Traits in Java

Master thesis

Kjetil Østerås

May 1, 2007
Abstract

A trait is a programming language feature which contains a collection of methods that can be reused across class hierarchies. Traits is a relatively new language feature that is beginning to be a part of some of the newest object-oriented programming languages. Traits have been implemented in some languages but it has not become a part of the Java language yet. In this thesis we apply traits to the Java 5 language by designing and implementing a traits aware preprocessor. The preprocessor translates a Java language extended with trait syntax into ordinary Java 5. The preprocessor builds abstract syntax trees of the input files and inserts traits by manipulating the abstract syntax trees. The preprocessor also contains a type-checker which ensures that the trait can safely be applied to a certain class.

This thesis describes the implementation of the traits preprocessor for Java and compares it to some other implementations of traits in statically typed languages.
Acknowledgments

First of all I would like to thank my supervisor, Professor Birger Møller-Pedersen for helping me finish this master thesis. By giving me the right guidance at the right time he gave me room to play with the subject and at the same time pushed me to finishing the thesis on time. He also provided numerous history lessons on the BETA language which helped me understand the subject better.

I would like to thank my friend and fellow student Olav Skjelkvåle Liggarden for reading through the thesis and providing me with comments and corrections.

I would also like to thank my family and friends for pulling me out of the computer room and exposing me to sunlight and fresh air from time to time, and for motivating me. The coffee grinder I got for Christmas 2006 could not have come at a better time.
2.3.5 Argument type ............................................ 29
2.3.6 Return value type ........................................ 31
2.3.7 Effects of adding traits .................................. 31

3 Type systems .................................................. 33
  3.1 Java type system ............................................ 34
    3.1.1 Java types ............................................. 34
    3.1.2 Type conversion ....................................... 37
    3.1.3 Conversion context .................................... 41
    3.1.4 Modifiers .............................................. 42
  3.2 Trait model ................................................ 44
    3.2.1 Java model ............................................. 44
  3.3 Semantics .................................................. 45

4 Trait Syntax .................................................. 47
  4.1 Trait declaration ......................................... 48
  4.2 Trait requirement ........................................ 48
  4.3 Trait member .............................................. 49
  4.4 Trait application ......................................... 49
  4.5 Trait operations ......................................... 50
  4.6 Examples with trait syntax ................................ 51
    4.6.1 Simple example ....................................... 51
    4.6.2 Operations example ................................... 51
    4.6.3 Synchronized wrapper example ....................... 52
    4.6.4 Diamond inheritance example ....................... 55

5 Method ......................................................... 59
  5.1 Language extension ....................................... 59
    5.1.1 Parser extension ..................................... 59
    5.1.2 Traits with OpenJava ................................ 60
    5.1.3 Stratego/XT ........................................... 62
    5.1.4 JavaCC ................................................ 62
    5.1.5 Another tool for language recognition, ANTLR .... 63
  5.2 JavaTrait preprocessor ................................... 63
    5.2.1 Requirements .......................................... 63
    5.2.2 The implementation ................................... 64
    5.2.3 Trait implementing an interface ...................... 73
    5.2.4 Interface with trait .................................. 76
    5.2.5 Trait type checking .................................... 78
    5.2.6 Preprocessing example ................................ 87
## 6 Discussion

6.1 Other work on traits ........................................ 91
6.1.1 C# .................................................. 91
6.1.2 Scala ................................................. 96
6.1.3 Perl 6 ................................................ 98
6.2 Evaluation of the preprocessor .............................. 99
6.2.1 Remaining work on the preprocessor ............... 100
6.2.2 New trait feature ..................................... 100
6.2.3 Alternative syntax ..................................... 101
6.2.4 Alternative requirement specification ............. 102
6.2.5 Checking requirements ................................ 103
6.2.6 Alternative design .................................... 104
6.3 Conclusion .................................................. 104

## 7 Summary

7.1 Further work ............................................... 107

## A Using the prototype

A.1 Getting the prototype ..................................... 109
A.2 Compiling the prototype ................................ 109
A.3 Running the prototype ................................... 110

## B Grammar

B.1 Extension of the Java syntax ............................. 111
Chapter 1

Introduction

“Programs must be written for people to read, and only incidentally for machines to execute.” – Abelson & Sussman, Structure and Interpretation of Computer Programs

From the early days of programming reuse has been an important part of software development. In object-oriented languages one of the main reuse mechanism is inheritance. The idea of inheritance is simply to base a new class definition upon existing classes [1]. With modern object-oriented class-based systems inheritance is done by programming with classes and subclasses. This has reduced the amount of code a programmer has to write in order to create abstractions of real world objects. Simple inheritance models have been used since SIMULA-67 [2], where one class could inherit state and behavior from a single superclass. Models for multiple inheritance have since then been tested and implemented in some object-oriented languages, but this has brought with it drawbacks like increased complexity and the infamous diamond-problem.

The goal of inheritance from the programmer’s perspective is to minimize the amount of code written and reuse common programming logic whenever appropriate. This can be as simple as calling a library function or building a class from a superclass made by another programmer. From a computer system perspective reuse can be applied to minimize the amount of memory needed for a software system, for instance by factoring out common operations into a function in a program.
1.1 Traits

Traits is a language feature made to increase the amount of code reuse. A trait is a composable unit of behavior [3], and they are meant to be entities which will serve as basic building blocks for classes by being units of method reuse. A trait is an entity with a collection of method signatures and method bodies, independent of any class hierarchy. Being hierarchy independent makes traits a good tool for factoring out common behavior in an object-oriented language.

When using traits, classes are still organized in a single inheritance hierarchy, and traits can be used to specify the incremental difference in behavior with respect to the applied classes [4].

Traits has currently been implemented in the Squeak programming language [5] and it has been used with success to refactor the Smalltalk-80 collection hierarchy where the amount of code was reduced by approximately 10% [6].

1.1.1 Example of trait benefits

Using traits, the programmer can reuse method bodies independent of the class hierarchy. Take for instance the class hierarchy of figure 1.1. In the figure there are three separate classes that are direct subclasses of Object, these are: AddressBook, Account and Collection. Analysis of these classes show that they lack some useful functionality. Assume that we need to extend some of these classes with two new methods that are based on existing methods. We want all the classes to have a method that checks if an instance of the class is empty and we want to make it easier to iterate over addressbooks and lists. All the methods needed to implement these two properties are provided by the classes. We can find out if something is empty by use of the size method, and we can accomplish iteration over addresses and lists by the use of the size method in combination with the getItem method.

These two methods can contain the same code no matter what class they are a part of, listing 1.1 shows the code for the isEmpty method and listing 1.2 shows the code used to create and return an iterator. When we notice that so much is common the usual way of inserting this code is to put the code in some superclass that is shared by all the classes which need the methods. However, in our case the only common superclass is Object and it is clearly not a good idea to put the isEmpty and iterator methods into the Object class. So we let the duplication begin.
CHAPTER 1. INTRODUCTION

Since we have no common superclass to put our shared methods into we have no choice but to insert the methods directly into the classes that need the methods. This is shown in figure 1.2. Collection is a superclass of Set, Map and List so we put the isEmpty method into the common superclass Collection. The duplication of code in this hierarchy brings with it a maintenance problem. The methods with exactly the same body are inserted into several different classes and must be maintained separately.

Traits comes to the programmers aid. In figure 1.3 traits are used to factor out the common methods and they are inserted only where they are needed, with no concern for the class hierarchy. There is no standard way of modeling traits, so traits are drawn using UML classes with a Trait stereotype in the figure. The application of traits are shown by dependency arrows using the with stereotype. The trait Emptiness contains the code from listing 1.1 and the trait ItemIterable contains the code from listing 1.2.

This example shows that candidates for trait refactoring are classes where methods are duplicated across class hierarchy or where methods are put too high in the class hierarchy to remove code duplication. Class hierarchies with no connection at all can sometimes contain the same pattern of behavior, this is where traits can be used to favor code reuse.
Listing 1.1: Emptiness method

```java
boolean isEmpty() {
    return size() == 0;
}
```

Listing 1.2: Iteration method

```java
public Iterator iterator() {
    return new Iterator {
        int current = 0;
        public boolean hasNext() {
            return current < size();
        }

        public Object next() {
            return getItem(current++);
        }
    };
}
```

Figure 1.2: Class hierarchy extension without traits
CHAPTER 1. INTRODUCTION

Figure 1.3: Class hierarchy extension with traits
1.1.2 Requirements

Traits are defined as units which cannot contain any state. This means that they only contain methods and not variables, but in order for the trait methods to do anything useful they need to manipulate variables of the class applying the trait. This interaction is done by calling methods that the trait requires the class to contain. We use the word requirement to describe that a trait needs some method from a class. For instance when we say that the trait \( t \) requires a method \( m \) it means that any class that wants to apply trait \( t \) needs to contain a method \( m \).

In the example shown in figure 1.3 we define the traits Emptiness and ItemIterable both of these have requirements. The Emptiness requires the method size and ItemIterable requires the methods size and getItem.

1.1.3 Composition

In order to make traits more usable they have the ability to be composed of other traits. Classes and traits can be composed from any number of traits, and the composition order is irrelevant. When traits are combined in this form the requirements of all the traits are combined to form a new set of requirements. Requirements posted by one trait can possibly be implemented by some other trait, so together they can form a composed trait with fewer requirements.

Composition can for example be used to collect associated traits into one larger trait. Take for instance a model of a collection hierarchy. Now assume that we want to use traits to enclose the properties of an ordered collection. An ordered collection in our case should have the ability to be sorted and reversed. We create the trait Ordered and we use composition to apply the traitsSortable and Reversable to the Ordered trait. Now when a class applies the Ordered trait it gets the methods from Sortable and Reversable as well as the methods from Ordered.

The idea of traits is that they should behave as if the methods inside the trait was defined directly into the class that applies the trait. This property is called the flattening property of traits. This means that the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the trait. Similarly, trait composition does not affect the semantics of a trait, a trait defined using traits is exactly the same as the trait constructed directly from all of the non-overridden methods of the composite traits.

Composition of classes by using the traits model can be describes with
the equation $\text{Class} = \text{Superclass} + \text{State} + \text{Traits} + \text{Glue}$ [3]. This shows that a class is composed of a superclass, state variables, the applied traits and the glue code which implement the requirements of the traits.

1.1.4 Conflicts

Since traits can be composed arbitrarily there can be cases where several traits provide the same methods. We say that a method conflict occurs when some unit, either a trait or a class applies several traits that contain methods with the same name and signature.

A conflict arises if and only if we combine two traits providing identically named methods that do not originate from the same trait. To grant access to conflicting or hidden trait methods traits support the alias operation. The alias operation makes a trait method available under another name. So if for instance two traits $t_1$ and $t_2$ are used in a class and they both provide the method $m$ a method conflict occurs. To resolve this conflict the class can provide its own $m$ method and override the $m$ methods of both the traits. However if the class wants to use one of the $m$ methods of $t_1$ or $t_2$ it has to be aliased to make it available under another name.

Trait composition also supports the exclusion operation. Using the exclusion operation a trait can exclude a given method from a composite trait. The exclusion operation allows a trait to avoid a conflict before it occurs. These two operations are useful when dealing with conflicting method signatures when composing traits or classes. Yet not all traits implementations support aliasing and exclusion. Using the same example as given in the previous paragraph where there is a conflicting method $m$ we can use exclusion to choose for instance to only get the $m$ provided by $t_1$.

1.2 Purpose

Abstraction mechanism is the way of organizing knowledge of a certain domain. When programming using object-oriented languages, it is important to have adequate abstraction mechanisms to be able to generalize the code. The purpose of traits is to decompose classes into reusable building blocks by providing representations for the different aspects of the behavior of a class [3].

Inheritance in object-oriented languages is well-established as an incremental modification mechanism that can be highly effective at enabling code reuse between similar classes [4]. With the multiple inheritance model
a class can inherit features from more than one parent class, this gives better code reuse and more flexible modeling capabilities. Traits provides a way for classes to inherit multiple method bodies, using traits is a way of getting the effect of multiple inheritance into a language.

1.2.1 Code reuse

One might ask, why would we need traits when there already exist a way of code reuse in the object-oriented languages like inheritance? There are several forms of inheritance, single inheritance, multiple inheritance, and mixin inheritance. But they all have their weaknesses when it comes to code reuse as identified in [3]. Traits are the proposed answer to the reuse problems caused by these ways of inheritance. Traits can be seen as the smallest unit of behavior reuse in an object-oriented programming language while class inheritance can be seen as structural reuse.

1.2.2 Understandability

Traits can also be used in a way that makes it easier to view and understand source code, in addition to being a unit of behavior reuse. There is a conflict between reuse and understandability [3]. Higher reuse by inheritance gives a harder time for people trying to understand the code. A programmer should therefore be able to view the code in multiple forms. Even when classes are composed of traits he should be able to view the class as a flat collection of methods, or as a composite entity built from traits. The flattened view of traits promotes understanding and the composite view of traits promotes reuse, so traits are a possible solution to these problems of reuse and understandability.

1.3 Goal

The goal of the thesis is to create an implementation of traits, where traits are a part of Java. The implementation is to provide some experiences with the introduction of traits into the type system of the Java 5 language, and to discuss the trait features with respect to Java. The implementation will be a preprocessor of the Java language that extends the language like described in [7] and that provides type checking mechanisms needed for traits. The goal is to make something as complete as the time given allows, and to sketch the needed extensions to make a complete implementation of traits in Java.
CHAPTER 1. INTRODUCTION

1.4 Organization

This thesis is organized in the following way:

**Chapter 1** is this chapter where traits are introduced and an example of traits usage is given.

**Chapter 2** contains the challenges faced when introducing traits into Java.

**Chapter 3** contains an overview of the type systems concerned when making the traits preprocessor. It contains a description of the type system of Java and sections on the trait model and semantics of traits.

**Chapter 4** contains the trait syntax used. Here we present the trait syntax divided into several sections. In the end of the chapter some examples of traits usage is provided.

**Chapter 5** contains a detailed description of the preprocessor created. It begins with a section on language extension in general and continues with the design and implementation of the Java preprocessor created.

**Chapter 6** is the discussion chapter. It contains a section where the preprocessor created is compared with other trait implementations. It also contains an evaluation of the preprocessor where remaining work and missing features are presented. The chapter ends with a conclusion.

**Chapter 7** is the summary chapter. It contains a short summary of the thesis and suggestions for further work.

**Appendix A** describes where to download and how to build the preprocessor. It also describes how to use the preprocessor from the command-line.

**Appendix B** contains the full grammar of traits in Java. This appendix was created to present the grammar in a concise way, since the syntax is split into several sections in Chapter 4.

