate classes and have a class inherit all those, but a class is still not the appropriate element to reuse. After all, the role of classes as instance generators requires that each class has a unique place in the class hierarchy. Therefore, programmers are unable to reuse code at arbitrary places in the class hierarchy. For small class hierarchies, this is not problematic, but in large hierarchies it is, resulting in duplicate code. Existing approaches do not solve this problem, because they maintain the duplicate role of classes (e.g. Multiple Inheritance), or introduce a restrictive new way of class composition (e.g. Mixins).

The Trait construct [SDNB02] has been designed to relieve classes of their role as units of reuse, and thus allowing a better way of reuse. Traits are lightweight entities that serve as primitive units of code reuse in object-oriented languages. In a Trait, a group of methods can be defined, which can be used at arbitrary places in an inheritance hierarchy. In a programming language that supports Traits, classes can be composed from multiple Traits, as well as define their own methods, while still allowing class inheritance.

The Traits model has been designed in the context of Smalltalk [GR85], an untyped language. The aim of the project that this thesis describes, is to design, develop and explore Traits in the context of the Java programming language. But before rushing into the details of our Java Trait Language compiler, we will first look at existing inheritance models as we answer the question:

What is wrong with existing inheritance structures?

One might wonder if the Trait constructs is really needed in Java. We will try to convince the reader that Traits are a better solution to programming problems concerning inheritance
and code reuse than existing approaches.

Instead of keeping a ‘what if?’-attitude towards Traits in Java, we wanted to actually be able to use them. Therefore, we need to answer the question:

*How do we add Traits to Java?*

This brings us to the design and implementation of the Java Trait Language (JTL) and matching compiler. Now that we can actually use Traits in Java, we can answer the question:

*How do Traits improve Object Oriented programming in Java?*

By answering these research questions throughout this thesis, we try to show that the availability of Traits in Java (which is provided by our compiler) yields significant benefits over regular Java. Also, we show the shortcomings of existing OO inheritance structures and how Traits solve them.

We proceed as follows: In Chapter 2, we look at various reuse structures that (object-oriented) programming languages use. Traits is one of these. We compare these structures to get an overview of the benefits and disadvantages of each of the structures. In Chapter 3, we describe Traits and how they should be used within an existing programming language. In Chapter 4, we present the language extension to Java, JTL (Java Trait Language), which extends the Java language with Traits, and we describe how we implemented the compiler for this new language. In Chapter 5, we present and evaluate a case study of using JTL. We come back to the research questions posed here, indicate future work and conclude our findings in chapter 6.
Chapter 2

Inheritance structures in OO

In this chapter an overview of inheritance mechanisms in Object Oriented programming is given. Each inheritance mechanism has its advantages and disadvantages, which will be described in the corresponding section. Also, we will look at other reuse structures that are alternatives to inheritance.

2.1 Single Inheritance

In object-oriented programming, programmers create classes that provide methods and fields (which is called functionality) to objects. Single Inheritance (SI) allows a programmer to write a class that inherits functionality from another class, the superclass or parent class. By doing so, it gains the functionality of the inherited class. Classes can inherit from only one other class, but multiple classes can inherit the same class (multiple subclasses). This way, a tree of inheritance can be built up for program.

An example of SI is given in Figure 2.1. The class Person has a field name. There are two classes that inherit from the superclass Person, which make them subclasses of Person. Student, has a field student number and Employee has a field salary, and through inheritance both subclasses receive the name field from their superclass.

![Figure 2.1: Example of Single Inheritance]
2.2 Single Inheritance

SI has, of course, the benefits that inheritance in general has. Since SI is the simplest form of inheritance, it does not have additional benefits. These benefits are:

- With inheritance, code duplication can be avoided. A programmer could write one superclass that has some common functionality and several subclasses of that class that all may use that functionality. Without inheritance, the programmer would have to write the same code for the common functionality in all the classes. That is called code duplication, and is something we want to avoid. In the example of Figure 2.1, class Person has a name. When writing the classes Student and Employee without inheritance, the programmer has to write the field name in both new classes. When using inheritance, this is not necessary, because the field is inherited. This saves the programmer work. Of course, the reuse of fields is not the strongest example here. Much more code reuse is achieved when methods are reused.

- Another important advantage of inheritance is abstract interface conformance. The abstract interface of a class are the fields and methods that are available to other classes. Since an interface of a class is always abstract, the word ‘abstract’ is superfluous. It is only used here to avoid confusion with the Interface structure of Java. A programmer may write a class C that defines common methods. All classes that inherit from that class will have those methods, and, where the type system expects an instance of C, any subclass (or even sub-subclass) of C may be used instead. The implementations of methods may be changed (i.e. overridden) by the different subclasses of C, but the way a method is called (e.g. with different parameter types) cannot be changed, which ensures the conformance to the superclass’ abstract interface.

Looking back at the example of Figure 2.1 through inheritance both subclasses conform to the abstract interface of their parent class Person. Any method or class constructor that takes an instance of Person as a parameter, may also be passed an instance of Student or Employee, because they are also Persons. So, not only the code of the parent class is reused with SI, but also other code that does operations on instances of the parent class is in a sense reused.

Though it clearly has benefits, the mechanism is too limited for programs with complex class hierarchies. To give an example, we look back at the Figure 2.1. Suppose we want to add a new class Tutor, which represents a student that teaches other student, and gets paid for that. A Tutor is conceptually a subclass of both Student and Employee, but since a class can only inherit from one superclass, we have to make a choice. Choosing to make Tutor a subclass of Student, causes code duplication of the salary field and makes it impossible to use an instance of Tutor when an instance of Employee is required. Choosing to make Tutor a subclass of Employee, causes code duplication of the student number field and makes it impossible to use an instance of Tutor when an instance of Student is required. SI is forcing us to make a choice, which we do not want, because there is no right choice possible.

Due to this limitation, modern OO languages use more advanced inheritance mechanisms. E.g. Multiple Inheritance (C++) or Single Inheritance complemented with interfaces (Java). These are discussed in the upcoming sections.

\[1\] This is not true the other way around: A Person is not always an Employee.
2.2 Multiple Inheritance

The problem mentioned in the previous section can be solved by Multiple Inheritance (MI). In the MI model [Mey88], classes can inherit from more than one superclass. The problem of the Tutor class at the end of the previous section can easily be solved with MI. The new class Tutor can just inherit from both Student and Employee. As a result, there is no code duplication and an instance of Tutor can be used whenever an instance of Person, Student or Employee is required.

Thus, by allowing multiple superclasses, better code reuse can be achieved than with SI. However, this also brings along several problems.

The diamond problem. For one, when a class inherits from the same base class via multiple paths, the diamond problem occurs, which is illustrated by the following example. In the example a fictional programming language is used, which has the syntax of Java, but, contrary to Java, does support MI (it allows putting multiple classnames behind the extends keyword).

Suppose we have the following class hierarchy and code to match in a language that supports MI (we call this the Jurassic Park example, after the novel by Michael Crichton [Cri91]):

```
abstract class Animal:
  abstract void talk();

class Frog extends Animal:
  void talk() { System.out.println("Ribit, ribit."); }

class Dinosaur extends Animal:
  void talk() { System.out.println("I'm a dinosaur..."); }

class Frogosaur extends Frog, Dinosaur
```

Figure 2.2: Jurassic Park example

Inheritance structures in OO
2.2 Multiple Inheritance

This class Frogosaur could give unexpected behavior, because when someone tries to invoke talk() on a Frogosaur object, the problem arises that the compiler does not know which talk() method should be called: the method in Dinosaur or the method in Frog. In this example, there are only conflicting methods, which can be easily disambiguated by explicitly naming the class for which the ambiguous method is called. It is imaginable that our fictional language would allow something like this:

Listing 2.2: Disambiguating conflicting method calls by explicit referencing

```java
Animal animal = new Frogosaur();
((Dinosaur)animal).talk();
((Frog)animal).talk();
```

This allows access to both ways of talking. When there are conflicting default attributes, there is a problem. Frogs are not extinct, but dinosaurs are. A Frogosaur is both a frog and a dinosaur, so is it extinct or not?

Listing 2.3: MI example using state

```java
class Frog extends Animal:
    boolean isExtinct = false;
class Dinosaur extends Animal:
    boolean isExtinct = true;
class Frogosaur extends Frog, Dinosaur
```

Since isExtinct is either true or false, the state of Frogosaur objects is ambiguous. To solve this kind of ambiguity, either the attribute needs to be redefined in Frogosaur, which leads to code duplication, or some kind of order or priority needs to be present in the compiler, which brings along problems on its own. Ambiguity that involves state variables cannot be solved as easily as when there are only methods, like in Listing 2.1 [Sak93].

The fragile base class problem. Another problem with MI is known as the fragile base class problem. It occurs when overridden features need to be accessed. To access identically named inherited methods and fields unambiguously, the superclass that is meant needs to be explicitly named (like in C++). For instance, in the Jurassic Park example (Listing 2.1), if we would want the Frogosaur’s talk() method to combine the talk() methods of Frog and Dinosaur, we would have to write something like this:

Listing 2.4: Using explicit references

```java
class Frogosaur extends Frog, Dinosaur:
    void talk() {
        System.out.println(
            ((Frog)super).talk() + " and " + ((Dinosaur)super).talk());
    }
```

In a language with SI, the super keyword would transparently point to the parent class, but now we have to put references to the superclass in the source code, making the code vulnerable to changes in the class hierarchy. For example, suppose the Frog class from the Jurassic Park example is replaced with a new class Toad. Then not only does a programmer have to change the line with extends in Frogosaur, but also all the superclass references to Frog, which were necessary because of MI. Explicit references outside the inheritance
Multiple Inheritance 2.3

A method read
method write

SyncA
method read
method write
method acquireLock
method releaseLock

B method read
method write

SyncReadWrite
method read
method write
method acquireLock
method releaseLock

(I)
(II)

Figure 2.3: In (I), the synchronization code is put directly in the subclass of A. (II) shows the programmer’s attempt to factor out the synchronization code into SyncReadWrite, which failed because methods there cannot refer to the read and write methods of A and B.

tree are also possible (as in Listing 2.2), which also need to be changed. This can mean a lot of extra work, which would not be necessary if there were no explicit references.

Limited compositional power. In addition to the problem with conflicting features and the problem with accessing overridden features, the MI model has limited compositional power. MI does not allow one to write a reusable entity that both uses and exports adapted forms of method implemented in unrelated classes, unlike Mixins and Traits (see Section 2.5 and 2.6).

The following example is illustrated in figure 2.3. Suppose we have a class A that reads and writes a data source. In (I), the programmer creates subclass of A with synchronized reading and writing, called SyncA. This class overrides the read and write methods of A. The new implementation of read first acquires a lock, then calls the original read method from A and finally releases the lock. The same goes for write.

There is also a class B that has unsynchronized reading and writing, which also needs to be synchronized. In (II), the programmer wants to factor out the synchronization code of SyncA, so that it can be used in both SyncA and SyncB. To do this, the programmer needs to put the synchronization code in a new class SyncReadWrite, that becomes a superclass of both SyncA and SyncB. However, it is not possible for a superclass to explicitly refer to a method such as read that a possible subclass will inherit from another superclass. It could be done implicitly with abstract methods read and write in SyncReadWrite that will eventually be implemented by a subclass. However, to do that, there need to be unsynchronized reads and writes available in SyncA and SyncB objects, and the whole point is that these should not be available there! This means that factoring out the synchronization code is a problem when using MI. Figure 2.3 illustrates this example. In Section 2.6, the same problem is solved (without difficulty) by using Traits. This tells us that MI lacks the compositional power to be able to factor out common code in all cases. [SDNB02]
2.3 Delegation

Inheritance is not the only way to achieve code reuse. It can also be done by delegation. With delegation, a new class that reuses features from an existing class creates an instance of that class and calls methods on it.

Consider the class Edge and VisualEdge in Java. The latter is a version of the former with an added feature: it can be drawn on screen. In VisualEdge, an instance of Edge is created, called edge. All methods from Edge are put in VisualEdge with as implementation edge.methodName(...). The code of VisualEdge is given in Listing 2.5.

```java
public class VisualEdge {
    // Use delegation to reuse the Edge class
    private Edge edge;

    public Vertex getFrom() {
        return edge.getFrom();
    }

    public Vertex getTo() {
        return edge.getTo();
    }

    // Code to make VisualEdge visual
    private java.awt.Color color;
    public VisualEdge(Vertex from, Vertex to) {
        edge = new Edge(from, to);
        edgeColor = java.awt.Color.black;
    }

    public void paint(java.awt.Graphics g) {
        g.setColor(color);
        g.drawLine(getFrom().x, getFrom().y, getTo().x, getTo().y);
    }
}
```

The benefits of delegation are:

- Reuse of existing code.
- Full control over which methods from which classes are reused. This in contrast to SI, where only one other class can directly be reused (viz. the superclass), and to MI, where all methods in the superclasses are reused, which can cause conflicts.

However, the benefits do - in most cases - not outweigh the disadvantages.

- Tedious programming. In the example, the code to delegate the getFrom() and getTo() methods in VisualEdge is just as large as the original methods. If we had used inheritance in this case, we wouldn’t have had to write these methods at all.
- A bigger disadvantage of delegation is the fact that by delegating, VisualEdge is not a subclass of Edge. For instance, a VisualEdge cannot be inserted in an array of Edges. If VisualEdge was a subclass of Edge, this would be possible.

Despite the limitations of delegation, it is useful for interface delegation, which we will discuss in the next section.
2.4 Interface delegation

Most OO languages have a single root class (e.g., `Object` in Java). Every class is a direct or indirect subclass of the root class. With MI, the inheritance relation forms a graph. With SI, it forms a tree, since directly inheriting from more than one class is not possible. However, since the appearance of Java, a second inheritance entity, the `interface`, has become a popular concept. It makes the SI model far more flexible. Interfaces are basically stripped-down versions of classes, with only the public method signatures remaining. A class can implement multiple interfaces. By implementing an interface, a class becomes a subtype of that interface. A non-abstract class (i.e., a class that can be instantiated) must implement all methods of the interfaces it has. This way, an interface can specify a certain behavior of a class, and classes can have more than one such behaviors. A class can thus implement multiple abstract interfaces, which is also possible with MI.

On the other hand, by allowing only one real parent (with implementation), the benefit of parent class reuse seems limited. By using interface delegation, the power of abstract interface conformance is used to simulate MI, thus creating the illusion of having multiple superclasses and reusing their code.

If a class has an interface entity, which describes the features of the class, it is possible to get a new class to conform to multiple such interfaces and to reuse the code of the corresponding classes. For example, if we have two classes `Reader` and `Writer` that allow us to read and write text files.

