UNIVERSITY OF OSLO
Department of Informatics

Compiling Package Templates

Kenneth Solbø Andersen

Spring 2013

Compiling Package Templates

Kenneth Solbø Andersen

Spring 2013

Abstract

In Java, the package concept is a means for modularization of the code, in the same way as e.g namespaces are for some other languages. However, the flexibility of packages for reuse of code is not very good. Introducing *Package Templates* is a proposal to improve this situation, by moving the package one step towards a generic concept.

The JPT language extends Java with the Package Template concept, and the JPT compiler has been developed alongside the development of the PT concept and the JPT language. This compiler is a necessary tool when evaluating the usefullness of Package Templates for large scale programming.

This thesis describes the JPT language through explanations of the concepts and through code samples showing how these concepts are applied. It also discusses the design and implementation of the JPT compiler and some of the technologies that are used for building this compiler, and finally, it documents some of the recent changes that have been made to the JPT compiler.

Acknowledgements

I would like to thank those who have helped me with this thesis:

- My supervisor, Stein Krogdahl, for providing very helpful feedback and guidance during my work on this thesis, and for helping me improve my writing skills.
- Eyvind W. Axelsen for his quick responses to every PT question I could throw at him.
- Eivind Gard Lund for helping me get started with working on the JPT compiler.

Contents

List of Figures

Chapter 1

Introduction

This thesis is the result of my work on a compiler for a language called JPT, which is short for "Java with Package Templates". Package Templates, or PT for short, is a mechanism for writing and combining separate modules in object oriented languages. It is not a language in itself, but rather a set of concepts and ideas that must be fitted to each object oriented language, usually by extending the language with new syntax to describe these concepts.

The main features of PT are:

- That modules can contain several classes.
- That modules can be fully type checked as separate entities.
- That modules (or rather, all their classes) are combined with the rest of the program at compile time.
- That the classes of a module can be adapted (e.g by renaming and by making additions) during the combination with the rest of the program.

The modules of PT are called "package templates", or often simply "templates".

The ideas of PT, in something like the current form, first appeared in the article "Exploring the use of Package Templates for flexible re-use of Collections of related Classes" [11] by Stein Krogdahl, Birger Møller-Pedersen and Fredrik Sørensen. Since then, several articles on PT have been published (many of them also including Eyvind W. Axelsen as an author), and the details have gradually been shaped, mainly in a setting where PT is applied to Java.

The JPT compiler has been developed alongside the development of the PT concept and the JPT language. My work on the compiler started when its development was well underway. The compiler already had most of the PT functionality, but there were unimplemented parts and a lot of bugs, and thus it was not suited for testing the usefulness of PT.

PT has also been applied to the languages Boo and Groovy. The work on Boo with PT was done as part of an earlier master thesis, which resulted

in a compiler with basic PT functionality [12]. Research into applying PT to dynamically typed languages was done by applying it to the Groovy language [3]. Both these were implemented, but the implementations were not nearly as complete as the JPT compiler.

To make it possible to evaluate the PT mechanism for larger programs, an implementation of a compiler, like JPT, is a necessary tool. It is therefore important that the compiler is as complete as possible and ready for use, so this is what my work has been geared towards. I have been able to take the compiler some steps closer to the goal of completion, and the compiler now works reasonably well and for most of the main mechanisms of PT. It is now regularly used for testing the ability of PT to support large scale programming.

There are, however, obviously still bugs in the compiler, and some aspects of JPT are not implemented. The two most important unimplemented features are templates as parameters to templates, and the access control mechanism connected to the package templates. The former is quite clearly defined, but some of the details on how the mechanism should work came so late in the process, that there was no time to implement it in the compiler as part of my work with this thesis. The latter is not implemented because it is still not clear how this should work, seen from the programmer.

At first, my thesis work started out in a different direction. The plan was to try to rewrite a quite extensive Java library to use JPT, and to see whether its structure and fexibility could be improved by using the PT concepts. However, after doing some preliminary work on this, it became clear that the compiler had limitations that blocked any further progress. Since there were no developers at work on the compiler, it was decided (by me and my supervisor) that I should shift my focus towards the completion of the compiler.

Even though a lot of work had gone into building a compiler for JPT, there did not exist any documentation describing this compiler and the JPT language as a whole. Most of the published articles on PT used Java syntax in the examples, but the information was fragmented across many articles, and some parts of JPT had not been described in any of these articles. Therefore, a significant part of my thesis work consisted of collecting this information and presenting it. The result of this work is presented in chapter 2 of this thesis.

Starting work on the compiler proved to be a challenge. Given the complexity of the code base and the lack of documentation, it was hard and time-consuming to understand how the compiler worked. Since there were no developers working on the compiler at the time, I was mostly on my own in getting to know the code base, without access to those who had best knowledge of it.

The work of understanding the JPT compiler's code base was further complicated by the fact that it is an extension of the *JastAddJ* compiler, which is a Java compiler built using *JastAdd*. JastAdd is a tool for building extensible compilers. It contains mechanisms tailored for this purpose, and uses special syntax for these mechanisms. Because of this, I had to learn how JastAdd works before I could go into the details of the JPT

compiler. The JPT compiler also uses some of the JastAdd mechanisms in a slightly different way than they were intended, which also made the learning process more complicated.

In the current compiler, the JPT part consists of around 12000 lines of code for the compiler, and around 7000 lines of test programs. JastAddJ has around 30000 lines of code in the parts that are relevant to the JPT compiler. While it was not necessary to know the entire JastAddJ compiler to start doing work on the JPT compiler, I had to understand some of it to make any sense of how the JPT part of the compiler extends it.

Although the JastAddJ compiler is a complete Java compiler, generating Java bytecode as its output, it was decided (before I started work on the compiler) to have the JPT compiler output standard Java source code, rather than bytecode. The code generated by the JPT compiler could then be compiled to bytecode using a standard Java compiler.

The rest of this thesis is organized in the following way: In part 1 we will look at some necessary background information, with an in-depth presentation of PT, as well as an introduction to JastAdd and the JastAddJ compiler. Part 2 is where we explore the JPT compiler, its design and implementation, and provide a presentation and some discussion of the work I have done since taking over the task of developing it. There will also be discussions on the decisions that were taken before my work started on the compiler, and some ideas for future work.

Part I

Background

Chapter 2

Package Templates

In standard Java, the package concept is a means for modularization of the code, in the same way as e.g namespaces are for some other languages. However, the flexibility of packages for reuse of code is not very good. Introducing *Package Templates* is a proposal to improve this situation, by moving the package one step towards a generic concept. Package templates (or just "templates" for short) also allow adjustments of the generic code each time it is used, so that even the same template can be used several times with different adjustments in the same program. Each time we use a template, we say that we *instantiate* the template.

PT is mainly targeted towards statically typed object oriented languages. It is meant to have good support both for writing flexible separate modules, and for combining these. The composition of separate templates and the program is done at compile-time. There has been some research into using PT in a dynamically typed language called Groovy [3], but this thesis will only focus on statically typed languages, and the PT implementation described here is based on Java, and is called JPT.

Java packages that instantiate templates are called *JPT packages*, and are written with a slightly different syntax than normal Java packages, see Figure 2.1. Templates use the same syntax as JPT packages, only replacing the **`package`** keyword with the **`template`** keyword.

In this chapter, we will see how JPT works, in mainly technical terms, without much discussion of why the particular solutions were chosen. While we will not go into any details of how the JPT compiler works until chapter 4, some of the discussions in this chapter will make assumptions that may be specific to how JPT is handled by the JPT compiler.

Java syntax	JPT syntax

`package P;`
`class A { ... }`
`class B { ... }`
	`package P {`
	`  class A { ... }`
	`  class B { ... }`
	`}`

Figure 2.1: The difference between Java packages and JPT packages

As mentioned in the introduction, the JPT compiler turns JPT programs into standard Java. Some examples in this chapter will show how source code in JPT will be transformed into standard Java. One can then use any Java compiler to compile the transformed code into bytecode.

2.1 Templates

A template, in its simplest form, is created with the **`template`** keyword followed by the name of the template and braces which enclose the template's body. A template `T` can be written and *instantiated* inside a JPT package `P` as follows:

```
template T {
  class A { ... }
  class B { ... }
}
package P {
  inst T;
  class K { ... }
}
```

When a template is instantiated inside a package, copies are made of the classes, interfaces, and enums in the template, and the copies are inserted into the package, usually after some adjustments. The package may also have its own classes, as shown by the class `K` in the above example.

When compiled with the JPT compiler, the resulting package `P` will contain the classes `A` and `B` from the template `T`, as well as the class `K`. In the resulting Java package, which is generated by the JPT compiler, it is, in almost every respect, not possible to distinguish between classes from the template and the classes defined in `P`. One can have more than one instantiation in a JPT package (even of the same template, when using renaming, as explained later).

A way of viewing the instatiation of templates, is to compare it to the instantiation of classes. A class is like a template for objects. The analog to instantiation of a class, is the generation of an object from the class. A package template, on the other hand, can contain several classes, as well as interfaces and enums. When it is instantiated, a copy of all these classes, interfaces and enums are made available inside the package in which the template is instantiated. As opposed to object generation from classes, which is executed at run-time, template instantiations are executed at compile-time.

Templates may also be instantiated inside other templates:

```
template T {
  class A { ... }
  class B { ... }
}
template U {
  inst T;
```

```
  class K { ... }
}
package Q {
  inst U;
}
```

The result here is first a template `U` that contains the classes `A` and `B`, as defined in the template `T`, and in addition to these, the class `K`. When this template is afterwards instantiated in the package `Q`, the package will be identical to the package `P` from the previous example. So although the two packages are created in different ways, the only difference between the two resulting packages is their names.

Several adaptations of the template classes are possible when instantiating a template. The adaptations are applied to the copies of the classes and interfaces made by the instantiation, not the original classes and interfaces in the template. So if the template is instantiated in a package or another template, previous adaptations will not disturb the new instantiations. The following sections will describe each of the possible types of adaptations.

2.2 Addition classes

When a template is instantiated it is possible to make additions to its classes, and these additions are described in the package or template it is instantiated in. This is done by writing an *addition class*, with the same name as the original class, which lists the additions (methods and fields) that should be added. The resulting class is a combination of the original class, as defined in the template, and the declarations in the addition class. Note that in addition classes one can refer to all declarations within the instantiating package/template, also those from other instantiations and addition classes. Below is an example of an addition class.

```
template T {
  class A {
    void f() { ... }
  }
}
package P {
  inst T;
  class A adds {
    void g() { ... }
  }
}
```

When the above JPT program is turned into standard Java, it would look somewhat like the code in the following example:

```
package P;
class A {
  void f() { ... }
  void g() { ... }
```

```
}
```

The result is similar to additions made in subclasses in a class hierarchy, but the difference is that class `A` from the template is combined with the (addition) class `A` from the package and there is no way to generate objects of the old version of `A`. The methods and fields in the resulting class is the union of the methods and fields from the class in the template and the addition class.

Templates may contain entire subclass hierarchies, and all classes in these hierarchies, not only the leaf classes, may get additions through their own addition classes during an instantiation.

2.2.1 Addition interfaces

It is also possible to make additions to interfaces that are defined in a template:

```
template T {
  interface I {
    void f();
  }
  class C implements I {
    public void f() { ... }
  }
}
package P {
  inst T;
  interface I adds {
    void g();
  }
  class C adds {
    public void g() { ... }
  }
}
```

In the example above, the method `g()` is added to the interface `I` in package `P`. As a result of this the class `C` in `P` also needs to add the method `g()` since this method now occurs in the interface it implements.

2.2.2 Overrides in addition classes

Even if additions do not create a subclass hierarchy, a method in the addition class may override methods in the template class. The overridden method will only be available by using the special *tsuper* keyword, which will be described in section 2.2.5. The example below shows an override in the addition class.

```
template T {
  class A {
    void f() {
```

```
      System.out.println("In template");
    }
    void g() {
      f();
    }
  }
}
package P {
  inst T;
  class A adds {
    void f() {
      System.out.println("In package");
    }
    void h() {
      f();
    }
  }
}
```

In the example above, the `f()` method from the template is overridden by the `f()` method in the addition class. As a result both `g()` and `h()` will call the `f()` method in the addition class.

