HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

Institute for
Internet Technologies
and Applications

Operator overloading for Java

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2011

Author(s):	Sandro Brändli
Advisor:	Prof. Dr. Josef M. Joller
External Co-Examiner:	Matthias Lips
Internal Co-Examiner:	Prof. Dr. Andreas Rinkel

Erklärung (German)

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.

Ort, Datum:

Name, Unterschrift:

Abstract

Introduction

This thesis analyzes the possibilities to extend the Java programming language. This is done by implementing operator overloading as an example of what is possible. An important goal of this implementation is to be easy to use for the developer. That means among other things that it should not require a special compiler or extra steps in the build process and it should work on a standard Java Runtime Environment (JRE).

Approach

There are several possibilities to extend the Java syntax. The obvious ones are:

1. Patch the compiler
2. Preprocess the source code
3. Develop a compiler with the desired features

The problem with patching the compiler or developing a new one is that it would require the developer to use a different compiler and thus requires extra effort from him. Preprocessing the code complicates the build process by introducing an additional step.

In this thesis the approach is using a Java 6 pluggable annotation processor to modify the Abstract Syntax Tree (AST) before the bytecode is generated. Annotations and annotation processors where introduced in Java 5, as tools like XDoclet showed the need for metadata in the code especially when developing Java EE applications. Annotation processors are used to generate source code or configuration files out of meta data (annotations) in the code.

Annotation processing in Java 5 was not integrated into the compiler, it required separate tools and an extra steps in the build process. With Java 6 and the introduction of pluggable annotation processors it became part of the compiling process. So an annotation processor can be registered via the Service Provider Interface (SPI) of Java and is then used by the compiler.

The transformation can be done by casting the objects passed to the annotation processor to their underlying classes and then modifying the AST. To stay independent of different compiler's AST implementations an abstraction layer is used.

Results

Operator overloading has been implemented and works with Eclipse, Netbeans and plain Javac. Also builds which use Ant or Maven both using Javac are working. The implementation is compiler independent and extendable. However it is not ready for productive use yet, because it still has some bugs. To extend it, a new transformations can be registrated with the SPI of Java.

Management Summary

1 Initial Status

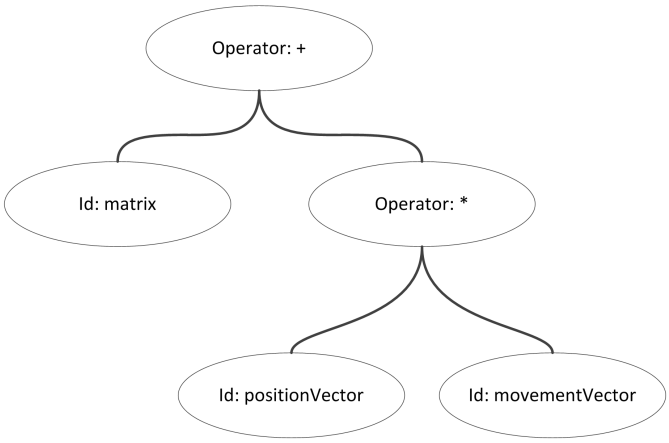

Figure 1: Simplified AST before transformation

Java does, in contrast to many other programming languages, not support operator overloading. Overloading an operator means to define it on a data type. An operator is a symbol or function representing a mathematical operation. It is likely, that the intent was to keep the language simple. Many features from C++ where not included in Java to keep Java simple.

The debate if Java should support operator overloading is almost as old as the language itself. The main advantage of having operator overloading in Java would be extensibility. The developer would be able to use operators not just for primitives or the +-operator for strings, but also for his own datatypes. It would also make it easier for Oracle to define operators for classes like Date, BigInteger and BigDecimal, which would greatly enhance their usability.

In this thesis operator overloading will be implemented as an example of how Java can be extended. It is an important goal of the implementation to be easy to use for the developer and not to introduce extra steps in the compiling process. Also the set of features should be easy to extend. The implementation should not only work with plain Javac, but also within Eclipse, Netbeans, Ant or Maven.

2 Approach

Concerning project management, the approach was very lightweight, as this was an explorative project and only one person was involved. A complex process would have only slowed things down, without any real benefits. Still the process was iterative and every iteration was consisting of phases. Every iteration started with defining the goals that should be achieved at the end. And is ended with discussing the results and defining the goals for the next iteration. An iteration typically consists of investigation, prototyping, programming and documenting.