Listing 2.6: A Reader and a Writer

```java
public class Reader {
    public void read(String f) { ... }
}

public class Writer {
    public void write(String f) { ... }
}
```

If we want to make a class both a `Reader` and a `Writer`, we cannot inherit directly from both of the classes, because we can only inherit from one class. Instead, we create interfaces the specify the behavior of the classes. The `Reader` interface defines the `read` method and the `Writer` interface defines the `write` method. Next, we make the `Reader` class implement the `Reader` interface, and the `Writer` implement the `Writer` interface.

Listing 2.7: A ReaderWriter class using Interface delegation

```java
public class ReaderWriter extends Reader implements Writer {
    Writer writer;

    public void write(String f) {
        writer.write(f);
    }
}
```

Listing 2.7 shows the `ReaderWriter` that uses interface delegation. It extends the class `Reader`, but also implements the interface `Writer`. The method `write` thereof is implemented by delegating to an instance of the class `Writer`. In Figure 2.4 the resulting class and interface hierarchy is shown.
ReaderWriter now conforms to Reader_I (by subclassing Reader) and to Writer_I (by implementing the methods in that interface). What is done here, is to create an instance of a second (or third, etc.) class that we want to implement. The interface that needs to be available for this class, must be implemented by the new class. Then for all the methods of the interface, we implement a forwarding call to the instance of the class. This is called interface delegation.

It seems a bit of a hack to do this, as it is tedious programming work to write forwarding calls to a lot of methods. Interface delegation tries to simulate something (i.e. MI) that has a number of inconvenient limitations and has a limited form of behavior reuse. For example, the problem involving synchronization coined in Section 2.2 and Figure 2.3 can still not be solved with interface delegation.

2.5 Mixins

The Mixin inheritance model [BC90] [FKF98] does have the compositional power that the MI model lacks. A Mixin is an abstract subclass specification that may be applied to various parent classes to extend them with the same set of features. Thus, a programmer may factor out similar class behavior into a Mixin and make the classes with that behavior use this Mixin. Figure 2.5 gives an example of the use of Mixins.

MColor and MBorder are Mixins. When a class uses the Mixin MColor, it is required to have a toString() method. The result of this method is then appended with a string by the Mixin. This can be done for every class that has a toString() method. The same holds for MBorder. It is possible to combine Mixins. The order of composition is significant. In the case of Figure 2.5, MColor is applied before MBorder on Rectangle. It results in a class MyRectangle that will return the original value of toString() from Rectangle appended with ” with color with border” when called. The classes with + in their names are intermediate classes, generated during Mixin composition.
This way of composing classes works well when just one Mixin is used per class, but when composing a class from many Mixins, there is a problem. Different Mixins often do not fit together well (i.e. their features may conflict), and inheritance is not expressive enough to solve such conflicts.

There are various ways in which this problem shows itself:

- **Mixin composition is linear**: classes inherit Mixins one at a time, with the later Mixins overriding all identically named features of the earlier Mixins. This is inflexible, as a right order of Mixin inheritance may not exist when selecting features from different Mixins.

- **Gluing Mixins together is not straightforward**, because identically named features cannot be accessed, except for the last ones inherited. In Figure 2.5, there are two Mixins: MColor and MBorder. They both provide a method toString(). The implementations of these methods work by calling the method toString on the superclass and then adding some information to it. We compose the class MyRectangle by using the two Mixins. We can choose which Mixin should come first, but we cannot specify how the two implementations should be composed, because Mixins must be added one at a time. In the class Rectangle + MColor + MBorder we can access the behavior of MBorder and the mixed behavior of Rectangle + MColor. We cannot, however, access the original separate behavior of MColor and Rectangle.

- **Mixin inheritance causes fragile hierarchies in regard to changes** to the Mixins. Adding a method to a Mixin may silently override the method of the same name in another Mixin. This causes unwanted behavior and even could make it impossible to restore
2.6 Traits

The synchronization code put in a Trait (TSync), which makes it possible to add synchronization to every class that has a read and a write method.

- Extending a class with a Mixin results in a new class with an explicit name. Since most object-oriented languages do not have anonymous classes, this could lead to an explosion of classes or Mixins. In the example of Figure 2.5, it would be nicer not to have to explicitly name the new classes Rectangle + MColor and Rectangle + MColor + MBorder (which in practice will probably need to be named ColoredRectangle and BorderedColoredRectangle), but to have them transparently integrated into MyRectangle. This would save the programming the hassle of coming up with a number of names and gives a cleaner, clearer class overview.

Although Mixins avoid many of the complications cause by MI while allowing more flexible class hierarchies and better code reuse than traditional SI, it has some limitations and minor flaws. It is possible to solve the problem from Section 2.2 involving synchronization using Mixins in a manner similar to the solution we will see in Section 2.6. However, the Traits-model does not have the problems of the Mixins-model described above.

2.6 Traits

The Trait inheritance mechanism [SDNB02] is an extension of the SI mechanism and can also be viewed as a different approach to Mixins. A Trait is a lightweight, object-oriented entity that contains methods. Classes that use a Trait inherit the Trait’s methods. The methods can contain an implementation, but may also be required. Required methods have no implementation, and must be provided by classes or other Traits using the Trait. Classes can use (i.e. inherit methods from) multiple Traits, and Traits themselves can use other Traits.

Looking back to the synchronized read/write example from Section 2.2 and Figure 2.3, it was not possible with MI to factor out the common synchronization code in the classes SyncA and SyncB. With Traits, this problem is easily solved, as illustrated in Figure 2.6. The solution: the Trait TSync is created, which contains the synchronization code (methods acquireLock and releaseLock). TSync also contains two required methods read and write. Although these methods do not have an implementation, they can be used by the synchronization methods, with the knowledge that if a class uses this Trait, the class is
required to provide the implementation. To create the SyncA class, it suffices to extend class A (inheriting the read and write methods, including implementation), and using Trait SyncA, inheriting the synchronization code. Class SyncB can do the same with B. In fact, for any class that has an unsynchronized read and write method, a synchronized version can be creating by extending it and using the Trait.

Similar to Mixins, Traits can be glued together, but contrary to Mixins, the programmer has full control over the way identically named methods are dealt with. This is done by exclusion and aliasing. With exclusion, a programmer can remove the implementation of a specified method of any used Trait, making that method required. The required method could immediately be provided by another used Trait that has the same method, thus solving naming conflict. With aliasing, methods can be duplicated with another name. This way, a programmer can access overridden or even excluded methods from used Traits. We explain Traits in full detail in Chapter 3.

2.7 Overview

In Table 2.1, an overview is given of the features of the different inheritance mechanisms in this chapter. Below we discuss each feature and its score for each inheritance mechanism.

Code reuse . The ability to 'recycle' code that is written elsewhere, avoiding code duplication. If a programmer is sometimes forced to write duplicate code, then the inheritance mechanism does not allow maximal code reuse. For example, programs using just SI often contain duplicate code. The other inheritance mechanisms have better possibilities for code reuse, especially Mixins and Traits, which are able to solve the problem of Figure 2.3.

Compositional power . The ability to combine and reuse behavior at arbitrary places within the inheritance hierarchy. This results in less bulky classes and easier reuse. Mixins and Traits have compositional power. However, multiple Mixins can not be glued together into a single entity, which is possible with Traits. As a result, Traits have the highest score.

Abstract interface conformance . Conformance of a class to a certain type. Interfaces are designed for this. With MI, you can achieve the same level of abstract interface conformance as with interfaces, because a class can inherit any number of classes, which may be abstract (which makes them equivalent to interfaces). Using simple delegation, Mixins or Traits does not help abstract interface conformance, because those techniques are not part of the inheritance hierarchy. However, in Section 3.4, we will introduce a solution for Traits, dealing with this problem.

Accessing overridden features . How inherited functionality that is overridden, can still be accessed. This is a problem with MI, because of the fragile base class problem explained in Section 2.2. The same problem holds for Mixins. For delegation, nothing is overridden, so this feature does not apply. Traits can use aliasing to access overridden features, and SI and interface delegation can use the super-keyword.
### Table 2.1: A comparison of the discussed inheritance structures

<table>
<thead>
<tr>
<th></th>
<th>SI</th>
<th>MI</th>
<th>Delegation</th>
<th>Interfaces</th>
<th>Mixins</th>
<th>Traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code reuse</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Compositional power</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Abstract interface conformance</td>
<td>+</td>
<td>++</td>
<td>−</td>
<td>++</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>Accessing overridden features</td>
<td>+</td>
<td>−</td>
<td>n/a</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Control over conflicts</td>
<td>n/a</td>
<td>−</td>
<td>n/a</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>‘Clean’ class hierarchy</td>
<td>++</td>
<td>+</td>
<td>−</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
</tbody>
</table>

+++ means: very good; does this implicitly.
+
means: good; only a small programming effort required.
− means: possible/present, but there are problems or serious limitations when doing this.
−− means: not possible/not present.
n/a means: not applicable in this mechanism.

Control over conflicts. How well conflicting features (inherited from multiple units of reuse) can be resolved. With Traits, control over conflicting features can be achieved by using exclusion. Mixins are inflexible with this, as it involves a linear way of composing classes with Mixins, explained in Section 2.5. MI scores low because of the fragile base class problem.

‘Clean’ class hierarchy. Whether a class tree in the inheritance structure is typically concise and not bloated with redundant composite classes. Mixins score low because of all the generated extra classes. SI automatically has a simple but clear structure, but simple delegation (through lack of inheritance) does not have a structure at all. With MI, interfaces or Traits, a clean structure can be achieved with some effort.

Looking at Table 2.1, it seems that the Traits mechanism is good or very good in every aspect. Adding this mechanism to an object-oriented programming language will probably make that language more powerful and more versatile.

### 2.8 Other reuse structures

In addition to inheritance, there are other ways to achieve reuse in a programming language. In this section we briefly look at some of them and compare them with Traits, as they might prove a better alternative to Traits.

#### 2.8.1 Aspect oriented programming

With aspect oriented programming (AOP), existing code can be changed without modifying the actual code. One must define an aspect in which pointcuts specify at which locations the existing code is to be changed. Examples of these pointcuts are “anytime the method print() is called” or “anytime a field starting with keyword... is set in class C”. In the aspect, additional code is defined that is set to execute before, after or around a pointcut. Also, by using introduction, new methods and variables can be added to classes or change which interfaces a class implements.
Suppose we have the source code of a large program that has logging functionality. The code for the logging is spread out over a lot of different packages, classes and methods. Changing the way of logging would be a tiresome and error-prone task. Logging is a typical task for AOP. Instead of putting logging in code and lose sight of all the places, all places where logging should occur could better be put in a pointcut, such that the logging code can be added in an aspect. Changing the way of the logging process would then require only modifications in a central place, i.e. the aspect.

Whereas Traits are part of the structure of a program, aspects are totally separated from the inheritance hierarchy. This can lead to an unclear program structure if one is not careful. AOP can also be a very dangerous way of programming. Since source code can be changed at a second level, testing is often problematic. This is not the case with Traits, which are completely integrated part of the source code. On the other hand, AOP has the benefit of being pluggable. As the AOP code is separate from the rest, it can be switched off and on at will, by not including the aspect during compilation. This is not the case with Traits, because Traits are more integrated in the code. Sometimes Traits can be ‘switched off’ without any problem (e.g. the TSync Trait from Section 2.6), but this requires actual modification of the source code.

In short, both Traits and AOP have their own benefits and limitations, but both systems work in such a different way that it is not possible to say that one is better than the other. For some tasks, Traits will probably be a better way (e.g. for class composition), for others, AOP (e.g. for logging). It would probably be best if a programming language had both Traits and aspects. [KLM+97]

### 2.8.2 Templates

Class and function templates are used in C++ and provide a basic form of reuse [Str94]. A programmer can write a class or function template with a parameter that a program can use to instantiate a real class. For example, a template of a list can be made, so a programmer can easily create a list-class specifically for some class, he want to put in a list.

```cpp
template <class T> class list
{
    private:
        T *first; // Pointer to 1st element of array of Ts.
        int size; // Number of elements of array of Ts.

    public:
        list(int); // Constructor for making an array of Ts.
        ~list(); // Destructor - frees memory.
        T& operator[](int); // Overloads [] operator so can access specific elements of the array of Ts.
};
```

The programmer can then instantiate a list-class of integer values by the following code.

```cpp
list<int> intList(100);
intList[1] = 1;
```
2.8 Other reuse structures

Through templates, code reuse is achieved. For example, if there are many different types for which a list is required, a programmer only has to write one list template class. But the downside is that templates are very loosely typed. For example, in a template with type parameter \( T \) and variables \( x \) and \( y \) of type \( T \), one could just write \( x + y \) and the compiler would not complain until the program is run.

Compared to Traits, templates work the other way around in a way. Traits can add reusable code to multiple classes, whereas templates can add multiple classes to reusable code. Therefore, Traits and templates are orthogonal and could be used together.

2.8.3 Generics

Generics is another way of code reuse, which is very similar to templates. The only difference is that generic classes are strongly typed and so operations on initiations of generic classes can be type-checked at compile-time. Among the programming languages that support generics are C# and the new version of Sun’s Java compiler (1.5). The presence of generics in such a programming language does not take away the need for Traits, because Traits and generics are orthogonal and the problems with inheritance will therefore remain after adding generics. [BOSW98]

2.8.4 Reuse variables

Reuse variables have been introduced by Keedy et al. [JLKM04] with the programming language Timor [Tim]. As with Traits, it decouples subtyping and code reuse. A reuse variable can appear in the state section of a Timor-type (which is similar to a class), but not as local variable in a method. The compiler can treat the public instance methods of a reuse variable as methods of the type it appears in. The order in which the reuse variables are declared is relevant, as a certain matching procedure goes over the instance methods of the class, and checks one by one if the method’s implementation is present (first in the type itself, then in the reuse variables).

This way, you can define a type with certain instance methods without implementation, and then add the implementation by using reuse variables. Suppose we have a List type that defines the methods insert, insertAtPos, remove, removeAtPos, getAtPos, position, contains, and size. The List can be used with different implementations (e.g. as a linked list, an array, etc.). Now we create a Bag type with only the following methods: insert, remove, contains and size. There are no implementations for the methods, so Bag (and List) can be viewed as Java-interfaces. And the methods can be viewed as ‘required’ methods. An implementation of Bag can use a reuse variable of a List-type like this (the reuse variable is preceded by a `^`-symbol):

```java
state:  
^List <:ELEMENT:>: aList;
maker: 
  init(int maxSize) {aList.init(maxSize);} 
}
```

The last line of this type is the maker (constructor), and performs delegation. The four required methods do not have to be delegated, because the compiler automatically takes them from the list type.
The List-type has all methods the Bag-type has and more, so it can be seen as a subtype of Bag. The implementation of Bag uses a subtype of Bag, which is reverse to the way of reuse we see with inheritance, where subtypes only reuse code from supertypes. However, both ‘directions’ of reuse (sub-supertype or super-subtype) are possible when using reuse variables. This makes this mechanism very flexible.