2.2.3 Subclass hierarchies and overrides

With class hierarchies, methods may be overridden both in subclasses and in addition classes. For these situations, well defined rules are needed to describe how these overrides work. If a method `f()` is called inside an addition class, and there exists a definition of `f()` both in a superclass and in the class that is being added to (called the **`tsuper`** class), the method in the class that is being added to is the one that is called.

The following example, combining overrides in subclass hierarchies with overrides in addition classes, shows how this works.

```
template T {
  class A {
    void f() {
      System.out.println("in T.A.f()");
    }
  }
  class B extends A {
    void f() {
      System.out.println("in T.B.f()");
    }
  }
}
package P {
  inst T;
  class A adds {
```

```
    void f() {
      System.out.println("in P.A.f()");
    }
  }
  class Main {
    public static void main(String[] args) {
      new A().f();
      new B().f();
      ((A) new B()).f();
    }
  }
}
```

When running the above example, the call `new A().f()` will print "in P.A.f()", as the `f()` method in the original `A` class has been overridden in the addition class in `P`. The call `new B().f()` will print "in.T.B.f()". The override in the addition class `A` does not override the `f()` method in class `B`. The call `((A) new B()).f()` will also call the `f()` method in class `B`, showing that polymorphism works for the class hierarchy.

2.2.4 Template abstract (tabstract) methods

Classes in Java may have abstract methods, which are expected to be implemented in subclasses. In addition, a similar concept is available in JPT. A method may be declared *template abstract* with the keyword **`tabstract`**. This means that it *has to* be implemented in an addition class upon instantiation of the template in a package. It *may* also be implemented in an addition class in a template, but as long as it is implemented at the latest in the package(s) that instantiates the template, it is not required.

```
template T1 {
  class C {
    tabstract void i();
    tabstract void j();
  }
}
template T2 {
  inst T1;
  class C adds {
    void j() { ... }
    tabstract void k();
  }
}
package P {
  inst T1;
  inst T2 with C => C2;
  class C adds {
    void i() { ... }
    void j() { ... }
```

```
  }
  class C2 adds {
    void i() { ... }
    void k() { ... }
  }
}
```

The example above shows how to use the **tabstract** keyword, and both templates contain classes with template abstract methods. Template abstract methods may be implemented in addition classes in other templates, but templates are not required to implement them. In the example, template `T2` implements the `j()` method, but leaves the `i()` method as template abstract. It also adds another template abstract method, `k()`. The package `P` is required to implement all template abstract methods.

This is similar to abstract methods in a class. If a class contains abstract methods it is an abstract class, and a subclass must provide an implementation of this method if you want to generate objects of it. However, a tabstract method does not prohibit generation of objects, as we can be sure that it is implemented at the latest when a package is formed.

2.2.5 Calls to the original method

In some cases of overriding in addition classes we may want to call a method in the original class from its addition class. This is similar to calling a method in a traditional superclass from its subclass using the **super** keyword, but in our case we must use the **tsuper** keyword. Instead of calling a method in a superclass, **tsuper** calls a method in a class in its "super-template".

In the following example, the method `f()` in class `A` gets overridden in the instantiating package. The overriding method then calls the original method, the one defined in the template `T`, by using the **tsuper** keyword.

```
template T {
  class A {
    void f() { ... }
  }
}
package P {
  inst T;
  class A adds {
    void f() {
      tsuper.f();
      ...
    }
  }
}
```

2.3 Renaming

When instantiating a template, the original naming of its classes, fields, and methods may not reflect their specific usage where it is instantiated. Thus it is an advantage to be able to rename them. JPT allows for this kind of renaming in the instantiation clause.

A renaming is done in a "semantic way", which is to first find the declaration that should be renamed, then find all accesses bound to this declaration, and then rename all of these.

One example of where renaming is useful, as described in [11], is when using a template to write a general implementation of a graph. This graph contains a class for nodes and a class for edges. When using these classes for something specific, like a map containing cities and roads, the code becomes more readable if we are able to rename these classes. The `Node` class could be renamed to `City`, and the `Edge` class could be renamed to `Road`.

2.3.1 Renaming classes

All renaming follows the keyword **`with`** in an instantiation statement. Several classes may be renamed by separating the rename clauses with a comma. The following example renames two classes from a template and leaves a third one as it was defined in the template.

```
template T {
  class A { ... }
  class B { ... }
  class C { ... }
}
package P {
  inst T with A => D, C => E;
  class D adds { ... }
  class B adds { ... }
}
```

This leaves the package `P` with the three classes `D`, `E` and `B`. The names `A` and `C` are not valid names in `P`. Also note that the addition classes must use the new names. Trying to add methods and fields to `A` using the statement **`class`** `A` **`adds`** `{ ... }` would result in a compile time error, since `A` is not a valid class name in `P`.

Note that the constructors of a class are renamed along with the class itself, and so are all occurences in **`new`**-statements, in **`extends`** clauses, and in type-casts in the instantiated template.

```
template T {
  class A {
    void f() { ... }
  }
  class B extends A {
    void f() {
```

```
      super.f();
    }
    A getA() {
      return new A();
    }
    A getB() {
      return (A) new B();
    }
  }
}
package P {
  inst T with A => NewA(f() -> g);
  class NewA adds {
    void f() { ... }
  }
}
```

In this example, `B` becomes a subclass of `NewA` in `P`, and the name `A` will no longer be valid. In the method `getA()`, the `new A()` statement will in `P` be changed to `new NewA()`, and in `getB()`, the cast to `A` will be changed to a cast to `NewA`. The `f()` method in `B` is an override of the `f()` method in `A`. In `NewA`, this method has been renamed to `g()`, and, as such, the `f()` method in `B` will also be renamed to `g()`. The new `f()` method declared in the addition class, is independent of the methods in `A` and `B`, and will keep its name. Note that the above code would cause the JPT compiler to give error messages about the `new` statements, as a certain detail is missing. There will be more about that in section 2.8.

2.3.2 Renaming methods and fields

Methods and fields may also be renamed. The renames are done in a parentesis after the class rename in the instantiation clause. If you do not want to rename the class, but want to rename some of its fields or methods, it is possible to "rename" a class to its old name. An example is provided below.

```
template T {
  class A {
    int i;
    void f() { ... }
    void g() { ... }
  }
  class B {
    int j;
    void u() { ... }
    void v() { ... }
  }
}
package P {
```

```
  inst T with A => A(i -> k, f() -> ff),
              B => C(v() -> vv);
}
```

In the example above, package `P` will contain two classes; `A` and `C`. Class `A` will contain the field `k` and the methods `ff()` and `g()`, and class `C` will contain the field `j` and the methods `u()` and `vv()`.

As seen in the previous example, method renames require the original method name to be followed by parentheses containing the parameter types. This is to allow for renames of specific methods when there are overloaded methods. However, instead of writing a rename clause for each of a number of methods with the same name, it is possible to use the wildcard character (∗) inside the parenthesis, to indicate that all methods with this name should be renamed regardless of parameters. An example of this kind of method renaming is shown below.

```
template T {
  class A {
    void f() { ... }
    void f(int i) { ... }
    void f(double d) { ... }
    void h() { ... }
  }
}
package P1 {
  inst A with A => A(f() -> g, f(int) -> g);
}
package P2 {
  inst A with A => A(f(*) -> g);
}
```

Now class `A` in package `P1` has got the methods `g()`, `g(int i)`, `f(double d)` and `h()`. In package `P2`, all methods originally named `f` are renamed to `g`, while `h` remains the same. So the resulting class `A` in package `P2` contains the methods `g()`, `g(int i)`, `g(double d)` and `h()`.

There are two additional rules concerning which renamings are legal. One says that renaming a field or method must not result in a name collision. The other says that fields and methods must be renamed at the subclass level where they are defined. For (virtual) methods this means that they must be renamed at the subclass level with the first occurrence of the method, i.e the one highest up in the hierarchy. The corresponding overrides will then also be renamed by the compiler. The following example shows violations of these rules:

```
template T {
  class A {
    void f() { ... }
  }
  class B extends A {
    void f() { ... }
```

```
    void g() { ... }
    void h() { ... }
  }
}
package P {
  inst T with B => B (f() -> ff, g() -> h);
}
```

The instantiation clause tries to rename the `f()` method in class `B` to `ff()`. However, as it is not done in `A`, where it first appeared, it is not allowed. Continuing to the next renaming, the method `g()` is renamed to `h()`. Since there already exists a method named `h` in class `B`, the renaming leads to a name collision, and thus, is not allowed. The next example shows how one can resolve the errors in the instantiation clause.

```
package P {
  inst T with A => A (f() -> ff),
              B => B (g() -> h, h() -> hh);
}
```

Here, the `f()` method is renamed in class `A`. The compiler will recognize that the method `f()` in class `B` is an override, and thus rename it so it has the same name as the method it is overriding. In class `B`, the method `g()` is still renamed to `h()`, but in this instantiation clause, the method `h()` is also renamed, so there is no name collision.

2.4 Class hierarchies, super, and tsuper

To understand the concepts of **`super`** and **`tsuper`** better, one may visualize a number of templates and a package instantiating them (directly or indirectly) as a graph lining up the classes from the packages and templates in two dimensions, where calls to **`super`** goes upwards and calls to **`tsuper`** goes to the left. Take the following example:

```
template T {
  class TA { ... }
  class TB extends TA { ... }
  class TC extends TB { ... }
}
template U {
  inst T with TA => UA, TB => UB, TC => UC;
  class UA adds { ... }
  class UB adds { ... }
  class UC adds { ... }
}
package P {
  inst U with UA => A, UB => B, UC => C;
  class A adds { ... }
  class B adds { ... }
  class C adds { ... }
```

```
}
```

Figure 2.2 shows the relationships between these classes and addition classes. Note that in this figure, all the classes and their addition classes are drawn separately. Figure 2.3 sketches how this system of templates and instantiations will finally appear in `P`. Here, all that remains of the two dimensional hierarchy has become a simple class hierarchy, like the following:

```
package P;
class A { ... }
class B extends A { ... }
class C extends B { ... }
```

Note that everything from `T` is renamed twice (`TA=>UA=>A`, `TB=>UB=>B`, and `TC=>UC=>C`), and everything from U is renamed once (`UA=>A`, `UB=>B`, and `UC=>C`).

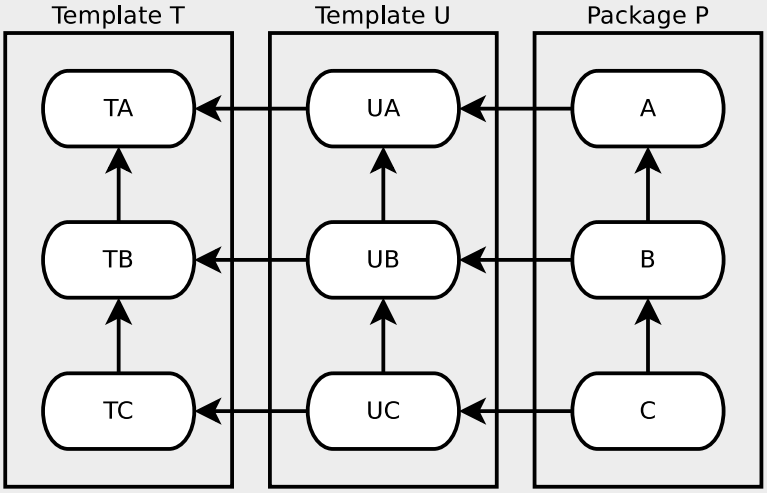


Figure 2.2: A two dimensional class hierarchy. Each class and addition class is drawn inside its template or package, with a vertical arrow pointing to its superclass, and a horizontal arrow pointing to its tsuperclass.