**CD** the appended CD contains the preprocessor and its source code. It is the same source code as can be downloaded from the web page [http://kjetilos.at.ifi.uio.no/master/](http://kjetilos.at.ifi.uio.no/master/).
Chapter 2

Challenges

"Multiple Inheritance is good, but there is no good way to do it!"
– Steve Cook, 1987 OOPSLA panel

A paper has been written with a road map of issues and possible strategies related to the integration of traits into statically typed languages [7]. The important issues that are listed in that paper are the representation of traits, the typing of trait methods and the adaption of the compilation and execution mechanisms.

In this chapter we will investigate possible difficulties with inserting traits into the Java programming language. The challenges will be presented briefly and research on the subjects by others will be presented. We will also see how traits can be used to solve the problems of some common inheritance models.

2.1 Inheritance

In this subsection the different inheritance models are reviewed with a description of their separate problems as mentioned in the introduction.

2.1.1 Single inheritance

Single inheritance has been used since the first days of object-orientation, starting with SIMULA-67 [2]. This is the simplest inheritance model, it allows a class to inherit from one single superclass and extend it further with additional fields and methods. Single inheritance is simple and therefore
CHAPTER 2. CHALLENGES

Figure 2.1: Illegal with single inheritance

easy to implement. The problem with single inheritance is that it lacks expressiveness when it comes to factoring out all the common features shared by classes in a complex hierarchy, and therefore lacks the features that promote maximum code reuse.

For instance we see this problem in a simple class structure where we want to model a stream which can both be read from and written to. In our model the stream is both an inputstream and an outputstream and we want to have this information in our model. We draw the model by making the class InputOutputStream a specialization of both class InputStream and OutputStream, see figure 2.1. We want the InputOutputStream to share the functionality of both InputStream and OutputStream and maybe include some extensions. This can not be implemented directly with the use of single inheritance.

Although single inheritance is a model which allows a low amount of reuse, it is still the model used in the major languages like Java and C#. Both of these languages provide interfaces mechanisms to allow multiple subtyping.

2.1.2 Multiple inheritance

Multiple inheritance is implemented in some programming languages, for instance C++ [8]. With the multiple inheritance model a class can inherit features from more than one parent class. This gives better code reuse and more flexible modeling capabilities. The model from figure 2.1 is legal in these languages, but the multiple inheritance model has some problems.

One known problem is diamond inheritance which is sometimes also called fork-join inheritance [1]. In diamond inheritance some class inherits from two
different super classes, and each of these super classes inherit from a common base class. For an illustration of diamond inheritance see Figure 2.2. In the illustration the example from the single inheritance section is reused, we have now added a common base class called Stream that both InputStream and OutputStream inherits from. The Stream class contains a method for closing a stream and a buffer which can be loaded with bytes read or bytes to write.

The problem with diamond inheritance is that the state variables from the common base class can be inherited in two ways. The state variable could be inherited twice in two different versions or it could be inherited once as a common copy. There is also an ambiguity in a call to a method defined in the common base class. It has to be resolved which class it inherits it from.

With single inheritance you sometimes have a super keyword to refer to overridden methods. When using the multiple inheritance model a reference to super could refer to several different classes, this is why languages supporting multiple inheritance often do not support the super keyword. C++ solves this problem by making the programmer prefix the method call with the name of the superclass, but as described in [3] this makes the code fragile because it mixes class references into the source code.

Multiple inheritance enables a class to reuse features from multiple base classes, but it does not allow one to write a reusable entity that “wraps”
methods implemented in as-yet unknown classes. A good example of this is the case where we want to synchronize access to methods defined in a superclass [3]. We have two separate classes A and B both providing a read and write method. Now assume that we want to create two subclasses SyncA and SyncB of these classes. These subclasses should synchronize the calls to the read and write methods of their superclasses and we want to reuse the synchronization code. Here we can use traits to “wrap” the synchronization code, and by apply the trait to the two subclasses we make the calls to the read and write methods of their superclasses synchronized.

2.1.3 Mixins

Mixin inheritance is the inheritance model that is similar to the traits model. In mixin inheritance you create certain mixin classes that have no superclasses. The mixin classes are especially created to add properties to other classes. When a class is created it can apply a number of mixin classes to get the properties defined in these mixin classes, in addition to inheritance from optional super classes. Mixin classes cannot be used alone, they must be combined with another non-mixin class to work.

Mixins [9, 1] were first used in the languages Flavors [10] and CLOS [11]. A mixin is a class specification that may be applied to various classes in order to extend them with a set of features. Mixins allow better code reuse than single inheritance while maintaining the simplicity of single inheritance, but the problem with mixins become apparent when conflicts arise. Since mixin composition is linear ordered, a mixin class is always inherited one at a time. This means that the last definition of a method silently overrides the previous versions. Silently overriding methods across class hierarchies makes mixins a fragile way of inheriting code.

Here is an example of silently overriding a method. In the Ruby programming language mixins are allowed by including a module within a class definition. To show that mixins override each other silently we have defined two modules A and B which both define a method hash. Now we want to define two classes C and D that should be combined with both A and B by mixin inheritance. The output from the example program in listing 2.1 is as follows:

c.hash = 15

d.hash = 17

The reason for this output is that the hash methods silently override each other. In the class C the hash method from B overrides the hash method
CHAPTER 2. CHALLENGES

Listing 2.1: Mixin in Ruby

```ruby
module A
def hash
  17
end
end

module B
def hash
  15
end
end

class C
  include A
  include B
end

class D
  include B
  include A
end

c = C.new()
d = D.new()

puts("c.hash=#{c.hash}")
puts("d.hash=#{d.hash}")
```

23
From the conceptual point of view, one of the less beneficial characteristics of mixin inheritance is that it tends to cause confusion in the relationship of object-oriented programming and conceptual modeling by splitting the role of classes [1]. The problem is that the ordinary class is a unit of reuse and object classification at the same time, while a mixin class is only a unit of feature reuse.

2.1.4 Trait solution

Traits are a good way to inherit method bodies by being a small unit of reuse, and since traits have no state it avoids problems like diamond-inheritance. When using traits the goal is to make traits the primary unit of reuse, and to make classes the unit of object creation and state encapsulation. This way traits can be used to solve the inheritance problems outlined in the subsections above. Traits can even wrap code for use with yet unknown classes, because as long as the method requirements of the trait is present the trait can be used in a class.

Figure 2.3 shows an example of how the problems with single inheritance and multiple inheritance could be solved with traits. Note that the same way of modeling traits as used in the introduction is used in this figure.
CHAPTER 2. CHALLENGES

2.2 Java traits

There are a number of challenges faced when including traits into the Java programming language. There is the question of whether traits fit into the Java type system and whether the Java specific modifiers have an impact on the traits implementation. In late September 2004 Sun released version 5 of the Java 2 platform standard edition, with the codename Tiger. The Java 5 release included many new language features which could have an impact on the trait implementation, as the trait implementation is an extension of the Java 5 language. The ultimate goal of any trait implementation is to create the ideal representation of traits. The properties of such an implementation are hard to define, but the following subsections will provide some of these properties.

2.2.1 Expressing traits in Java

When considering Java and traits one has to think of all the challenges that arise when inserting traits in a statically typed programming language. We want to express traits as general as possible to make them work, and we want them to fit neatly into the Java type system. Nierstrasz [7] has written a paper which describes some issues when applying traits to a statically typed language. This paper focus on a trait implementation in C#, but many of the same issues exist when inserting traits in the Java language.

Traits in Java is not a revolutionary new idea. There have been attempts to add traits functionality to the Java language before. One of the earlier implementations of traits in Java simulate traits by the use of the AspectJ system [12], which is a system that provides an aspect-oriented [13] extension to Java. Another trait implementation has been done in a subset of the Java language called mini-Java [14]. This implementation works in much the same way as the preprocessor that we have created, but it only operates with a subset of Java. It takes as input the mini-Java language extended with trait syntax and spits out ordinary mini-Java.

2.2.2 Java modifiers

Many statically typed languages like C# and Java define access modifiers to control and specify access to fields and methods. Modifiers are keywords that prefix the field or method signature. In Java such modifiers are private, public and protected, there is also an implicit modifiers when no modifier keyword is present and we call this the none modifier. A private method or
field cannot be accessed from other classes at all, a public method or field
can be accessed by anyone, and a protected method or field can only be
accessed in the class and subclasses. When a method or field has the none
access modifiers it can be accessed from classes in the same package.

What needs to be done when extending the Java language with traits
is to figure out the effect of the different access modifiers on requirements
and on trait methods. Java also presents other method modifiers that could
have an effect when implementing traits. These modifiers are:

- **static**: states that the associated method is a class method.
- **abstract**: provides a method signature but not an implementation.
- **native**: the method is implemented in platform-dependent code.
- **synchronized**: the method requires a monitor before it executes. This
  modifier is used to synchronize execution of methods.
- **strictfp**: expressions within the method body are FP-strict [15].
- **annotations**: user-defined “modifiers”.
- **final**: prevents subclasses from overriding or hiding this method.

We are only concerned with the modifiers effect on method signatures
since traits as implemented in the preprocessor does not contain fields.

In addition to method modifiers, it is natural to consider associating
modifiers to the trait. An interesting point is what type of modifiers the
trait should be allowed to have, and what kind of effect they have on the
trait.

### 2.2.3 Java 5 features and traits

With the release of Java 5, Sun changed its version number by a large jump
from 1.4 to 5. This lead to some confusion with version numbering, and
Sun made it clear that version 5 could also be regarded as Java 1.5. Java 5
and 1.5 are the same version with the same features. In this thesis the Java
5 name will be used consistently.

The Java 5 language enhancements like enumerations, generics and an-
notations could have a great effect on the way traits behave with the Java
language. It could also be that some of these language structures should be
a natural part of traits e.g when a trait method is included in a Java class,
CHAPTER 2. CHALLENGES

Listing 2.2: Trait method with annotation

class A {
    @TraitMethod("Ta")
    void m() { ... }
}

the method could be inserted with an annotation which contains information on what trait it originated from. In this way we provide marking of methods originating from traits. Listing 2.2 shows an example of a method that has been injected from a trait $Ta$. After the method is injected this has been marked with an annotation.

2.2.4 Language extension

Implementation of traits in any language is described in [7] by a simple and generic strategy. To extend a language $L$ with traits it is proposed to create a language $L_T$ that differs from $L$ in that it has an extended syntax to allow the definition and application of traits and trait operations like aliasing and exclusion. Now we can write a translator or preprocessor that takes a program written in $L_T$ and in-lines all the traits and outputs code written in $L$ which can be compiled with any compiler for $L$. This will yield semantically equivalent code because of the flattening property of traits.

The method chapter in this thesis describes the method used when extending Java with traits. It is influenced by the ideas of language extension described in [7].

2.2.5 Ideal representation of traits

An ideal representation of traits should exist at both compile-time and runtime according to [7]. They should exist in compile-time to separate trait compilation units from class compilation units. By having a separate compilation unit for traits you make the using classes free from the trait source code, and it will be possible to compile the units separately.

Traits should also exist in the runtime environment to enable runtime reflecting applications, for instance to support traits in the reflection api of the Java language or when debugging code made with traits.
2.3 Typing issues

Typing issues are issues associated with the type systems of a programming language. A type system defines how a programming language classifies values and expression into types, how it can manipulate those values and how they can interact. A type can be something like a primitive/predefined type or a user-defined type by means of a class.

In this thesis we will use two different language classifications based on their type systems: dynamically and statically typed languages. In dynamically typed languages the types do not exist at compile-time. In statically typed languages every variable and expression has a known type at compile-time.

This section contains subsections which outline the typing issues focused on in this thesis when applying traits to the Java language. The issues are concerned with return types and argument types of method signatures and the effects of adding traits into a type system.

2.3.1 Traits in a dynamically typed language

Traits have been implemented in both dynamically and statically typed languages. The first implementation of traits was implemented in Squeak [5], a dialect of the Smalltalk programming language [16]. Smalltalk is a dynamically typed programming language, where type checking is deferred until run-time. This implies that the Smalltalk compiler does not know the types of any of the variables used in a class, and a variable can have several different values of any type/class during execution.

By implementing traits in a dynamically typed language one did not need to worry about type checking issues at compile-time like the typing of method arguments and return values of methods provided by a trait. For instance the checking of whether a required method is present is decided at runtime when the actual method lookup is made, and not at compile-time which is required in a statically typed language. When inserting traits into a language like Java, these issues need to be identified and resolved.

2.3.2 Traits inside the Java type system

By creating a trait aware preprocessor for Java we attempt to answer the following questions:

- Do the original traits as outlined in [3] fit into the Java type system?
How can the trait definition possibly be extended to adapt to a statically typed language, and specifically the Java language?

Should the implementation of traits contain type checking or not?

2.3.3 Semantics of traits

In numerous traits papers it is emphasized that traits enjoy the flattening property. This property says that a class $A$ with a trait $Ta$ has the same semantics as the class $A$ with all the methods of $Ta$ in-lined, but what does a trait actually represent in a statically typed programming language, and how does the compiler handle the traits at compile-time? Are they only treated as a set of methods that can be inserted into classes or are they something bigger, something that occasionally has a side effect on the class using the trait.

2.3.4 Trait type

The relationship between type and traits has to be considered when implementing traits in a language. Questions like whether or not a defined trait also defines a type with the same name needs to be answered. For instance if we have a trait $Ta$ inside a statically typed language we need to know if the type $Ta$ is a valid reference or pointer type to be e.g. used as formal parameters.

Another issue that arises when traits are implemented in a statically typed language is concerned with type-checking requirements. A strategy for how required methods should be type-checked must be implemented in a preprocessor. The strategy must enable the trait to be reused across multiple classes correctly, and it should not cause unnecessary restrictions of use.

2.3.5 Argument type

Consider a trait method and its associated argument list. What if we wanted a method to be able to take as input-argument a class which applies a specific trait. More formally we want to somehow define that parameter $p_i$ has the type $t$ where $t$ is any class with trait $T$. If we extend the type system of the language to give each trait its own corresponding type and create a subtype relation whenever a trait is applied, we would be able to use the trait type as parameter $p_i$. If the trait does not have a corresponding type then we need some other way of specifying the type of parameter $p_i$. 
CHAPTER 2. CHALLENGES

Listing 2.3: Trait type as argument

```scala
trait TLinkable {
    public boolean includes(TLinkable linkable) {
        ...
    }
}
```

Listing 2.4: Interface type used as argument

```scala
trait TLinkable {
    public boolean includes(ILinkable linkable) {
        ...
    }
}
```

This argument type problem is presented in [7] with a possible solution. The solution applies when a trait does not define its own corresponding type, but the language provides us with interfaces. We could then use an interface as argument type. That interface would then need to be implemented by all classes that are passed as an actual parameter.