At first glance, reuse variables seem a fancy version of delegation, but there are many advantages, which will not all be described here. There are similarities with Traits, such as the decoupling of subtyping and code reuse, the ‘copying’ of methods into classes. Just like Traits there are rules for the order of this ‘method importing’, but there is a linear order of the reuse variables, which can cause difficulties. In Timor, there is also a way of mapping method names, which is similar to aliasing. A last similarity is that both mechanisms do not allow the reuse of state.

In conclusion, a combination of Traits and reuse variables in one language would probably be superfluous, as both mechanism do pretty much the same. We believe however, that Traits is the best choice of the two, as the programmer has slightly more control over the reused methods.
Chapter 3

Traits

This chapter gives an explanation of the Traits inheritance mechanism. In Section 2.6, we have already seen an example of a Trait (TSync). Common code was factored out into this Trait and the Trait could be used in several classes, which need that common code. This example illustrates the idea of Traits: they are entities, which can be used as building blocks for classes, and serve as units of code reuse. Each Trait defines specific, reusable behavior. For TSync, this behavior is synchronizing existing read- and write-methods.

3.1 Language and graphical conventions

We assume a programming language with single inheritance, which supports Traits. The language constructs are as follows.

- **Method dictionary**: $D ::= \{ \} \mid \{ x_1 \mapsto m_1, \ldots, x_n \mapsto m_n \}$
- **Trait expression**: $T ::= N_T \mid T + T \mid T - x \mid T[x \mapsto y]$
- **Trait definition**: $T_{\text{Def}} ::= N_T = D \mid N_T = D \text{ with } T$
- **Fields**: $F ::= \{ \} \mid \{ f_1, f_2, \ldots, f_n \}$
- **Class expression**: $C ::= \text{nil} \mid N_C \mid F \text{ in } D \mid F \text{ in } D \text{ extends } N_C \mid F \text{ in } D \text{ with } T \mid F \text{ in } D \text{ extends } N_C \text{ with } T$
- **Class definition**: $C_{\text{Def}} ::= N_C = C$

A **method dictionary** ($D$) is either empty or contains a number of methods. Method names can be overloaded, i.e. a method dictionary may contain equally named methods if and only if their parameter types are different. A **method** is denoted in the form $x \mapsto m$. The letter $x$ comprises the method’s signature, i.e. the name plus parameters types. The letter $m$ comprises the method’s body. A **required** Trait method has no implementation, so its body will be denoted by the empty set symbol (e.g. $x \mapsto \emptyset$).

A **Trait expression** ($T$) is either a named Trait ($N_T$), a symmetric composition of Traits ($T + T$) or a Trait subject to an exclusion ($T - x$) or an alias ($T[x \mapsto y]$). Composition of Traits is explained in Section 3.2.1, exclusion in Section 3.2.3 and aliasing in Section 3.2.4.

A **Trait definition** ($T_{\text{Def}}$) binds a named Trait ($N_T$) in two forms: just a method dictionary ($D$) or a method dictionary that uses a Trait ($D \text{ with } T$). The latter, a Trait that uses another Traits, is explained in Section 3.2.2. Observe that classes cannot contain Fields. This is explained in Section 3.3.
A class expression \((C)\) can either be empty, a named class \((N_C)\), a conventional class \((F \text{ in } D)\) with fields \((F)\) and methods, or a class that uses Traits \((F \text{ in } D \text{ with } T)\). A class (whether with or without Traits) may have a superclass, which is denoted by the \texttt{extends} keyword. A class definition \((C_{\text{Def}})\) simply binds a class name to a class.

In the example diagrams, there are some notational conventions maintained. A Trait is represented by a rounded rectangle. Its name is printed in the top of the rounded rectangle and starts with the letter T. A class is represented by a rectangle and its name does not start with a T. The required methods of Traits are italicized and the glue methods are emboldened. If a Trait \(T\) is used by a class \((\text{or other Trait}) C\), an arrow with a double arrowhead will point from \(C\) to \(T\). Pieces of code in the diagrams are put in rectangles in sans-serif font and have a curved line connecting to the corresponding class or Trait.

### 3.2 Trait specification

A Trait contains a set of methods, which can be divided into two sets: provided methods and required methods. The behavior of the Trait is provided by the \textit{provided methods}. A Trait may in addition use a number of methods that have no implementation, called the \textit{required methods}, similar to abstract methods in Java and C++. These are used to parametrize the behavior of the Trait.

\[
\begin{align*}
D_T^R \cup D_T^P &\equiv D_T \land D_T^R \cap D_T^P \equiv \emptyset \\
\forall x \in D_T (x \not\rightarrow \emptyset &\rightarrow x \in D_T^P) \\
\forall x \in D_T (x \rightarrow \emptyset &\rightarrow x \in D_T^R)
\end{align*}
\]

The method dictionary of a Trait \((D_T)\) can be divided into two orthogonal subsets: the required method dictionary \((D_T^R)\) and the provided method dictionary \((D_T^P)\). In the example from Section [2.6], the provided methods are acquireLock and releaseLock, and the required methods are read and write.

#### 3.2.1 Composing classes with Traits

In this chapter, we use a running example of Traits for geometric shapes. A Trait representing a shape (e.g. a circle or a rectangle) has methods to perform certain geometric calculations (e.g. circumference and surface). In the example of Figure [3.1] we want to construct a class that represents a circle with a certain color. For this, we use the Trait TCircle, which provides the shape. There is also a Trait TColor available to provide methods that deal with color. The class ColoredCircle is a class that uses two Traits: TCircle and TColor. This means that the class is composed from (amongst others) these Traits. Trait TColor has one required method \((\text{getRGB})\), as has TCircle \((\text{getRadius})\). The rest of the Trait methods are provided and their implementations are visible in the figure.

Every class that uses a trait with required methods must provide these methods. This 'filling in' of required methods is called \textit{gluing}. The resulting code is called \textit{glue code}. In the example, the glue code is provided by the class ColoredCircle. The class has two
Basic class composition. A class can be represented by the equation \( \text{Class} = \text{Traits} + \text{States} + \text{Glue} \) \cite{NSN02}. Traits and state will be addressed in Section 3.3. To achieve the equality in the equation, the flattening property applies. This property dictates that the semantics of a method defined in a Trait is identical to the semantics of the same method defined in a class that uses the Trait. So, a class plus the Traits it uses are conceptually flattened or merged into one class, also containing the methods from the Traits.

\[
N_C = F \text{ in } D_C \text{ with } N_T \land N_T = D_T
\]

\[
N_C = F \text{ in } (D_C \cup D_T)
\]  \hspace{1cm} (3.4)

A few extra rules apply for class composition:

Class method precedence. Class methods take precedence over Trait methods. If the method dictionary of a class contains a method, whose name is also in the Trait method dictionary, the method from the class is taken. The rule for class composition hereby becomes more specific:

\[
N_C = F \text{ in } D_C \text{ with } N_T \land N_T = D_T
\]

\[
N_C = F \text{ in } (D_C \cup (D_T \setminus D_C))
\]  \hspace{1cm} (3.5)

Superclass method precedence. Trait (provided) methods take precedence over superclass (\(SC\)) methods. This can be inferred from the flattening property, which says that
3.2 Trait specification

Trait methods behave as if they were implemented in the class itself.

\[
N_C = F \text{ in } D_C \text{ extends } N_{SC} \text{ with } N_T \land N_T = D_T
\equiv
N_C = F \text{ in } (D_C \cup ((D_{SC} \setminus D_C) \setminus D_T) \cup (D_T \setminus D_C))
\] (3.6)

However, if the Trait method in question is a required method, then the superclass method does provide the implementation.

\[
N_C = F \text{ in } D_C \text{ extends } N_{SC} \text{ with } N_T \land N_T = D_T
\equiv
N_C = F \text{ in } (D_C \cup ((D_{SC} \setminus D_C) \setminus D_T^R) \cup ((D_T \setminus D_C) \setminus (D_T^R \cap D_{SC})))
\] (3.7)

**Symmetry.** The composition order is irrelevant.

\[
F \text{ in } D \text{ with } T_1 + T_2 \equiv F \text{ in } D \text{ with } T_2 + T_1
\] (3.8)

**Conflicting methods.** Any conflicting Trait methods must be explicitly disambiguated (how this is done is described in Section 3.2.3 and 3.2.4)

3.2.2 Composing Traits

Not only can classes be composed with Traits and glue code, but Traits themselves can be nested, allowing Traits to be composed from other Traits. When Traits act as building blocks for more complex Traits, a very flexible modular framework is created. A base Trait consists of only methods, but composite Traits can be summed up with the equation: Trait = Traits + Glue. Required methods in the ‘superTraits’ may be provided by the composite Trait, and otherwise just become required methods of the composite Trait. Methods that are already provided in superTraits may not be redefined as required in the composite Trait (however, this effect can be achieved with exclusion, which is covered in Section 3.2.3).

Again, composition order is irrelevant, and the composite Trait’s methods take precedence over the methods of the superTraits. The flattening property remains in order, even with deeply nested Traits. A method’s semantics does not depend on the place where that method is defined: a method defined in a superTrait acts the same as if it were defined in the composite Trait, and vice versa.

\[
N_{T_1} = D_{T_1} \text{ with } T_2 \land N_{T_2} = D_{T_2}
\equiv
N_{T_1} = D_{T_1} \cup (D_{T_2} \setminus D_{T_1})
\] (3.9)

In the example depicted in Figure 3.2, T2DShape is a superTrait for TRectangle and TCircle. There are two required methods in T2DShape, which represent two properties of geometric shapes: circumference and surface. There are no standard formulas for these properties, which are valid for every 2-dimensional shape, so these methods have been left open (i.e. required).

A rectangle has a length and a width property, which were translated to required methods in TRectangle. With these two properties provided, the circumference and surface can
be calculated, and so the getCircumference and getSurface methods are provided, by 'gluing' together length and width. The same holds for a circle, when its radius is known.

A square is a specific version of a rectangle, that has equal width and length, so when the length of one side is known, we know the width and length properties. Accordingly, the TSquare method gets a new required method getSideLength, and the getLength and getWidth methods are provided using the new required method. The calculations of the circumference and surface inherited from TRectangle, are still valid. A class that represents a square now only has to provide one method that returns the length of a side and gets circumference, surface, length and width 'for free'.

### 3.2.3 Exclusion

Following from the rules of precedence for composition with Traits, conflicts can occur if and only if two or more Traits are used at one composition, that provide methods with equal signatures.

\[
N_C = F \text{ in } D_C \text{ with } N_{T1} + N_{T2} \quad \text{and} \quad N_C = F \text{ in } D_C \cup \{x \mapsto m, x \mapsto m'\} \\
\wedge \quad N_{T1} = \{x \mapsto m\} \\
\wedge \quad N_{T2} = \{x \mapsto m'\} \\
\rightarrow \quad \downarrow \quad \text{Conflict!}
\]

(3.10)

In Equation 3.10, there are two Traits that define a method with signature \(x\). Class \(C\) uses both Traits, so both methods are added to \(D_C\). This results in two methods of equal signature available in the class, which is a conflict. Note that if \(x \in D_C\) were true, there would not be a conflict, because methods defined in the class take precedence over Trait methods. As a result, both methods from \(T_1\) and \(T_2\) would not be added to \(D_C\).
It is possible that a class uses two Traits, which both use the same supertrait. If both Traits do not override a provided method from the supertrait, the class will inherit the same method with implementation via two paths. In such a case, the Trait model dictates that there is no conflict. Only one Trait method is inherited by the class, because they are the same. This example is shown in Equation (3.11). This can also occur with a Trait instead of a class, and with multiple supertraits.

\[
N_C = F \text{ in } D_C \text{ with } N_{T1} + N_{T2} \\
\land \quad N_{T1} = D_{T1} \text{ with } N_{T3} \\
\land \quad N_{T2} = D_{T2} \text{ with } N_{T3} \\
\land \quad N_{T3} = \{x \mapsto m\} \\
\land \quad x \notin D_C \cup D_{T1} \cup D_{T2} \\
\land \quad N_{T1} = D_{T1} \cup \{x \mapsto m\} \\
\land \quad N_{T2} = D_{T1} \cup \{x \mapsto m\} \\
\land \quad N_C = F \text{ in } D_C \cup D_{T1} \cup D_{T2} \cup \{x \mapsto m\}
\]

(3.11)

In Equation (3.11) there are two Traits \((T_1 \text{ and } T_2)\), which use a third Trait \(T_3\) that provides a method \(x\). There is a class that uses the two Traits \(T_1\) and \(T_2\). Although both Traits have a method with the same signature defined, since it is the same method, it is added only once to \(D_C\). If one of the two Traits define a new implementation for \(x\), there would still be a conflict, because the methods are not the same.

By this, we can deduce that conflicts can only occur when there are methods with equal names and signatures present with different implementations in two or more used Traits. Following the rules of Section 3.2, these conflicts must be explicitly resolved. This is done by exclusion or by providing a new implementation of the method (i.e. overriding) in the class or Trait where the conflict occurs. By exclusion, a provided method is converted to a required method. An entity (i.e. a class or Trait) that uses a Trait can specify which (provided) methods from the Trait it wants to exclude.

\[
N_{C1} = F \text{ in } D_{C1} \text{ with } N_{T1} + (N_{T2} - x) \\
\land \quad N_{C2} = F \text{ in } D_{C2} \text{ with } (N_{T1} - x) + N_{T2} \\
\land \quad N_{C3} = F \text{ in } D_{C3} \text{ with } N_{T1} + N_{T2} \\
\land \quad N_{T0} = D_{T0} \text{ with } (N_{T1} - x) + (N_{T2} - x) \\
\land \quad N_{T1} = \{x \mapsto m\} \\
\land \quad N_{T2} = \{x \mapsto m'\} \\
\land \quad x \notin D_{C1} \cup D_{C2} \cup D_{T0} \\
\land \quad x \notin D_{C1} \cup \{x \mapsto m\} \\
\land \quad N_{C2} = F \text{ in } D_{C2} \cup \{x \mapsto m'\} \\
\land \quad N_{C3} = F \text{ in } D_{C3} \\
\land \quad N_{T0} = D_{T0} \cup \{x \mapsto \emptyset\}
\]

(3.12)

In Equation (3.12) four examples of conflict solving are shown. The classes \(C_1, C_2\) and \(C_3\) and the Trait \(T_0\), each solve a method conflict in a different manner. Class \(C_1\) excludes the \(x\)-method from \(T_2\), leaving the implementation from \(T_1\). Class \(C_2\) excludes \(x\) from \(T_1\), working the other way around. The class \(C_3\) also solves the conflict, but not with exclusion. Instead, it overrides the \(x\)-method in the class, thus not adding the \(x\)-methods from its Traits. Trait \(T_0\) excludes both \(x\) methods from the Traits, which leaves \(x\) as a required method in the composite Trait.