2.4.1 Constructors using tsuper

In some well defined cases (see below) the JPT language requires that you call a constructor in a template class from a constructor in a corresponding addition class. JPT provides special syntax for this, as shown below.

```
template T {
  class C() {
    C() {
      System.out.println("Constructor in template T.");
    }
```

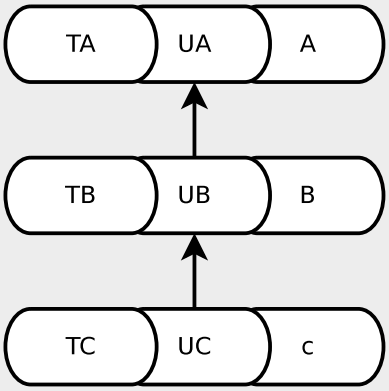


Figure 2.3: A more compact view of the class hierarchy from Figure 2.2.

```
  }
}
package P {
  inst T;
  class C adds {
    C() {
      tsuper();
      System.out.println("Constructor in package P.");
    }
    public static void main(String[] args) {
      new C();
    }
  }
}
```

Compiling and running the above example would first print the line "Constructor in template T.", then print the line "Constructor in package P.".

In Java, calls to the superclass's constructor must be done using the `super()` statement in the first line of a class's constructor. However to avoid multiple calls to constructors in the same addition class, the following rules apply:

- In an addition class in a template you should not call `super()`, but instead call `tsuper()` for the `tsuper` class.
- In an addition class in a package, you should first call `super(...)`, and then call `tsuper(...)` for the `tsuper` class.

Thus, calls to constructors using `tsuper()` will happen after calls to `super()`. In a class hierarchy like the one in Figure 2.2 calls to the constructors in each class and addition class using `super()` and `tsuper()` would take the form of a backwards "E"(∃).

Figure 2.4 shows an overlay over the class hierarchy in Figure 2.2. The constructors are represented as numbered circles. When the constructor in `C` is called, the following happens:

- The `super()` call in `C`'s constructor calls the constructor in `B`.
- The `super()` call in `B`'s constructor calls the constructor in `A`.
- There is no `super()` call in `A`'s constructor, so `UA`'s constructor is called with `tsuper()`.
- `UA`'s constructor calls `TA`'s constructor using `tsuper()`.
- In `TA`'s constructor there are no calls to `tsuper()` or `super()`, so the rest of the statements in the constructor are executed.
- When `TA`'s constructor has finished, the rest of `UA`'s constructor runs.
- When `TA`'s constructor has finished, the rest of `A`'s constructor runs.
- When `A`'s constructor has finished, the `super()` call in `B`'s constructor returns. Then `B`'s constructor will continue with the `tsuper()` call to `UB`'s constructor, and the calls continue for `B`'s tsuper classes in the same way as with the `tsuper()` call from `A`'s constructor.
- When `B`'s constructor has finished, `C`'s constructor will do the same by calling `tsuper()` before, finally, finishing with its remaining statements.

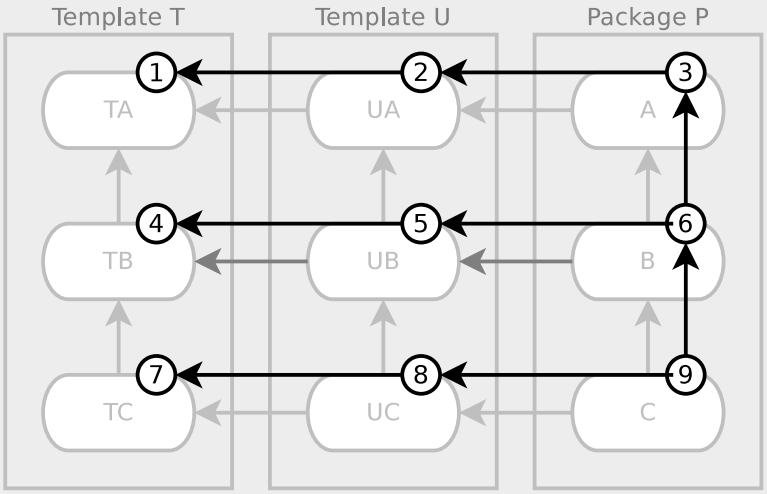


Figure 2.4: Each circle represents a constructor call, with the number denoting which order the constructors are finishing their execution.

2.5 Multiple instantiations

It is possible to instatiate a given template several times in another template or in a package. Some of the names of the classes in each instantiation then need to be changed, so that name collisions between the instantiations are avoided. The instantiations are independent of each other, so a class from

one instantiation looks like any other class to the corresponding class in the other instantiation.

The following example defines a package with two instantiations of the template `T`:

```
template T {
  class A { static int i; ... }
}
package P {
  inst T;
  inst T with A => B;
  class A adds { ... }
}
```

In the package `P`, the template is instantiated twice, with one of the instantiatons renaming the class `A`. The resulting package contains the classes `A` and `B` which, although they are instantiated from the same class in the same template, are independent of each other, just as if they were written separately. Note that the static member of `A` is repeated for each instantiation; setting `A.i` does not affect the value of `B.i`.

2.6 Merging

We have stated above that we must avoid name collisions during renaming and multiple instantiations. However, there is one exception to this rule, which is that classes are allowed to have the same name. Then these classes are *merged* to one class (as explained below) with the actual name. Mostly, merging is done with classes from different templates, but it is possible to merge classes from the same template also.

It is not possible to merge fields and methods, so name collisions between fields and methods inside the merged classes must be avoided by renaming. This is done as described in section 2.3. Figure 2.5 shows an example of a hierarchy containing merged classes.

```
template Merge1 {
  class M1 {
    int i;
    void f() { ... }
    void g() { ... }
    int h() { ... }
  }
}
template Merge2 {
  class M2 {
    int i;
    void f(int i) { ... }
    void g() { ... }
    double h() { ... }
  }
```

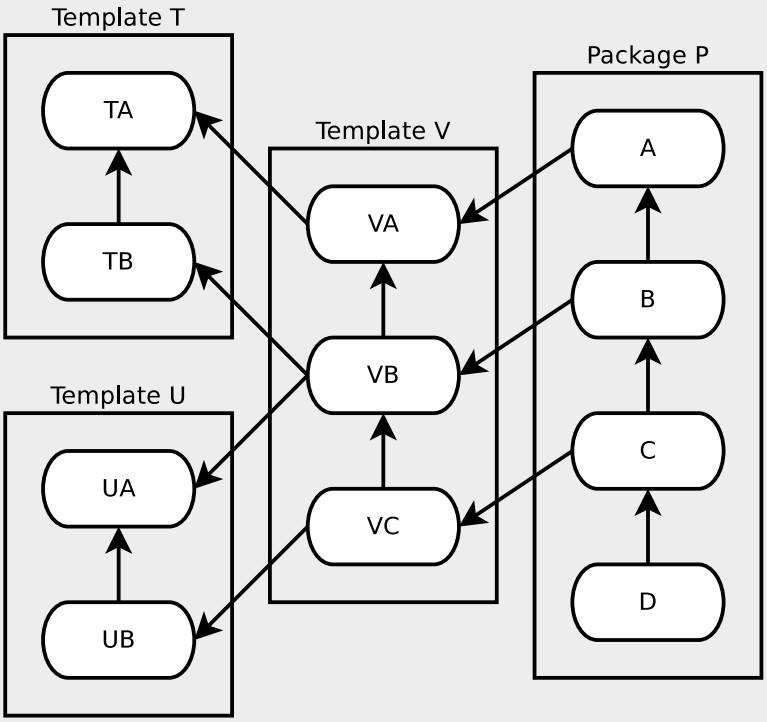


Figure 2.5: A two dimensional class hierarchy containing merged classes. Template `V` instantiates the templates `T` and `U` drawn in the same style as Figure 2.2. The class `TB` from template `T` and the class `UA` from the template `U` are merged to form the new class `VB`.

```
}
package P {
  inst Merge1 with M1 => M(g() -> gg);
  inst Merge2 with M2 => M(i -> j, h() -> hh);
}
```

In the example above the classes `M1` and `M2` are merged into the single class `M`. There are potential name collisions that must be resolved during the instantiations:

- The collision between `M1.i` and `M2.i` is resolved by renaming `M2.i` to `M2.j`.
- The collision between `M1.g()` and `M2.g()` is resolved by renaming `M1.g()` to `M1.gg()`.
- The collision between `M1.h()` and `M2.h()` is resolved by renaming `M2.h()` to `M2.hh()`.

Note that the methods named `f` do not cause collisions because the **`int`** parameter in `M2.f(int i)` makes it different from `M1.f()`.

2.6.1 Constructors in merged classes

With merging, the scheme for calling constructors becomes slightly more complicated. Then the regular syntax using a call to **tsuper()** will not work because it is ambiguous. To solve this, there is a special syntax for this case:

```
template T1 {
  class A1 {
    A1() { ... }
  }
}
template T2 {
  class A2 {
    A2() { ... }
  }
}
package P {
  inst T1 with A1 => A;
  inst T2 with A2 => A;
  class A adds {
    A() {
      tsuper[T1]();
      tsuper[T2]();
    }
  }
}
```

This works in most cases, but in one case it doesn't work, and that is when we merge two classes from the same template. In this case we need to use *named instantiations*. The following example shows how this works.

```
template T {
  class C { ... }
}
package P {
  T1: inst T;
  T2: inst T;
  class C adds {
    C() {
      tsuper[T1]();
      tsuper[T2]();
    }
  }
}
```

2.7 Hierarchy preservation

PT was explicitly designed so that the underlying object oriented language should not need multiple inheritance. Thus, class merging introduces a

problem as we could merge classes that have different superclasses and thus get a class with multiple superclasses. To avoid this, a rule in PT says that if classes to be merged have different superclasses (the `Object` class is not taken into consideration in this case; it is defined as being equal to all other classes), then their superclasses must also be merged during the same instantiations. The following example shows this.

```
template T1 {
  class A { ... }
  class AA extends A { ... }
  class C { ... }
}
template T2 {
  class B { ... }
  class BB extends B { ... }
  class D { ... }
  class E extends D { ... }
}
package P {
  inst T1 with A => AB, AA => AABB, C => CE;
  inst T2 with B => AB, BB => AABB, E => CE;
}
```

As seen above, the classes `AA` in `T1` and `BB` in `T2` both have explicit superclasses, so when they are merged, their superclasses must also be merged by giving them the same name. The class `C` in `T1` does not have an explicit superclass, so it will have the `Object` class as its superclass. As shown in the example, this does not need an explicit merge of superclasses when it is merged with the `E` class. However, the resulting class `CE` will have `D` as its direct superclass, not `Object`.

2.7.1 External superclasses

The rule that superclasses of merged classes should also be merged is obviously not enforceable for superclasses external to the template itself (that is, in other packages), as these classes might also be used in other parts of the program. As a consequence of this, some additional rules are needed for external superclasses:

1. A class which extends an external class must say **`extends external`**, not only **`extends`**.

2. Names from the external superclass may not be changed.

3. Two classes with different external superclasses may not be merged.

4. Two classes where one has an internal superclass and the other has an external superclass may not be merged.

2.8 Assumed constructors

To create new objects of a template class inside a template, we need to be certain that a constructor matching the **new** statement will exist in that class when it finally is instantiated in a package. In JPT this problem is solved with the **assumed** keyword. Consider the following example:

```
template T {
  class C {
    assumed C();
    assumed C(int i);
    void f() {
      C c = new C();
      ...
    }
    void f(int i) {
      C c = new C(i)
      ...
    }
  }
}
package P {
  inst T;
  class C adds {
    C() { ... }
    C(int i) { ... }
  }
}
```

Here, the methods named `f()` each generates a new object of type `C`. The **assumed** keyword makes sure that the needed constructors exist, otherwise one will get a compile-time error. In package `P`, the addition class is required, since it must provide the constructors.

2.9 Required types

There are two kinds of generic parameters to packages in JPT: *type parameters* and *template parameters*. Type parameters allows for formal types to be given actual types during instantiation inside a template. Template parameters make it possible to pass a template as a parameter to another template. Template parameters will be explained in section 2.10.

In earlier versions of JPT, type parameters were passed into a template in a way similar to generics in Java. This was later replaced by the *required types* mechanism, due to certain limitations with the original approach [4].