The technical approach is to use some new technologies introduced in Java 6 like pluggable annotation processing and dynamic attach. And combine them with existing ideas to modify the AST. The annotation processor is used as a hook to the compiling process. Dynamic attach in conjunction with Javassist is used to change the parser of Eclipse to use the annotation processor. To stay compiler independet an abstraction of the AST called Java Unified Abstract Syntax Tree (JUAST) is used.

3 Results

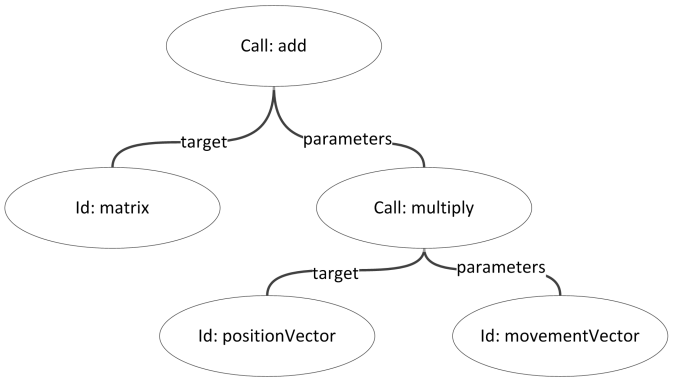

Figure 2: Simplified AST after transformation

An annotation processor has been implemented, which is able to transform the AST and allows the implementation of new features for the Java programming language by simply implementing a visitor that transforms the AST. It works with Eclipse, Netbeans, Javac, Ant and Maven. Because of problems related to classloading and lack of time, for Eclipse the transformation is implemented as a plugin. The implemenatation still has some bugs, but it shows what is possible.

Table of Contents

Chapter 1

Scope of work (German)

1.1 Organisation

Autor: Sandro Brändli, Student Abteilung für Informatik HSR Betreuer: Prof. Dr. Josef M. Joller, Abteilung für Informatik HSR Gegenleser: Prof. Dr. Andreas Rinkel, Abteilung für Informatik HSR Experte: Matthias Lips, MediData AG

1.2 Ausgangslage, Problembeschreibung

An mehreren Tagungen und Vorträgen wurde immer wieder über das Thema "Operator Overloading (OO) in Java" diskutiert und debattiert. Es gibt Gründe, die für ein Operator Overloading sprechen, wie etwa in den Vorträgen von Guy Steel erläutert wurde. Traditionalisten/Puristen sehen im Operator Overloading eher eine mögliche Quelle für problematischen, sprich unzuverlässigen Programmcode.

In der Vergangenheit sind verschiedene Ansätze für das OO publiziert. Diese basieren in der Regel auf älteren JDK bzw. Javac Versionen. Die Einführung von Annotations in Java 5 zeigen einen neuen Weg auf:

- Annotation Processors

Damit hat man die Möglichkiet, speziell ab Version 6 auf den AST zuzugreifen und Manipulationen vorzunehmen.

1.3 Aufgabenstellung

Es ist eine Lösung für das Operator Overloading zu finden (zu entwickeln, evtl. bestehende Ansätze zu erweitern), welche sowohl in standalone Javac als auch in den gängigen Entwicklungstools (Eclipse, evtl. NetBeans) funktioniert. Dabei ist der Annotation Processor Ansatz der naheliegende. Es ist aber dem Studenten / dem Team überlassen, welcher Lösungsansatz verfolgt wird. Das Ergebnis steht im Vordergrund!

1.4 Durchführung

Die Durchführung richtet sich nach den geltenden Durchführungsbestimmungen für Bachelorarbeiten.

1.5 Dokumentation und Abgabe

Wegen der beabsichtigten Verwendung der Ergebnisse in weiteren Projekten wird auf Vollständigkeit sowie (sprachliche und grafische) Qualität der Dokumentation erhöhter Wert gelegt.

Die Dokumentation zur Projektplanung und -verfolgung ist gemäss den Richtlinien der Abteilung Informatik anzufertigen. Die Detailanforderungen an die Dokumentation der Recherche- und Entwicklungsergebnisse werden entsprechend dem konkreten Arbeitsplan festgelegt. Die Dokumentation ist vollständig auf CD in drei Exemplaren abzugeben.