These are four possible ways to resolve a conflict of two Trait methods. The way in which \(T_0\) does this, can only be used by Traits, as a class cannot have required methods. The ways
Figure 3.3: Exclusion is used to resolve conflicting methods

in which the three classes resolve a conflict, could also be used by Traits with supertraits. The disadvantage of these ways of dealing with conflicts is that methods that are overridden or excluded cannot be used. Fortunately, by using aliasing, programmers can still access overridden and excluded methods.

An example of exclusion is given in Figure 3.3. There is a class ColoredCircle, which uses the Traits TCircle and TColor. Each Trait provides a specific functionality to ColoredCircle. The geometry is provided by TCircle and the color by TColor. The Traits both provide a different implementation for the equals method. The programmer has decided that two instances of ColoredCircle are equal if their radius is equal, even if their colors differ. To accomplish this, the equals method of TColor is excluded when composing the class. As a result, that method becomes required, but it is immediately ‘filled in’ by TCircle. The equals method of ColoredCircle is therefore the same one as in TCircle, which solves the conflict.

3.2.4 Aliasing

If the programmer had decided that two instances of ColoredCircle are equal if and only if their size and their color are equal? A new equals-method would then be required in the class, combining both equality methods from the Traits. Putting a new method in ColoredCircle will resolve the conflict, as it will take precedence over the Trait methods, but how can the equals methods from the Traits be accessed? A programmer who is used to Multiple Inheritance would probably come up with something like Listing 3.1.

This will not work because the super keyword has no special meaning for Traits (because of the flattening property). A class that uses Traits is the same as the class without Traits, but with the methods of the Traits in the class. This has as advantage that there is no fragile
base class problem, but it may seem that overridden Trait methods cannot be accessed anymore. Fortunately, it is possible using a mechanism called aliasing. With aliasing, a method from a used Trait can be copied to a method with another name: an alias.

\[
N_C = F \text{ in } D_C \quad \text{with} \quad N_T[x \mapsto y] \\
\wedge \quad N_T = \{x \mapsto m_1\} \\
\wedge \quad x \notin D_C \\
\rightarrow \quad N_C = F \text{ in } D_C \cup \{x \mapsto m, y \mapsto m\}
\]

(3.13)

In Equation [3.13] a class aliases the method \(x\) from its Trait to \(y\). As a result, both \(x\) and \(y\) are added to the \(D_C\). Note that if \(x \in D_C\) was true, then only \(\{y \mapsto m\}\) would be added to \(D_C\).

In the colored circles example we can use this. In ColoredCircle the equals method from TCircle is aliased to circleEquals. Likewise, the equals from TColor is aliased to colorEquals. These two methods are added to ColoredCircle and contain the original implementations. The new equals glue method in ColoredCircle now can use these two methods to achieve the desired equality. By aliasing we can still use the equals methods from the Traits, even though they are overridden by the one in ColoredCircle. This example is illustrated in Figure [3.4]. Equation [3.14] shows the example in the language used throughout this chapter.

\[
N_{\text{ColoredCircle}} = F \quad \text{in} \quad D_{\text{ColoredCircle}} \quad \text{with} \quad N_{\text{TCircle}}[\text{equals} \mapsto \text{circleEquals}] \\
\quad + \quad N_{\text{TColor}}[\text{equals} \mapsto \text{colorEquals}] \\
\wedge \quad N_{\text{TCircle}} = D_{\text{TCircle}} \wedge D_{\text{TCircle}} \supseteq \{\text{equals} \mapsto m_1\} \\
\wedge \quad N_{\text{TColor}} = D_{\text{TColor}} \wedge D_{\text{TColor}} \supseteq \{\text{equals} \mapsto m_2\} \\
\wedge \quad D_{\text{ColoredCircle}} \supseteq \{\text{equals} \mapsto m_0\} \\
\wedge \quad D_{\text{ColoredCircle}} \cap D_{\text{TCircle}} \equiv D_{\text{ColoredCircle}} \cap D_{\text{TColor}} \equiv \{\text{equals}\} \\
\rightarrow \quad N_{\text{ColoredCircle}} = F \quad \text{in} \quad D_{\text{ColoredCircle}} \cup (D_{\text{TCircle}} \setminus \{\text{equals} \mapsto m_1\}) \\
\quad \cup (D_{\text{TColor}} \setminus \{\text{equals} \mapsto m_2\}) \\
\quad \cup \{\text{circleEquals} \mapsto m_1, \text{colorEquals} \mapsto m_2\}
\]

(3.14)

Aliasing can also be used to access Trait methods that are explicitly excluded. Here, both equals-methods from the Traits are in a sense implicitly excluded from ColoredCircle by the new equals-method, which takes precedence over the Trait methods. If we had explicitly excluded those methods, the result would have been the same, and the class definition for ColoredCircle would be as defined in Equation [3.15].
Figure 3.4: Aliasing is used to access overridden or excluded Trait methods

\[ F \text{ in } D_{\text{ColoredCircle}} \text{ with } \quad N_{\text{ColoredCircle}} = \]
\[ (N_{\text{TCircle}}[\text{equals } \rightarrow \text{circleEquals}] - \text{equals}) \]
\[ + (N_{\text{TColor}}[\text{equals } \rightarrow \text{colorEquals}] - \text{equals}) \] (3.15)

### 3.3 Traits and state

In the examples from the previous paragraphs we have seen that get-methods are used within a Trait to access the state of classes that will eventually use the Trait. Traits were designed without state. A Trait is just a collection of methods. The absence of state avoids problems that occur when there are conflicting state variables, as we have seen in Section 2.2. Although this is an advantage of not having state, the absence of state can also be a disadvantage.

For example, consider the example with the colored circle from the last section (we hereby ignore the TEquality Trait). Suppose we want to have two classes, which use TCircle. One (Circle) is just a circle and the other (ColoredCircle) also uses the TColor Trait. In both classes we would have to write glue code for getRadius, which is the same in both classes. This is code duplication, which we want to avoid. This is illustrated in Figure 3.5. In this case, we can solve this problem by using single inheritance. If we make ColoredCircle a subclass of Circle, there is no more code duplication, as illustrated in Figure 3.6.

Unfortunately, it is not always possible to avoid the code duplication. Take for instance the example from Section 2.6. Suppose there is a class Lock that we want to use for the
synchronization, which has the methods lock and unlock. We cannot have a Lock-object in the state of a Trait, so instead, we add an extra required method getLock, which must return a Lock-object to use. The acquireLock and releaseLock methods can call the getLock method, even though the method is not implemented within the Trait. Any class that uses the TSync Trait must provide an implementation for getLock, as well as for read and write. The implementation can be provided by the class itself, or by a direct or indirect superclass, as is the case here with read and write.

Classes SyncA and SyncB both use the Trait TSync. There are now three required methods in the Trait. These methods must be ‘filled in’ by any class that uses the Trait. In this case, no extra glue code needs to be added for read and write, because the required methods are already present in both classes. For the getLock method, a Lock object must be returned. For this, we create an instance of Lock in the Sync-classes and write the appropriate glue code: return lock.

As you can see in Figure [3.7] this leads to code duplication in the classes SyncA and SyncB. Unfortunately, this is a disadvantage of Traits, brought along by the fact that Traits do not have state. But the absence of state avoids problems with conflicting state values, which the designers of the Traits model considered to be more important.

Fisher and Reppy [FR03] proposed a way to have ‘required’ state variables. These variables cannot be instantiated within Traits, but they may be used there. This eliminates the need for (duplicate) get- and set-methods, while still avoiding problem of conflicting state. However, naming conflicts can occur this way. For example, a lock-variable in one Trait of type Lock and one in another Trait of type boolean, gives a conflict when both Traits are combined. This concept was not incorporated in our design, because as it is the first
3.4 Typing Traits

In the overview of Section 2.7, we concluded that on the subject of abstract interface conformance, Traits do not score as high as some other, more mainstream inheritance mechanisms. In this section, we introduce a new way of typing Traits, which will put Traits on the same level as regular interfaces on this subject.

As a first application, the Trait mechanism was added to the programming language Smalltalk, which is an untyped language [BSD02]. Later, Fischer and Reppy introduced a typed calculus of Traits [FR03]. The purpose of this calculus was to provide a means of integrating Traits into a statically-typed language such as Java [AG98]. While designing the compiler for Java with Traits, we came up with another, but similar way to type Traits. It uses the existing typing structures of Java in order to accommodate Traits in the type system. Thus, the structure of Traits can neatly be converted to structures that are available in regular Java.

Suppose we want to add Traits to an object-oriented programming language. Because of the flattening property, it is possible to convert the new classes that use Traits into regular classes. The regular classes are extended with the methods from the Traits, following the precedence, exclusion and aliasing rules of the Traits model. If the language we are extending is typed, situations in which a type is needed for a Trait need a solution. For instance, consider the situation of Figure 3.3. If we have a typed language, we need to have
a type for the other parameter of the equals-methods. In such situations, we want classes that use traits to not only conform to its own interface, but also to the interface of the Traits. However, in the original Trait model, Traits do not provide interface conformance. Their are just added to the collection of method of the class that uses them. In a typed language, we want to have interface conformance for Traits.

An obvious, but unsatisfactory way to achieve this is to add the abstract interface of a Trait to the type of the class that uses it. A person programming a Trait could then refer to the Trait’s own name (we call this self) as its type, knowing that a class that will eventually use that conforms to at least the interface of the Trait. In the flattening process, references to self within a Trait can be converted to references to self of a class that uses a Trait. This ‘merging’ of abstract interfaces discards the individual types of the Traits. When multiple classes use the same Trait, the methods of that trait which have references to self, become different methods in the classes, and therefore, incompatible. An example of this problem is shown in Figure 3.8, in which the method equals from TCircle ends up as two different, incompatible methods in the classes Circle and ColoredCircle.

In our design, we have chosen a different approach to achieve interface conformance for Traits. This approach is possible when the programming language in question already has interfaces at its disposal. In the flattening process, we can convert all Traits to interfaces, have the nested Trait extend their superTraits' interfaces, and make all classes that use Traits, implement the corresponding interfaces. The Trait methods (provided and required) become the interface methods. This causes no naming issues, because the names of the Traits cannot overlap with the names of classes and interfaces in the first place.
Figure 3.8: The flattening property is useful, but also ‘flattening’ the types yields incompatible methods.

Traits become interfaces, but in the process donate implemented methods to classes that use them. Exclusion yields no problems here, because exclusion does not ‘delete’ methods from a Trait; it merely makes methods required. Aliasing also causes no problems, because aliasing merely copies methods to other methods (which will eventually also appear in the subinterface or implementing class), and does not rename methods. In Figure 3.9 we show the interface hierarchy corresponding to the Traits of the example of Figure 3.8.

The equals-methods of Circle and ColoredCircle are now exactly the same and so, compatible. When the Traits are flattened into the classes, the resulting equals-methods still make use of TCircle, even though that Trait no longer exists. This is possible, because TCircle now exists as an interface. The code of the equals-method (of both classes) is the same as that of the equals-method of TCircle, which can be found in the left upper corner of Figure 3.8.

3.5 Summary of features

Throughout this chapter, we have seen how the Traits model works, and how an existing type system can be mapped onto the Trait hierarchy using interfaces. In this section, the major advantages and disadvantages of Traits are summed up. First, the advantages.
### 3.5 Summary of features

#### TCircle
- `double getRadius()`
- `double getCircumference()`  
- `double getSurface()`  
- `boolean equals(Object other)`

#### TColor
- `int getRGB()`  
- `boolean equals(Object other)`

#### TEquality
- `boolean equals(Object other)`

#### Circle
- `double getRadius()`  
- `double getCircumference()`  
- `double getSurface()`  
- `boolean equals(Object other)`

#### ColoredCircle
- `double getRadius()`
- `double getRGB()`  
- `double getCircumference()`
- `double getSurface()`
- `boolean equals(Object other)`

---

**Figure 3.9:** The hierarchy of classes interfaces (in light-yellow) as they will be generated by the flattening process.

- The use of Traits separates code reuse from the inheritance hierarchy. This allows programmers to avoid problems with methods defined too high in the inheritance hierarchy while sharing common implementation.

- Specific behavior can be factored out into a Trait, instead of putting all different behavior together in a class. This leads to cleaner code.

- Lack of state and a clean way of dealing with method conflicts prevents the **diamond problem**.

- Full control over conflicting methods using exclusion and/or aliasing.

- Access to every method of the used Traits without the **fragile base class problem** with aliasing.

- Simple structure because of the **flattening property** prevents the **fragile base class problem**.

- Traits can have ‘interface conformance’ unlike e.g. delegation.

But there are also some disadvantages:

- The lack of state sometimes demands an inconveniently indirect way of dealing with variables.

- Not (yet) available in most programming languages, and, because they are not widespread, most programmers have no knowledge of Traits.
Chapter 4

Implementation

We have designed a Trait extension for Java 1.4, and implemented a compiler for this language, based on the specifications of Chapter 3. We call this new language JTL, which stands for Java Trait Language. Source files of this language should have the extension .jtl. This language is backwards compatible with (regular) Java, as it is of course possible to not use Traits (resulting in regular Java). In this chapter, we discuss the implementation of this compiler.

4.1 Framework

For the implementation of the JTL compiler we use the extendible Java compiler framework Polyglot [NCM03]. It is implemented in Java and distributed as Open Source. Polyglot without extensions is a base compiler, which compiles regular Java. Polyglot allows language extensions to Java to be written, based on the base compiler. Extensions are written in Java and in PPG, which we will discuss below. When such an extension is compiled, the result is a compiler for the new language. That compiler is a source-to-source compiler that accepts programs written in the new language and translates it to regular Java source code. Optionally, the compilation process involves the compilation of the resulting Java code to bytecode, with a Java compiler.

The compilation process the Polyglot framework is shown in Figure 4.1. In the figure, the name Ext stands for the name of the extension. The process offers several opportunities for programmers of a language extension to customize the behavior of the framework.