2.9.1 Concretization of required types

When instantiating a template with required types, the required type may be *concretized* in the **inst** statement by using the left pointing arrow

operator, <=. Upon instantiation in another template, it is not necessary to concretize a required type, but required types must be concretized at the latest when their template is instantiated in a package. Below follows an example which shows how to declare required types in a template and how to concretize them in an instantiation.

```
template T {
  required type R {
    int f();
  }
  class C {
    int g(R r) { ... }
  }
}
template T2 {
  inst T;
}
package P {
  inst T2 with R <= A;
  class A {
    int f() { ... }
  }
  class Main {
    public static void main(String[] args) {
      A a = new A();
      C c = new C();
      int i = c.g(a);
      ...
    }
  }
}
```

In the above example the required type `R` is concretized by the class `A`. So when the `g()` method in class `C` is called from the main method, the actual parameter has type `A`, and will therefore use the method `f()` from the class `A`. Note also that the required type is not concretized in template `T2`, and because of that it will still be a required type in `T2`. However, when T2 is instantiated in a package (here `P`), it must be concretized. The class `A` could also have concretized `R` if it had been defined in an external package, or in `T` or `T2`.

The required type in template `T` or `T2` could just as well be concretized by an interface. Below is another example which, among other things, shows this.

```
template T {
  required type R {
    int f();
  }
  class C {
    int g(R r) {
```

```
      return r.f() + 1;
    }
  }
}
package P {
  inst T with R <= I;
  interface I {
    int f();
  }
  class A implements I {
    int f() { ... }
  }
  class B implements I {
    int f() { ... }
  }
  class Main {
    public static void main(String[] args) {
      A a = new A();
      B b = new B();
      C c = new C();
      int i = c.g(a);
      int j = c.g(b);
      ...
    }
  }
}
```

Here, the interface `I` concretizes the required type `R`, and because both class `A` and class `B` implement `I`, both are valid parameters to the `g()` method.

2.9.2 Required type constraints

Required types may be constrained by naming a type, by giving structural constraints, or both.

```
template T {
  interface I {
    void f();
  }
  required type R1 {
    void f();
  }
  required type R2 implements I {}
  required type R3 implements I {
    void g();
  }
}
package P {
  inst T with R1 <= C, R2 <= C, R3 <= C;
```

```
  class C implements I {
    public void f() { ... }
    void g() { ... }
  }
}
```

In the example above, the class `C` satisfies all constraints set by the required types in `T`, and because of that it may concretize all of them. To satisfy `R1` it would be enough for `C` to contain an implementation of a method named `f()` with no return type. `R2` must be concretized by a type which implements the interface `I`. It is not enough to provide the `f()` method, it must also explicitly implement `I`. For `R3` the concretizing type must explicitly implement `I` and provide the method `g()`.

2.9.3 Required classes and required interfaces

As shown earlier, using the **`required type`** statement puts no restriction on whether the concretizing type is a class or an interface. Sometimes, however, it may be necessary to impose such a restriction. An example of when a required type needs to be a class, is when objects are created from the required type. If a class implements the required type, the required type must be an interface.

To enforce these restrictions JPT contains the declarations **`required class`** and **`required interface`**. An example of their usage is provided below.

```
template T {
  required interface I {
    int i();
  }
  required class C {
    int j();
  }
  class A {
    int add(I i) {
      return i.i() + (new C).j();
    }
  }
}
package P {
  inst T with I <= II, C <= CC;
  interface II {
    int i();
  }
  class CC {
    int j() { ... }
  }
  class B implements II {
    public int i() { ... }
  }
```

```
  class Main {
    public static void main(String[] args) {
      B b = new B();
      A a = new A();
      int i = a.add(b);
    }
  }
}
```

Here, the interface `II` concretizes the required interface `I` and the class `CC` concretizes the required class `C`. Trying to concretize `I` with a class, or concretize `C` with an interface, would result in a compile time error.

2.9.4 Default concretization

In some well defined cases, it is allowed to not concretize a required type upon instantiation in a package. If the required type not being concretized is a required interface, it is then automatically concretized to an interface without any changes. For classes, this would not be as straightforward. Only if a required class is constrained nominally by another class, without adding any new methods, will it get such a default concretization. In that case this default concretization would be the class which constrains the required class.

In the example in section 2.9.3, the `II` interface does not add anything new to the required interface it concretizes. The required type could just as well be used directly as a default concretization.

2.9.5 Required type constructors

In Java it is not possible to generate objects of a generic type in a simple way. If a required type is resolved in the currently compiled template or package, the compiler knows which type concretizes that required type. As a result of this it is possible to generate objects of a required class, but the class must then specify a constructor.

```
template T {
  required class R {
    R(int i);
  }
  class A {
    public static void main(String[] args) {
      R r = new R(42);
    }
  }
}
package P {
  inst T with R <= C;
  class C {
    C(int i) {
```

```
      System.out.println(i);
    }
  }
}
```

As shown in the example above it is possible to instantiate objects of the required class `R`, as its constructor is explicitly required to be implemented in the concretizing class. During concretization all mentions of `R` are replaced by `C`. This means that the statement **`new`** `R(42)` is transformed into **`new`** `C(42)` inside the package `P`. Note that, as opposed to constructors in regular template classes, no **`assumed`** keyword is necessary in a required class.

2.9.6 Additions to required types

A template which instantiates a template containing a required type without concretizing it, may make additions to the required type. This is done in a way very similar to addition classes. An example is shown below.

```
template T1 {
  required type R {
    void f();
  }
}
template T2 {
  inst T1;
  required type R adds {
    void g();
  }
}
```

Any type that concretizes `R` from template `T2` must now provide both an `f()` and a `g()` method.

2.9.7 Merging of required types

Required types may also be renamed, and as such, they may also be merged by giving two or more required types the same name during a set of instantiations. As opposed to renaming in ordinary merging, there is no need to resolve name collisions when merging required types.

```
template T1 {
  required type R1 {
    void f();
    void g();
  }
template T2 {
  required type R2 {
    void g();
    void h();
```

```
  }
}
template T {
  inst T1 with R1 => R
  inst T2 with R2 => R;
}
package P {
  inst T with R <= C;
  class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
  }
}
```

As shown in the example above, the required type `R` contains all the methods from `R1` and `R2`. The signatures of the `g()` methods are identical, and because of that, `R` contains only one method named `g()`. There is no need to resolve the names when merging, since there is no implementation in either of the merged required types, so this will work much like multiple inheritance for interfaces.

2.10 Template parameters to templates

In addition to required types, templates may also have *template parameters*. Template parameters allow us to write generic code which will work for several templates.

Templates may declare formal template parameters inside angular brackets in the template declaration. The actual template parameters may then be given during an instantiation. A simple example follows.

```
template T {
  class C {
    int i() { ... }
  }
}
template V <template TT subof T> {
  inst TT;
  inst TT with C => CC;
  class C adds {
    int k() { ... }
  }
}
```

Here, `TT` is a formal template parameter, and the actual parameter for `TT` must be a "sub-template" of `T`, i.e a template given as an actual parameter for `TT` must instantiate `T` in a special way.

To satisfy the **`subof`** requirement on a template parameter, we use the **`subof`** keyword in the template declaration. An example follows below.

```
template T {
  class C {
    int i() { ... }
  }
}
template U subof T {
  class C adds {
    int j() { ... }
  }
}
```

With this construct the template `U` has an *implicit instantiation* of the template `T`. `U` may also expand the class `C` with an addition class, just as it could have done if it had a normal instantiation of `T`. However, as opposed to normal instantiations, when a template is implicitly instantiated, there is no way to rename anything inside it. If this was possible, it would defeat the purpose of the **`subof`** concept, because we want this declaration to be a guarantee that the template `U` contains, at least, the exact contents of `T`.

More than one template may occur after the keyword **`subof`** in a template heading. Since we cannot change names in these templates it follows that any name collisions between the classes in the templates named as bounds are forbidden. The following example shows the syntax used for more than one implicit instantiation.

```
template T1 {
  class C {
    int i() { ... }
  }
}
template T2 {
  class D {
    int j() { ... }
  }
}
template U subof T1, T2 {}
```

The template `U` can now be used as an actual template parameter to both

```
template V <template G subof T> { ... }
```

and

```
template W <template H subof U> { ... }
```

This concludes the presentation of the JPT language in this thesis, but there are still rules and corner cases in the language that we have not touched upon. However, what is presented here should be quite comprehensive, and should indeed be enough for understanding the rest of the thesis.

Chapter 3

JastAdd and JastAddJ

The JastAdd system is developed at Lund University in Sweden by Görel Hedin and her research group. Several articles (among them [7], [8], and [10]) have been published about JastAdd and about other systems built using JastAdd.

JastAdd is a compiler building system where you describe nodes of abstract syntax trees (AST's) and extend them with attributes for forming an attribute grammar. One of the benefits of JastAdd is that when you use it to build a compiler, this compiler may later be extended without making changes to the old code. The new code which extends the old is kept separate.

When describing the nodes of an AST with their attributes, one uses a special declarative language defined for JastAdd. The AST nodes become classes, and their attributes may be accessed through "get" and "set" methods. The AST node classes form a class hierarchy with the system defined `ASTNode` class as the top-level superclass.

JastAdd supports the use of third-party scanner generators and parser generators. These will build AST's by generating objects of the AST node classes generated by JastAdd.

JastAdd supports *inherited* and *synthesized* attributes, as well as *reference attributes*. The calculations which determines the value of each attribute are given in *attribute declarations*.

The attribute declarations and the evaluation of their values are given to JastAdd through *aspects*. Aspects provide a way of gathering all code related to one given concept in one place, regardless of which class it will eventually end up in. So code from an aspect may span several classes, and classes may contain code from several aspects. An example showing aspects in use is given in section 3.1.

When JastAdd generates the node class hierarchy it will *weave* all the aspects together. This means that it will take code from the aspects and place it in the node classes. In the end, all the generated code will be standard Java code.

Compilers built using JastAdd may easily be extended. Scanner tokens may be added by writing additional description files for the scanner, and the parser may be extended by writing additional grammar rule

descriptions. Extending the AST is done in the same way, by adding additional AST descriptions. And, finally, the attribute grammar may be extended by writing additional aspects, or by writing extensions to the aspects that already exist.

JastAddJ is a Java compiler built using JastAdd. Its base is a Java 1.4 compiler. Support for Java 1.5 is built upon the Java 1.4 compiler, only adding what is different from Java 1.4 to Java 1.5. Java 7 is then supported by extending the Java 1.5 compiler.

In this chapter, two kinds of hierarchies are described, the AST, built for each compiled program by the compiler, and the class hierarchy describing the nodes of the AST. Therefore it is important to differentiate beetween child nodes in the AST and subclasses. Nodes in the AST may have child nodes, but these child nodes are usually not objects of subclasses of their parent AST node.

Some parts of JastAdd will not be discussed here. We will stick to what is relevant for the rest of this thesis. Also, some examples given in this chapter may not be directly relevant to the JPT compiler.

3.1 Aspects

Aspects provide a way of grouping code together by functionality rather than grouping it together by which class the code belongs to. This means that code implementing functionality that is provided by several classes can be written in one aspect instead of spreading it across many different files.

To demonstrate how this works in JastAdd, we will define an AST node class hierarchy of numbers, where the root node is a class called `Number` which has two subclasses, named `Integer` and `Float`. The hierarchy is shown in Figure 3.1. The classes in the hierarchy are described outside of the aspects, so the aspects only expand existing classes. In JastAdd the classes are described using a special language described in section 3.3.

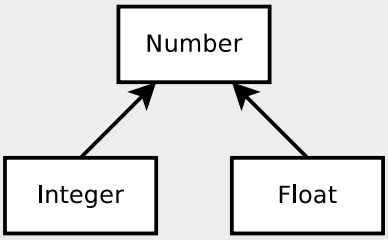


Figure 3.1: A simple AST node class hierarchy.