Neben der Dokumentation sind abzugeben:

- ein Poster zur Präsentation der Arbeit
- alle zum Nachvollziehen der Arbeit notwendigen Ergebnisse und Daten (Quellcode, Buildskripte, Testcode, Testdaten usw.)
- Material für eine Abschlusspräsentation (ca. 20')

Chapter 2

Introduction

As the title of this Thesis states, it is about implementing operator overloading for Java. Therefore we first need to define what operator overloading means. We will start with the definition of what an operator is:

„An operator is a symbol or function representing a mathematical operation"[1]

Java only defines arithmetic operators for primitive data types and the operator + for Strings. Overloading an operator means defining it on a data type. On the next page you will see an example of how operator overloading works in C++. Java supports the operators listed in Table 2.1.

Operator	Description
`x++, x--`	Postincrement, Postdecrement
`++x, --x`	Preincrement, Predecrement
`>, <`	Greater than, Less than
`>=, <=`	Greater than or equals, Less than or equals
`==`	Equal
`!=`	Not equal
`&, |, ^`	Logical, bitwise AND, OR, XOR
`!`	Logical negation
`&&, ||`	Conditional AND, OR
`~`	Complement operator
`obj instanceof Class`	Evaluates if `obj` is of type `Class`
`=`	Assignment Operator

Table 2.1: Operators supported by Java [AGH05, Chapter 9.2]

To keep the sourcecode readable, not all of this operators should be overloadable. For some of them I can not see an overload that would make sense.

[1] `http://wordnetweb.princeton.edu/perl/webwn?s=operator`

```
1  #include <iostream>
2
3  class myclass
4  {
5    int sub1, sub2;
6
7  public:
8    myclass(){}
9    myclass(int x, int y){sub1=x;sub2=y;}
10   myclass operator +(myclass);
11   void show(){std::cout<<sub1<<", "<<sub2;}
12 };
13
14 myclass myclass::operator +(myclass ob)
15 {
16   myclass temp;
17   temp.sub1=sub1 + ob.sub1;
18   temp.sub2=sub2 + ob.sub2;
19   return temp;
20 }
21
22 int main()
23 {
24   myclass ob1(10,90);
25   myclass ob2(90,15);
26   ob1=ob1+ob2;
27   ob1.show();
28   return 0;
29 }
30 // Output: 100, 105
```

Listing 2.1: Operator overloading in C++

Operators in C++ are overloaded by implementing special methods. The rest is performed by the compiler, meaning that it is just a compiler feature.

2.1 Motivation

Java has been around for more then ten years without support for operator overloading. This rises the question why we should need it. There have been many arguments about operator overloading. The syntax of Java is largely derived from C++, but some features were suppressed to simplify the language [GM96, Chapter 2.2].

The supporters say, that the user should be able make his own data types behave like the built in ones [Ste99, p. 233].

The opponents counter, that code might become harder to read and maintain with heavy use of operator overloading, because it allows programmers to give operators completely different semantics depending on the types of their operands.

Still many popular programming languages like C++, Ruby and Python offer operator overloading and also some newer ones like C#, Scala and Groovy [Str00, FM08, VR03, HWG03, OO04, KGK+07].

2.2 Thesis structure

The rest of the thesis is structured as follows: Chapter 3 presents, what has already been done in finding ways to extend the Java syntax. It also contains the decision of how operator overloading will be implemented from a technical point of view.

Chapter 4 shows how other languages have implemented operator overloading. It gives a short overview about what features it has in these languages and how they are used. It ends with a description of how it should be done in this thesis.

The chapter 5 describes the passes of the compiling process and how to do annotation processing. It includes an explanation of how to debug an annotation processor.

After the chapter about Javac, chapter 6 about IDE-support follows, it explains how the IDE-support works.

The thesis continues with chapter 7, which looks at the thesis from a project management point of view. It describes the iterations and phases of the project.

At the end, the results are summarized and an outlook is given in chapter 8.

Chapter 3

Related Work

There have been other attempts to implement operator overloading in Java or to otherwise extend it. This chapter gives a short overview about the approaches taken, and how they might be usable for this thesis. Luckily none of them completely solves the problem, as this would render this thesis useless.

3.1 Preprocessing code

JFront[1] is an implementation that works by transforming the source code before compiling. This approach is interesting, as it would theoretically allow everything. With preprocessing the source it would even be possible to transform code from any programming language to Java. As Java is touring complete, every program in any programming language could be translated to Java. Long story short, there are no limits when using a preprocessor to implement new features.