The Polyglot framework contains the extendible parser generator PPG, which is based on the CUP LALR parser generator for Java [Hud]. We use PPG to define the Trait extension to Java. Since PPG is an extendible parser generator, programmers only have to define

---

Figure 4.1: Compilation process of a language extension made with Polyglot
4.2 The JTL extension

The JTL extension follows the Polyglot extension architecture. It defines additional syntax constructs for Traits, which are written in PPG. These syntax constructs require new AST nodes for Traits and modifications to existing nodes for classes. With the syntax extension and new AST nodes, the generated parser generates ASTs for programs. With the addition of Traits to the language, a lot of existing compilation passes need to be extended and modified. Also, new compilation passes are needed to translate the JTL AST to a Java AST. In the following sections, we go into more details on the implementation of the various steps of the compilation process. Below we give a summary of the implementation of the JTL extension.

In JTL, it is possible to define Traits and have classes use Traits. To give an idea of the syntax, here is pseudo-code of how a Trait is defined and of how a class can use Traits.

```
trait TraitName with Trait1 excluding {methodName(parameters), ...}, Trait2 aliasing {alias=methodName(parameters), ...} {
    ReturnType method1(parameters) {
        ...
    }
    ...
}
class ClassName extends SuperClass implements Interfaces with Traits {
    ...
}
```

The class that is used in Polyglot for AST nodes for class declarations (ClassDecl), needs to be extended to be able to use Traits. To do so, we make an extension to that class (calledJTClassDecl), which has a list of used Traits. Because Trait declarations are similar to class declarations, we make the class for AST nodes of Trait declarations an extension to JTClassDecl (called TraitDecl). In TraitDecl, we disallow fields to be declared, leaving only methods.

The behavior of a number of compilation passes that perform checking is also extended in order to get appropriate error messages when erroneous code (concerning Traits) is compiled. In a new compilation pass, we add Trait methods to the class declaration AST nodes. This is a collection of all the methods that a class gains from its Traits when it is ‘flattened’, following the rules of the Traits model. As a last step, the Trait declarations are translated.
to interface declarations, and their method bodies are discarded. The Traits used by a class become interfaces of the class in the class declarations.

### 4.3 Syntax

The language extension for JTL requires the following additions to the regular Java syntax:

- There is a new entity Trait (indicated by the keyword `trait`) that needs to be in a separate `.jtl` file like a (non-nested) Class or Interface.
- Trait entities can only have methods.
- Methods in Traits can be required or provided. Provided methods are regular methods with implementations. Required methods have no implementation (like methods in interfaces) and are indicated by the `required`-keyword.
- Classes and Traits can use other Traits with the `with`-keyword, which is similar to the `implements`-keyword for interfaces.
- Trait aliasing and exclusion is done with the `aliasing` and `excluding` keyword.

Next we give the formal definition of the extended syntax, using EBNF. In order to avoid giving the syntax definition for the entire Java language instead of just the extension, we use *italics* to indicate existing syntax structures that have not been modified. The non-terminals `traitName`, `className`, `interfaceName` and `methodName` are identifiers.

```
traitDeclaration = "trait" traitName [usedTraits] "{" traitMethod* "}"  
usedTraits = "with" usedTrait ("," usedTrait)*  
usedTrait = traitName [aliases] [exclusions]  
aliases = "aliasing" "{" alias ("," alias)* "}"  
alias = methodAlias "=" methodWithSignature  
methodAlias = methodName  
exclusions = "excluding" "{" exclusion ("," exclusion)* "}"  
exclusion = methodWithSignature  
traitMethod = requiredMethod | providedMethod  
requiredMethod = "required" methodWithoutBody  
providedMethod = methodWithBody  
methodWithoutBody = returnType methodWithSignature ";"  
methodWithBody = returnType methodWithSignature "{" method body "}"  
methodWithSignature = returnType methodName "(" signature ")"  
signature = formal parameter list  
returnType = "void" | type
```

The syntax for class declarations has been modified as follows:

```
classDeclaration = "class" className [superClass] [implementedInterfaces] 
                  [usedTraits] "{" class body "}"  
superClass = "extends" className  
implementedInterfaces = "implements" interfaceName ("," interfacesName)*
```
4.4 Compilation scheme

As mentioned before, Polyglot uses passes to do the compilation. In the first pass, the source files are parsed and become a tree structure of nodes. This tree is the abstract syntax tree (AST). Each pass is performed by an extension of NodeVisitor. Each node in the AST can be visited by a NodeVisitor. The NodeVisitor decides which operations it performs on a Node and which sub-nodes it visits. This whole idea is based on the visitor design pattern \cite{GHJV94} and allows for easy modification and extension of the passes.

In Figure 4.2, the passes of the Polyglot compiler are shown, as well as the new passes scheme of JTL. The visiting mechanism for a number of passes had to be modified (shown in italics). Also, two extra passes were added plus a barrier pass (shown in bold font).
4.4.1 Barrier passes

There are different jobs when multiple source files are compiled at once (every source file forms a separate job). Also, every class/interface/Trait that is used by the source files needs to be taken into account by the compiler, and thus becomes a job. A barrier pass is a special pass that serves synchronisation. It ensures that all jobs that a given job \( J \) depends on have completed at least as far as the last barrier pass that \( J \) has completed. A global barrier pass is a special pass that ensures that before a job \( J \) can run a global barrier pass, every currently active job must have completed all passes up to the same global barrier pass.

The use of (global) barrier passes is necessary when source files depend on each other. With Traits, classes (and Traits) that use Traits depend heavily on the Trait source files because source code is more or less copied into the classes. Because of this, in order for the JTL compiler to work correctly, we had to add barrier passes at two points in the compilation scheme.

4.4.2 Modified passes

For compilation with Traits, several passes had to be modified. Below we will discuss the major changes to the various existing passes. We will not give a technical discussion of every detail.

Parse. The parse phase runs the parser and constructs the AST for each job. For the new language constructs (Traits and classes that use Traits) new AST nodes have to be created. For a large part, we could make use of existing constructs such as classes, interfaces and methods, and extend them. For aliasing and exclusion, entirely new constructs were made, although not very complex and similar to each other (they both use ‘methods with signature’, but aliasing pairs a name (alias) to each one).

Build types. The type system is constructed in this pass, based on the information in the AST nodes. For Traits, this needs to be done as well. To be able to type check Traits and classes that use Traits, modifications were made to the type classes (each AST node gets an instance of a type class in this pass). For aliasing and exclusion, new type classes were created.

Clean superclasses/-interfaces. This pass is performed by AmbiguityRemover, which is a visitor in Polyglot. It looks at the inherited methods and variables in the superclass-/interface-hierarchy and adds related jobs to the job pool. Now that there are Traits as similar entities to classes and interfaces, they have to be looked at as well. That is why the pass had to be modified. If a class uses a Trait, that Trait is added to the job pool and its methods will be marked for checking in upcoming passes.

Clean signatures and Disambiguate. These two passes are also performed by the AmbiguityRemover. They look up and link the type references that AST nodes contain. Like with the ‘Clean superclasses/-interfaces’ pass, the addition of Traits require extra work for the AmbiguityRemover. In the methods that this visitor calls, we added code to facilitate this. Within Traits, we
4.4 Compilation scheme

used an extension (TraitMethod) of the existing construct for method declarations, and we made the Trait-construct itself an extension of the existing construct used for classes and interfaces. This allowed us to use existing operations that work on classes and interfaces for Traits. Changes of this nature were in fact made at many places, amongst which the passes 'Add members', the passes performed by AmbiguityRemover and some of the 'checking passes'.

Checking . With ‘checking passes’, we mean the passes from ‘Type checking’ up to ‘Circular constructor call checking’. These passes give errors when certain things are badly programmed. For example, if there is a return statement that will be executed before some other code in the method, the reachable checker will throw an error. In most passes, the checking is done at source-code level (within methods). This automatically goes well for Traits in most check passes, because Trait methods can be seen as regular methods by these passes.

At some points however, there are a number of things that have to be added for Traits. For example, aliased methods need to be added to the list of possible methods to call, methods from Traits must be added to the list of possible methods in the first place, you cannot have a required method in a class. Most of this extra checking and functionality is performed by the two new passes, so only small modifications were required in the methods that are called by the checking passes.

Translation . This pass makes sure that the output of the compiler is regular Java source. It translates AST nodes into String values. Because JTL uses new AST nodes, this pass needed to be changed. The regular Java output should contain no errors, so any errors that are in the source code have to be detected by one of the passes of our compiler. Two new passes ‘Trait methods finding’ and ‘Trait methods filling in’ will take care of the methods of classes that are inherited from Traits (including aliased ‘copies’ of methods). The ‘Translation’ pass uses the ‘Traits become interfaces’ technique that was presented in Section 3.4 to convert JTL to regular Java.

4.4.3 New passes

In the compilation scheme of JTL, there are two new passes, which are discussed in this section. Both passes are performed by an AST visitor class called TraitMethodCopier.

‘Trait method finding’-pass The TraitMethodCopier class runs through the entire AST, but only does something with ClassDecl (class declaration) nodes (which can be classes, interfaces or Traits). For the ‘Trait method finding’-pass, the TraitMethod copier looks at the Traits used by a ClassDecl.

If a ClassDecl does not use Traits, nothing happens. If it does, the methods that will be added to the ClassDecl by the Traits are collected. This is done by a complex method that works recursively (bottom-up) on the used Traits and takes in account all the rules concerning combination, exclusion and aliasing that are defined in Chapter 3. A list of Trait-methods is constructed this way. Each Trait method that is also a provided method contains a reference to the original definition (in a Trait) of the method. Now we just need to copy the Trait methods into the ClassDecl.
However, if we would copy the Traits methods entirely (including their implementation) into the ClassDecl, this will lead to unnecessary duplicate (type) checking of Trait methods that are used in multiple classes. Moreover – as we noticed in practise – it was not possible to copy a piece of the AST to another place, due to the bindings to the type system. Therefore we create new TraitMethod nodes in the Methods field of the ClassDecl, with correct bindings to the type system, but without method bodies. These TraitMethods redirect to their original method definitions when (type) checking, thereby creating an efficient way of (type) checking. You could compare this with the benefits gained (no redundancy) with pointers to (instead of copies of) objects in object-oriented programming languages.

To put the method bodies of the Trait methods into the class declaration at a later stage, we let this pass copy every method defined in a Trait (the actual AST node) along with its original position into a static collection of Trait methods that can potentially be used elsewhere and let the ‘Trait method filling in’-pass deal with the output.

‘Trait method filling in’-pass The ‘Trait method finding’-pass has done most of the work regarding the addition of Trait methods to classes and to Traits that use them. Only the output classes that use Traits will end up with methods without bodies. To fix this, the AST nodes of the method’s bodies need to be copied into the methods before the output is done.

The TraitMethods contain a reference to the original position of the method’s implementation. By looking up that position in the static collection of Trait methods that was built by the ‘Trait method finding’-pass, we can easily access the correct AST node. Now we can copy that piece of the AST (the method + its body with everything in it) into the ClassDecl, because all (type) checking is behind us (because of the added global barrier pass); The bindings to the type system will still not match, but that is now irrelevant. Thanks to the copied AST nodes of the method bodies, the output will now have the correct (and checked) copied source code in it.

4.5 Type safety

Now let’s look closer at a very important part of this project: type safety. Traits have in the past only been implemented for untyped languages. In Section 3.4, we have seen how we could translate Traits to interfaces. In Section 4.4, we have seen how this is done in the various passes.

According to the flattening property, classes that use Traits are the same as ‘flattened’ classes. The addition of Trait methods to a class yields a normal class. Using such classes for type checking will result in unclear error messages. Therefore, we want to type-check Traits themselves and – when a Trait contains errors – give error messages that refer to the Trait, and not to classes that use the Traits.

To do this, we define Traits as a sort of class that can define only methods. Methods are of course existing type constructs in Polyglot. Traits are based on the existing type construct of classes, and so get a lot of checking for free. There can be no instances of Traits; they can only be used by classes. This leaves us with only three major additions to the type system.
4.6 Remarks

• Classes (and Traits) can now use Traits. This is new, but very similar to implementing interfaces. The class type gets a new field (list of) Traits. Thanks to the visitor design pattern used in Polyglot, we can just make each visitor visit the Traits at each point where interfaces are visited in the regular Java compiler.

The methods from the Traits are added to the type of the class. Method calls on an instance of that class will now also take into account the class’ Trait methods while type checking.

Of course, the checking of Trait methods is different from interface methods, because Trait methods can have an implementation, and thus do not need to be implemented by a class.

• Exclusion of Trait methods is possible at the point where they are used. Only provided methods can be excluded and then just become required methods. Classes and Traits that exclude methods from used Traits, are still able to call the method. So exclusion is not a big issue. We only have to check whether classes do not end up with required methods, and that check is simple to add.

• Aliasing a method from a used Trait results in an extra method in the class. That extra method is just added to the list of Trait methods.

Most checks regarding the composition of Traits, exclusion and aliasing were built into the findValidTraitMethods method in the class JTParsedClassType_c (the type of class and Trait). The source code of the method is included in Appendix A.

4.6 Remarks

In this section, we have placed some remarks about the result of and choices made when implementing the compiler:

• The abstract keyword could have been used instead of the new required keyword. But we made the decision to introduce a new keyword for clearer distinction between abstract classes and Traits.

• In JTL, interfaces almost become obsolete. Traits are more powerful than interfaces because the methods can have implementations. Interfaces however, can define static final variables, something that cannot be done within Traits. Instead of writing an interface that has some methods, one could also write a Trait with the same (required) methods. However, we think that it is better to keep using interfaces as strictly implementationless entities, and start using Traits as partly implemented entities. This way, you have different types of entities available, for different programming techniques. In other words: hammering a nail in a board and pulling it out leaves a hole, but it is more sensible to use a hammer for what it’s meant (hammering nails in), and not for things a drill is meant to do (making holes). The drill here being interfaces and the hammer Traits.

• In JTL, the same overloading rules apply as in Java. Methods with different return types and the same parameter types are not accepted.

• The super-keyword is unchecked within Traits. Polyglot outputs regular Java source code and compiles that with the standard Java compiler, which only at that stage
detects errors with super-keywords in class methods that were inherited from Traits. This is a bit of a flaw in the compiler and should probably be fixed in a new version of the JTL compiler. In Chapter 6, we look at this problem in more detail.

- The documentation of Polyglot is limited, and because of that we found it quite hard to get to know. We suggest that better documentation be made. That way, the framework will become more appealing to programmers that want to make language extensions to Java.