If we wanted to create a simple calculator which could do additions using the `Number` classes, we would need to add methods to the classes. A way of doing this using standard Java would be to write methods that would do additions in the code of each of the classes. When using aspects, all those methods can be grouped into one file, even if they eventually end up inside the different classes. The code example below shows how this

could be done. Note that the code in this example is a simplified version of a JastAdd aspect.

```
aspect Add {
  abstract Number Number.add(Number x, Number y);
  Number Integer.add(Number x, Number y) {
    return x.intValue() + y.intValue();
  }
  Number Float.add(Number x, Number y) {
    return x.floatValue() + y.floatValue();
  }
}
```

In this example each of the method names (here, all are named "add") are qualified by the name of the class it is supposed to end up inside (`Number`, `Integer`, and `Float`). When the aspect is weaved, the resulting Java code for the AST nodes for the `Number` class and its subclasses will look something like this:

```
abstract class Number {
  ...
  abstract Number add(Number x, Number y);
}
class Integer extends Number{
  ...
  Number add(Number x, Number y) {
    return x.intValue() + y.intValue();
  }
}
class Float extends Number {
  ...
  Number add(Number x, Number y) {
    return x.floatValue() + y.floatValue();
  }
}
```

The aspects are placed in *.jadd* and *.jrag* files. Although the JastAdd system treats these two kinds of files the same way, it is recommended to use these different file types in special ways to make it clearer what the content of a file contributes to the system. The JastAdd reference manual [2] describes their difference in the following way:

- Use .jrag files for declarative aspects, i.e., where you add attributes, equations, and rewrites to the AST classes

- Use .jadd files for imperative aspects, i.e., where you add ordinary fields and methods to the AST classes

```
abstract Expr;
abstract AssignExpr : Expr ::= Dest:Expr Source:Expr;
abstract AssignAdditiveExpr : AssignExpr;
AssignPlusExpr : AssignAdditiveExpr;
AssignMinusExpr : AssignAdditiveExpr;
```

Figure 3.2: Part of the AST description file in JastAddJ

3.2 Attributes

Attributes in JastAdd can be declared as *inherited* or *synthesized* using the **inh** and **syn** keywords, respectively. When an attribute is declared as inherited in an AST node class `A`, each AST node class which may have a child node of type `A` must also define this attribute. When an attribute is declared as synthesized in an AST node class `A`, each concrete subclass of `A` must provide a definition for the attribute.

Also, the value of a synthesized attribute may depend only on the value of the corresponding attribute in the child nodes. Likewise, inherited attributes can only depend on the value of this attribute in the parent nodes (with a few exceptions). In JastAdd, it may be easier to think of the difference in terms of where the attribute calculations are described. So when a node has an inherited attribute, this attribute is calculated and set by a node further up in the AST, and when a node has a synthesized attribute, this attribute is calculated and set by a node further down in the AST node class hierarchy.

Attribute calculations may not have side-effects. The reason for this is that JastAdd needs a guarantee that the order of the attribute calculations should not matter. JastAdd also uses caching of calculated values extensively, so having side-effects would make the compilers which are built by JastAdd less efficient, as possible caching would be limited by this.

3.3 Creating and extending the AST description

As described earlier, the AST which is built by JastAdd has an object oriented structure. All nodes in the AST are objects of classes in a subclass hierarchy with the class `ASTNode` as root. The node classes are defined in one or more files using a special syntax to describe the tree. Figure 3.2 shows an example taken from the Java 1.4 frontend of JastAddJ. JastAdd translates this into a simple class hierarchy where inheritance is denoted by a colon followed by the name of the superclass. The `::=` operator allows us to specify attributes of the node class. Set and get methods will be generated for all these attributes. Figure 3.3 shows the resulting class hierarchy as a UML diagram. Extending the AST can easily be done by adding a file containing more AST descriptions, like those shown above. AST nodes

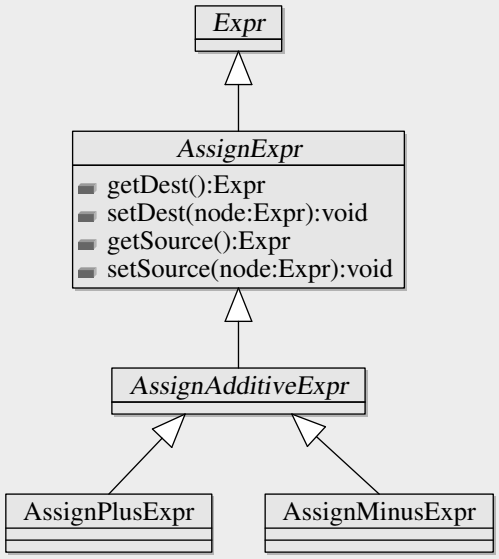


Figure 3.3: Class hierarchy of the additive expressions in JastAddJ's AST

from other files are available in the new file, so only the new AST nodes must be described.

3.4 Extending functionality with jadd and jrag files

If we want to add a method called `toString()` to the classes generated by the AST description shown in Figure 3.2, this should be done inside a jadd file, since it would be an imperative addition to the class. The following example shows how this could be done inside an aspect called `StringAddition`.

```
aspect StringAddition {
  public String Expr.toString() { return ""; }
  public String AssignPlusExpr.toString() {
    return getDest().toString() + "+=" + getSource().toString();
  }
  public String AssignMinusExpr.toString() {
    return getDest().toString() + "-=" + getSource().toString();
  }
}
```

As mentioned earlier, the method names in these files are preceeded by the name of the class they belong to. JastAdd will then add the method to its class when weaving the aspects. The classes `AssignExpr` and `AssignAdditiveExpr` inherit the method from `Expr` while `AssignPlusExpr` and `AssignMinusExpr` override it with their own definitions.

To avoid code duplication, we could add a method returning just the operator string. This would be an attribute, and should thus be placed in a

jrag file.

```
aspect OperatorString {
  syn String AssignExpr.operatorString() = "";
  eq AssignPlusExpr.operatorString() = "+=";
  eq AssignMinusExpr.operatorString() { return "-="; }
}
```

Here we are adding a new synthesized attribute called `operatorString` to `AssignExpr`. The subclasses of `AssignExpr` may override this attribute with their own value. Attribute calculations may be given using a special short form, as in `AssignExpr` and `AssignPlusExpr`, or like an ordinary method, as is the case for `AssignMinusExpr`.

Using this new attribute, we may now modify the `StringAddition` aspect.

```
aspect StringAddition {
  public String Expr.toString() { return ""; }
  public String AssignExpr.toString() {
    return getDest().toString()
           + operatorString()
           + getSource().toString();
  }
}
```

Here we avoid duplicating the code in `AssignPlusExpr` and `AssignMinusExpr` by moving the implementation into their superclass.

3.4.1 Collections

JastAdd includes a mechanism for defining named collections of nodes. A collection is declared as an attribute of a node class (using the keyword **`coll`**), and the collection gets its contents by nodes *contributing* to it. Continuing the example from above, if we wanted the `AssignExpr` node to contain a collection of its `AssignPlusExpr` child nodes, we could use the following collection attribute.

```
aspect AssignPlusCollection {
  coll LinkedList<AssignPlusExpr> AssignExpr.getAssignPlusExprs()
      [new LinkedList<AssignPlusExpr>()]
      with add root AssignExpr;
  AssignPlusExpr contributes this
      when getParentClass(AssignExpr.class) != null
      to AssignExpr.getAssignPlusExprs()
      for getParentClass(AssignExpr.class);
}
```

The first line in the aspect declares the attribute using the **`coll`** keyword. The statement inside brackets in the second line is the initialization of the collection. The first call to `getAssignPlusExprs` will run this statement to initialize the collection. The third line tells JastAdd to use the `add()` method

when adding elements to the collection, and sets the root of the subtree which is searched to find the elements to put into the collection. When JastAdd computes the collection, it will first search upwards in the AST, starting at the current `AssignExpr` node, and stopping at the first appearance of a node of the root type. In this case the root is also `AssignExpr`, so the search for the root will stop immediately. When the root has been found, the search continues downwards the AST, looking for contributors.

The *contribution* statement tells JastAdd that all nodes of type `AssignPlusExpr` in the subtree should be added to the collection. The **`when`** statement is used to set conditions for when a node should be added to the collection. In this case we only verify that there is an `AssignExpr` node above the `AssignPlusExpr` node in the AST. The **`to`** keyword tells JastAdd which collection to add the node to, and the **`for`** keyword tells JastAdd which class holds the collection to add to. The `getParentClass` is a method in every AST node which can be used to find a class above the current class in the class hierarchy.

3.4.2 Rewrites

Sometimes it is useful to be able to rewrite, i.e transform, an AST, and a mechanism that does this exists in JastAdd. As an example we introduce another AST definition, which describes a simple AST:

```
abstract Expr;
abstract Binary : Expr ::= Left:Expr Right:Expr;
AddExpr : Binary;
IntegerLiteral : Expr ::= Value:Integer;
```

When the compiler finds an instance of an `AddExpr` where both the left and right operand are integer literals, it is possible to optimize the generated code by calculating the value at compile time. This can be done by using JastAdd's rewrite mechanism. The following example shows how this is done.

```
rewrite AddExpr {
  when (getLeft() instanceof IntegerLiteral &&
        getRight() instanceof IntegerLiteral)
  to IntegerLiteral {
    int newValue = getLeft().getValue() + getRight().getValue();
    return new IntegerLiteral(newValue);
  }
}
```

The `AddExpr` node will now be replaced by the new `IntegerLiteral` node in the AST.

Rewrites are initiated when a node is accessed in the following way: In the root class of the AST, `ASTNode`, there is a method, called `getChild`, which will get a child of the node it is called from. When it is called, the `getChild` method first checks if the child node has any rewrites defined. Each of the rewrites will have certain prerequisites (defined using the **`when`**

keyword) that must be satisfied before they can be done. If there are any rewrites where these prerequisites are satisfied, they are run before returning a reference to the child node. If a rewrite takes place, the node which is returned may be a different node than the original child node. In the example above, when getting a reference to a child node of type `AddExpr`, the `getChild` method will return a reference to a node of type `IntegerLiteral` if both operands are `IntegerLiteral` objects.

3.4.3 Refines

When extending a compiler built by JastAdd, it may be useful to be able to make changes to attribute calculations. The *refine* mechanism in JastAdd makes this possible while also making it possible to reuse the old attribute calculations in the new ones.

Refining an attribute calculation is done by prefixing it with the **`refine`** keyword, followed by the name of the aspect that is being refined. The calculation done in the refine is no different than a regular attribute calculation, but it has access to the attribute calculation it is refining through the special `refined()` method call.

3.5 Further information

The description of JastAdd in this chapter is brief and does not provide a complete introduction to JastAdd. For further information on JastAdd, the JastAdd reference manual [2] is a good start. Some of the functionality is also described in tutorial form in [10].

Part II

The JPT implementation

Chapter 4

Main design

The JPT compiler is a source-to source compiler that translates a set of templates and packages to a program in standard Java. It is written as an extension of the JastAddJ compiler for Java, by using the mechanisms in the JastAdd tool. JastAddJ is a complete Java compiler, so the JPT compiler only provides the parts that are specific to JPT.

When I started working on the JPT compiler most major design decisions were already taken, and I would have to live with those decisions, good or bad. It was also decided early on to base the compiler on JastAddJ, rather than choose an open source Java compiler to extend. Also, most of the features of JPT were implemented, but the compiler was not tested to a degree that made it viable as a tool for testing the usefulness of the PT mechanisms.

The paper "Package Templates: A Definition by Semantics-Preserving Source-to-Source Transformations to Efficient Java Code" [5] describes a process of transforming a subset of the language accepted by the JPT compiler into standard Java code. Throughout this chapter and the next, we will use concepts described in that paper. We will also describe how the JPT compiler does the transformations compared to the transformations described in that paper.

This chapter will start off with a general overview of the design of the compiler. In the next chapter, there is a more detailed description of the part of the compiler which rewrites the AST, and also some discussions on what is missing in this part of the compiler and also some discussion on the choices that were made during its implementation. The changes I have made to the compiler will be discussed in chapter 6, and towards the end of that chapter we will look at some ideas for future work on the compiler.

The JPT compiler extends the JastAddJ compiler by making additions in the form of AST nodes with attributes, lexemes, non-AST classes, and methods added to existing AST nodes. Some parts of JastAddJ are also replaced, mostly through JastAdd's **`refine`** mechanism.