We want to create a trait that can be applied to a class that should implement linked list functionality. We create the trait TLinkable which offers a method `public boolean includes(...)`. This method `includes` wants to take a class which applies the trait `TLinkable` as input-argument. If the trait `TLinkable` does not define a type of the same name then we can not use `TLinkable` as input-argument of the `includes` method. What we could do instead is to make every class using `TLinkable` implement an interface `ILinkable` and insert `ILinkable` as the input-argument of the `includes` method. The final method signature would look like this: `public boolean includes(ILinkable linkable)`.

The example in listing 2.3 shows the situation where the trait `TLinkable` defines a type with the same name, and can therefore be used as formal parameters in methods. While the example in listing 2.4 shows the situation where the trait `TLinkable` does not define a type with the name `TLinkable` and we must instead make the `includes` method take an interface `ILinkable` as formal parameter.
CHAPTER 2. CHALLENGES

2.3.6 Return value type

Return values of trait methods brings another challenge. A problem occurs when a return value of a trait method is supposed to be the same type as the class which applies the trait. It is not sufficient to solve this by typing traits because the returned object will only have the operations contained in the specific trait. The solutions proposed in [7] are either to use generics to give the trait its class as a parameter or one could introduce a new keyword that signifies that this method returns the type of the using class. The keyword proposed is selftype or ThisType. The listing 2.5 contains a small example of a trait which provides linked list functionality. In a linked list you typically want a method for getting the next node in the linked list chain. This is implemented in the trait so it can be reused in different classes.

Listing 2.5: Return type value

```
trait TLinkable {
  public ThisType getNext () {
  ... 
  }
}
```

2.3.7 Effects of adding traits

When adding traits to a language, two properties should hold. First you want a small program, traits should not only reuse source code but also reuse the compiled code. Second you want a good execution performance, this means that even when adding traits you do not want it to have negative effects on the execution speed. The program should execute as fast as the corresponding flattened program that does not contain any traits. An ideal implementation of traits should satisfy these requirements [7].
Chapter 3

Type systems

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.” – Benjamin C. Pierce, Types and programming languages [17]

Type systems are the part of a programming language that are used to classify statements in a program. They contain the logic used by the type-checkers that does the error-checking of expressions. The type-checkers are often built into the compiler and the run-time system. Type systems are constructed to make the programmer write logically sound code, and to make sure that certain errors never occurs. However type systems cannot be used to check every kind of error possible. For instance consider the situation where a division $a/b$ is made. The type system can be used to ensure that $a$ and $b$ are always integers, but there is no way a type system can ensure that the $b$ of this division will never be zero.

We have outlined some of the abilities of the type systems to detect errors, but type systems can also be used for other tasks. They can be used to enforce programmers to use proper abstractions. They can also be used to make a program easier to read by being a form of documentation. For instance a variable $x$ is document by declaring it as an $int$ in Java. This would mean that $x$ is always an integer numerical value, in the range of $-2^{31}$ to $2^{31} − 1$. 
3.1 Java type system

This section provides necessary information about the Java type system to understand the issues when including traits into the Java language. Most of the information in this section is gathered from the third edition of the Java languages specification [15] which describes the Java 5 language. This section will go through the different types of Java, like primitive and reference types. Since the traits preprocessor contain operation on methods and type-checking of method signatures we need to introduce Java type conversions and the different conversion context used in Java.

3.1.1 Java types

A type system defines how a programming language classifies values and expressions into types, how it can manipulate those types and how they interact. The Java programming language is a strongly typed language, which means that every variable and every expression has a type that is known at compile-time [15].

The types of the Java programming language are divided into two categories: primitive types and reference types. There are eight primitive types and three reference types in the Java language.

Primitive types

Primitive types are types that are native to the language, you cannot make new primitive types. The primitive types of Java are used to store values that are close to values represented in the CPU. Primitive values are not composed of other types. The reason for using primitive types is that operations on these will often be quicker than operations on reference types. They are also easier to use because a set of operations over the primitive types is already implemented in the language.

The Java language contains three categories of primitive types. There is the truth value type which only contains the boolean type, the integral types which contain byte, short, int, long and char, and the last category is the floating-point types which contains float and double. Following is a list of the primitives and the values they can hold.

- boolean represents truth values, can be either true or false.
- byte is an integral type, and the possible values range from $-2^7$ to $2^7 - 1$. 

34
CHAPTER 3. TYPE SYSTEMS

- *short* is an integral type, and the possible values range from $-2^{15}$ to $2^{15} - 1$.
- *int* is an integral type, and the possible values range from $-2^{31}$ to $2^{31} - 1$.
- *long* is an integral type, and the possible values range from $-2^{63}$ to $2^{63} - 1$.
- *char* is an integral type which represents a single 16 bit unicode character. The possible values range from `'\u0000'` to `'\uffff'`, which is from 0 to $2^{16}$.
- *float* is a floating-point type of the single-precision 32-bit floating-point format, where the possible values are as defined in the IEEE 754 standard.
- *double* is a floating-point type of the double-precision 64-bit floating-point format, where the possible values are as defined in the IEEE 754 standard.

Reference types

Reference types are used whenever the primitive types does not cut it. The reference types can be composite types containing primitive types or other reference types. There are three kinds of reference types in the Java 5 programming language.

- *Class type* represents a class. A class type is created by a class or enum declaration.
- *Interface type* represents an interface. An interface type is created by an interface declaration.
- *Array type*. This type is created by an array expression and is a fixed-size list of values. An array object is an instance of the array type. The array object contains a number of variables which are called the component of the array.

In Java 5, Sun introduced generics into the language giving the possibilities of class types having parameterized types. This is an interesting feature with respect to trait implementation, but due to time constraints we have only looked at traits in Java without Java 5 generics. However traits with generics should be looked at in further work.
Type relation

Understanding the Java type relations is important when adding traits to the language. The type relations have an impact when the type system of the language is used to perform type-checking and type conversion. The subtype and supertype relations are binary relations on types.

**Supertype relation.** The supertypes of a type are obtained by reflexive and transitive closure over the direct supertype relation, written $S >_1 T$. We write $S :> T$ to indicate that the supertype relation holds between $S$ and $T$. $S$ is a proper supertype of $T$, written:

$$S > T, \text{ if } S :> T \text{ and } S \neq T$$

**Subtype relation.** The subtypes of a type $T$ are all types $U$ such that $T$ is a supertype of $U$. We write $T <: S$ to indicate that the subtype relation holds between types $T$ and $S$. $T$ is a proper subtype of $S$, written $T <: S$, if $T <: S$ and $S \neq T$. $T$ is a direct subtype of $S$, written:

$$T <_1 S, \text{ if } S >_1 T$$

In the Java language the following rules define the direct type relations among primitive types. Looking at the list one can for instance see that a double is a direct supertype of float, and that float is a direct supertype of long.

- double $>_1$ float
- float $>_1$ long
- long $>_1$ int
- int $>_1$ char
- int $>_1$ short
- short $>_1$ byte

Let $C$ be a class declaration then $C <: S$ if $S$ is a superclass in the superclass chain of $C$ or if $S$ is one of the interfaces implemented by $C$. The following rules define the direct subtype relation among array types.

- if $S$ and $T$ are both reference types, then $S[] >_1 T[]$ iff $S >_1 T$.
- $\text{Object} >_1 \text{Object}[]$
- $\text{Cloneable} >_1 \text{Object}[]$
CHAPTER 3. TYPE SYSTEMS

- `java.io.Serializable` > `Object[]`
- If `p` is a primitive type then
  - `Object` > `p[]`
  - `Cloneable` > `p[]`
  - `java.io.Serializable` > `p[]`

This subtype relationship with arrays leads to the fact that any method with the formal parameter `Serializable` can take any array as the associated actual parameter, it also means that an array can be passed to any method as a reference to an `Object`.

The class `Object` is a superclass of all other classes. A variable of type `Object` can therefore hold a reference to any object. The class `Object` is also a superclass to arrays in Java. An object is said to be an instance of its class and of all classes in the superclass-chain, therefore an instance of the `String` class is also considered an instance of `Object` since `Object` is a superclass of `String`.

### 3.1.2 Type conversion

Every expression written in the Java programming language has a type that can be deduced from the types of the elements inside the expression. It is possible to write an expression in a context where the type of the expression is not appropriate. In some cases, this leads to an error at compile time. In other cases the expression might be converted into a type which is acceptable. The Java language allows the programmer to convert an expression explicitly, but it also provides implicit conversions to make an expression fit into a context. An implicit type conversions can for instance be to use a byte value in an integer expression.

In every conversion context, only certain specific conversions are permitted. The conversions that are possible in Java are grouped into several categories and there are five conversion contexts in which conversion may occur. Each of the different conversion contexts allows only certain conversions to be done. The important conversion context when making a trait preprocessor is the method conversion context. So in the next subsections we will present only the conversions that are applicable in the method conversion context.
**Widening primitive conversion**

Widening conversion is the conversion where a subtype can be converted to its supertype. Widening primitives conversion allows any numeric primitive to be widened to its supertype without casting, i.e. any primitive type can be widened to double since double is a supertype of all the numeric primitives. This is written

\[ \forall p \in \{\text{byte, short, int, long, float, double}\} : \text{double} \Rightarrow p \]

Here is a complete list of the numeric primitives and which other numeric primitive they can be converted to using widening primitive conversion.

- byte to short, int, long, float or double.
- short to int, long, float or double.
- char to int, long, float or double.
- int to long, float or double.
- long to float or double.
- float to double.

Widening primitive conversion is significant with traits when it comes to checking if a required method is provided by the class applying the trait.

**Widening and narrowing primitive conversion**

From byte to char is a special conversion, first the byte is converted to int via widening primitive conversion, and then the resulting int is converted to a char using narrowing primitive conversion. This leads to the fact that a method with one formal parameter of the type char will not be able to take a byte as an actual parameter. Listing 3.1 shows one example of code that is illegal due to the fact that char is not a supertype of byte written: \texttt{char} \not\Rightarrow \texttt{byte}

To make the code from listing 3.1 legal we have to explicitly cast \texttt{b} to a char using the casting operation inside the call statement. This special handling of byte and char variables have to be respected by traits.

**Widening reference conversion**

Widening a references is analog to the way widening a primitives is done. A widening reference conversion exists from any type \(T\) to any type \(S\), pro-
CHAPTER 3. TYPE SYSTEMS

Listing 3.1: Illegal call

```java
void foo(char c) {
    ...
}

void test() {
    byte b = 8;
    foo(b);
}
```

Listing 3.2: Widening reference

```java
class RedirectInput {
    public void redirect(FileInputStream fis) {
        System.setIn(fis);
    }
}
```

vided $T$ is a subtype of $S$. A widening reference conversion never require a special action at run time and therefore never throw a runtime exception.

Take for instance the class InputStream and the class FileInputStream, both in the java.io package. InputStream is an abstract superclass of FileInputStream, that makes FileInputStream a subtype of InputStream:

```
java.io.FileInputStream <: java.io.InputStream
```

So by using the widening reference conversion we can use an instance of FileInputStream in any reference requiring a reference to InputStream.

Listing 3.2 shows an example of using this widening reference conversion to redirect standard input from a file using Java.

**Boxing**

Collections like the ones in the utility package in Java can only hold object references, so when a programmer wants to put a primitive value into a collection he has to box the value into a wrapper class. Boxing can be described as the process of turning a primitive type into a reference type.
CHAPTER 3. TYPE SYSTEMS

Wrapper classes are the classes that contain the value of the primitive. The wrapper classes for each primitive is listed here:

- **Boolean** is the wrapper class for the boolean values.
- **Byte** is the wrapper class for the byte values.
- **Character** is the wrapper class for the char values.
- **Short** is the wrapper class for the short values.
- **Integer** is the wrapper class for the int values.
- **Long** is the wrapper class for the long values.
- **Float** is the wrapper class for the float values.
- **Double** is the wrapper class for the double values.

To use the values boxed inside the collection the programmer has to do an unbox operation. Unboxing is the process of getting the primitive values from the instance of a wrapper class. Each wrapper class has its own method for getting the value stored inside:

- **Boolean** has the `booleanValue` method.
- **Byte** has the `byteValue` method.
- **Character** has the `charValue` method.
- **Short** has the `shortValue` method.
- **Integer** has the `intValue` method.
- **Long** has the `longValue` method.
- **Float** has the `floatValue` method.
- **Double** has the `doubleValue` method.

**Boxing conversion**

All this boxing and unboxing caused a lot of clutter in Java code, and in the release of Java 5 a new feature called autoboxing/unboxing was released. Autoboxing/unboxing is a language feature that introduces two new conversions, boxing conversion and unboxing conversion. The new conversion is meant to make the programmer able to use the primitives and wrapper classes interchangeable with no explicit conversions between them. This means that a method void `foo(int i)` that has a single formal parameter of the primitive type `int`, now can be called with the actual parameter `Integer` without having to call the `intValue()` explicitly.
CHAPTER 3. TYPE SYSTEMS

Boxing conversion converts values of primitive type to corresponding values of reference type automatically at compile-time. The following eight conversion are called boxing conversions.

- From type boolean to type Boolean
- From type byte to type Byte
- From type char to type Char
- From type short to type Short
- From type int to type Integer
- From type long to type Long
- From type float to type Float
- From type double to type Double

Unboxing conversion

Unboxing is the opposite operation of boxing, unboxing conversion converts values of reference type to corresponding values of primitive type.

- From type Boolean to type boolean
- From type Byte to type byte
- From type Char to type char
- From type Short to type short
- From type Integer to type int
- From type Long to type long
- From type Float to type float
- From type Double to type double

3.1.3 Conversion context

There are several contexts where conversion rules in Java apply, the Java language specification lists all of them [15]. When working with traits we are only concerned with the conversion contexts which have an effect on the type-checking of traits inclusion and checking on whether required methods are provided. The only conversion context which have an effect with traits is therefore the method invocation conversion context.
CHAPTER 3. TYPE SYSTEMS

Method invocation conversion context

Method invocation conversion is applied to each argument value in a method or constructor invocation. The type of the argument expression must be converted to the type of the corresponding parameter. Method invocation contexts allow the use of one of the following conversions.

- Identity conversion (a conversion from a type to that same type is permitted for any type).
- Widening primitive conversion.
- Widening reference conversion.
- Boxing conversion optionally followed by a widening reference conversion.
- Unboxing conversion optionally followed by a widening primitive conversion.

Understanding the method invocation conversion context is important when it comes to trait implementation because when a trait is flattened within a class, one wants to be sure that every call within the trait is legal with respect to the rules of the method invocation conversion context of the Java language.

3.1.4 Modifiers

Modifiers exist for both field declarations and method declarations. We are only interested in the modifiers applied to methods when working with traits.

Method modifiers

Method modifiers are associated with the method declaration and are used to mark the method with special properties. There is a group of modifiers that Java allows for method declarations, this group includes these modifiers:

- public: a public method is accessible everywhere.
- protected: a protected method is accessible from within the class declaring the method, and its subclasses.
- **private**: a private method is only accessible from within the class declaring the method.