- If a method is defined in a class, which would otherwise be inherited from a Trait, which that class uses, this is accepted by the compiler. It is assumed that the programmer overrides the Trait method on purpose. It might be better to deal with such a case by yielding an override warning or even an error when compiling. By doing so, programmers would be forced to explicitly exclude Trait methods that they want to override. Although this overriding is allowed in the Traits model, it can lead to unexpected behavior. For example, when a programmer is not aware of the methods of a used Trait, which a class uses, and ignorantly adds a method to that class which is equal to a Trait method. Compiling would now yield no warning, but this example shows that there is something to be said for one. On the other hand, by forcing method exclusion, headers of classes that use Traits could become littered with exclusion statements, which otherwise would not have been necessary.

4.7 Package

The JTL compiler is an extension to Java 1.4 built upon the compiler front end framework Polyglot 1.2. In order to run the compiler, one needs to have a Java software development kit (SDK) installed, for example, Sun’s JDK [Sun]. To compile the source code, one needs to have Ant installed [Ant]. The source code and compiled Java bytecode binaries are available on SourceForge.net. Installation instructions and example .jtl files are also included. The URL is [http://sourceforge.net/projects/javatraits/](http://sourceforge.net/projects/javatraits/)
Chapter 5

Case study

In this chapter we will discuss case studies concerned with the Java class library. Sun’s Java class library contains a lot of duplicate code. We ran a simple copy-paste detection scanner (CPD) \cite{CPD}, which uses the fast (but naive) Karp-Rabin string matching algorithm \cite{KR87}. Over the \texttt{java.*} classes, a simple scan reported over 15 thousand lines of duplicated code spread over 205 top-level classes or interfaces. This is more than 17 percent of the total number of classes. If we would use a more sophisticated scanner the amount would probably be even greater.

As a result, we came up with two case studies that use Traits to improve the Java source code. The first (small) case study is called \textit{Printing protocol} and is described in Section \ref{sec:printing}. The second, larger case study is called \textit{AWTListeners} and is described in Section \ref{sec:awtlisteners}. In Section \ref{sec:evaluation}, we will evaluate the case studies. The resulting source code of both case studies can be found in the tests-directory of the JTL package. The exact location of these files is mentioned at the end of each corresponding section.

5.1 Small case study: \textit{Printing protocol}

In Figure \ref{fig:printstream}, the class hierarchy is shown for the PrintStream and PrintWriter classes of the Java class library. We chose this example because these classes were mentioned as an example of duplicate code in a paper by Quitslund and Black \cite{QB04}.

Both classes have \texttt{print} and \texttt{println} methods for the Java primitive types \texttt{boolean}, \texttt{char}, \texttt{char[]}, \texttt{double}, \texttt{float}, \texttt{int}, \texttt{long}, \texttt{Object}, and \texttt{String}. These methods are more or less the same for both classes.

The only difference is that for the \texttt{println}-methods, a PrintStream uses its current instance (\texttt{this}) to make the operation thread-safe, whereas a PrintWriter uses a variable \texttt{lock}. First we made them truly duplicate by introducing a \texttt{getLock()} method to both classes, which will return the \texttt{lock}-variable in PrintWriter, and the \texttt{this}-reference in PrintStream. Then we remove these duplicate methods by refactoring them to a Trait \texttt{TPrintTypes}. Since the \texttt{println}-methods now use a \texttt{getLock()} method, we must make this a \texttt{required} method of the Trait. The \texttt{println()}-method is also duplicated in both classes, so we put that in the Trait as well. All that the \texttt{println()}-method does, is call the \texttt{newLine()}-method, so we must make \texttt{newLine()} a \texttt{required} method in the Trait.
Duplicate code!

Figure 5.1: Classes PrintStream and PrintWriter do not share a common superclass to provide the various print methods that both classes have, which leads to duplicate code.

The print-methods all make use of the write(String) method that is provided by both classes. In the Trait this becomes a required method as well.

Both classes will use this Trait. This way, the duplicate code is successfully removed by using Traits. The total file size has become 28 percent smaller and the code has become more maintainable, because future changes to the protocol can be done at one point (the Trait), instead of in both classes. This Trait can be used by other classes that need to print Java types. This increases code reuse, speeds up the development process and makes it more error-resistant and uniform.

The result of the refactoring of the two classes is illustrated in Figure 5.2. The source code of the Trait TPrintTypes is listed in Appendix B. The source code of the refactored classes and the Trait, as well as the original classes, is included in the JTL package (see Section 4.7), in the subdirectory src/polyglot/ext/jt/tests/printprotocol.

5.2 Larger case study: AWTListeners

As a larger case study, we have looked at Java’s Abstract Windowing Toolkit (AWT), which provides user interface components. We detected duplicate code in the source code of classes that allow EventListeners to be added. We chose this as a case study, because these classes are widely used by Java programmers, and because all of these classes were reported by CPD when we did a duplicate code scan on java.*. Event listeners receive a
method call from the class they are added to when a certain event occurs. For example, if a button is pushed or if text is typed in a text field. Classes are event listeners if they implement the interface of the event listener. There are a lot of different event listeners in Java, but for this case study, we have only looked the ones that can be added to one or more subclasses of Component or MenuComponent. These are:

- ActionListener. The listener for receiving action events, such as the push of a button. If such an event occurs in an object, all its ActionListeners receive a call to the method actionPerformed(ActionEvent e).

- ItemListener. The listener for receiving changes to the items of a component, such as the selection or deselection of a checkbox. If such an event occurs in an object, all its ItemListeners receive a call to the method itemStateChanged(ItemEvent e).

- TextListener. The listener for receiving changes to a text. If such an event occurs in an object, all its TextListeners receive a call to the method textValueChanged.

- AdjustmentListener. The listener for receiving adjustments to a movable part of a component, such as the sliding of a scroll bar. If such an event occurs in an object, all its AdjustmentListeners receive a call to the method adjustmentValueChanged.
5.2 Larger case study: AWTListeners

Figure 5.3: The class hierarchy of AWT components that are relevant in our case study.

In Figure 5.3, we see the classes that allow the relevant listeners to be added. The code for dealing with listeners is repeated in each class, because this functionality does not belong to the common base class Component. Therefore, if we compare the code that allows listeners in CheckBox.java and Choice.java, we see a lot of completely duplicate code. These duplicate methods are listed in Listing 5.1.

By putting the methods that handle adding and removing listeners into Traits, we can eliminate the duplicate code. Since TextComponent is the only component that has code which allows ItemListeners to be added, we could choose to leave it alone and focus on the other two listeners. But, since TextField allows both ActionListeners and (through inheritance) ItemListeners, we need to combine these two Traits in that class.

5.2.1 ItemListeners

We created the Trait TAllowsItemListeners, and modified Choice, Checkbox, and CheckboxMenuItem so that they used this Trait. The duplicate methods from both classes (Listing 5.1) were moved to the Trait. Access to the local variables ItemListener, eventMask and newEventsOnly within these methods was changed into get/set methods. These methods are required in the Trait and are regular get/set-methods in the classes. In fact, we added the methods setNewEventsOnly and getEventMask to the classes Component and MenuComponent, because these methods refer to variables, which...
Larger case study: AWTListeners

Listing 5.1: Duplicate code in CheckBox.java and Choice.java

```java
public synchronized void addItemListener(ItemListener l) {
    if (l == null) {
        return;
    }
    itemListener = AWTEventMulticaster.add(itemListener, l);
    newEventsOnly = true;
}

public synchronized void removeItemListener(ItemListener l) {
    if (l == null) {
        return;
    }
    itemListener = AWTEventMulticaster.remove(itemListener, l);
}

public synchronized ItemListener[] getItemListeners() {
    return (ItemListener[]) (getListeners(ItemListener.class));
}

public EventListener[] getListeners(Class listenerType) {
    EventListener l = null;
    if (listenerType == ItemListener.class) {
        l = itemListener;
    } else {
        return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
}

boolean eventEnabled(AWTEvent e) {
    if (e.id == ItemEvent.ITEM_STATE_CHANGED) {
        if (!(eventMask & AWTEvent.ITEM_EVENT_MASK) != 0 ||
            itemListener != null) {
            return true;
        }
    }
    return super.eventEnabled(e);
}

protected void processEvent(AWTEvent e) {
    if (e instanceof ItemEvent) {
        processItemEvent((ItemEvent)e);
        return;
    }
    super.processEvent(e);
}

protected void processItemEvent(ItemEvent e) {
    ItemListener listener = itemListener;
    if (listener != null) {
        listener.itemStateChanged(e);
    }
}
```
are defined in those classes. This way, any subclass of Component or MenuComponent that uses the Trait TAllowsItemListeners, already implements two of its required methods. The two other methods must be implemented in the subclass like this:

```java
public ItemListener getItemListener() { return itemListener; }
public void setItemListener(ItemListener il) { itemListener = il; }
```

So, once we have the Trait, the changes to the class are easy:
- Add with TAllowsItemListeners to the class header.
- Add the two get/set-methods shown above.
- Remove the methods that are provided by the Trait.

The source code of TAllowsItemListeners can be found in Appendix C.1. The component List allows ItemListeners as well as ActionListeners to be added. This is covered in Section 5.2.4.

5.2.2 ActionListeners

For ActionListeners we created the Trait TAllowsActionListeners. This process was largely the same as for ItemListeners. A difference was that with the class MenuItem, its superclass MenuComponent didn’t define the method getListeners, because it can add no listeners. Because of that, it defines a method getListeners that is slightly different from the one used in e.g. Button. This causes the super-call to getListeners from the following piece of code from TAllowsActionListeners to be invalid:

```java
public EventListener[] getListeners(Class listenerType) {
    EventListener l = null;
    if (listenerType == ActionListener.class) {
        l = getActionListener();
    } else {
        return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
}
```

To solve this we maintained the getListeners method in MenuItem, because it is not duplicate. The rest of the methods are duplicate, so they can be deleted from the code. This way, the getListeners-method in MenuItem overrides the (wrong) Trait method from TAllowsActionListeners, resulting in the desired behavior.

The source code of TAllowsActionListeners can be found in Appendix C.2.

5.2.3 TextListeners and AdjustmentListeners

The class TextComponent is the only class covered here that allows TextListeners to be added. The class TextField does not have duplicate methods, because it inherits them from its superclass TextComponent. However, for consistency reasons, we have also refactored TextComponent into a class that uses a Trait (called TAllowsTextListeners) to gain this functionality. The process of doing so was very similar to that of the ItemListeners,
and therefore not worth discussing here. The source code of T AllowsTextListeners can be found in Appendix C.3.

The same holds for Scrollbar, which is the only class covered here that allows listeners of the type AdjustmentListener to be added. Again, we refactored this AWT class so that it uses a Trait to gain this functionality. This Trait is called T AllowsAdjustmentListeners. The source code of T AllowsAdjustmentListeners can be found in Appendix C.4.

What we achieve with this is separation of concerns and scalability. Separation of concerns because the code for the ability to add listeners is kept in a separate file. And scalability because we can now build more classes that can easily adapt this functionality (by using the Trait), instead of having to write duplicate code.

5.2.4 Combined ActionListeners and ItemListeners

The class list uses both ActionListeners and ItemListeners. It would be an easy solution to just add "with T AllowsActionListeners, T AllowsItemListeners" to the class header. However, these two Traits have overlapping methods. This means that if this is done, the JTL compiler will tell us that methods getListeners, eventEnabled and processEvent conflict.

The conflicts could be solved by excluding conflicting methods from one Trait. But that would mean we will miss some functionality. Aliasing the conflicting methods and providing a new method that calls the aliased method one at a time is another option. Unfortunately, that does not work in this case, because these methods have super-calls in them. This could lead to unwanted behavior if a superclass of the class in which the trait is used, also allows Item- or ActionListeners. use a mechanism with which the superclass’ method is called.

Ultimately we have chosen to create a new Trait that combines both other Traits and – for the conflicting methods – provides its own implementation (taken from List.java.

The entire source code of T AllowsActionAndItemListeners can be found in Appendix C.5

This Trait does not define required methods, because they are ‘inherited’ from the Traits it uses. Despite being equal in both used Traits, the methods setNewEventsOnly and getEventMask do not conflict here, because they are required methods. As explained in Chapter 3, required methods do not cause conflicts.

The source code of all refactored classes and the Traits, as well as the original AWT-classes, are included in the JTL package, in the subdirectory src/polyglot/ext/jt/tests/listeners.

5.3 Evaluation of the case studies

We have seen that Traits can be effectively used to refactor existing code into better structured code that contains significantly less duplicate code. As a result we can achieve a better separation of concerns.

The case studies show two (not entirely orthogonal) key situations in which the use Traits is useful:

- When classes that are at different places in an inheritance hierarchy share a common protocol. An example of this are the PrintStream and PrintWriter classes, which
print types as a string in the same way, but are not related in the class hierarchy. In such a case, a Trait can be created that provides the methods that implement the protocol. The classes that use the protocol can subsequently use the Trait. If there are any changes to the protocol in the future, then modifications only have to be made in one place (the Trait), instead of at all places where the protocol is used.

- When sibling classes share specific features, which should not be available in their common superclass. An example of this is the use of various listeners in the AWT components. The Choice and Checkbox components both allow ItemListener to be added and removed, but their common superclass, Component doesn’t. In some cases it would be possible to define an in-between common superclass (a class that inherits from Component and also allows ItemListeners), and have the classes that share code inherit from that class. In this case that would mean there also needs to be an in-between common superclass for components that use ActionListener – which is not available in Java – to inherit from both classes. Fortunately, Traits can be a container for shared functionality, and a class can use multiple Traits. By putting specific functionality in separate entities (Traits in this case), thus detaching it from the actual classes, the class structure also becomes clearer and the code more to the point.

In short, Traits can be used to eliminate duplicate methods, and to avoid them, but they also help in improving the structure of software systems.
Chapter 6

Discussion

In the previous chapters, we have already answered the research questions posed in the introduction. These were ‘What is wrong with existing inheritance structures?’, ‘How do we add Traits to Java?’, and ‘How do Traits improve Object Oriented programming in Java?’. In this chapter, we will discuss the benefits we have gained by adding Traits to Java. Nevertheless, there are also some limitations to our new programming language, that have to be taken into account. These are also described. Most of these limitations could very well be lifted by improving the compiler. Possible improvements to the compiler are discussed in Section 6.3: Future work. We sum up our findings in a conclusion in Section 6.4.

6.1 Benefits of JTL

In Chapter 3, we have already seen what benefits the Trait model has. Because JTL is an implementation of the Trait model, the new language also has those benefits. We discuss the benefits of JTL below.