When the JPT compiler is started, it will generate an object of class `Program`. This object is the root node in the AST that will be built from the JPT code. This code to be compiled is located in files, and the name of these files are given to the compiler as command line parameters. Each of

these filenames are given to the `Program` node in turn, and the `Program` node will read the files and pass their content through the scanner and parser. The output from the parser for each template and package is an AST which will be added as a subtree to the `Program` node. After all files are parsed, all these subtrees will be part of an AST with the `Program` node as its root.

Thus, all packages and templates in the program will have their own subtree below the `Program` node, and the root of each subtree is either of class `CompilationUnit` or class `PTCompilationUnit`. The former is generated for files which contain standard Java code, and the latter is generated for files containing JPT code. Each of these compilation units then go through a *rewrite phase*. During this phase the compiler traverses the entire AST and transforms some of the nodes. These transformations are done using JastAdd's rewrite mechanism.

The largest of the JPT rewrites is the one where a node representing an **`inst`** statement is transformed into a (sub)tree that describes the content of the instantiated template with the adaptations given in the **`with`** clause and the addition classes. Thus, most of the JPT parts of the compiler revolve around this.

Inside the templates and packages, large parts of the code tends to be standard Java code, and these parts can be handled by JastAddJ, so what remains to do for the JPT compiler is the JPT specific parts of the code.

The result of the full rewrite phase (treating all **`inst`** statements) is an AST describing standard Java code (with a few small exceptions, which will be covered later).

The compiler finishes by creating output files that are placed in a directory structure defined by the package names. A simple build system using the Apache Ant tool [1] is also written to the output directory. The resulting files may then be compiled using Apache Ant and a standard Java compiler.

4.1 Open and closed templates and packages

An important part of template instantiations is the concept of *open templates* and *closed templates*, as well as *open packages* and *closed packages*. These are described for a subset of JPT in [5], but we will also give an overview of these concepts here.

4.1.1 Closed templates

A closed template is a template which does not instantiate any other templates. As such, it does not depend on any other templates, and may be type checked separately from the rest of the program, except for classes from imported packages. In [5] the content of a closed template is described as "a true subset of plain Java." This is true for the subset of JPT that is described in the paper, but it is not true for the complete language accepted by the JPT compiler. An example is required types, which may occur inside a closed template.

4.1.2 Open templates

An open template contains instantiations of other templates. Before any transformations can be done on an open template, all the templates it instantiates must be treated by the compiler and stored as a subtree for later use in instantiations of this template. To transform an open template into a closed template, the subtree of each instantiated template must be found and a copy must be made of it. Then this copy is inserted into the AST of the instantiating template, and renames, additions and merges are done according to declarations in the **`inst`** statement and the addition classes. As described above, all this is part of the transformation of the AST.

4.1.3 Open and closed packages

Open and closed packages are like open and closed templates in that open packages contain template instantiations and closed packages does not. In fact, closed packages are standard Java packages except for the special JPT package declaration syntax (**`package`** `{...}`). As with open templates, open packages cannot be transformed into closed packages until all templates they instantiate have been transformed into closed templates.

4.1.4 Transformation of a template hierarchy

Figure 4.1 shows how a template hierarchy is transformed into a single closed Java package. The arrows show instantiations. In (a) template `T1` and template `T2` are closed, since they are "leaf" templates, i.e they do not instantiate any other templates. Template `T3` instantiates both `T1` and `T2`, so it is open. Package `P` is also open, since it instantiates template `T3`. Since both templates instantiated by `T3` are closed, it is possible to start by transforming `T3` into a closed template.

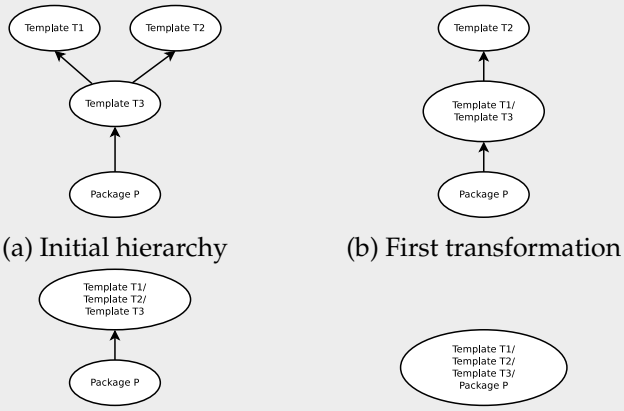
In (b), this transformation on `T3` has been done. All interfaces, classes, and enums have been copied from `T1`, and all renames declared in the instantiation clause for `T1` have also been done and the contents of the addition classes are inserted. `T3` is still open, since it still instantiates `T2`.

In (c), the contents of `T2` have been copied into `T3` in the same way. Now both instantiation clauses are gone from `T3`, and it has become a closed template. `P` is still an open package, but as the template it instantiates, `T3`, is now closed, and it can therefore be transformed in the same way as above.

In (d), the transformations are finished. `P` is now a closed package, and its content is standard Java code.

4.2 Directory structure for the JPT compiler

Figure 4.2 shows the main structure of the input to JastAdd for generating the JPT compiler. In the top-level directory of the directory tree shown in the figure, the JPT part is found inside the "PTFrontend" directory, while the JastAddJ part is found inside the appropriately named "JastAddJ"



(a) Initial hierarchy (b) First transformation

(c) Second transformation (d) Finished

Figure 4.1: Transformation of a template hierarchy. Rewriting the instantiations is the main part of the work the JPT compiler does. This is done for arrows in this figure, from the outermost point to the innermost.

directory. The Java 7 parts of the JastAddJ have not been included in the JPT compiler yet.

Figure 4.3 shows how the JPT compiler is generated. The first part of the figure (marked with the number 1) is the input (as shown in Figure 4.2). In part 2 the input is given to the build system, which uses JastAdd together with the Beaver scanner generator and the JFlex scanner generator to generate the JPT compiler. Part 3 is the generated code for the JPT compiler. The generated code can then be compiled into bytecode by using a Java compiler.

When work started on the compiler, the then current JastAddJ release was downloaded and placed in the "JastAddJ" directory, and at that time all the changes that were needed inside JastAddJ, could be given as standard JastAdd extensions. Over time, a few issues occured where there was no (reasonable) way to change parts of the JastAddJ code without operating directly on it. These changes needed to be preserved so it would be possible to replace JastAddJ with a newer version at a later stage. To preserve these changes, patch files, which could be applied to the JastAddJ code to redo the changes, were placed in the "JastAddJChangesLog" directory inside "PTFrontend".

4.3 Overview of a JPT compiler run

When running the JPT compiler, it goes through a number of steps, as illustrated in Figure 4.4.

1. JPT source code is given as input to the compiler in a number of files. This source code may contain packages using the standard Java package declaration syntax, and templates and packages using JPT

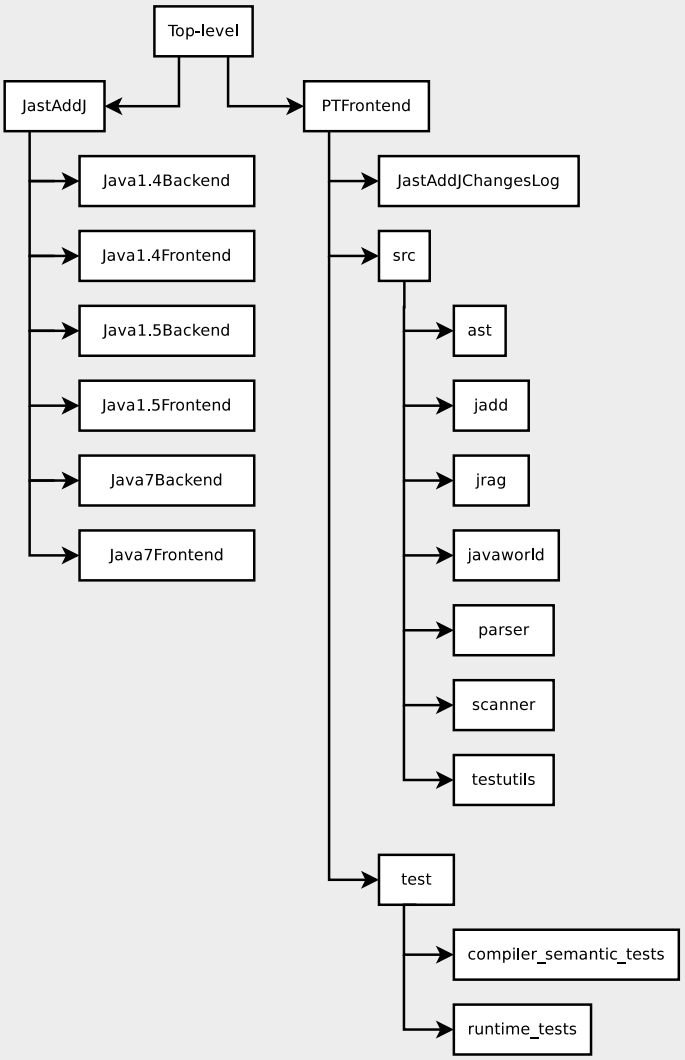


Figure 4.2: The directory structure of the input to JastAdd, when it generates the JPT compiler.

syntax.

2. All this JPT source code goes through the scanner and parser one file at a time. Source code from each file is turned into an AST which is then added as a subtree to the main AST (the AST representing the entire program).

3. When all files have been parsed the result is an AST containing nodes from standard Java (i.e nodes defined by JastAddJ), and nodes which are JPT specific. In this AST, instantiations of templates have not yet been performed, so the PT transformations, like renaming or merging, have not been done yet.

4. The AST goes through the rewriting process. This is done by traversing the tree. During this traversal, a node will be rewritten

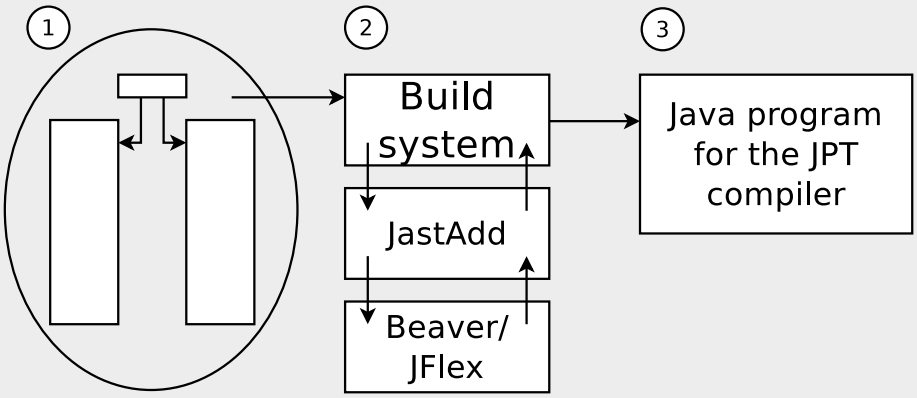


Figure 4.3: JastAdd with JFlex and Beaver is used to generate the JPT compiler.

when it is accessed through its parent, but only if all the prerequisites for rewriting are satisfied. Each time a package or template has been turned from open to closed, a semantic check is performed by the semantic checker of the JastAddJ compiler, which is extended so that it can also treat the JPT constructs that may occur in a closed package or template

5. The result of the full rewriting process is a tree which to a larger extent only contains standard Java nodes. As the output of the compiler is standard Java packages, the AST has been transformed so that it mostly does not contain JPT specific nodes. Some JPT specific nodes remain, but these will be handled by the compiler when it outputs the Java code.

6. The final step turns the transformed tree into textual Java packages, which is the output from the JPT compiler. To generate these Java packages, the AST is again traversed by using "pretty printer" methods on the AST nodes. These methods stem from the JastAddJ compiler, but they are slightly modified to also work for the remaining JPT specific nodes in the AST.

4.4 JPT additions to JastAddJ

4.4.1 Scanner and parser additions

Since JPT uses some keywords and operators that are not used in standard Java, the JastAddJ scanner is extended with additional lexemes. Figure 4.5 lists these additions.