- **abstract**: an abstract method declaration introduces the method as a member. Providing its signature, return type, and throws clause, but the abstract method does not provide an implementation. A declaration of an abstract method must appear directly within an abstract class.

- **static**: a method declared static is called a class method. A class method is always invoked without reference to a particular object.

- **final**: a method can be declared final to prevent subclasses from overriding or hiding it.

- **synchronized**: a synchronized method acquires a monitor before it executes. For a static method, the monitor associated with the Class object for the method’s class is used. For an instance method, the monitor associated with `this` (the object for which the method was invoked) is used.

- **native**: a method that is native is implemented in platform-dependent code, typically written in another programming language. The body of a native method is not present, indicating that the implementation is omitted.

- **strictfp**: the effect of the strictfp modifier is to make all float or double expressions within the method body be explicitly FP-strict. Within any expression that is FP-strict all intermediate values must be elements of the float value set or the double value set, implying that the results of all FP-strict expressions must be those predicted by IEEE 754 arithmetic on operands represented using single and double formats.

If a method has no modifier it is said to have default access, this means that the method is accessible within the package where it is declared. In additions to the modifiers above a method can also be marked with user-defined annotation, but this does not give the method any special meaning when it comes to the usage of traits.
3.2 Trait model

The formal model of traits [18] used for describing and reason about the original traits is made for the Smalltalk implementation. Where methods of a class can be modeled as a simple dictionary where names are mapped to method bodies. When inserting traits into Java, one way to proceed is to create a model of the Java language with traits, and create a flattening function that takes a class with traits and turn it into a standard Java class. Basically, to create a model of traits we could create a function flatten that defines a way to flatten classes with traits, like this:

$$\text{flatten} : C_T \rightarrow C$$

Where $C_T$ contains any Java class $C$ with any number of traits $t$ of $T$, and this function would produce an ordinary Java class $C$. To produce this model we would have to create a rigorous model for the whole Java language and make models for operations on traits. This would be very time consuming because all the rules of the Java language specification must be added, for example method overriding and return type substitution would have to be represented in some way. Instead of making traits in a theoretical way like this we have implement them directly using the Java programming language without basing the implementation on a formal model.

3.2.1 Java model

A flattening-based calculus for traits have been made for a subset of Java [7]. The calculus is based upon a minimal core calculus for the Java programming language called featherweight Java and the calculus is extended throughout the paper to include traits and interfaces as well as generics. This calculus shows the basics of flattening, and it treats the set of methods in a trait or a class as a simple dictionary where lookup is made using the method name. This model is insufficient and leads to errors when creating an application of the calculus because it takes away important properties of method signatures and method overriding, for instance take the exclusion operation of the calculus which is described in figure 4 of the referred paper. The definition of the exclusion operation as is also stated here:

$$\overline{M} - m \defeq \overline{M} \setminus \text{lookup}(m, \overline{M})$$

In this model $\overline{M}$ is a set of methods, $m$ is a method name and the lookup operation that return methods with a given name from a set of methods.
CHAPTER 3. TYPE SYSTEMS

The exclusion operation takes one operand, an identifier $m$. The definition of the exclusion operation is that the methods with name equal to $m$ are removed from a method set $M$. The problem is illustrated best in languages that allow method overloading. This is languages which allow several method bodies to have the same name but with different formal parameters. When we in this case exclude a certain name without thinking about their signature we get an unwanted side-effect that all methods with the same name are excluded when we in reality only wanted to exclude one specific method.

For instance we create a trait $t$ with two separate methods both named $foo$. One of the $foo$ methods take an integer parameter and the other take a double parameter. We call them $foo_i$ and $foo_d$. Now when a class is using $t$ that wishes to exclude $foo_i$ but not $foo_d$ we get a problem since this cannot be done with the model of featherweight trait Java. The reason is that only method names are used in the exclude operation. The same problem arises with the alias operator. The problem originates from the models treatment of methods and method lookup by using a simple method name.

In order to use the model to reason about traits in the full Java language we have to extended the model. This would generate a larger more complex model that would most likely be too complex for real applications.

3.3 Semantics

Traits are designed to not effect the semantics of a class. A class with a certain trait has the same semantics as if all the trait methods were inserted into the class body. This design has resulted in that traits have the flattening property. This property states that the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits.

The flattening property is important to preserve when a preprocessor is made for the Java language. The design of the preprocessor has been based on this property.
Chapter 4

Trait Syntax

This chapter describes the syntax for the trait implementation for Java. It provides the grammar and some examples of trait usages. Stefan Reichhart has done an implementation of traits for C# and the syntax created for traits in the Java language is strongly influenced by his paper [19]. We have tried to make the syntax as close to this as possible. After defining the grammar a few examples are included to make the syntax and the flattening effect clear.

To provide a concrete grammar for the Java implementation we have based our grammar on the Java language reference grammar, which can be found in the Java language specification [15]. The implementation is an extension of the Java language so it is only natural to describe Java traits the same way as the Java language is described.

When a rule below is not listed then it will be found in the Java language reference, e.g. the Identifier. Some of the rules has been abbreviated as there is no need to copy everything the language reference specify in order to add a rule, i.e. in the TypeDeclaration rule three dots are inserted instead of writing the full contents of the rule. The general design idea is to reuse as much of the Java grammar as possible.

The grammar below uses the same BNF-style conventions as in the Java language specification, with the following rules:

- \([x]\) denotes zero or one occurrences of \(x\).
- \(\{x\}\) denotes zero or more occurrences of \(x\).
- \(x | y\) means one of either \(x\) or \(y\).

The whole grammar of the trait extension is located in appendix B. The appendix contains the same rules as stated in the next sections, but in the
appendix they are all collected in a compact way to give an easy overview of the grammar.

4.1 Trait declaration

A trait has to be declared somehow, and declaration is described by the grammar in this section. Note that an ellipsis in TypeDeclaration is used in order to signify that the grammar rule is extended and the rest can be found in the Java language specification [15].

TypeDeclaration :

     ...
    TraitDeclaration

TraitDeclaration :

    trait Identifier [implements TypeList] TraitBody

TraitBody :

    { [TraitRequirements] {TraitMemberDecl} }

A trait declaration consists of a keyword \emph{trait} followed by an identifier and an optional list of interface names that are implemented by the declared trait. The trait body consists of two parts, the first part contains required methods used by methods inside the trait and the second part contains trait members as described in section 4.3. Trait members can be method declarations and composite traits.

4.2 Trait requirement

The requirement part is the first part of a declared trait. It consists of a set of method signatures. These signatures are called the \emph{requirements} of the trait. InterfaceMethodDeclaratorRest refers to a rule defined in the Java language specification.

TraitRequirements :

    requires { {TraitMethodRequirement} }

TraitMethodRequirement :

    {Modifier} Type Identifier InterfaceMethodDeclaratorRest
CHAPTER 4. TRAIT SYNTAX

Note that requirements are explicitly listed opposed to the way requirements are handled in the original Smalltalk implementation. In the Smalltalk implementation a requirement is generated by a call to an undefined method inside a trait method. The reason for explicitly adding method requirements when adding traits to Java is that the type-checker part of the preprocessor should be able to control if a trait can be flattened or not. Declaring a requirement inside a trait creates a precondition stating that flattening can only happen if a method with the same signature or with any compatible signature exists in the class.

4.3 Trait member

The second part of a trait declaration is the part containing the trait members. A trait member can be one of two elements, either a with block where other traits are used, or a method declaration which will add a provided method to the enclosing trait entity.

TraitMemberDecl :
    WithTrait
    TraitMethodDecl

TraitMethodDecl :
    {Modifier} Type Identifier MethodDeclaratorRest

There is nothing special about the TraitMethodDecl it is just like any ordinary method declaration in a Java class. The reason for creating yet another rule for method declaration and not refer to the Java method declaration is because the Java language specification has divided the method declaration in a way that makes it harder to reuse.

4.4 Trait application

In order for traits to be useful you have to supply a way to include the declared traits inside a class body. Inclusion of traits is done by creating a with-block within a class or interface body. A with-block can also be within a trait body.

ClassBodyDeclaration :
    ...
CHAPTER 4. TRAIT SYNTAX

WithTrait

InterfaceBodyDeclaration :

...  

WithTrait

WithTrait :
   with { {WithDeclaration ;} }

WithDeclaration :
   Identifier [TraitOperations]

A with block inside a class will signify that this class needs to be flattened before it can be compiled by an ordinary Java compiler. The preprocessor will locate these with blocks and execute the flattening. With-blocks can contain any number of traits, and the order of inclusion is not important.

4.5 Trait operations

Whenever traits are applied optional operations can be used on the included trait methods. Operations are added in order to resolve method conflicts that occur when composing traits. There are two different operations: aliasing and exclusion. Aliasing is an operation which takes two method signatures $m_1$ and $m_2$ as parameters. The effect of the alias operation is that the body of $m_1$ is reachable by calling $m_2$ as well as by calling the original $m_1$.

Exclusion is an operation which takes a single method signature as parameter. The effect of the exclusion operation is that the method with the associated signature is excluded from the composing trait entity. Use of exclusion operation is needed when two traits with the same method signature are supposed to be combined/flattened and you do not want to cause a method conflict. By exclusion the programmer can choose one of the conflicting methods.

TraitOperations :
   { {TraitOperation ;} }  

TraitOperation :
   Aliasing
CHAPTER 4. TRAIT SYNTAX

Exclusion

Aliasing:
MethodSignature -> MethodSignature

Exclusion:
~ MethodSignature

MethodSignature:
{Modifier} Type Identifier MethodSignatureRest

MethodSignatureRest:
FormalParameters [throws QualifiedIdentifierList]

4.6 Examples with trait syntax

In this section a few examples are presented to show the use of the trait syntax.

4.6.1 Simple example

We will start off with a traditional example of an application which does nothing other than print the well-known line “Hello world!” to the console. Listing 4.1 shows the code which first defines a trait called Greetable which contains one simple method sayHello. To use the defined trait we create a class Hello and inserts a with-block. When flattened the sayHello method is inserted into the class body of Hello. This is a very simple trait which does not post any requirements to the using class, and it can therefore be inserted into any Java class.

4.6.2 Operations example

Let’s look at more of the possibilities of traits. We are now going to define some traits and use both the alias and exclusion operations and show how a trait can express requirements. In this example we need to build a collection of integer values, and we need to do operations on the collection like reverse, sort and finding max values. Listing 4.2 shows the outline of the source code. The idea is that we factor out the reversing part and the sorting part in two separate traits, but both of the traits provide a method called
Listing 4.1: Hello world

```java
trait Greetable { // from file Greetable.trait
    public void sayHello() {
        System.out.println("Hello world!");
    }
}

class Hello { // from file Hello.java
    with {
        Greetable;
    }

    public static void main(String[] args) {
        new Hello().sayHello();
    }
}
```

max. When we use these traits in the collection class we have to exclude the version we do not need: the max method of the \texttt{Reverse} trait is excluded in this example. The alias operation is used as well when we insert the \texttt{Sort} trait. This is because we want to override the sort method in our collection class with a faster sorting algorithm than the one provided by the \texttt{Sort} trait. We still want to have access to the sort method of the \texttt{Sort} trait so we apply the alias operation to be able to access it as the method \texttt{slowSort} instead.

### 4.6.3 Synchronized wrapper example

In papers about traits there is one example that keeps turning up, and that is the synchronized read-write example. In this case we have two classes that both wants a synchronized read-write method. Listing 4.3 is an example of using traits to construct such a synchronized wrapper code that can be used in classes as long as the requirements are present.

\texttt{Read} and \texttt{Write} are traits with an implementation of the \texttt{read} and \texttt{write} methods. Classes using these traits need only include them to get their functionality. When making the synchronized version of \texttt{read} and \texttt{write}, we make a new trait which overrides the \texttt{read} and \texttt{write} methods and synchronizes calls on the methods. This shows how the super keyword
Listing 4.2: Operations

```java
trait Reverse {
    requires {
        int[] getValues();
    }

    int[] reverse() { ... }
    int max() { ... }
}

trait Sort {
    requires {
        int[] getValues();
    }

    int[] sort() { ... }
    int max() { ... }
}

class MyCollection {

    with {
        Reverse { ^ int max(); };
        Sort { int[] sort() -> int[] slowSort(); };
    }

    int[] values;
    int[] getValues() { return values; }

    int[] sort() {
        if (values.length > 100) {
            return quickSort();
        } else {
            return slowSort();
        }
    }

    int[] quickSort() { ... }
}
```

53
Listing 4.3: Sync read-write

```java
trait Read {
    int read() { ... }
}
trait Write {
    void write(int b) { ... }
}
trait SyncRead {
    synchronized int read() {
        return super.read();
    }
}
trait SyncWrite {
    synchronized void write(int b) {
        return super.write(b);
    }
}
class A {
    with { Read; Write; }
}
class B {
    with { Read; Write; }
}
class SyncA extends A {
    with { SyncRead; SyncWrite; }
}
class SyncB extends B {
    with { SyncRead; SyncWrite; }
}
```
can be used to generate wrapper functions.

4.6.4 Diamond inheritance example

The diamond inheritance problem does not exist when composing traits. Listing 4.4 is an example of a trait $T_d$ that uses traits $T_b$ and $T_c$ which both uses trait $T_a$. When the Diamond class uses trait $T_d$ it gets all the methods from $T_a$, $T_b$, $T_c$ and $T_d$ but since everything is flattened in the class it only gets one copy of each method so even if $a()$ is inherited in $T_b$ and $T_c$ the method is only included once in the final class. The end result of the flattened Diamond class is shown in listing 4.5

If a conflicting method name should occur in multiple traits you can always use the ability to exclude or alias the conflict methods.
Listing 4.4: Diamond inheritance

```r
trait Ta {
    void a() {}
}

trait Tb {
    with {
        Ta;
    }

    void b() {}
}

trait Tc {
    with {
        Ta;
    }

    void c() {}
}

trait Td {
    with {
        Tb;
        Tc;
    }

    void d() {}
}

class Diamond {
    with {
        Td;
    }
}
```

CHAPTER 4. TRAIT SYNTAX

Listing 4.5: End result

class Diamond {
    void a() {}
    void b() {}
    void c() {}
    void d() {}
}

57
Chapter 5

Method

This chapter contains the method used when implementing the traits preprocessor. It contains a section on language extension in general, a section on the preprocessor that has been made.

5.1 Language extension

This thesis contains an implementation of traits in the Java language by building a preprocessor for the Java compiler. This preprocessor is traits aware and it take as input code written in an extended form of Java and output ordinary Java. The extended form of Java is the language described by Chapter 4.

In this section we introduce different ways of extending a language, and some specific tools used for making these extension.