**Better and easier software design.** Traits can be used as *units of reuse*, encompassing as group of methods, which can be used anywhere in the inheritance hierarchy. Regular Java only has classes as *units of reuse*, which can lead to awkward choices at the point of inheritance hierarchy design. By using Traits in the *software design*, this problem is solved. As we saw in Chapter 5, by putting a characteristic behavior that is shared by multiple classes into a Trait, the inheritance hierarchy becomes easier to design, whilst avoiding duplicate code. If at one point in the future, the shared behavior changes (for example, a used communication protocol), the change can be made at a single, centralized point, namely in the Trait that defines that behavior.

**Existing code can be improved.** As seen in the case study, Traits can be used to *refactor* source code that contains duplicate code. When doing so, the common (duplicate) methods are placed in a Trait. The methods are deleted from the classes defined them, and instead the classes use the Trait. This way, the source code becomes more uniform and smaller, and thereby more *maintainable*. 
6.2 Limits of JTL

**Exclusion and aliasing.** Because of exclusion, aliasing and method overriding, using Traits is very flexible. Conflicting methods can be neatly resolved while maintaining access to each original method. Classes can still define their own methods, overriding methods from a Trait, if there is need for a different implementation than what the Trait provides (as discussed in Section 5.2.2).

**Compositional power is gained.** Traits can be easily combined, forming new Traits. This allows for a fine-grained system of light-weight Traits, and reuse of Traits by other Traits. Having small and clear Traits results in clean source code and improves understandability and maintainability of software systems.

**The problems found in other inheritance structures are solved.** The problems described in Section 2.2 of using Multiple Inheritance are solved when using Traits instead. The problem described in Section 2.5 of using Mixins is also solved when using Traits instead. Furthermore, using Traits fills the void of reusability that interface delegation (Section 2.4) has.

### 6.2 Limits of JTL

The JTL compiler has some limitations, which are described below.

**Limited possibility for references to superclasses in Traits.** The super-keyword is used in Java to give subclasses access to methods, constructors and fields from the superclass that are overridden in the subclass. Traits have no subclasses, so super has no meaning. According to the Traits specification, super-calls are allowed in Traits; they will just result in super-calls in the flattened code. But in a strongly-typed language like Java, we must also check whether every method call is valid – including super-calls in Traits. Since Traits are separate entities when compiling, it is not known at compile-time by which classes the Trait will be used. In other words, a Trait does not know which classes will use it. Consequently, it also does not know what the type of their superclass is. Therefore we cannot type check any references to super in traits at compile time.

There are three possible kinds of super references: field references, constructor calls and method calls. Superclass field references are not allowed, because state is not allowed in Traits. A Traits does not have constructors, so super-constructor-calls are not applicable. So the problem lies with super-method-calls. Therefore, only superclass method calls are relevant in the context of traits.

We decided to allow super-method-calls in Traits, because by disallowing them Traits would lose a lot of expressive power. But we allowed these calls on the condition that they are calls to methods also defined in the Trait. That way, super-calls can be checked as if they were normal method calls within a Trait. So instead of not checking the super-calls at all, they are checked, only not entirely safely. The programmer is forced to add methods to the Trait (either required or provided) that have to correspond to superclass-method from the classes that will use the Trait. These added methods will not bloat the resulting classes, since their superclass(es) should already have that method defined.
Super-method-calls could be made safe by doing an extra check at the point where classes use Traits. If a Trait is used, in which super-method-calls are present, the superclass of the class that uses the Trait must have those methods. However, we have not (yet) implemented this check. The entire JTL compiler exists of two parts: the translation of JTL to Java, and the compilation of the Java sources to bytecode. All checking is to be done in the first part, so the second part won’t produce any errors at all. Because of the problem described above, this claim is not entirely true. However, these errors will be reported by the regular Java compiler. These errors will report a bad super-call, and can thus be easily tracked down by the programmer.

**Limited support for visibility modifiers.** Trait methods do not support all of Java’s visibility modifiers (public, private, and protected). This can be attributed to the fact that Traits are translated into interfaces in compilation, and interfaces can only define public methods. As a result, Traits can also only define public methods. This shows the drawbacks of this otherwise effective approach.

If we choose to allow protected and private methods in Traits and change them to public in the part of the compilation process where the translation to interfaces happens, then the classes that use them will have private and protected methods. But then the regular Java compiler will complain that the visibility of the implemented method (in the class) becomes less than where it is defined (in the interface that was translated from the Trait), so that is not an option. If it was possible to define protected and private methods in Traits, JTL would be a better language than it is now. Unfortunately, while the interfaces-approach is maintained, this will not be possible.

For the same reason, synchronized methods are not allowed in Traits. Fortunately it is allowed to add the keyword synchronized to methods in classes, even if they are defined in an interface without that keyword. However, making a Trait method synchronized requires extra code in a class, in combination with aliasing the unsynchronized method.

**No state.** Traits have no state, and cannot access state variables. This leads to get- and set-methods. If there are a lot of these methods necessary for one Trait, and the Trait is used in many classes, the get- and set-methods that those classes are required to implement are likely duplicate methods, which is what we were trying to avoid in the first place! Fortunately, these methods are very simple and cause only one line of duplicate code per method.

### 6.3 Future work

**Upgrade to Java 1.5.** JTL is basically Java 1.4 extended with Traits. While developing it, Sun released Java 1.5. This new version has a lot of new features. Polyglot – the extensible compiler front-end – is as of yet only available for Java 1.4. The developers of Polyglot will start with version 2.0 in the spring of 2005. If that version supports Java 1.5, JTL could probably be adapted easily to this new version of Polyglot, because JTL is not integrated in the Polyglot framework, but developed as an extension. Undoubtedly some upgrades will need to be made to get Java-with-Trait working for Java 1.5.
Fix the limitations that were given in Section 6.2.

- The problem of super-method-calls described in Section 6.2 should be solved. This can be done by adding required super-methods, a new concept.

For example, if you define required super void find() in a Trait, it becomes possible for methods of the Trait to call super.find(). And, classes that use that Trait should have a superclass that defines the find()-method. This will however rather limit the amount of classes that can use such a Trait, so perhaps a better solution can be thought of. Anyway, for refactoring purposes, required super-methods can be very useful. For instance, when dealing with duplicate code, which contains super-calls to a common ancestor of a set of classes (like the subclasses of Component in Section 5.2).

- The problem of duplicate get- and set methods caused by the lack of state in Traits can be solved. A possible solution is to introduce required fields. As with required methods, classes that use Traits with required fields need to provide these fields. Traits can then use the required fields as if they were actual fields. An extra check needs to be added to make sure that classes do provide all the required fields.

Another approach to required fields is to translate each one of them automatically into a required get- and set-method. In classes, programmers can then use differently named variables to implement the required fields. But then, there would still be code duplication in the get- and set-methods. This can be solved by a special keyword (e.g. property) that automatically will transform a variable to a get- and/or set-method, thus providing the required methods. The keyword could have an optional parameter that allows get- and set- methods of different names than the variable’s name to be created. So for example in a Trait, you would have required String x. A class that uses that Trait could provide that field by either defining property String x, or a differently named property{x} String y.

By using only required fields, the diamond problem will not occur, because the fields do not have values in the Traits.

- The approach that we have chosen to translate Traits into interfaces, is practical, but does have its disadvantages, as seen in Section 6.2. Time is needed to work on a different approach, that will improve this weakness.

Improving Swing with Traits. As shown by Quitslund and Black [QB04], the amount of duplicate code in the Java Swing package is very large. By refactoring the Swing classes using Traits, this amount could be reduced to a minimum.

Traits in other languages. As this project has successfully implemented Traits in a strongly-typed language, it could well be imagined that other such languages will be extended with Traits. An entirely new language could also be developed that features Traits and uses them as building blocks for the main class library. This will greatly help to avoid duplicate code therein, as is the case with Java (as seen in Chapter 5).

Trait library. A Trait library could be created for Java, combined with a Trait browser. With this, programmers could develop classes in a new way, using the existing Traits as
building blocks. New Traits can of course be added to the library.

**Design patterns with Traits.** Existing design patterns can perhaps be made easier when Traits are possible in the programming language. This can be researched and as a result, a collection of new design patterns for JTL can be delivered.

**Automatic refactoring.** A tool could be made that detects duplicate code and automatically refactors it into a Trait, which is consequently used in the classes that contain the duplicate code. Although the result of such an automated tool will probably not be as good as doing the work manually, it will be a lot faster and can help to jumpstart a later refactoring process, which will be done manually.

**Information hiding with Traits.** Traits can also be used for code generation. For example a software company may use the JTL compiler to generate regular Java source code. The company uses Traits internally, but it sends out regular Java source code, or Java bytecode to its customers. The company thereby keeps its real (JTL) source code somewhat secret, and the shipped source code contains duplicate code. The company has an advantage, in that it does not reveal its real source code. With Java, exposure of source code is sometimes problematic as even bytecode can mostly be easily decompiled (translated back into source code), and companies do not want their source code exposed publicly. By making extensive use of Traits, the real source code is not exposed as easily as with regular Java. Also, this gives a company the benefit of reuse, without requiring their customers to have a Trait compiler installed.

**Compiling and running independently of regular Java.** Traits are translated into regular Java, and from there into Java bytecode. By doing so, duplicate code is avoided at the JTL level, but reappears in the compiled code. It may be possible to translate JTL directly into bytecode. This bytecode will also need to be an extension of regular Java bytecode, as it must contain codes for Traits, aliasing, etc. For this bytecode, a special virtual machine must also be written.

If all this is done, JTL is a fully fledged programming language, with its own run-time environment (now it uses that of normal Java). This might seem like writing a new language altogether, but just like we did with the Java compiler Polyglot – we only need an extension to the existing Java bytecode and virtual machine. Most of the work is already done: only the Trait parts need to be added.

First, one needs to research if such extensions are possible and necessary. Advantages that we can think of are independence of the regular Java compiler, and less compiled code (since Trait method do not need to be copied into classes).

### 6.4 Conclusion

In this thesis, we have seen that Traits are a welcome addition to Java, and to OO programming in general. We have looked at various inheritance structures and other ways of code reuse in object-oriented programming languages. By comparing them, we have seen that by using Traits, many shortcomings of other inheritance structures (and therefore of
many modern programming languages) can be solved. In combination with single inheritance and interfaces – which are already present in Java – the use of Traits leads to more code reuse, better maintainability and understandability of code, a finer-grained approach to class composition, and less duplicate code. We have seen how adding Traits provides a solution to inheritance problems that lead to duplicate code.

We have shown how Traits work, what rules apply when using them, and how they can be made to fit in with Java's type system. Next we presented our compiler, which type-checks and translates the JTL language into regular Java, and ultimately into Java byte code using a regular Java compiler. We have shown how we did this and what issues we have dealt with.

In the case study presented in Chapter 5 we have shown that the use of Traits improves the source code of software that almost every Java programmer uses, namely Sun’s Java class library. From this we can conclude that JTL makes Java a better and more complete programming language.
Appendix A

Java method findValidTraitMethods

Method findValidTraitMethods from the file JTParsedClassType.java in package polyglot.ext.jt.types (of the JTL compiler). Recursively finds all methods that should be actually added to this class. The recursion works with a bottom-up traversal. Returns a List of MethodInstance.

```java
protected List findValidTraitMethods() throws SemanticException {
    if (!flags.contains(TraitFlags.TRAIT) && traits().isEmpty())
        return new LinkedList();

    LinkedList reqAll = new LinkedList(); // required methods from supertraits
    LinkedList provAll = new LinkedList(); // provided methods from supertraits

    for (Iterator it = traits().iterator(); it.hasNext();)
        LinkedList req = new LinkedList(); // required methods from this supertrait
        LinkedList prov = new LinkedList(); // provided methods from this supertrait

        JTClassType trait = (JTClassType) it.next();
        // Recursive call to find the basic supertrait's method suite
        List traitMethods = trait.findTraitMethods();

        // Divide the required and provided methods
        for (Iterator it2 = traitMethods.iterator(); it2.hasNext();)
            MethodInstance mi = (MethodInstance) it2.next();

            if (mi.flags().contains(TraitFlags.REQUIRED)) {
                req.add(mi);
            } else {
                prov.add(mi);
            }

    // Exclusion, from Fisher, Reppy: Statically typed traits:
    // Excluding a method from a trait causes its definition to be
    // removed from the traits methods, but it also causes the excluded
    // method to be added to the list of required methods, which is
    // necessary because the method may be mentioned in one of the
    // traits remaining methods.
    for (Iterator it1 = trait.exclusions().iterator(); it1.hasNext();)
        MethodSignatureInstance msi = (MethodSignatureInstance) it1.next();

        boolean found = false;
        for (Iterator it2 = prov.iterator(); it2.hasNext() & !found;)
            MethodInstance mi = (MethodInstance) it2.next();

            if (msi.equalsMI(mi)) {
                it2.remove();
                mi = mi.flags(mi.flags().set(TraitFlags.REQUIRED)).
                    set(TraitFlags.REQUIRED);
                req.add(mi);
                found = true;
            }
```
```java
public MethodInstance[] findValidTraitMethods() {
    ArrayList<MethodInstance> reqAll = new ArrayList<MethodInstance>());
    ArrayList<MethodInstance> provAll = new ArrayList<MethodInstance>());
    LinkedHashMap<MethodInstance, MethodInstance> aliased = new LinkedHashMap<MethodInstance, MethodInstance>();

    // Aliasing
    for (Iterator it1 = trait.aliases().iterator(); it1.hasNext();)
        MethodAliasInstance mai = (MethodAliasInstance) it1.next();
    boolean found = false;
    // Then in the provided methods
    for (Iterator it2 = prov.iterator(); !found && it2.hasNext();)
        MethodInstance mi = (MethodInstance) it2.next();
    if (mai.method().equalsMI(mi)) {
        prov.add(mi.name(mai.alias()));
        found = true;
    }
    if (!found) {
        // Look in the required methods
        for (Iterator it2 = req.iterator(); it2.hasNext();)
            MethodInstance mi2 = (MethodInstance) it2.next();
        if (mai.method().equalsMI(mi)) {
            reqAll.addLast(mi);
            throw new SemanticException("Cannot alias a required method", mi.position());
        }
    }
    throw new SemanticException("Aliased method not found", mai.method().position());
}

while (!req.isEmpty()) {
    reqAll.addLast((MethodInstance) req.removeFirst());
}
while (!prov.isEmpty()) {
    provAll.addLast((MethodInstance) prov.removeFirst());
}

// Check for conflicting provided methods
LinkedHashMap<MethodInstance, MethodInstance> checkedProv = new LinkedHashMap<MethodInstance, MethodInstance>(); //list of provided methods from supertraits, checked for conflicts
while (!provAll.isEmpty()) {
    MethodInstance mi = (MethodInstance) provAll.removeFirst();
    if (mi.requiresTrait) {
        for (Iterator it = reqAll.iterator(); it.hasNext();)
            MethodInstance mi2 = (MethodInstance) it.next();
        if (mi2.flags(mi2.flags().clear(TraitFlags.REQUIRED)).isSameMethod(mi)) {
            it.remove();
        }
    }
    boolean conflict = false;
    for (Iterator it = provAll.iterator(); it.hasNext();)
        MethodInstance mi2 = (MethodInstance) it.next();
    // Check for methods with same signature
    if (mi.name().equals(mi2.name()) && mi.formalTypes().equals(mi2.
formalTypes()) {
        // If they have the same implementation (i.e. defined at same position),
        // then there is no real conflict: the duplicates will just be removed.
    }
```
if (!mi.position().equals(mi2.position())) {
    conflict = true;
    //if (conflict) System.out.println("Conflict:"+mi2);
}

if (!mi2.position().equals(mi.position()))
    it.remove();