Still this approach has some disadvantages. The flexibility of preprocessing comes at the price of an extra step in the compiling process. This means that it requires extra effort from the developer and introduces a source of very subtle errors, like bugs in the preprocessor. Preprocessing also makes it difficult to implement IDE-Support and almost certainly require a plugin.

Because of the problems and disadvantages mentioned above, this is not a viable solution. IDE-support is an important requirement and it would be to difficult to implement. Also requiring the developer to configure an extra step in the build process and to install a plugin in his IDE is to much effort to remain usable.

3.2 Patching the compiler

Another approach is to patch the compiler.[2] The linked solution implements operator overloading, but does not have IDE-support. This is almost the same as using a preprocessor. The difference is that it does not require an extra step in the compiling process and thus not complicates the build process.

[1] `http://www.gginc.biz/jfront/`
[2] `http://www.java-forum.org/540038-post1.html` (German)

The problem with this approach is, that it requires the developer to use a different compiler. It is difficult for a developer to be sure, that a patched compiler is still trustworthy. Using a patched compiler may also violate policies in a company. Besides there might be copyright problems when patching the compiler. As Eclipse uses its own compiler and parser, it would be necessary to also patch them.

The main reason for not using this approach are the copyright concerns. It would require a lawyer to determine, what changes are allowed and under what conditions. It would also be to much work to patch the compiler of Eclipse and write a plugin to support it in the editor.

3.3 lambdaj

The listing below shows what is possible with lambdaj[3]. It is taken from lambdaj's website.

```
1 List<Person> personInFamily = asList(
2     new Person("Domenico"),
3     new Person("Mario"),
4     new Person("Irma")
5 );
6 forEach(personInFamily).setLastName("Fusco");
7 List<Person> sortedPersons = sort(persons, on(Person.class).getAge());
```

Listing 3.1: Example of using lambdaj

Lambdaj strives to facilitate working with collections by eliminating loops and manipulate them in a pseudo-functional and statically typed way. It first seems, that lambdaj needs to modify Java in some way to achieve its functionality. Further investigation showed, that this is not the case. Everything lambdaj does can be achieved with plain Java code. Lambdaj works by using dynamic proxies and thread locals. So lambdaj does not show a new way of modifying the Java syntax, it is just an internal Domain Specific Language (DSL).

Investigation showed, that with lambdaj's approach it is not possible to do operator overloading, as it does not allow to change Java's syntax.

3.4 AST-modifying annotation processor

JUAST[4] works by directly transforming the syntax tree during the compile process. JUAST already has a basic demo that does operator overloading in special cases. It also provides an abstraction layer for the AST-representations of different compilers. So it is a good start. What is missing is IDE-support.

[3] http://code.google.com/p/lambdaj/
[4] https://bitbucket.org/amelentev/juast/wiki/Demo

Another project which works the same way is Lombok[5]. It has IDE-support, but does not implement operator overloading and does not have an AST-abstraction. However it seems very stable and the knowledge in which way you can support Eclipse is very useful.

This is the way to go. It does not require any extra steps or a modified compiler. And by using JUAST the solution is compiler independent. Annotation processing is part of the Java standard and thus can be used on any compliant compiler. Also IDE-support should not be to hard to implement, since the code of the annotation processor is executed by the same Virtual Machine (VM) as the IDE. One big constraint of using an annotation processor is, that the sourcecode must be parsable by the compiler, because the annotation processor works on the AST provided by the compiler. For operator overloading, however this is not a problem, due to the fact that the code is parsed before type resolution is done. As long as the compiler does not know, the type of an expression, it can not complain about using operators, since it might be of a primitive type.

[5] `http://projectlombok.org/`

Chapter 4

Analysis

This chapter describes how operator overloading is implemented in other languages. Then the decision is made, how it will be implemented.

4.1 Potential misuse

Code could become hard to read if operators like = or == are overloaded. For some operator there is no meaningful overload, for example for the instanceof operator.

4.2 Current implementations

This section shows in which form other languages implement operator overloading. It gives a short overview about the features and how to use them, this is done by using sample code.

4.2.1 C++

Chapter 2 contains sample code for C++, so it is not repeated here. In C++ an operator is overloaded by implementing a method with a special name. As C++ also supports extension methods, it is possible to overload operators even on classes of the standard library or other classes not under control of the developer.

C++ does not allow to change the precedence or associativity of an operator, which is a good thing, as this would only make the code less readable. If a developer needs different associativity or precedence, he can use parentheses to achieve it. Because of the special method names of overloaded operators, they look a bit out of place.