5.1.1 Parser extension

Parsing is the process of analyzing a sequence of tokens to determine its grammatical structure with respect to a given grammar. In most compiler designs the parser is the part that contains the syntax analysis and the construction of an abstract syntax tree. The abstract syntax tree is a tree structure containing nodes and edges which together represent the structure of the source code with respect to a grammar.

To extend the Java programming language two options were considered. The first one was to use some kind of language framework that lets the programmer extend the syntax of a language easily, the other option
was to create a parser that contains the syntax of the whole language extension. The first of our implementation attempts of traits were done by using an existing language framework for the Java language and adding the constructions which are needed for traits. However, the lack of good framework support for the Java 5 language transformation lead us to create a full parser for the traits extended Java language.

The next subsections contains the discussion of different tools encountered while deciding how to implement the preprocessor for Java.

5.1.2 Traits with OpenJava

The first alternative that was considered was OpenJava [20], which is an extensible language based on Java. OpenJava provides a Metaobject Protocol, and through this the programmer can customize the language to implement a new language mechanism. It is an interesting way of extending a programming language because you do not have to create your own parser for the extension.

OpenJava sounds like a good choice for implementing traits, but it has some drawbacks. First of all, OpenJava is outdated. The latest build at the current time is OpenJava 1.1 and it was released on August 4, 2002. The release naturally does not support Java 5. When using OpenJava you must also use a special syntax to activate the MetaObject.

A small example was produced in OpenJava that copies methods from a trait to a class. The code is shown in listing 5.1. The metaobject is called TraitAwareClass and does the copying of methods. First we define two traits, TraitHello which contains a method sayHello and a trait TraitBye which contains a method sayBye.

These two traits are used in the class MyObject shown in listing 5.2, by extending the class definition syntax with a uses keyword which takes a list of comma separated traits as argument. MyObject then gets both the method sayHello and sayBye.

The problem when using OpenJava in addition to its lack of Java 1.5 support, is its intrusive syntax. To specify that a class can use traits you must use the instantiates keyword after the class identifier with the specified class TraitAwareClass. This is a serious drawback in our case since we wanted to be able to control the exact syntax of trait usage. If OpenJava was extended to support Java 5 syntax then traits could be implemented in a clean way without making your own parser and abstract syntax tree.

Because of all the serious drawbacks further effort of creating traits in OpenJava was stopped. So the only trait like behavior that was created was
CHAPTER 5. METHOD

Listing 5.1: Two traits in OpenJava

```java
class TraitHello instantiates TraitAwareClass {
    public void sayHello() {
        System.out.println("Hello World!");
    }
}

class TraitBye instantiates TraitAwareClass {
    public void sayBye() {
        System.out.println("Goodbye");
    }
}
```

Listing 5.2: Trait usage in OpenJava

```java
class MyObject instantiates TraitAwareClass uses TraitHello, TraitBye {
    public static void main(String[] args) {
        MyObject myObject = new MyObject();
        myObject.sayHello();
        myObject.sayBye();
    }
}
```
a simple copying of methods from a trait into a class.

5.1.3 Stratego/XT

Stratego/XT [21] is a language and tool-set for program transformation. The Stratego language supports the development of transformation components at a high level of abstraction according to their web-site.

The Stratego system also supplies a parser for the Java 5 language, this is part of the Java-front package, a package that provides the syntax-related support for implementing Java transformation systems. There was not enough time to look into how the Stratego tools works in detail, but traits could possibly be implemented using the Java-front package for source code transformation since it provides an abstract syntax tree representation of source code and functions for transformation and printing the abstract syntax tree.

5.1.4 JavaCC

JavaCC is an abbreviation for Java Compiler Compiler, and is a parser generator that produces parsers in the Java language. It is one of the popular parser generators for use with Java applications, and there exists a good deal of documentation and examples for JavaCC [22]. There already exists a good Java 5 grammar written by Sreenivasa Viswanadha for the JavaCC parser generator system, and that is the grammar we choose to work with.

The preprocessor created uses JavaCC to generate the parser and JJTree [23] to generate the abstract syntax tree. We have extended the Java 5 grammar file for generating a parser to include the traits syntax described in chapter 4. JJTree can be used with JavaCC to generate a parser which also builds an abstract syntax tree. JJTree works as a preprocessor for JavaCC, that inserts the node building actions needed for creation of abstract syntax tree nodes at various places in the JavaCC grammar source. The output of JJTree is run through JavaCC to create the final parser.

The reason for using abstract syntax trees when operating with traits is that it is easy to work with. When you do a trait inclusion you typically copy the methods from the trait into the class definition. This is done with abstract syntax trees by just copying a whole subtree from one branch to another. See figure 5.1 for a graphical representation of an abstract syntax tree transformation. The nodes in the abstract syntax tree is represented by circles. The class nodes and trait nodes have child nodes which represents
method declarations. Solid arrows indicate references, and dotted arrows indicate node copying.

5.1.5 Another tool for language recognition, ANTLR

ANTLR is another tool with many of the same capabilities as JavaCC. It is a parser generator that has the possibility to create a Java parser based on a grammar file. There is also a possibility to create abstract syntax trees from the grammar source in the same way as JJTree is used for JavaCC. An existing grammar for ANTLR which parses Java 5 and generates a full abstract syntax tree also exists. This grammar could have been used to create the preprocessor instead of using JavaCC, but using one or the other is a matter of personal preferences.

5.2 JavaTrait preprocessor

This section will describe the preprocessor made to extend Java with traits. It will be presented as any other system development process, beginning with requirements, general design of the preprocessor and continue with more detailed implementation information. At the end we present one example of flow through the system.

5.2.1 Requirements

The preprocessor should be an implementation of the traits model as described in the traits paper [3], with adoptions to make traits fit into the Java
programming language. The preprocessor should allow:

- Definition of traits.
- Application of traits in class bodies.
- Support composition of traits.
- Implement the two operators for aliasing and exclusion of trait methods.

These requirements are listed in the syntax chapter as well by listing what the input language should look like, but the important part in the preprocessor is that the flattening of traits is implemented.

### 5.2.2 The implementation

The preprocessor inserts trait syntax into the Java 5 language. The preprocessor implements the flattening of Java classes and traits as well as type-checking of required methods. It is only a prototype and is by no means complete or bug free. There are still many aspects of the preprocessor that can be improved.

#### Design

The design of the preprocessor is based upon operations on abstract syntax trees. It is a simple concept: the source code is parsed by an extended Java 5 parser that creates an abstract syntax tree. The abstract syntax tree is transformed according to the rules of the trait model, which includes copying method declarations from the trait into the class. In the end the abstract syntax tree is printed using a PrettyPrinter of some sort. A PrettyPrinter is a program that takes as input an abstract syntax tree and walks the tree producing text output. The produced text is often indented to make it easier to understand and better looking, hence the name PrettyPrinter. The resulting program now has the trait methods inserted just as if they were written as methods in the class, see figure 5.2.

The parser used in the preprocessor is based on a JavaCC parser for the Java 5 language and the abstract syntax tree is created by using JJTree. Java classes are located in the collection of abstract syntax trees and the program iterates over the found Java classes and for each one flattens the used traits.

Figure 5.3 shows a class diagram of some of the important classes of the preprocessor and their associations. This is included to show how traits
Figure 5.2: Preprocessor Design
concepts can be modeled in a preprocessor or any system working with traits and classes. A different class hierarchy which is a big part of the preprocessor is the abstract syntax tree classes. There is approximately one class for each grammar rule, but these classes are all generated from the grammar file used by JJTree and are best viewed as part of the source code and not in any class diagram.

**Trait type system**

The Java type checking system is extended with methods for checking the type of the required method against provided methods using rules that resembles the rules for checking if one method overrides another. The preprocessor has an implementation of such a type checking system. Trait methods are located from the abstract syntax trees and injected into the blocks in the Java classes using them. This injection is made by copying the node for the method declaration in the trait abstract syntax tree. Since Java is statically typed we want to check these injection operations before they are performed.

**Flattening of traits**

Traits are often viewed as a set of methods, and flattening of traits is the process of collecting all the declared methods of the trait and all its composed traits. In the preprocessor traits are modeled as two sets of methods, a set of provided methods and a set of required methods. All the traits have a method used for flattening itself.

Flattening of a trait is the process of gathering all the methods of a trait and all composed traits into one single set of methods. The flattening of traits in the preprocessor is a recursive process where all the composed traits of trait $t$ have to be flattened before the trait $t$ itself can be flattened. Flattening a trait also involves execution of alias and exclusion operations.

When flattening traits, it is important to remember that methods in the current trait override the methods declared in the composed traits for instance in listing 5.3 we create a trait $Ta$ that defines a method $a$ and trait $Ta2$ that also defines method $a$. If trait $Ta2$ uses trait $Ta$ then the a method in $Ta2$ will override the a method in $Ta$.

Figure 5.4 shows the pseudocode used for flattening a trait in the preprocessor.
Figure 5.3: Class diagram of the preprocessor concepts
Listing 5.3: Traits overriding methods

```scala
trait Ta {
  void a() { }
}

trait Ta2 {
  with {
    Ta;
  }

  void a() {
    // overrides Ta.a
  }
}
```

for each trait t in used traits:
  flatten trait t
  execute operations on the flattened trait
  insert the flattened trait locally

insert provided methods locally

Figure 5.4: Flattening of a trait
CHAPTER 5. METHOD

Operations on traits

As mentioned before there are two operations one can perform on traits, aliasing and exclusion. The model of figure 5.3 shows a superclass Operation with two subclasses Aliasing and Exclusion. Operations in the preprocessor are modeled as classes with an execute method which does something to a method within a context. This context within the preprocessor is a set of methods since traits are modeled as a set of methods. When executing an aliasing operation, a new method is added to the context. When executing an exclusion operation, a method is removed from the context. This treatment of operations leaves room for extending the possible operations one can perform on traits.

Checking methods

The type checking when it comes to trait usage is mostly concerned with method signatures. It is therefore important to include a solid representation of method signatures that reflects the method signatures of the programming language. Method signatures in Java are composed of modifiers, return type, a method name, a list of formal parameters and a list of checked exceptions the method can throw.

The implementation of method signatures must contain the type checking mechanisms that are present in the language. In Java these mechanisms are the rules for when two method signatures are the same, when one method signature is a subsignature of another, when two signatures are override equivalent and when they are return type substitutable. Type checking of when two method signatures are exception compatible have also been implemented.

Same signature Two method have the same signature if they have the same name and argument types.

Subsignature The signature of a method \( m_1 \) is a subsignature of the signature of a method \( m_2 \) if either \( m_2 \) has the same signature as \( m_1 \) or the signature \( m_1 \) is the same as the erasure of the signature of \( m_2 \). Erasure of a signature is a signature without parameterized types or type variables as formal parameters.

Override equivalent Two methods \( m_1 \) and \( m_2 \) are override equivalent iff either \( m_1 \) is a subsignature of \( m_2 \) or \( m_2 \) is a subsignature of \( m_1 \).
Return type substitutable  $R_1$ and $R_2$ are return types of methods $m_1$ and $m_2$, if $R_1$ and $R_2$ are primitives, then they must be the same primitive. If $R_1$ is a reference type then $R_1$ is either a subtype of $R_2$ or $R_1$ can be converted to a subtype of $R_2$ by unchecked conversion. if $R_1$ is void then $R_2$ is void. These rules makes Java support a specialization of the return type, but only when returning a reference type. This is called covariant returns.

Exception compatible  Two methods $m_1$ and $m_2$ are exception compatible if there is no throws clause conflict between them. A throws clause conflict in Java is when a method $m_2$ overrides $m_1$ and $m_2$ includes at least one exception class that is not a subclass of any of the exception thrown by $m_1$.

Compatibility

In the preprocessor these rules have significance when it comes to checking method signatures for compatibility. Checking of compatible method signatures is done when the preprocessor needs to find out if a required method is provided at all by the applying class or by another trait. The rules above are used together to find compatible method signatures. The way they are used is by looking at the signature of the required method and check if there exists a method with a compatible signature in the set of provided methods. If a method signature compatible with the requirement is not found then a compile-time error which states that the method is missing is generated.

Checking of exception compatibility

To understand the need for the preprocessor to check exception compatibility we show a little example of a trait with a single requirement. The required method signature is defined as a signature throwing an exception. Listing 5.4 contains the example where a trait $T_a$ requires a method $m$ which throws an IOException.

What type of method can fulfill this requirement? There are four alternative methods that can be considered here. First a method which throws no exception, second a method that throws an exception which is of a superclass of IOException, third a method that throws a subclass of IOException, and fourth a method which is exactly the same.

1. void m()
CHAPTER 5. METHOD

Listing 5.4: Required method throws exception

```
trait Ta {
  requires {
    void m() throws IOException;
  }
}
```

Listing 5.5: Trait with try-catch block

```
trait Ta {
  requires {
    void m() throws IOException;
  }

  void foo() {
    try {
      m();
    } catch (IOException ex) { }
  }
}
```

2. void m() throws Exception  
3. void m() throws EOFException  
4. void m() throws IOException

The fourth method can obviously provide the trait with the required method because it has the same signature, but let's look at the rest. It turns out that method signatures three and four are the only ones that can provide the trait with the required method, and the following paragraphs explains why.

Traits methods are inserted directly into the using class, so naturally when a trait has a requirement, a call to this required method should work as if the trait method had been defined locally in the class. The reason why method one and two does not work can be showed with an example. Suppose the trait Ta used this required method and enclosed its use within a try-catch block as in listing 5.5.

It is important that this method foo and its try-catch block is legal when
CHAPTER 5. METHOD

Listing 5.6: Requirement throws no exception

class A {
    void m() { }

    void foo() {
        try {
            m();
        } catch (IOException ex) { }
    }
}

Listing 5.7: Requirement throws superclass exception

class B {
    void m() throws Exception { }

    void foo() {
        try {
            m();
        } catch (IOException ex) { }
    }
}

inserted into a class using the trait Ta. This is what restricts the class methods that are compatible with the requirement. We now provide four examples of classes providing the four different methods above.

The class A in listing 5.6 provides a method which throws no exceptions. A compilation of this class will fail due to the fact that the contents of a try-catch block has to be able to throw an exception. In this case m never throws an exception and it fails at compile-time.

The class B of listing 5.7 provides a method which throws an exception that is a superclass of the required method. This will also fail at compile-time since the try catch block will not catch all the possible exceptions.

The class C from listing 5.8 provides a method which throws an exception that is a subclass of the required method. This will work since catching
Listing 5.8: Requirement throws a subclass exception

class C {
    void m() throws EOFException { }

    void foo() {
        try {
            m();
        } catch (IOException ex) { }
    }
}

Listing 5.9: Requirement throws the same exception

class D {
    void m() throws IOException { }

    void foo() {
        try {
            m();
        } catch (IOException ex) { }
    }
}

An IOException will also catch an EOFException.