// The conflicting method become a required method
if (conflict)
    mi = mi.flags(mi.flags().set(TraitFlags.REQUIRED));
    reqAll.add(mi);
else
    checkedProv.addLast(mi);

// The conflicting method become a required method
if (mi2.flags().contains(TraitFlags.REQUIRED))
    for (Iterator it = checkedProv.iterator(); it.hasNext(); ) {
        MethodInstance mi2 = (MethodInstance) it.next();
        if (reqAll.iterator().hasNext()) {
            MethodInstance mi = (MethodInstance) it.next();
            if (mi.isSameMethod(mi2))
                throw new SemanticException("Method in this + this +
                Use exclusion to make it required.", mi.position());
        }
    }
else
    // It is a provided method
    // Check if a previously provided method is provided by the class
    for (Iterator it = reqAll.iterator(); it.hasNext(); ) {
        MethodInstance mi = (MethodInstance) it.next();
        if (mi.isSameMethod(mi2))
            it.remove();
    }

// Check if a previously provided method is provided by the class (trait method is overridden)
for (Iterator it = checkedProv.iterator(); it.hasNext(); ) {
    MethodInstance mi2 = (MethodInstance) it.next();
    if (mi.isSameMethod(mi2))
        it2.remove();
}

if (flags().contains(TraitFlags.TRAIT))
    localMethods.add(mi);

} // Compare the required methods to the methods provided by the superclasses

if (!flags().contains(TraitFlags.TRAIT) && superType() != null) {
    JTParsedClassType superClass = (JTParsedClassType)superType();
    while (superClass != null)
        for (Iterator it = superClass.methods().iterator(); it.hasNext(); ) {
            MethodInstance mi = (MethodInstance) it.next();
            if (mi.flags().contains(TraitFlags.PUBLIC))
                // Check if a previously required method is provided by the superclass
                for (Iterator it2 = reqAll.iterator(); it2.hasNext(); ) {
                    MethodInstance mi2 = (MethodInstance) it2.next();
                    if (mi.isSameMethod(mi2))
                        it2.remove();
                }
        }
    // A trait method must override a method that is also provided by the superclass,
    // so we leave the trait method in and this will happen automatically.
}
superClass = (JTParsedClassType) superClass.superType();
}

localMethods.addAll(reqAll);
localMethods.addAll(checkedProv);
//debug
System.out.println(name()+".."+localMethods+"\n");
//return localMethods;
}
Appendix B

**Trait** TPrintTypes.jt

Here listed is the source code of the trait that eliminates the duplicate code in the Java classes java.io.PrintStream and java.io.PrintWriter. The (JavaDoc) comments have been removed from the code for easier reading.

```java
trait TPrintTypes {
  required Object getLock();
  required void newLine();
  required void write(String s);

  public void print(boolean b) {
    write(b ? "true" : "false");
  }

  public void print(char c) {
    write(String.valueOf(c));
  }

  public void print(int i) {
    write(String.valueOf(i));
  }

  public void print(long l) {
    write(String.valueOf(l));
  }

  public void print(float f) {
    write(String.valueOf(f));
  }

  public void print(double d) {
    write(String.valueOf(d));
  }

  public void print(char s[]) {
    write(s);
  }

  public void print(String s) {
    if (s == null) {
      s = "null";
    }
    write(s);
  }

  public void println() {
```

```java
```
public void println(boolean x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(char x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(int x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(long x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(float x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(double x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(char x[]) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(String x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}

public void println(Object x) {
    synchronized (getLock()) {
        print(x);
        println();
    }
}
C.1 TAllowsItemListeners

```java
package java.awt;
import java.awt.event.*;
import java.util.EventListener;

trait TAllowsItemListeners {
  public required ItemListener getItemListener();
  public required void setItemListener(ItemListener il);
  public required void setNewEventsOnly(boolean b);
  public required long getEventMask();

  public synchronized void addItemListener(ItemListener l) {
    if (l == null) {
      return;
    }
    setItemListener(AWTEventMulticaster.add(getItemListener(), l));
    setNewEventsOnly(true);
  }

  public synchronized void removeltemListener(ItemListener l) {
    if (l == null) {
      return;
    }
    setItemListener(AWTEventMulticaster.remove(getItemListener(), l));
  }

  public synchronized ItemListener[] getItemListeners() {
    return (ItemListenr []) (getListeners(ItemListener.class));
  }

  public EventListener[] getListeners(Class listenerType) {
    EventListener l = null;
    if (listenerType == ItemListener.class) {
      l = getItemListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }

  public boolean eventEnabled(AWTEvent e) {
    if (e.id == ItemEvent.ITEM_STATE_CHANGED) {
```

C.2 T AllowsActionListeners

```java
package java.awt;
import java.awt.event.*;
import java.util.EventListener;

trait T AllowsActionListeners {
  public required ActionListener getActionListener();
  public required void setActionListener(ActionListener al);
  public required void setNewEventsOnly(boolean b);
  public required long getEventMask();

  public synchronized void addActionListener(ActionListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.add(getActionListener(), l));
    setNewEventsOnly(true);
  }

  public synchronized void removeActionListener(ActionListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.remove(getActionListener(), l));
  }

  public synchronized ActionListener[] getActionListeners() {
    return (ActionListener[]) (getListeners(ActionListener.class));
  }

  public EventListener[] getListeners(Class listenerType) {
    EventListener l = null;
    if (listenerType == ActionListener.class) {
      l = getActionListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }

  public synchronized void addItemListener(ItemListener l) {
    if ((getEventMask() & AWTEvent.ITEM_EVENT_MASK) != 0 ||
        getItemListener() != null) {
      return true;
    }
    return false;
  }

  public synchronized void removeItemListener(ItemListener l) {
    if (l == null) {
      return;
    }
    AWTEventMulticaster.remove(getItemListener(), l);
  }

  public synchronized ItemListener[] getListeners(Class listenerType) {
    ItemListener l = null;
    if (listenerType == ItemListener.class) {
      l = getItemListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }

  public boolean eventEnabled(AWTEvent e) {
    if (e instanceof ItemEvent) {
      processItemEvent((ItemEvent) e);
    }
    return super.eventEnabled(e);
  }

  public synchronized void addItemStateChangedListener(ItemListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.add(getItemListener(), l));
    setNewEventsOnly(true);
  }

  public synchronized void removeItemStateChangedListener(ItemListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.remove(getItemListener(), l));
  }

  public synchronized ItemListener[] getListeners(Class listenerType) {
    ItemListener l = null;
    if (listenerType == ItemListener.class) {
      l = getItemListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }

  public synchronized void addItemPropertyListener(PropertyChangeListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.add(getItemListener(), l));
    setNewEventsOnly(true);
  }

  public synchronized void removeItemPropertyListener(PropertyChangeListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.remove(getItemListener(), l));
  }

  public synchronized PropertyChangeListener[] getListeners(Class listenerType) {
    PropertyChangeListener l = null;
    if (listenerType == PropertyChangeListener.class) {
      l = getItemListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }
}
```

C.2 T AllowsActionListeners

```java
package java.awt;
import java.awt.event.*;
import java.util.EventListener;

trait T AllowsActionListeners {
  public required ActionListener getActionListener();
  public required void setActionListener(ActionListener al);
  public required void setNewEventsOnly(boolean b);
  public required long getEventMask();

  public synchronized void addActionListener(ActionListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.add(getActionListener(), l));
    setNewEventsOnly(true);
  }

  public synchronized void removeActionListener(ActionListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.remove(getActionListener(), l));
  }

  public synchronized ActionListener[] getActionListeners() {
    return (ActionListener[]) (getListeners(ActionListener.class));
  }

  public synchronized void addItemListener(ItemListener l) {
    if ((getEventMask() & AWTEvent.ITEM_EVENT_MASK) != 0 ||
        getItemListener() != null) {
      return true;
    }
    return false;
  }

  public synchronized void removeItemListener(ItemListener l) {
    if (l == null) {
      return;
    }
    AWTEventMulticaster.remove(getItemListener(), l);
  }

  public synchronized ItemListener[] getListeners(Class listenerType) {
    ItemListener l = null;
    if (listenerType == ItemListener.class) {
      l = getItemListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }

  public boolean eventEnabled(AWTEvent e) {
    if (e instanceof ItemEvent) {
      processItemEvent((ItemEvent) e);
    }
    return super.eventEnabled(e);
  }

  public synchronized void addItemStateChangedListener(ItemListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.add(getItemListener(), l));
    setNewEventsOnly(true);
  }

  public synchronized void removeItemStateChangedListener(ItemListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.remove(getItemListener(), l));
  }

  public synchronized ItemListener[] getListeners(Class listenerType) {
    ItemListener l = null;
    if (listenerType == ItemListener.class) {
      l = getItemListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }

  public synchronized void addItemPropertyListener(PropertyChangeListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.add(getItemListener(), l));
    setNewEventsOnly(true);
  }

  public synchronized void removeItemPropertyListener(PropertyChangeListener l) {
    if (l == null) {
      return;
    }
    addActionListener(AWTEventMulticaster.remove(getItemListener(), l));
  }

  public synchronized PropertyChangeListener[] getListeners(Class listenerType) {
    PropertyChangeListener l = null;
    if (listenerType == PropertyChangeListener.class) {
      l = getItemListener();
    } else {
      return super.getListeners(listenerType);
    }
    return AWTEventMulticaster.getListeners(l, listenerType);
  }
}
```
if (e.id == ActionEvent.ACTION_PERFORMED) {
    if ((getEventMask() & AWTEvent.ACTION_EVENT_MASK) != 0 ||
        getActionListener() != null) {
        return true;
    }
    return false;
}
return super.eventEnabled(e);
}

public void processEvent(AWTEvent e) {
    if (e instanceof ActionEvent) {
        processActionEvent((ActionEvent)e);
        return;
    }
    super.processEvent(e);
}

public void processActionEvent(ActionEvent e) {
    ActionListener listener = getActionListener();
    if (listener != null) {
        listener.actionPerformed(e);
    }
}

C.3 TAloowsTextListeners

package java.awt;
import java.awt.event.*;
import java.util.EventListener;

trait TAloowsTextListeners {
    public required TextListener getTextListener();
    public required void setTextListener(TextListener tl);
    public required void setNewEventsOnly(boolean b);
    public required long getEventMask();

    public synchronized void addTextListener(TextListener l) {
        if (l == null) {
            return;
        }
        setTextListener(AWTEventMulticaster.add(getTextListener(), l));
        setNewEventsOnly(true);
    }

    public synchronized void removeTextListener(TextListener l) {
        if (l == null) {
            return;
        }
        setTextListener(AWTEventMulticaster.remove(getTextListener(), l));
    }

    public synchronized TextListener[] getTextListeners() {
        return (TextListener[])getListeners(TextListener.class);
    }

    public EventListener[] getListeners(Class listenerType) {
        EventListener l = null;
        if (listenerType == TextListener.class) {
            l = getTextListener();
        } else {
            return super.getListeners(listenerType);
        }
        return AWTEventMulticaster.getListeners(l, listenerType);
    }
}
public boolean eventEnabled(AWTEvent e) {
    if (e.id == TextEvent.TEXT_VALUE_CHANGED) {
        if (((getEventMask() & AWTEvent.TEXT_EVENT_MASK) != 0 || getTe xtListener() != null) {  // Added missing '||' to ensure proper logical operator usage
            return true;
        }
        return false;
    }
    return super.eventEnabled(e);
}

public synchronized void addAdjustmentListener(AdjustmentListener l) {
    if (l == null) {
        return;
    }
    setAdjustmentListener(AWTEventMulticaster.add(getAdjustmentListener(), l));
    setNewEventsOnly(true);
}

public synchronized void removeAdjustmentListener(AdjustmentListener l) {
    if (l == null) {
        return;
    }
    setAdjustmentListener(AWTEventMulticaster.remove(getAdjustmentListener(), l));
}

public synchronized AdjustmentListener[] getAdjustmentListeners() {
    return (AdjustmentListener[])getListeners(AdjustmentListener.class);
}

public EventListener[] getListeners(Class listenerType) {
    EventListener l = null;
    if (listenerType == AdjustmentListener.class) {  // Added missing '}' to close the if block
    } else if (listenerType == TextListener.class) {  // Added missing '}' to close the if block
    }
    return null;
}
public boolean eventEnabled(AWTEvent e) {
    if (e.id == AdjustmentEvent.ADJUSTMENT_VALUE_CHANGED) {
        if (((getEventMask() & AWTEvent.ADJUSTMENT_EVENT_MASK) != 0 ||
            getAdjustmentListener() != null) {
            return true;
        }
    }
    return super.eventEnabled(e);
}

public void processEvent(AWTEvent e) {
    if (e instanceof AdjustmentEvent) {
        processAdjustmentEvent((AdjustmentEvent)e);
    }
    super.processEvent(e);
}

public void processAdjustmentEvent(AdjustmentEvent e) {
    AdjustmentListener listener = getAdjustmentListener();
    if (listener != null) {
        listener.adjustmentValueChanged(e);
    }
}
C.5 TAllowsActionAndItemListeners

    return false;
    default:
        break;
    }
    return super.eventEnabled(e);
}

public void processEvent(AWTEvent e) {
    if (e instanceof ItemEvent) {
        processItemEvent((ItemEvent)e);
        return;
    } else if (e instanceof ActionEvent) {
        processActionEvent((ActionEvent)e);
        return;
    }
    super.processEvent(e);
}
Bibliography


[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, USA, 1994.


Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behaviour. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-012, OGI School of Science & Engineering, Beaverton, Oregon, USA.


Timor. [http://www.informatik.uni-ulm.de/rs/projekte/Timor/].