4.2.2 Scala

Listing 4.1 shows an example[1] of operator overloading for a class representing a complex number.

[1] `http://www.scala-lang.org/node/224`

```
1  object complexOps extends Application {
2      class Complex(val re: Double, val im: Double) {
3          def + (that: Complex) =
4              new Complex(re + that.re, im + that.im)
5          def - (that: Complex) =
6              new Complex(re - that.re, im - that.im)
7          def * (that: Complex) =
8              new Complex(re * that.re - im * that.im,
9                re * that.im + im * that.re)
10         def / (that: Complex) = {
11         val denom = that.re * that.re + that.im * that.im
12         new Complex((re * that.re + im * that.im) / denom,
13                                  (im * that.re - re * that.im) / denom)
14 }
15     override def toString =
16         re + (if (im < 0) "-" + (-im) else "+" + im) + "*i"
17     }
18     val x = new Complex(2, 1); val y = new Complex(1, 3)
19     println(x + y)
20 }
```

Listing 4.1: Operator overloading in Scala

It seems like Scala implements operator overloading, actually this is not the case. There is no chapter about operator overloading in the Scala Language Specification ([OO04]). Scala just has less constraints on how to name a method, than for instance Java. As shown in the example, a method in Scala can be named „*". With the property that any method which takes a single parameter can be used as an infix operator, operator overloading is possible without explicitly implementing it.

Scala is a very flexible language and thus also allows to change associativity and precedence of operators.

4.2.3 Groovy

Groovy does operator overloading in a way similar to C++. What is different are the method names. In Groovy the methods have normal names as shown in listing 4.2. The set of operators that can be overloaded is limited to comparison operators, arithmetic operators and the array access operator.

```
1  class JukeBox {
2      def songs
3
4      JukeBox(){
5          songs = []
6      }
7
8      def plus(song){
9          this.songs << song
10     }
11
12     def minus(song){
13         def val = this.songs.lastIndexOf(song)
14         this.songs.remove(val)
15     }
16
17     def printPlayList(){
18         songs.each{ song -> println "${song.getTitle()}" }
19     }
20 }
```

Listing 4.2: Operator overloading in Groovy

The way Groovy implements operator overloading seems very Java-like. With normal methods as operators it is possible to implement the overloading using an annotation processor, as the sourcecode remains parsable. Groovy does not allow to change associativity or precedence.

4.3 What to implement

Operator overloading should be implemented in a similar way as it works in groovy. If the operators are represented as normal methods, the code of a class with overloaded operators remains normal Java. It can therefore even be compiled without operator overloading enabled. Operator overloading only needs to be enabled for classes, that really use it.

The set of operators is limited to the arithmetic operators +, -, *, /, ^, since the overloading of operators like the assignment operator would cause a lot of confusion. Also changing precedence or associativity is not allowed. It would be confusing to use, difficult to implement and not really bring any advantage.

4.4 How to implement it

As already stated in chapter 3 the best way to implement operator overloading seems to be using an annotation processor which modifies the AST. To stay compiler independet, JUAST can be used. Eclipse can be supported by changing the parser to transform the AST after parsing using the Dynamic Attach API and a bytecode manipulation library.

Chapter 5

The Java Compiler

5.1 Passes of the compiling process

The compiling process consists of the passes shown in Figure 5.1

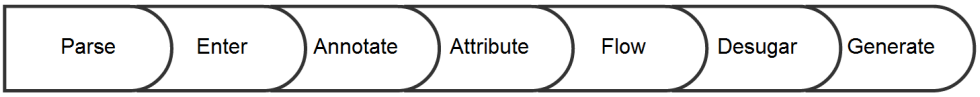

Figure 5.1: Passes of the compiling process

parse Reads a set of *.java source Files and maps the resulting token sequence into AST-Nodes.

enter Enters symbols for the definitions into the symbol table.

process annotations If requested, processes annotations found in the specified compilation units.

attribute Attributes the syntax trees. This step includes name resolution, type checking and constant folding.

flow Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.

desugar Rewrites the AST and translates away some syntactic sugar.

generate Generates Source Files or Class Files.

The above description of the compiling process is taken from [EK08]. For implementing operator overloading the ”process annotations”-phase is important. It is described in detail in the next section.