It is easy to see that this class D from listing 5.9 will compile with no difficulty, since the exception declared thrown by the required method signature is exactly the same as the exception thrown by the provided method.

5.2.3 Trait implementing an interface

A natural extension of traits when inserting them into the Java language is to make traits implement interfaces. This is a simple extension of the trait syntax and has also been done in a traits prototype for C# [19]. When a Java class implements an interface it means that the class provides methods that are defined as signatures in the interface. When a trait implements
Listing 5.10: Trait implementing interface

```scala
trait TEqual implements Comparable {
    public int compareTo(Object o) {
        return 0;
    }
}
```

Listing 5.11: Use of the equal trait

```scala
class UniversalEqual {
    with {
        TEqual;
    }
}
```

an interface it would mean that the trait provides bodies for methods that are defined as signatures in the interface. When a class applies a trait that implements a certain interface \( i \), this means that the class also provides the methods that are defined in the interface \( i \) and hence the class now also implements interface \( i \). However, there are exceptions to this behavior. Since the trait model provides the class with the exclusion operation on traits, there is a possibility of the class excluding the method needed to implement a certain interface.

Here we will present a clarifying example. Ordering in Java is often implemented by using the `Comparable` interface. The `Comparable` interface contains a single method with the signature `int compareTo(Object o)`. This method returns a negative integer, zero or a positive integer as the object is less than, equal to or greater than the specified object `o`. The `Comparable` interface is often used when sorting a collection of objects. In the example in listing 5.10 we define a trait that can be used by a class which should be evaluated to equal to any object with respect to the `compareTo` method of the `Comparable` interface. We define a trait `TEqual` which implements the `Comparable` interface.

Now we want to define a class with this trait to make the class equal to anything else. We define a class `UniversalEqual` with the trait `TEqual`, look at listing 5.11.
CHAPTER 5. METHOD

Listing 5.12: Result of flattening
class UniversalEqual implements Comparable {
    public int compareTo(Object o) {
        return 0;
    }
}

Listing 5.13: Violating the interface
class UniversalEqual {
    with {
        TEqual {
            public int compareTo(Object o);
        };
    }
}

The end result when we combine these two is the code of listing 5.12. We see that when the trait TEqual is used, all its methods are included and therefore all its implemented interfaces are added to the class. Now consider another example shown in listing 5.13 which is kind of pointless but nonetheless legal code. Here we exclude the only method TEqual provides.

In these cases we have the choice to carry the interface down to the class or not. Since the method is excluded from the trait it causes a compile time error to propagate the implements clause since the UniversalEqual does in fact not implement Comparable. On the other hand, exclusion is an operator that is meant to deal with conflicting methods. Now if we had two conflicting compareTo methods and excluded one, we would still implement the desired interface. This is the reason why in the preprocessor the implements clause is propagated down with no extra checks that the methods are provided because this is up to the Java compiler that handles the flattened class.

It is unfortunate that the preprocessor can cause such violations, but a type-checker can in theory check if the exclusions are causing an interface violation or not. This can be done after flattening of the Java class, when all the trait methods are inserted into the class body. The type-checker can
iterate over the interfaces and check if the required methods of each interface is present. This check is a part of the Java compiler and is therefore not needed in the preprocessor.

5.2.4 Interface with trait

Traits can implement interfaces, but let's look at the idea to have interfaces with traits. This idea would enable interfaces to carry traits, in the sense that interfaces are able to contain with-blocks that are inserted into classes implementing the interface before flattening is done.

In the preprocessor this special possibility has been implemented. The way it works is that the syntax is extended so that interface bodies can contain with-blocks in the same way as classes. Now we can naturally not flatten interfaces in the same way as we flatten classes, because it would lead to illegal code. So interface flattening is implemented in the preprocessor like this: when an interface is implemented by a class the with-blocks that are contained inside this interface is injected into the class as if the with-blocks were written directly into the class body.

So an interface with traits can be thought of like a carrier of with-blocks, and hence you can have multiple inheritance of method bodies only by extending the interface syntax.

We will show that this can be a good idea when you want to provide a default implementation of the methods in a certain interface. Assume that you want to implement a `MouseListener`, which is an interface with five methods: `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased` and assume that you only want to catch the event where the mouse is clicked, the other methods are only supposed to be empty. This can be solved by using the new trait feature.

First we define a trait containing the empty mouse methods as done in listing 5.14. We could have decided that the default implementations of the listener methods should have some side effect for instance logging events, but right now we keep the method bodies empty. Then we define the `MouseListener` interface with `EmptyMouseListener` in listing 5.15.

In the end we define our class that should listen to the mouse clicked event. The class contains one simple method which prints out the number of times a mouse click event has happened. The code is shown in listing 5.16.

This problem could also be solved by using single inheritance. We could have created a superclass of `PrintClickCount` that provides us with empty methods of the `MouseListener` methods, and overriding only the
CHAPTER 5. METHOD

Listing 5.14: Empty mouse listener

```java
trait EmptyMouseListener {
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
}
```

Listing 5.15: Interface with trait

```java
interface MouseListener {
    with {
        EmptyMouseListener;
    }

    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

Listing 5.16: Using the mouse listener

```java
class PrintClickCount implements MouseListener {
    private int num_clicks = 0;

    public void mouseClicked(MouseEvent e) {
        num_clicks++;
        System.out.println("Number of clicks: " + num_clicks);
    }
}
```
mouseClicked method in PrintClickCount. This solution would however manipulate the class hierarchy and that might not always be a good idea.

This feature is similar to something called interfaces with default implementations [24], where an interface can contain a default implementation of any method signature inside the interface. The class implementing this interface can choose to use the default implementation of the interface method or provide its own.

5.2.5 Trait type checking

This section will describe the trait type checking in more detail, and the different sections will bring up some interesting points when it comes to implementation decisions.

Class fulfilling requirements

This thesis uses the term fulfilled about requirements of traits that are provided by classes. This section will contain a formal description of what it means that a class fulfills the requirements posed by the applied traits.

A class is compatible with a trait when all the requirements of all the composed traits of the class is provided. A requirement is a method signature listed in the requires block of a trait. In order for a class to apply a trait all its requirements need to be fulfilled. To fulfill a requirement is to provide a method with a compatible signature in the class or in any superclass.

Method signature $ms_2$ is compatible with $ms_1$ if $ms_1$ could be replaced by $ms_2$ inside a class body and all calls to $ms_1$ would still be legal. In Java this implies that $ms_2$ and $ms_1$ must at have the same name and parameter count, and they must also be return type substitutable and exception compatible. Any parameter $p_i$ in $ms_1$ could be replaced by a supertype of $p_i$ in $ms_2$. This leads to the fact that a requirement $\text{foo}(\text{int } i)$ could be fulfilled by a method $\text{foo}(\text{double } d)$, since double is a supertype of int.

Method modifiers

Section 2.2.2 stated that it is important to know how modifiers have an effect on the trait methods insertion into the Java class. This section will describe how this is handled in the preprocessor.

The first group of modifiers we handle are concerned with access restrictions, these are the modifiers public, private, protected. When the implementation was made there was a question of whether access modifiers
should be strictly matched when checking for provided methods that fulfill requirements or not. To solve this issue we have to see how method access is handled within a Java class.

In Java any method within a class has access to any non-private methods of superclasses as well as all methods declared directly inside the class. Trait methods are inserted directly into the class body and are treated as though they were members of a class, so any trait method has access to any method defined inside the class body with any access modifier, but the trait method does not have access to private methods defined within superclasses. That is the only restriction that effects trait methods, so in effect when we have a requirement with the signature `private void foo()` it means that the foo method must be a part of the class using the trait and not a part of its superclass.

In the preprocessor the private access modifiers is taken into account when checking for requirement compatibility. When a trait requires a private method that method or any compatible method must be a part of the class which applies the trait. If the compatible method is not defined within the class applying the trait an error is generated. An error is also generated if a requirement compatible method is found inside a superclass and has the private access modifier. This is due to the effect of the private modifier which protects a method from access by all other classes than the one it is defined in.

**Static**  An interesting discussion is whether traits should be able to contain static methods. Since trait methods are injected as if they were written directly into the class there is no problem to let traits be able to define static methods, but there is an issue when checking calls from the static method. Since a static method in Java only can call other static methods, this needs to be checked by the preprocessor or the Java compiler. A traits preprocessor could check that every static required method signature is fulfilled by a compatible static method.

For this to work then a preprocessor has to enforce that any required method declared with a static modifier has to be provided as a method with static modifier. The preprocessor created supports checking of static requirements. An error is generated if a static requirement is compatible with a non-static method, or if a static method is compatible with a non-static requirement.
Synchronized  Synchronization is a handy feature in Java made possible with a single method modifier, but should a provided method which is unsynchronized be able to fulfill the requirement of a synchronized method. There are two options, either it can or it can not. If it can, then that means that even if a method invocation inside the trait thinks its synchronized that might in fact not be the case. If it can not then you are assured that a a synchronized invocation always is synchronized. In the preprocessor a unsynchronized method can fulfill the requirements of a synchronized one. This is done to make the implementation more useful, although it can lead to code that is not thread-safe.

Other modifiers  Abstract methods are methods which does not yet have an implementation. When a requirement is fulfilled by a method with the abstract modifier it means that the method is not part of the class but should be a part of any non-abstract subclass. The abstract modifier is not treated specially in the preprocessor because the call to an abstract method from within a trait method is a perfectly legal call.

The native modifier is treated the same way as the abstract modifier. A native method is a method marked as an externally implemented method, but a call to any native method is just like any other call so it is not treated specially.

A required method marked with the strictfp modifier does not have to be fulfilled by a provided method with the strictfp modifier. This comes from the rules of overriding of the Java language specification which states that the presence or absence of the strictfp modifier has no effect on the rules for overriding methods.

Annotations are user defined modifiers and they are allowed by the syntax to be present in requirement signatures, but they have no effect when checking for the presence and compatibility of a provided method. The final modifier prevents subclasses from overriding or hiding the marked method, but a required signature with the final modifier has no special meaning it is treated as though the final modifier was not present.

Widening conversion and provided methods

Widening conversion is, as mentioned in section 3.1.2, the mechanism that provides the programmer with the possibility of invoking a method with an actual parameter that is a subtype of the formal parameter specified in the method signature.
Widening conversion can be applied in a method invocation context, when a method invocation happens within a trait method the same widening conversion can happen, and this is why the type checking of required versus provided methods must allow provided methods to fulfill the requirement from a trait method signature with the supertype as formal parameter.

Boxing conversion

With traits in Java the boxing conversion also has to be supported. This means that a primitive and a wrapper class could be used indifferently when it comes to method invocation. This leads to the fact that required methods could be declared with a primitive type as formal parameter, and the provided method could have a method signature that uses the wrapper class associated with the primitive. The opposite is also true, as when a required method is declared to have a formal parameter of a wrapper class type. The requirement can then be fulfilled by a method using a signature with the primitive associated with the wrapper class.

For instance we want to create a trait \( T \) which requires a method \( \text{int foo(int i)} \) that does some operation on an integer value and returns a result. Lets just say that it calculates the foo value of any integer \( i \). Now when you invoke this \( \text{foo} \) method within your trait, there are some alternatives as to what signature \( \text{foo} \) actually can have. Because of the autoboxing/unboxing features of Java 5 the actual signature of the \( \text{foo} \) method can by any of the following:

- \( \text{int foo(int i)} \)
- \( \text{Integer foo(int i)} \)
- \( \text{int foo(Integer i)} \)
- \( \text{Integer foo(Integer i)} \)

The same rules apply for the rest of the wrapper classes associated with primary types. This is solved in the preprocessor by adding a check inside the reference type and the primitive type, so that any wrapper class is considered equal to the associated primitive, and any primitive is considered equal to the associated wrapper class.
Autoboxing and widening

Most of the primitives in Java can be used in a widening conversion to invoke a method which takes a supertype as formal parameter. The same rule must be applied with traits and method invocation within traits. This leads to the fact that required methods with a certain primitive type can be fulfilled by a method with a supertype as formal parameter. From section 3.1.2 we have one supertype chain described by:

$$\forall p \in \{\text{byte, short, int, long, float, double}\} : \text{double} \rightarrow p$$

This chain says that $\text{double}$ is the supertype of the types $\text{byte, short, int, long, float or double}$. It leads to the fact that any method providing a formal parameter of type $\text{double}$ can fulfill the requirements of a method with a formal parameter of any of the subtypes $\text{byte, short, int, long, float or double}$.

The hierarchies for primitives and wrapper classes are very different, figure 5.5 shows the primitive hierarchy and figure 5.6 shows the wrapper class hierarchy. We see from figure 5.5 for instance that every numerical primitive is a subtype of the $\text{double}$ primitive. However the wrapper class hierarchy does not have the same structure at all, since it is the $\text{Number}$ class and not the $\text{Double}$ class, that is the supertype of all numerical types. This wrapper class hierarchy has consequences when it comes to checking if a required method is provided by the enclosing class.

Due to autoboxing we can allow a method with the signature $\text{void foo (Integer i)}$ to fulfill the requirement of a signature $\text{void foo (int i)}$, but a method with the signature $\text{void foo (Double d)}$ cannot fulfill the same requirement which is due to the fact that the class $\text{Double}$ is not a supertype of the class $\text{Integer}$ as seen in figure 5.6. To make a general method that takes a wrapper class as argument we need to define a method which has the signature $\text{void foo(Number n)}$.

If we look at unboxing we get a different view. Suppose a trait requires a method $\text{void foo(Integer i)}$. This can be fulfilled by a method signature $\text{void foo(int i)}$ due to unboxing conversion, and since we can also apply primitive widening conversion a method signature $\text{void foo(double d)}$ could also fulfill the same requirement.

Checking exclusion and aliasing

The two operations on traits can also be a target for checking from the preprocessor. An exclusion removes a method body from a trait composition and a proper implementation of traits should include methods that check
CHAPTER 5. METHOD

Figure 5.5: Primitive hierarchy

Figure 5.6: Wrapper class hierarchy
that the excluded method actually exists. An alternative is to allow the exclusion of non-existing methods without giving any warnings or errors. In the preprocessor there is implemented a check that excluded methods actually exist. If an excluded method does not exist then a compile-time error is generated. This is done to hinder misuse of exclusion due to typing errors.

Aliasing involves two method signatures, the original method signature and the alias method signature. A trait implementation should check if the original method signature is present, and give an error if its not. There should also be a check to see if the alias method signature already exists inside the flattened trait. If one is found then an error can be thrown.

In the preprocessor the aliasing operation involves two whole signatures. This is done to get a clean implementation, and to get the possibility to alter the modifiers of a trait method in addition to its name. The features of trait method aliasing could be used in cases where a synchronized method is wanted, which can be accomplished by aliasing a method name and putting the synchronized keyword in front of it. The same operation can involve the static modifier. It can be seen as a feature that the programmer can turn any method from static to non-static, and the other way around only by aliasing.

Trait modifiers

Since the type declaration of classes and interfaces in Java can be composed of modifiers like abstract and final, then why not allow for traits to have modifiers that resemble the semantics of modifiers on other type declarations. A static trait could for instance be a trait where every method is declared static, and a final trait could be a trait where every method is final and cannot be overridden. In making the syntax for traits there was not found any convincing reason for having modifiers on traits, and that is why it is not included in the preprocessor.

Variable lookup

Variable lookup within a trait body should not be allowed in any implementation of traits, since traits cannot hold state and the traits model does not have a way of require state variables. In the preprocessor variable lookup in the trait body is allowed due to the reuse of the Java 5 grammar. There is nothing that restricts a method body so that it can’t look like the code in listing 5.17.
Listing 5.17: Trait method with field lookup