As mentioned briefly above, the JastAddJ compiler (and thereby the JPT compiler) uses the JFlex scanner generator, which is easily extended with new lexemes. Additions to the scanner are simply added to a file which JastAdd puts together with the original input files to the scanner when the compiler is built.

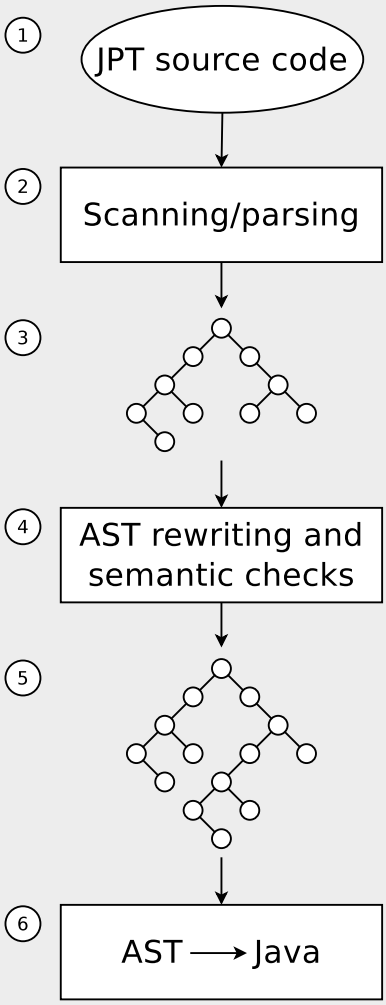


Figure 4.4: The process of compiling JPT source code to Java code.

In much the same way as the scanner, the parser generator also needs additions to its grammar for the JPT specific parts of the compiler. As with the scanner, these additions come in the form of a file which is appended to the original grammar files when the compiler is built. JastAddJ (and the JPT compiler) uses the Beaver parser generator, which is of the LALR type.

The addition file to the scanner generator is placed in the "scanner" directory in the source code directory hierarchy. Likewise, the addition file to the parser generator is placed in the "parser" directory.

4.4.2 AST additions

As explained in section 3.3 AST additions are descriptions of AST nodes used by JastAdd to expand the AST class hierarchy. The JPT compiler reuses some of the AST nodes from the JastAddJ compiler, and also adds many AST nodes of its own. Some of these are used directly during the building of the AST, while some are used during the transformation of the

template	**adds**	**inst**
with	=>	->
(*)	**tsuper**	**tabstract**
external	**assumed**	**required**
type	**subof**	

Figure 4.5: Lexemes added to the scanner by the JPT compiler

AST after the parsing has finished.

Like with the scanner and parser additions, additions to the AST are placed in a file which is then appended to the other AST descriptions by JastAdd when the compiler is built. These additions are placed in the "ast" directory in the JPT compiler's source code directory hierarchy.

A description of all the JPT additions to JastAddJ's AST is found in appendix A.

4.4.3 Aspects

A lot of the functionality of the JPT compiler is added through aspects defined in "jrag" and "jadd" files (These file types are described in section 3.4). All the jadd and jrag files for the JPT compiler are placed inside the "jadd" and "jrag" directories. The different aspects will not be described in detail here, because it would mostly just be a listing of method names, with a few comments.

4.4.4 Putting it all together

JastAdd takes the additions described in the previous sections, and weaves everything together into a compiler taking JPT code as input. The "testutils" directory contains the actual runnable part of the JPT compiler. The name "testutils" was given to that directory early in the development process, and it is not an appropriate name anymore, since it now contains much more than some utilities used for testing the implementation.

Inside "testutils" there is a class named `PTToJavaPackage` which contains the `main` method for the JPT compiler. This method will invoke the generated JPT complier, and compile JPT source code into standard Java source code, as described above.

4.4.5 Transforming the AST

The AST transformation step is JPT's largest addition to JastAddJ. All of the transformations that are made are triggered by JastAdd's rewriting mechanism. Since these rewrites are so extensive, they require a lot of code. Leaving all this code inside JastAdd rewrite clauses would make the rewrite methods very large, and thus hard to work with, so this rewriting code has been moved from these rewrite methods and into separate classes called *rewriter* classes. The rewrite methods then generate objects of these

classes and run a special `mutate()` or `run()` method on these, which will do the rewrite.

The source code for the rewriter classes are stored inside the "java-world" directory, alongside many other classes which are used by the rewriter classes during the transformations.

4.5 A note on access objects

The concept of *access objects* is used by the JastAddJ compiler and therefore also by the JPT compiler. An access object is created when one needs access to something that is declared elsewhere in the JPT/JastAddJ program. For instance, when the compiler reads the name of a method in the program, it will create a `MethodAccess` object which points to the declaration object (`MethodDecl`) of the corresponding method.

Every concept in the language which has this kind of interaction (where there is a declaration which is accessed through other statements) is represented by a `Decl` node type and an `Access` node type in the compiler.

Chapter 5

The rewrite phase

In this chapter we will discuss some details of the rewrite phase in the JPT compiler. As mentioned earlier, the rewrite phase is initiated after the AST for all packages and templates has been built by the parser. In Figure 4.4 the rewrite phase is numbered as step 4. After the full rewrite phase, the AST will contain mostly nodes which are also found in an AST built by JastAddJ for standard Java code, as most of the JPT specific nodes have been rewritten and transformed into standard Java language constructs.

The rewrite phase in the JPT compiler is initiated by a full recursive traversal of the AST. For this the `getChild` method is called at every level. As described in section 3.4.2, this method will trigger a rewrite if the preconditions for that rewrite are satisfied.

5.1 A note on PT declarations

In this chapter some of the description of what the JPT compiler does will apply equally to packages and templates. When referring to both packages and template we will use the term *PT declarations*. This term is interchangeable with the term "templates and packages".

5.2 The rewrite of one instantiation

The largest part of the rewrite phase is the rewrite of all the **`inst`** statements in the code. The basic step here is to treat one **`inst`** statement node in the AST by replacing it with a subtree representing the actual template that is instantiated, including the given adjustments. Each basic step must be repeated for all **`inst`** statements in the code. Since this basic step is so large, it is itself divided into four sub phases, which will be discussed in the following sections.

5.2.1 Adding "missing" rename clauses

The first sub phase of the basic step operates on the **`inst`** statement nodes, and can only be initiated after the template that is being instantiated has

been rewritten to a closed template. Its function is to add "missing" rename clauses to the **inst** statement. The "missing" rename clauses are the ones where classes are not renamed, so the rewrite consists of "renaming" these classes to the same name they had. After this sub phase, the **inst** statement node will contain rename clauses for every class, interface, and required type in the template it instantiates.

5.2.2 Rewriting the PT declarations

The precondition to this sub phase is that "missing" rename clauses have been added to all the **inst** statement nodes in the PT declaration that is to be rewritten, i.e the previous sub phase has been done for all the **inst** statement nodes. At this stage all the templates that are being instantiated in this PT declaration have been closed. When this sub phase has finished the PT declaration that is being rewritten will also be closed.

This sub phase is, by far, the largest of the sub phases, so the discussions of it will be divided into discussions of each step that is performed during the rewrite.

Find virtual methods

The first operation done in this sub phase involves determining which methods are *virtual* and which are *overrides*. Virtual methods are methods that appear for the first time in a class hierarchy, i.e that the method's signature does not appear in any superclasses. *Overriding* methods are methods which, as the name says, overrides methods declared in a superclass. The overridden method may itself be an override, or it may be a virtual method.

To find the virtual methods the compiler loops through each class in the PT declaration, and for each method in the class, looks upwards in the class hierarchy for methods matching the method's signature. If no methods matching the signature is found, the method is marked *virtual*.

The reason for marking the methods as virtual is to prevent virtual methods from becoming overrides when classes are merged. An example of this is given below.

```
template T1 {
  class A {
    void f() { ... }
  }
  class B extends A {}
}
template T2 {
  class C {
    void f() { ... }
  }
}
package P {
  inst T1;
```

```
  inst T2 with C => B;
}
```

This could translate into the following Java code, but note that an error message is issued before that code is generated.

```
package P;
class A {
  void f() { ... }
}
class B extends A {
  void f() { ... }
}
```

The problem in the above example, is that the method `f()` in `C` is a virtual method. When `T2` is instantiated in `P`, `C` is renamed to `B`, and thereby merged with class `B` from `T1`. `B` in `T1` is a subclass of `A` which also has a virtual method `f()`. The result of merging `C` and `B` is a subclass of `A`. This makes the method `f()` that came from `C` an override of method `f()` in `A`. Since the method was virtual in `C` before the merge this is an error.

Create a new `PTDeclRew` object

The `PTDeclRew` class is a helper class for rewriting a PT declaration. An object of type `PTDeclRew` contains a reference to the AST node of the PT declaration that is being rewritten. The following steps in the PT declaration's rewriting process are done by calling methods on this object.

Check the type parameters

This step involves the "old style" type parameters, which have been replaced by required types. However, these parameter, though deprecated, are still handled by the JPT compiler, so we will also describe this step.

The checks done in this step are:

- Check that the number of actual parameters matches the number of formal parameters.
- Check that each actual parameter is a class.
- Check that each actual parameter satisfies the constraints set by the corresponding formal parameter.

Each of the type parameters are added to an object which contains mappings from formal parameters to actual parameters. This object will later be used to replace all accesses to the formal parameters with accesses to the actual parameters. After all type parameters are added to this object, a reference to it is kept internally in the `PTDeclRew` object.

Add enums to the PT declaration

The next step is to find all enums in the instantiated templates in the PT declaration, and add them as enums to the PT declaration. In the compiler, enums may be renamed, but not merged. Therefore the compiler can just copy the enums directly from the originating template, and rename the copied enums according to the rename clauses in the **`inst`** statement.

Create "missing" addition classes

The compiler creates empty addition classes for all classes in the PT declaration that do not have an explicit addition class. An explicit addition class is one that was defined in the input to the compiler using the **`adds`** keyword. Addition classes form the base for what will be the finished classes in the PT declaration when it has been closed, so all classes need to have a corresponding addition class.

If the class is being renamed, the addition class will get the new name, so this step introduces the first part of class renaming. The addition classes that are created will contain empty constructors.

Merging of interfaces and of required types

The compiler treats the merging of interfaces and the merging of required types as two steps. However, the only difference between the two steps is that one is for interfaces and one is for required types, so we will describe both steps in this subsection. We will mainly describe how the merging is done for interfaces, but this description can also be directly applied to required types.

This step starts with something similar to what the previous step does; it creates addition interfaces for the interfaces that do not have explicit addition interfaces. The addition interfaces are used as base interfaces for the merging. When looping through the instantiations, fields and methods from each interface in an instantiation is added to the corresponding addition interface.

Before adding the fields and methods from an interface to its addition interface, the interface is copied and all renaming is done on the copy. In this way, we ensure that everything that is added to the addition interface is renamed. The renaming is done by first finding the declaration to be renamed, and then finding all the places where a name refers to this declaration, and finally rename all these.

Merging of classes

Merging of classes is a more complex process than the merging of interfaces and required types, so this step is larger than those steps. Similar to interfaces and required types, this step operates on addition classes. The "missing" addition classes were added in an earlier step, so in this step all the classes already have corresponding addition classes.

Also, here we must follow the rule that a class cannot be handled before its superclass has been handled. This is solved by marking classes as "visited" when they have been handled. Before a class is handled, a check is done for whether its superclass has been "visited". If it has not, the handling of the class is deferred and it will try again later.

The first part of this step is the renaming of the original classes and their methods and fields. This is done in the same way as for interfaces and required types; a copy of the class is made and the renaming is done on the copy.

After the renaming has been done, the compiler will loop through all the methods in each class and check if there are any naming conflicts. If there is a conflict, the compiler will report it as an error.

The next action is to set the correct superclass for the addition class. The compiler here also checks for multiple superclasses. Since each class's superclass has been fully handled before the class itself is handled, there should only be one superclass left, because if there were more they should be merged by now. If there are more superclasses, the compiler will report it as an error.

The implemented interfaces must also be set for the addition class. This is less complicated for superclasses, as classes may implement several interfaces. However, there is a possibility that two merged classes implement the same interface. In that case the interface could be listed twice, which would give an error in the generated Java code, so this is handled by setting the addition class to implement a union of all the interfaces implemented by the merged classes.