5.2 Annotation processing

Java 6 has introduced a new Java Specification Request (JSR) called JSR 269[1], which is the Pluggable Annotation Processing API. With this API, it is possible for the developers

[1] `http://www.jcp.org/en/jsr/detail?id=269`

to write an annotation processor which can be plugged-in to the compiling process to operate on the set of annotations that appear in a Source File. The idea is typically used to create new source file or configuration files to be used in conjunction with the original code.

In this thesis the parameters given to the annotation processor are casted to their underlying implementation. With this technique the AST becomes accessible and also modifiable.

5.2.1 Debugging

It is difficult to debug an annotation processor, since it is executed during the compiling process. Therefore, it is needed to start the compiler in debug mode to render it possible to debug the annotation processor. The compiler can be launched in debug mode by starting the underlying launcher in debug mode.

The easiest way to accomplish this is by using Ant or Maven and start them in debug mode. As the compiler is running in the same process as Ant or Maven, it will also be in debug mode. Maven can be set to debug mode by setting the environment variable MAVEN_OPTS.

```
1 set MAVEN_OPTS=-Xdebug -Xnoagent -Djava.compiler=NONE
2 -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000
```

Listing 5.1: Debugging Maven

5.2.2 Manipulating the AST

To manipulate the AST, a good way is to use JUAST. It provides an abstraction of the AST and is therefore compiler independent.

JUAST provides an API using visitors to manipulate the AST. The interface `juast.visit.Visitor` can be implemented to manipulate the AST. It contains enter and leave methods for the nodes of the AST. Alternatively the class `juast.visit.DefaultVisitor` can be extended, it contains default implementations of every method of the interface.

5.2.3 Service Provider Interface

The SPI is a standardized way to provide services in a Java application. A service is registered by putting its fully qualified classname on a new line in a file with the name of the interface implemented by the service. This file must be put under the path „META_INF/services” in the published jar file.

Chapter 6

IDE Support

6.1 Netbeans

To use operator overloading in Netbeans, annotation processing has to be enabled. This can be done by enabling „annotation processing" and „annotation processing in editor" under the compiler settings of the project. No additional steps are required.

Implementing operator overloading for Netbeans did not require any extra effort, because Netbeans internally uses Javac. Thus implementing operator overloading for Javac also means implementing it for Netbeans.

6.2 Eclipse

Using operator overloading in Eclipse requires to register and enable the annotation processor. The annotation processor can be enabled by enabling „annotation processing" and „annotation processing in editor" under the compiler settings of the project. To register an annotation processor, the factory path under the annotation processing settings has to be set.

Eclipse uses its own parser and compiler. As mentioned in Chapter 4, JUAST is used to handle the different AST implementations. However, when using Eclipse, there remains a problem. Eclipse not only compiles the sourcecode on-save, but also parses it when typing and reports errors. The problem here is, that the parser does not run annotation processors and therefore operates on a non-transformed AST. This results in the parser reporting errors because of unsupported operators.

The behavior of Eclipse seems legitimate as annotation processors were originally not intended to modify the AST, so it should not be needed to run the annotation processors for parsing the sourcecode. Fortunately, there is a solution for this problem. The code to run the annotation processor can be injected into the parser during runtime, using the Dynamic Attach API[1] introduced in Java 6.

[1] `http://download.oracle.com/javase/6/docs/technotes/guides/attach/index.html`

6.2.1 The Attach API

This section shortly describes the Attach API, how it is actually used to patch the parser of Eclipse is explained in Section 6.2.2. The attach API provides a way to modify and reload classes in an application by using an agent. Since Java 6 it is not needed to declare the agent when starting an application, it can also be attached later.

A dynamically attached agent is used to patch the parser of Eclipse at runtime. To do this, the annotation processor, which is called by the compiler of Eclipse, creates the agent and loads it into the current VM. The agent then patches the parser.

Patching the parser is done by modifying it with Javassist. With its instance of a `java.lang.instrument.Instrumentation`, the agent reloads the parser class. It achieves this by using the following methods of the Instrumentation:

addTransformer with this method a transformer can be added, which then transforms classes when loaded or unloaded.

retransformClasses causes transformation of given classes, even if they where already loaded.