```java
trait TInc {
    void inc () {
        i += 1;
    }
}
```

Listing 5.18: Trait method with field requirement

```java
trait TInc {
    require {
        int i;
    }
    void inc () {
        i += 1;
    }
}
```

One solution could be to alter the trait model in order to pose requirements on what fields that must be available in the applying class, much in the same way as the requirement specification now states what methods should be available. Take the example from listing 5.18, where we require a field with the type `int` and the name `i`. The trait `TInc` does not have any state, it only requires that the class using the trait has a field with the type `int` and the name `i`.

**Argument and return type**

The argument type and return type problem as described in section 2.3.5 and section 2.3.6 are not handled in any good way in this preprocessor for Java. A trait declaration does not implicitly create a reference type which can be used as formal arguments to the methods of the trait body.

The `ThisType` keyword is not implemented due to time constraint, but the idea was that each trait should have an implicit reference type with the name `ThisType`. When flattening a method with `ThisType` in the method signature or with `ThisType` in the method body then the flattening process
replaces every ThisType reference to the enclosing class name. For instance we define a class Node and a trait Linkable. The trait Linkable has a single method with the following signature ThisType next(). When the Node class is flattened the method ThisType next() is injected into Node and the method signature is altered to become Node\_next().

The example is shown in listing 5.19 and the resulting code when flattened is shown in listing 5.20.

**Runtime reflection**

To get the runtime and compile-time information about applied traits we propose to use the annotations mechanism from Java 5. Annotations are markups in the code that is compiled into the class file and can be located at compile time as well as at runtime.

The annotation declaration in listing 5.21 can be used to annotate meth-
CHAPTER 5. METHOD

Listing 5.21: Trait annotation declaration

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface TraitMethod {
    String value();
}
```

Listing 5.22: Annotated method

```java
public class A {

    @TraitMethod("Ta")
    void m() {
    }

}
```

ods with what trait the method originated from. It is obvious that such information can be useful when debugging code produced by a trait preprocessor. The annotation could be extended to include the path of the source file containing the trait declaration and the line number where the trait method is gathered from. The annotation of listing 5.21 is used to annotate trait methods in the preprocessor. In comparison, the C# prototype annotate trait methods by inserting the trait name inside an ordinary comment associated with the method.

Listing 5.22 is an example where a class has applied a trait Ta which contained one method m. Annotations is used to preserve the trait information after flattening of the class.

5.2.6 Preprocessing example

This is an example of what happens in the preprocessor when a set of files are flattened. The example is one that has been used in some trait papers and we call it the synchronized IO example. The example mentioned here is a variant of the synchronized wrapper code of section 4.6.3 where only the read class is implemented for simplicity.
Input files

The source code for this example is provided to the transformer through the command line. The input files are:

**TRead.trait**  Contains the source code for the trait TRead. It provides one method with the signature `public char readChar()` and it requires one method with the signature `public InputStream getInputStream()`.

**TSyncRead.trait**  Contains the source code for the trait TSyncRead. It uses the trait TRead and it provides one method with the signature `public synchronized char readChar()` which overrides the one defined in TRead. To access the overridden method the old `public char readChar()` is aliased to `public char unsyncReadChar()`.

**SyncDataReader.java**  Contains the source code for the class SyncDataReader which uses the trait TSyncRead and uses the method `readChar()`. It also provides a method with the signature `public InputStream getInputStream()`.

Abstract syntax trees created

The first thing that happens is that three abstract syntax trees are created from the input files. See figure 5.7, 5.8, and 5.9.
Chapter 5. Method

Figure 5.8: AST created from TSyncRead.trait

Figure 5.9: AST created from SyncDataReader.java
CHAPTER 5. METHOD

Abstract syntax tree transformed

The next step is to transform the abstract syntax trees. Since there is only one abstract syntax tree containing a class there is only one transformed abstract syntax tree as output. As seen in figure 5.10 the class SyncDataReader has now got the method declarations together with the right signatures into its body.

Signature checking

In this example there is one method that needs to be checked, and that is the required public InputStream getInputStream(). The requirement comes from TRead and is brought down to the class SyncDataReader since neither TRead or TSyncRead contains a method signature that is compatible with public InputStream getInputStream().

The type-checker of the preprocessor will locate the getInputStream method in the class SyncDataReader. The type-checker will then check if the methods are compatible. Since they have exactly the same signature there is no problem, but if the getInputStream in SyncDataReader for instance only returned an Object then an error would be created.

PrettyPrinting

When the transformation is done the pretty printer goes through the abstract syntax tree and produces the method bodies one by one, just as if the trait methods had originated from the Java file.
Chapter 6

Discussion

The work done with traits in Java is not unique and in this chapter we have a section containing comparisons of the preprocessor with other trait implementations. This chapter also contains a section where the preprocessor is evaluated, and missing features and alternative designs are presented.

6.1 Other work on traits

The following section compares the preprocessor to other implementations of traits, with respect to syntax, features, representation and design.

6.1.1 C#

This section will discuss the differences between the Java preprocessor and the C# implementation [19]. This implementation is the one closest related to the preprocessor made in this thesis, because it is implemented by using the same idea of language extension.

Implementation

The implementation is made with Smalltalk and is a simple prototype of a preprocessor. It is meant to be a study case for a clean implementation of traits in C#. The reason it was written in Smalltalk is unknown and seems like a strange choice when both the input and the output language of the preprocessor is C#.

The C# traits preprocessor is made with the VisualWorks implementation of the Smalltalk programming language. It uses a parser generator
called SmaCC to generate an abstract syntax tree from the given input, and it operates on the abstract syntax tree when flattening the trait and spits out C# code when done.

Syntax

There is a small difference in syntax between the C# prototype and the Java preprocessor. The syntax used to declare that a C# class applies a trait is borrowed from the Squeak implementation. The difference in syntax is shown in listing 6.1, where the first Node class uses the C# notation and the second Node class uses the Java notation.

Here is a description of the traits grammar in C# using the same notation as used when describing the traits grammar in Java.

---

Listing 6.1: Trait usage comparison between C# and Java

```csharp
class Node { // C#
    uses {
        TLinkable;
    }
}

class Node { // Java
    with {
        TLinkable;
    }
}
```

---

TraitDeclaration:

```
trait Identifier TypeSignature TraitBody
```

TypeSignature:

```
[TypeParameters] [Interfaces] [Constraints]
```

TraitUses:

```
uses { {TraitUse} }
```

Requirements:

```
requires { {Signature} }
```

---
Method signatures are special in the C# implementation. The Signature rule allows method signatures to be written in a compact way. Method signatures can be written without braces if the signature has an empty list of formal parameters. If the method signature has any formal parameters then these are written as a comma separated list of type names, without any identifier. In the Java preprocessor formal parameter must be written using a type name and an identifier making the signature more verbose.

Requirements are written using the same syntax as in the Java preprocessor. When it comes to trait application there is an interesting difference as shown in listing 6.1. In the C# implementation the keyword uses is used to declare that a class applies a trait. This could have been adapted in the Java preprocessor as well, but the reason for choosing the keyword with instead is the meaning of the word trait. From biology a trait can be defined as a distinguishing characteristic or quality of an organism, which implies that a trait is something that the subject has, not something it uses. From that point of view a trait or a characteristic of a class is described as some property. One could say for instance that a list which can be reversed could be modeled as a class List with the trait Reversable.

The syntax of traits in C# allows the programmer to write generic traits.
A generic trait is a trait which is defined with generic type parameters. Since generics is a part of Java 5 as well, the possibility to write generic traits should be included when traits are introduced in Java, but the generics properties of traits and how they effect the trait model will have to be further researched before they can be inserted into a type-checking trait preprocessor.

The generic traits of C# suffers from the fact that the C# prototype is unfinished. The prototype restricts the names of the generic type parameters of a class and the applied traits. The restriction is so hard that if the class and the applied trait has different generic type parameters then an error is produced.

Traits in C# are able to implement interfaces and the interfaces are inserted in the class using the trait when flattening is done. This is the same way as interface behavior is implemented on the preprocessor for Java.

Requirements

Requirement blocks are allowed in the C# implementation to specify required methods used by a trait in the same way as in the Java preprocessor. The C# implementation also provides type-checking of the trait requirements and the classes that apply the trait. However required methods are checked more restrictively in the C# preprocessor than in the Java preprocessor. In the Java preprocessor we check if a class contains a method with a compatible signature of the required method, while the C# preprocessor checks if a class contains a method with exactly the same signature as the required method.

This restrictive checking of requirements leads to unnecessary errors in some cases. We could for instance create a trait that could be flattened manually with no problem, but we would get an requirement error when flattening with the C# preprocessor. This contradicts the flattening property of traits.

Let’s say that a trait requires a method $m$ with a single formal parameter type $t_1$. Now with the restricted checking of parameters this requirement can only be fulfilled by a method $n$ with exactly the same formal parameter type $t_1$. When comparing with the Java preprocessor, a required method $m$ with a single parameter type $t_1$ can be fulfilled by any method $n$ with a single formal parameter type $t_2$ where $t_2 > t_1$.

The code in listing 6.2 and listing 6.3 produced an UnfullfilledTraitRequirement error when attempting to flatten, but the code is legal C# code when flattening manually. The TRequire trait in listing 6.3 provides a method for
emptying a list, and requires a method that can take a single parameter of the class List which is a subclass of the class Collection as shown in listing 6.2.

Checking of return types is also very restrictive, there is no check on whether a provided method is return type substitutable or not. The only check made is whether the return type is exactly the same as the required return type.

Implementation details

The general design of the C# implementation is based on creation and transformation of the abstract syntax tree of the compilation units. A trait is
CHAPTER 6. DISCUSSION

represented by a class, and the visitor pattern [25] is used to flatten a trait. The flattening is done by passing a flattener object to each compilation unit. Flattening is the process of removing trait declarations from a class, adding libraries and extensions, and adding all the methods of the trait.

This design is similar to the design used in the Java preprocessor. Abstract syntax trees are used in both implementations to do the flattening. The greatest differences in design is the checking of requirements, where the Java preprocessor checks if methods are available that are compatible with the required signature. The C# implementation is more strict with requirement checking.

Local variable lookup is legal in the C# implementation as well as in the Java preprocessor. This has been done to simplify the process of generating a parser or to simplify the method body checking. Both implementations do not have any restrictions as to what code a method body can contains, so a trait method can contain method calls to undefined methods and variable lookup even if it is unknown if the class applying the trait supplies this variable.

6.1.2 Scala

Scala is a general purpose programming language which combines features of object-oriented and functional languages. Its syntax resembles Java and Scala programs are constructed to be fully interoperable with Java [26].

Design

Scala does not contain the notion of interfaces, it contains traits instead. Traits in Scala define an interface that classes can implement. In addition to being an interface definition the trait can also provide default implementations of methods. In Scala a trait naturally defines a type with the same name, since a Scala trait is treated as an interface.

The Scala traits are applied to classes like mixin classes. A class can be defined with a chain of traits that are a part of the class. As pointed out in section 2.1.3 the problem with mixin inheritance is its possibility to silently override methods. However the Scala language does not allow trait methods to silently override each other, and the Scala compiler produces an error when two traits have conflicting methods.
CHAPTER 6. DISCUSSION

Syntax

The syntax of traits in Scala is defined according to this grammar:

\[ TmplDef: \]
\[ \text{trait TraitDef} \]

\[ TraitDef: \]
\[ \text{id [TypeParamClause] [requires AnnotType] TraitTemplate} \]

\[ TraitTemplate: \]
\[ \text{[extends MixinParents] [TemplateBody]} \]

\[ MixinParents: \]
\[ \text{AnnotType \{with AnnotType\}} \]

The \textit{with} keyword is used to define that a class is mixing in a trait. We see from the above grammar that the with keyword can also be a part of the trait definition. By using the with keyword in the trait definition we can create composite traits. Having the with keyword in the signature in Scala is natural since traits in Scala are mixins and are used to apply additional type relations. Since the Java preprocessor does not treat traits as mixins the with-blocks are a part of the class body instead of a part of the class or trait signature.

Requirements

Traits in Scala can give requirements by declaring abstract methods as part of the trait body. Abstract methods in Scala are method signatures with an empty method body.

The \textit{requires} keyword of the trait definition is not used to specify method requirements. The requires clause of the trait definition is used to set requirements as to what types can use this trait. In effect the requirement specifies what type \textit{this} can refer to, so the use of \textit{this} within a trait body can be type checked. The requires keyword can for instance be used to declare that any class using a trait \textit{t} should be a subtype of the class \texttt{Node}, in this way you can use \textit{this} as a reference to the \texttt{Node} class within the trait body.
CHAPTER 6. DISCUSSION

Conflicts

Scala does not provide any of the original traits operations like: exclusion and aliasing. The reason might be that traits are used to define interfaces and traits should therefore provide a final unchangeable collection of methods. Method conflicts can therefore not be handled with exclusion in the composing class. Method conflicts can still appear, for instance when creating a class with two traits $T_1$ and $T_2$ which both provide a method with the same signature we get a method conflict. However Scala allows this method conflict to be resolved by overriding the conflicting method within the final class.

Scala is an interesting language since it has accomplished to introduce traits into a statically typed language. The Scala model of traits is different from the original model as implemented in Smalltalk. The main difference from the Scala traits and the traits of the preprocessor is that in Scala traits define a corresponding type, while in the preprocessor a trait does not define a type. Another difference is the handling of traits. In Scala traits are mixed into the class, while in the preprocessor traits are injected. The difference is that when injecting methods there is no side-effects, the flattening property is treated with respect. While when a trait is mixed into a class a type relation is created.

6.1.3 Perl 6

Perl 6 [27] is the next version of the once very popular scripting language Perl. The design of Perl 6 is based on a community driven rewrite of Perl 5. The idea is to fix what did not work in Perl 5 while adding missing features and introduce some completely new ideas. One of the completely new ideas is the introduction of traits in Perl 6. Although in Perl 6 they are named roles instead of traits. The Perl 6 roles are inspired by traits as defined in the traits paper [3].