6.2.1.1 Application lifecycle with an agent

Normally the first method invoked when starting a Java program is the main method. When working with agent, this might not be the case. When starting a Java application with the -javaagent option and pointing it to a jar file, that contains an agent and has an appropriate manifest, the premain method of the agent is executed before the main method of the application. This behavior is guaranteed. Since Java 6 there might also be a agentmain method which can be executed before or after the main method. So we have the following methods when using an agent:

premain is invoked first prior to running the main method and used for loading the agent statically at startup.

main this one should be familiar. It is always invoked after premain.

agentmain can be invoked at any time and is the hookpoint when loading the javaagent dynamically and attaching it to a running process.

The methods premain and agentmain have, except for the name, the same method signature. They are throwing Exception, are public, static and take two arguments.

6.2.2 Hotpatching with an agent

6.2.2.1 Creating the agent jar

Within the context of this thesis the class `ch.gizmo.overload.agent.AgentInjector` has been implemented. It is a helper class for injecting agents into its own VM. As the Dynamic Attach API requires the path to a jar file to load an agent, the `AgentInjector` is able to create a compliant jar.

With the method `loadAgent(Class<?> agentClass)` it is possible to directly use a class as an agent. The `AgentInjector` creates a jar file containing the given class and all its dependencies with an adequate manifest and loads the agent into the current VM.

6.2.2.2 Injecting the agent

An agent class needs to have a method with the signature `public static void agentmain(String, java.lang.instrument.Instrumentation)`. The agent has to be deployed in a jar file with a manifest like Listing 6.1.

```
1 Manifest-Version: 1.0
2 Agent-Class: agent.AgentTest
3 Can-Redefine-Classes: true
4 Can-Retransform-Classes: true
```

Listing 6.1: Manifest of agent jar

The method `com.sun.tools.attach.VirtualMachine.loadAgent(String)` loads the agent, the parameter is the absolute path to the jar containing the agent. The class `VirtualMachine` is part of Sun's Java Development Kit (JDK), so the JDK is needed to inject an agent.

It is also possible to inject an agent into the own VM. To achieve this, a reference to the VM has to be retrieved. The method in Listing 6.3 accomplishes this.

```
1  public VirtualMachine getCurrentVm() throws Exception {
2      RuntimeMXBean mxbean = ManagementFactory.getRuntimeMXBean();
3      Field jvmField = mxbean.getClass().getDeclaredField("jvm");
4
5      jvmField.setAccessible(true);
6      VMManagement mgmt = (VMManagement) jvmField.get(mxbean);
7      Method method = mgmt.getClass().getDeclaredMethod("getProcessId");
8      method.setAccessible(true);
9      Integer processId = (Integer) method.invoke(mgmt);
10
11     return VirtualMachine.attach("" + processId);
12 }
```

Listing 6.2: Get reference to own VM

6.2.3 Debugging an Annotation Processor

The most convenient way to debug an annotation processor within Eclipse is by exporting it as an Eclipse plugin. Launching the plugin in debug mode causes a new instance of Eclipse to launch in debug mode. The first instance can then be used to debug the second one. The following plugin.xml is needed for a plugin to export an annotation processor.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.4"?>
3  <plugin>
4    <extension
5    point="org.eclipse.jdt.apt.core.annotationProcessorFactory">
6         <java6processors enableDefault="true">
7                  <java6processor
8                 class="ch.gizmo.overload.processor.RootProcessor"/>
9           </java6processors>
10   </extension>
11 </plugin>
```

Listing 6.3: Exporting an annotation processor via an Eclipse plugin

Chapter 7

Project Management

This project is not based on a strict heavy-weight process. In an explorative one-man project, complex processes would only slow things down. Nevertheless, this project was not unorganized. The organization is iterative and agile. Every iteration started with a meeting to define the goals. After accomplishing the goals, a new meeting to discuss the iteration and define goals for the next iteration was arranged. An iteration typically consisted of investigation, prototyping, programming and documenting.

The project consisted of the following iterations:

7.1 Investigating

This was the shortest phase, it took one week, but of course there was some investigation during the whole project. This phase consisted of finding out, how operator overloading works in other languages and how it should be implemented for this thesis.

7.2 Exploring

With a duration of about two weeks, this was also a rather short iteration. In this phase I researched about related projects. The results are documented in chapter 3.

7.3 Prototyping

This iteration was dedicated to experimenting with all sorts of technologies that are needed for this thesis, with the goal of making a prototype that does operator overloading. It was three weeks long. In this iteration also prototypes of what is possible with agents and the Instrumentation API where made.

7.4 IDE-Support

It was the longest and hardest iteration of this project. It was underestimated a lot, what effort it needs to make the overloading, therefore this iteration was eight weeks long.

7.5 Finishing

The last two weeks were dedicated to finishing tasks and make everything ready for delivery.

Chapter 8

Conclusion

8.1 Experience report (German)

Da ich das Thema selbst vorgeschlagen habe, fand ich es besonders interessant. Ich konnte viele Technologien ausprobieren, mit welchen ich zuvor noch nicht gearbeitet hatte. Einige davon, wie zum Beispiel Dynamic Attach oder Pluggable Annotation Processing wurden erst in Java 6 eingeführt, waren also noch sehr neu.

Es war spannend, die Möglichkeiten der oben genannten Technologien bis an die Grenzen auszureizen um Dinge möglich zu machen, welche in der Sprache nicht vorgesehen waren. Ausserdem habe ich viel dabei gelernt.

Weil es sich um ein Thema handelte, mit dem sich bisher wenige beschäftigt hatten und neue, relativ unbekannte Technologien involviert waren, liessen sich nur wenige Informationen finden. Vieles war auch sehr spärlich oder gar nicht dokumentiert. Somit brauchte ich viel Zeit um Lösungen zu erarbeiten.

Den Code für Eclipse zu implementieren benötigte besonders viel Zeit. Dies war vor allem der Fall, weil Eclipse es einem nicht erlaubt, auf Klassen des JDKs zuzugreifen. Eine weitere Schwierigkeit war es, den Parser von Eclipse zu patchen, damit nach dem Parsen die AST-Transformation durchgeführt wird. Andererseits war es auch spannend, zu experimentieren, wie zur Laufzeit Code eingeschleust werden kann.

Auch wenn die Integration in Eclipse interessant und lehrreich war, würde ich dies beim nächstem Mal weglassen. Es hat mich viel zuviel Zeit gekostet und daran gehindert, neben dem Operator Overloading weitere Transformationen zu implementieren und alle gesetzten Ziele zu erreichen.

8.2 Results

8.2.1 Annotation processor

An annotation processor that implements operator overloading and works with Netbeans and Javac build, including Maven and Ant. It was implemented by extending JUAST. It is not ready for production use, since it has some bugs. Nevertheless it demonstrates what is possible with annotation processors.

8.2.2 Eclipse plugin

For Eclipse operator overloading is implemented as a plugin that provides an annotation provider. The original idea was to convert the plugin to a plain annotation provider that also works with Netbeans. This turned out to be more difficult than expected because classloading works different in annotation processors than in plugins.

8.2.3 AgentInjector

AgentInjector is a utility class, which does a lot of useful things regarding agents. It can be used to generate a agent jar file containing all dependencies and necessary manifest entries. It is also able to inject agents in its own VM.

8.2.4 Injector

The Injector is a small Swing application, that can inject agents into any running Java applications. It was mainly created for testing purposes and proved useful.

8.2.5 Agent

An agent has been developed, it patches the Eclipse parser to do operator overloading. It can be injected into a running Eclipse instance using the Injector.

8.3 Outlook

The results of this thesis lays the base to implement other features like extension methods. Although it needs some bugfixing and refactoring first.

For the future I suggest to drop support for other compilers than Javac and not to focus on IDE-Support. It should be used to experiment with new features. After the experiences I made, I do not think it is possible to make AST-transforming annotation processors production ready with the current APIs of Java. There where many non public APIs involved in realizing operator overloading.

Bibliography

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Pearson, 4th edition, 2005.

[EK08] David Erni and Adrian Kuhn. The Hacker's Guide to Javac. `http://scg.unibe.ch/archive/projects/Erni08b.pdf`, 2008.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, first edition, 2008.

[GM96] James Gosling and Henry Mcgilton. The Java language environment: A white paper. Technical report, Sun Microsystems, 1996. `http://java.sun.com/docs/white/langenv/`.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[KGK+07] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[OO04] Martin Odersky and Others. The Scala Language Specification. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2004.

[Ste99] Guy L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999. `http://dx.doi.org/10.1023/A:1010085415024`.

[Str00] Bjarne Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, 3rd edition, 2000.

[VR03] Guido Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., 2003.

Appendix A

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
DSL	Domain Specific Language
JDK	Java Development Kit
JRE	Java Runtime Environment
JSR	Java Specification Request
JUAST	Java Unified Abstract Syntax Tree
SPI	Service Provider Interface
VM	Virtual Machine

Appendix B

List of Figures

Appendix C

List of Tables

Appendix D

List of Listings