Syntax

A role is declared like a class, but with a role keyword. The does keyword is used when a class wants to apply a role. The use of roles can be declared in the class signature or within the class body, the choice like many Perl language features is up to the programmer. Here is an example where class Dog is a subclass of Mammal and applies two roles Pet and Sentry.

class Dog is Mammal does Pet does Sentry { ...}
class Dog {
  is Mammal;
  does Pet;
  does Sentry;
  ...
}

A role contains method signatures and default method bodies. If a role
only contains method signatures is the same as an interface. A class method
definition overrides any role methods with the same signature and a role
method overrides any methods inherited from other classes. These over-
riding rules are the same as in the original trait model.

A surprising feature of Perl 6 roles is that they may also contain at-
ttributes. Attributes inside a role like this can be shared with the class using
the role or be totally private inside the role.

Conflicts

If there are two roles which introduce a method with the same name then
a method conflict occurs. This conflict is found at the time when the class
is composed. There are ways to resolve this conflict. One can provide a
method in the class that override the conflicting methods like in Scala. The
overriding method has the opportunity to call the appropriate role method.

Perl is well-known language that many programmers respect and the
inclusion of something similar to traits into this language can possibly be a
step in the direction of wider acceptance of traits as a unit of reuse.

6.2 Evaluation of the preprocessor

The preprocessor is an implementation of the stated requirements of sec-
tion 5.2.1. It produces the flattened Java classes when the trait syntax is
used in a proper way. Although the preprocessor gives the right results in
the test cases when applied, it does not mean it is an ideal solution. The
preprocessor is more of a proof of concept implementation than a stable
production quality traits preprocessor.

It is hard to say how the ideal solution to traits in statically typed lan-
guages should look. The ideal solution should at least implement the syn-
tax of traits, and it should also contain a type-checker which asserts that
trait flattening can be done safely and is capable of producing decent error messages when something is wrong.

The usability aspect of the preprocessor has been down-prioritized in favor of feature implementations, that is why the error messages of the preprocessor sometimes can be vague or incomplete.

6.2.1 Remaining work on the preprocessor

There still remains work to make the implementation a fully usable traits preprocessor.

- The preprocessor lacks support for generic traits. Generics which has become a natural part of the Java 5 language should be implemented in any ideal trait preprocessor.

- Support for packages. e.g. when writing InputStream the qualified name is really java.io.InputStream. This is not implemented yet and requires a loading of class information from classpaths to build up a symbol table. Package structure should also be a part of a full Java implementation of traits.

- There are many places where optimizations could be inserted. Smart use of hashing and limiting the traversing of lists. Since the preprocessor is only a prototype the main goal was to get some of the basic functionality to work and not to produce something that was fast. There also needs to be extensive testing of the application.

- The preprocessor lacks support for traits referring to the applying class type. Implementation of the selftype keyword or the ThisType keyword would solve this. The full implementation would then translate the use of the keyword into the applying class name when flattening.

6.2.2 New trait feature

The idea of an interface carrying a with-block is an original trait feature which is suggested to have useful applications. This feature can for instance be used to create default implementations of interface methods so the class implementing an interface might only need to provide or redefine some of the methods defined in the interface. Interfaces are currently used to simulate multiple inheritance in Java, but with this new feature the
model of inheritance is one step closer to being multiple inheritance where method bodies are inherited as well as method signatures.

This feature is similar to AspectJ's possibility to insert methods into classes using certain interfaces. This feature is called inter-type declarations within the AspectJ system. With this feature the aspects can declare that a method body is a part of interface $i$ and the method body is then inserted into all classes implementing the interface $i$. A class can decide to provide its own method instead of the one inserted from the aspect as methods defined in the class overrides the ones inserted by inter-type declaration.

Scala traits and Perl 6 roles also provide similar features, as they both give the programmer an opportunity to create interfaces with default method implementations.

### 6.2.3 Alternative syntax

The syntax for traits is under development, and since there is no de facto standard syntax on traits in statically typed languages it should be further discussed and tested. The syntax chosen for the Java preprocessor is a bit verbose, it should be refined if it is to be used in a production environment. In this section other alternatives to the trait syntax are presented.

#### Operations

Aliasing could be written in several ways, figure 6.1 shows some of the alternatives that was considered under the development of the Java preprocessor. The choice landed on the same as used in the traits implementation in C#. Exclusion could also be written in several ways, the different alternatives considered are listed in figure 6.2. Again the choice landed on the same syntax as used in the C# implementation.

When it comes to aliasing there are more ways of controlling the aliasing operation. One could implement a strict form of aliasing where only...
the method names are allowed to be changed, or as suggested in [14] one could alias the whole method signature. An even more advanced form of aliasing can use pattern-matching schemes to alias a set of methods. This pattern-matching scheme could for instance be used to make a whole set of methods synchronized by adding the synchronized modifier to the aliased signatures.

**Requirement syntax**

Requirements are described in a block structure, but they could also have been described in other ways, e.g. by marking method signatures inside the body of a trait with a keyword. As pointed out in the C# implementation of traits there is a possibility to reuse the keyword abstract as a marker for methods that are required. The use of abstract would work, but the meaning of an abstract method is somewhat wrong with respect to method requirements. An abstract method is often used to create a place-holder for a method that can be overridden later by a subclass, while a required method does not define any place-holder. A required method could also be marked by using Java 5 annotations like in listing 6.4 where we have the required methods `read` and `write`:

```java
@RequiredMethod int read();
@RequiredMethod void write(int b);
```

**6.2.4 Alternative requirement specification**

A big part of the preprocessor features is concerned with checking that a class is compatible with the traits it uses. That is the main concern of the preprocessor. Right now requirements are stated in a very simple way, they are just a list of method signatures written inside a requirement block. An
CHAPTER 6. DISCUSSION

Listing 6.5: Extended interfaces as requirements

```java
trait TRead extends Comparable, StreamProvider {
    // Require the methods from both
    // Comparable and StreamProvider
}
```

Listing 6.6: Class or Interface used as requirement

```java
trait TCompare {

    requires {
        Comparable;
    }

    public boolean greaterThan(Object obj) {
        return compareTo(obj) > 0;
    }
}
```

interesting new feature of traits is the possible use of classes to declare requirements.

Interfaces in Java are a list of method signatures in the same way as requirement blocks contain a list of method signatures. We will now show that it is possible to use a new trait syntax to pose requirements of a trait. Listing 6.5 shows an example where the keyword extends is used to apply requirements to a trait. Listing 6.6 contains the trait TCompare that would require methods contained in the Comparable interface.

6.2.5 Checking requirements

The Smalltalk implementation of traits does not contain the same kind of requirement checking as the preprocessor created. An interesting discussion is whether the preprocessor could be created without any requirements checking at all. What would be the effect of such an implementation?

The implementation of a preprocessor which does no requirements check-
ing would just do the insertion of methods and skip the phase containing checking of methods. By skipping the checking of methods phase the programmer would not be able to get the same detailed error messages caused by incompatible class and trait combinations, he would in fact get no error messages at all. All the error checking would be deferred to the final compilation of the flattened Java class. When the class is flattened most of the trait information is lost, and therefore a missing requirement would not be reported correctly as a missing requirement. The requirements specify methods that the trait methods can use, so the use of the required methods would possibly be reported as missing methods or missing symbols instead of missing trait requirements. On the other hand, by skipping the checks the preprocessing phase would be faster if speed of preprocessing is an important factor.

6.2.6 Alternative design

One idea is that traits could be implemented using the Java annotation framework. The Java 5 package includes a tool for working with annotations, which is called annotation processing tool or apt for short. Using the annotation processing tool the annotations of a class can be found and the contents of these annotations could be used to do some code transformation or code generation. The alternate design proposed here is that one can annotate a Java class with traits used, and by using the Java annotation tools be able to insert method bodies from traits. This would make use of annotations instead of with-blocks in the code, which might produce a cleaner code. However there would still be a need to define traits. Here annotations could be used as well. For instance a class could be annotated with a special marker annotation in order to signify that the class is actually a trait and not an ordinary class like in listing 6.7. Of course this does not give the opportunity to modify the syntax of the language, that is why this design was not implemented. It would however be interesting to see is if such a design could be implemented to introduce traits into Java.

6.3 Conclusion

Flaws of the single inheritance model are well known and language designer are trying to come up with ways of improving it. The increasing use of the word trait in programming language design is pointing towards a wider acceptance of traits as a unit of reuse. New languages are being made
Listing 6.7: Alternative trait design

```java
@Trait class A {
    @RequiredMethod int size();

    public boolean isEmpty() {
        return size() == 0;
    }
}
```

that incorporate traits or similar concepts. Examples of new languages with traits are as previously mentioned Scala and Perl 6. Sun is currently working on a programming language designed for high-performance computing called Fortress [28]. In Fortress traits are a part of the language as a unit containing methods. This shows that traits have come to stay. Although the traits units of these new languages resemble traits as first implemented in Squeak, they have made some adjustments to introduce traits into their type system.

There is still no standard way of implementing traits in a programming language and many attempts must be made to find the best. This thesis has made an attempt to insert traits into the Java programming language with the use of a preprocessor. The preprocessor has successfully proven that traits can be implemented in a given language by extending its syntax, and creating a translation from the extended syntax to the original syntax. It also contains type checking methods that ensure that a class is compatible with the traits applied to the class.
Chapter 7

Summary

In this thesis a preprocessor was created that inserts the concept of traits into the Java programming language. The preprocessor was created by extending the Java 5 grammar with rules for defining and applying traits. The extended grammar is used to produce an abstract syntax tree that is manipulated to do the flattening of classes and traits. The flattening is the process where method declarations are copied from the traits into the classes applying the traits. The created preprocessor can be used in order to increase the reuse of method bodies between several Java classes contained in separate class hierarchies. By using the preprocessor the programmer can create classes that are composed from multiple traits, and in that way get to reuse methods like in multiple inheritance.

An important feature of the implemented preprocessor in addition to its ability to increase code reuse, is the type-checking of requirements. This type-checking is responsible for making sure that a class and its applied traits can be flattened without problems. Before the trait flattening is done the preprocessor checks if the class actually is compatible with the applied traits by checking if every trait requirement is fulfilled. By having these requirement checks inside the preprocessor before a Java program is compiled we can report errors early when something is wrong with the the trait composition.

7.1 Further work

Further work includes further development of the preprocessor to implement missing features and to make it stable and reliable. When the preprocessor is stable it can be used to test traits effect on large software develop-
ment projects. A part of further development of the preprocessor should be to provide tool support for development with traits. IDEs should be extended with the knowledge of traits. Eclipse is an IDE that is used in many Java projects, if an eclipse plugin was created to support the editing of trait source files the adaption of traits in software development would be easier. A preview of such a plugin is provided in [29], where the plugin is a part of a larger project to support multiple views of a system.

After Sun made its Java development kit open-source an option to implement traits directly into the Java compiler opens up. We have shown that traits can be implemented as a source code transformer, but maybe traits should be included in a complete compiler for Java. When developing traits directly into the compiler it will be possible to implement runtime support for Java traits, including debugger and reflection api extension.

The trait syntax needs some extra work before it can be standardized. A larger community of programmers should discuss the syntax for traits and agree on how it should be defined. A need for proper modeling with traits is also present, and the UML notation could be extended to include a way of modeling traits.

In the evaluation of the preprocessor an design alternative was considered where annotations played a large role. The idea was to not extend the syntax of the language but instead use the annotation tools of Java 5 to implement traits. Further work could indeed be to see if such a design could be used to implement traits. If such an implementation is successful the adaption of traits could be made easier by software projects which already have started, due to the fact that annotations already are a part of the Java 5 language.
Appendix A

Using the prototype

A.1 Getting the prototype

The prototype source as well as the pre-compiled binary can be downloaded from the url http://kjetilos.ifi.uio.no/master/.

A.2 Compiling the prototype

The build script is an apache ant script which is located at the root of the project. Before you compile the prototype be sure to have downloaded JavaCC and setup an environment variable called JAVA_HOME to point to the root of the JavaCC installation. JavaCC is needed to produce the parser used in the prototype.

To compile and create the jar file just issue the command below at the project root, where the build.xml file is located:

> ant dist

This will produce the parser by using JavaCC and compile all the sources from the src directory. The compiled class files will be put into the classes directory, and packed into a jar file which is put into the dist directory. If wanted the source documentation (Javadoc) can be created by issuing the command:

> ant javadoc

The documentation will then be available inside the doc directory.
A.3 Running the prototype

The prototype is packaged into a jar file called `jt.jar` located in the `dist` directory and is used from command line like this:

Usage: `java -jar jt.jar <options> <source files>`
where possible options include:

- `-d <directory>` Specify where to place generated java files
- `-v` Verbose information, prints AST
- `-t` Test, print output code on screen instead of generating java files
- `-a` Annotate methods inserted from traits
Appendix B

Grammar

The grammar below uses the same BNF-style conventions as in the Java language specification,

- \([x]\) denotes zero or one occurrences of \(x\).
- \(\{x\}\) denotes zero or more occurrences of \(x\).
- \(x \mid y\) means one of either \(x\) or \(y\).

B.1 Extension of the Java syntax

TypeDeclaration :
   ...
   TraitDeclaration

ClassBodyDeclaration :
   ...
   WithTrait

InterfaceBodyDeclaration :
   ...
   WithTrait

TraitDeclaration :
   trait Identifier [implements TypeList] TraitBody

TraitBody :

111
\[
\{ \text{TraitRequirements} \} \{\text{TraitMemberDecl}\} \}
\]

**TraitRequirements**: requires \{ \{\text{TraitMethodRequirement}\} \}

**TraitMethodRequirement**: \{\text{Modifier}\} \text{Type} \text{Identifier} \text{InterfaceMethodDeclaratorRest}

**TraitMemberDecl**: WithTrait
\text{TraitMethodDecl}

**TraitMethodDecl**: \{\text{Modifier}\} \text{Type} \text{Identifier} \text{MethodDeclaratorRest}

**WithTrait**: with \{ \{\text{WithDeclaration} ;\} \}

**WithDeclaration**: \text{Identifier} \{\text{TraitOperations}\}

**TraitOperations**: \{ \{\text{TraitOperation} ;\} \}

**TraitOperation**: Aliasing
Exclusion

**Aliasing**: \text{MethodSignature} \rightarrow \text{MethodSignature}

**Exclusion**: ^ \text{MethodSignature}

**MethodSignature**: \{\text{Modifier}\} \text{Type} \text{Identifier} \text{MethodSignatureRest}

**MethodSignatureRest**: \text{FormalParameters} \{\text{throws} \text{QualifiedIdentifierList}\}
Bibliography


[4] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.


[7] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart, and Nathanael Schärli. Adding Traits to (statically typed) languages. Technical Re-


BIBLIOGRAPHY