The compiler will then loop through each original class and check if it is abstract. If any of them are abstract, the corresponding addition class is also set as abstract.

Next, if there are any formal type parameters with corresponding actual type parameters, all accesses to the formal type parameters are replaced with accesses to the actual types.

The final part of this step is to add all methods and fields from the original classes to the addition class. All renames was done earlier in this step, so there are no name collisions at this time. Any constructors in the original class are also copied to the addition class. These constructors are transformed into regular methods and are given unique new names before being copied.

Name resolution

As the renamings have now been done, the JastAddJ part of the compiler can now resolve a special type of ambiguous accesses that may occur. These are accesses where it is unclear whether they access a package or a type.

An example of where an access becomes ambiguous is when a static method in a class from an instantiated template is accessed. The method is accessed using the `<classname>.<methodname>` syntax. Before the addition classes are present there is no declaration of the class, so the compiler doesn't know if the name belongs to a package or a type.

When the addition classes have been created and all methods and fields have been copied, these ambiguous accesses may be resolved. This is done by JastAddJ in a traversal of the subtree of the PT declaration node.

Copy import statements

As each of the instantiated templates may have had import statements, these must be copied to the PT declaration that is being rewritten in order for all parts of the instantiated templates to work.

Add `tsuper` calls in the generated constructor

If the constructor was generated for the addition class, a call to the method that was created from the original class's constructor is added.

Concretize required types

Now that all types are present in the PT declaration, it is possible to concretize the required types. The names of the required types that have concretizations are replaced with the names of their concretizations. All the accesses are then updated to reflect the name change. If the PT declaration is a package, and there are required types without concretizations, the compiler will try to create default concretizations.

5.2.3 Rewriting the addition classes

After a PT declaration has been rewritten, all its addition classes will be rewritten into AST nodes of type `PTClassDecl`. The nodes are not turned into standard Java class nodes yet, as they may still contain **`tabstract`** methods.

5.2.4 Rewriting accesses to `tsuper` methods

In the addition classes there may be accesses to **`tsuper`** methods. When the addition class nodes have been rewritten into `PTClassDecl` nodes, the **`tsuper`** methods are copied into the `PTClassDecl` nodes they are accessed from, and then given a unique name. The name is generated using the name of the originating template, the name of the originating class (before renaming), and the method name itself. In this rewriting the name gets the form **`tsuper`**`[<TemplateID>.<ClassID>].<MethodName>`. This name will again be changed before generating the output from the compiler, since a simple method name cannot contain brackets or dots. How this renaming is done is discussed in section 6.4.1.

5.2.5 Discussion of the rewrites

The rewrites that have been discussed in this section, are similar to the transformations described in [5]. These transformations are:

1. The fortifying transformation
2. The renaming transformation
3. The addition-handling transformation
4. The composing transformation

The JPT compiler does not implement all the transformations described in the article, and some of the transformations are done slightly different.

The fortifying transformation

Only a few of the steps in the fortifying transformation are implemented in the JPT compiler. Some of the known bugs in the compiler are results of these missing steps of the transformation.

The first step in the fortifying transformation is to give variable declarations within a method, and the method's formal parameters, unique names. This step is not done in the JPT compiler. The following program is an example of a bug that is the result of this missing step.

```
template T {
  class C {
    int j = 0;
    void f(int i) {
      j = i;
    }
  }
}
package P {
  inst T with C => C (j -> i);
}
```

Not renaming the formal parameter `i` in the method `f()` will cause the assignment in the method to become `i = i`. Obviously this is not what was expected, so this is a bug in the compiler.

The second step in the fortifying transformation involves adding the `@override` annotation to all methods that are overrides of other methods and giving virtual methods in anonymous classes unique names. The first part is very similar to the marking of virtual methods that is done by the compiler. However, the compiler only does the marking of the methods and does not give unique names to any methods. The lack of renaming leads to similar problems as described in the example for the first step.

Marking the virtual methods rather than the overridden methods introduces a superfluous concept to the compiler. It would probably be better to do as described in [5]. This would re-use an existing concept (`@override` annotations) rather than adding a new one. If this change is done, it is possible to decide that a method is virtual by verifying that it is not an override.

The third step in the fortifying transformation is to introduce explicit casts to the formal parameters of methods. This is done to avoid

unintentional overloads. In the JPT compiler this is done as part of the merging of classes.

The fourth step in the fortifying transformation changes some unqualified accesses to qualified accesses, so renamings will not change the semantics of these accesses. This step is not implemented in the JPT compiler. The reason why I have not corrected these errors is simply lack of time (but see 6.6.1).

The renaming transformation

The renaming transformation provides preconditions that must be satisfied before a renaming is done. All but one of these preconditions are checked by the JPT compiler.

The precondition that is not tested for is the following: A precondition for renaming a class is that, if a template `T` is instantiated, a class `C` in `T` cannot be given the same name as another class in `T` in an instantiation clause. Since the JPT compiler allows merging of classes from the same template, this precondition is not possible to satisfy without removing that functionality.

The addition-handling transformation

The addition-handling transformation involves two steps. One is to give unique names to any variables in a class that have the same name as a variable declared in the corresponding addition class. The other is to mark methods which become overridden in addition classes with the `@TOverridden` annotation.

None of these are done in the JPT compiler. However, the second one is handled in another way. The compiler keeps track of which methods have been overridden in an addition class by keeping references to them in a set. I consider the compiler's way of doing it as (technically) better than using the annotation, since it provides quicker access to the methods than one would have if it was necessary to look through all methods and sort out those with the `@TOverridden` annotation. However, for the purposes of describing this transformation step in a clear way, the `@TOverridden` approach may be better.

The composing transformation

The preconditions in the composing transformation are all checked by the JPT compiler. In the steps of the transformation described in [5], however, there are some differences.

The first step in this tranformation says that all the methods that are overridden in an addition class should be deleted. This was OK in the simplified version of JPT treated in [5], but because it is possible that the overridden methods are called using the **`tsuper`** keyword, they are given a unique name rather than being deleted. However, it should not be a

problem to look through each of the **`tsuper`** method calls and find out which methods are actually called.

The next step is to create "missing" addition classes, but the JPT compiler does this early in the PT declaration rewriting. The compiler also performs the two last steps as described in [5].

Chapter 6

My work on the JPT compiler

Because of the complexity of the JPT compiler, a lot of my time has been spent studying its code to get an understanding of how it works. So, even though some of the work that we will look at in this section may look like small changes, they did consume a lot of time since I was mostly on my own in figuring out where in the code the changes would go, and how the different parts of the compiler would be affected by the change.

Before going into the details of the work I have done on the compiler, there are some terms that should be clarified to avoid confusion. The compiler as it was when I started working on it will be referred to as the *original* implementation. When describing that compiler, we will use the phrases *originally* and *used to*. As an example, the sentence "originally, the compiler used to work with this particular kind of statement", describes a feature of the compiler before I got involved in its development. The compiler in its current form will be referred to as "the JPT compiler" or just "the compiler".

6.1 Testing the implementation

The test system for the JPT compiler consists of two sets of test programs (*compiler semantic tests* and *runtime tests*), a Java program that runs the tests from one of the sets, and a Python script that runs the tests from the other set. These sets are placed in the "test" directory in the compiler's directory tree.

Compiler semantic tests

The "compiler semantic tests" check that the compiler performs as expected internally. If the compiler reports errors when trying to compile a test, or if an exception occurs during the compilation, the test has failed. Some tests are supposed to fail, e.g tests that check if the compiler detects semantic errors in the code. These tests have filenames that end with "_fail", and the test will be considered to have failed if it is compiled without errors. Figure 6.1 shows an example of a test program from the "compiler semantic tests" directory.

```
template SimpleTemplate {
  class A {
    int k;
    assumed A();
    A getA() {
      return new A();
    }
}
```

Figure 6.1: An example of a simple test program from the "compiler semantic tests" directory.

These tests are run by issuing the command "ant testall" from the "PTFrontend" directory. This command starts a Java program called "TestScenario", which is part of the "testutils" package. This program will run the compiler on each of the test programs in the "compiler_semantic_tests" directory. It does not inspect the output of the compiler, it only checks if the compiler completed successfully or failed with an error message or an exception.

One weakness with the "TestScenario" program, is that it is not possible to specify how a program that is supposed to fail should fail. So if a program is supposed to trigger a specific error, but triggers another one, the program will report the test as passed, even though the error was not the one that was expected.

Runtime tests

The "runtime tests" will check the result of compiling and running the test programs. A typical runtime test will be a program that outputs some characters when run. The output of the test program is then compared to the contents of a file with the expected output of the test program. If the comparison does not result in an exact match, the test has failed. Figure 6.2 shows the source code of a simple runtime test.

These tests are performed by running the python script named "runtime_tests.py", which resides in the "PTFrontend" directory. The script will run the test programs in the "runtime_tests" directory. For each test program the script performs four steps:

1. Run the JPT compiler with the test program as input.
2. Run a Java compiler on the generated code.
3. Run the program generated by the Java compiler.
4. Compare the output of the program to the expected output.

```
package hello_world {
  class Main {
    public static void main(String[] args) {
      System.out.println("Hello, World!");
    }
  }
}
```

Figure 6.2: An example of a simple runtime test. A text file containing the text "Hello, World!" accompanies this program, which is compared to the actual output of the program when it's run.

6.1.1 Tests added during my study of the compiler

While I was studying the compiler I discovered a number of bugs in the implementation. For each of these bugs I wrote a small JPT program which would either fail to compile, or would produce erroneous output when compiled, because of the bug. These programs were added to the test system for the compiler. We will not look at every test program I added, but to get a sense of them, we will look at two examples.

One of the missing features of the JPT compiler is template parameters (see section 6.6.2). As soon as the details of how the template parameters should work was beginning to take shape I wrote a number of test programs in which template parameters are used. As an example we will discuss one of them. The test program is shortened below, as not all parts are relevant to the discussion. This test program is expected to fail.

```
template T {
  class A {
    void m() { System.out.println(``m()''); }
  }
}
template T2 subof T {
  class A adds {
    void n() { System.out.println(``n() in T2''); }
  }
}
template U<template V subof T> {
  inst V;
  class A adds {
    void n() { System.out.println(``n() in U''); }
  }
}
package P {
  inst U<T2>;
}
```

What happens here is that when the `n()` method is added to `A` in `T2` it causes a name collision when `T2` is used as a template parameter for `U`, since `U` also adds a method named `n()`. When running the test it is currently always reported as passed, as the compilation fails. However, the reason for the failure is because template parameters are not implemented, not because there is an error in the program.

The second test program shown here is one I wrote to trigger a bug in the merging algorithm.

```
template T {
  interface FooRunnable {
    void fooRun();
  }
  class A implements FooRunnable {
    public void fooRun() { }
  }
}
template U {
  interface BarRunnable {
    void barRun();
  }
  class B implements BarRunnable {
    public void barRun() { }
  }
}
package P {
  inst T with FooRunnable => MyRunnable, A => AB;
  inst U with BarRunnable => MyRunnable, B => AB;
}
```

In this program the interfaces `FooRunnable` and `BarRunnable` are merged, and the classes `A` and `B` are merged. This used to fail because the compiler did not correctly check for duplicates in the **`implements`** clause, so the `MyRunnable` interface was listed twice in the resulting class `AB`. How this was fixed is described in section 6.5.

6.2 Upgrading to a newer version of JastAdd and JastAddJ

Both JastAdd and JastAddJ has had several releases since work began on the JPT compiler, and the JPT compiler had not been upgraded with any of these new versions. For reasons that will be explained below, I decided to upgrade the JPT compiler with what was then the newest version of JastAddJ. This new version of JastAddJ also included a newer version of JastAdd.

This upgrade was the most time consuming part of my work with the JPT compiler. Since no such upgrade had been done on the JPT compiler before, it was a large change to the code base. Thus several parts of the

compiler were affected, and I had to use a lot of time to understand each of these parts, and how to change them so that they worked as they did before the upgrade.

Throughout this process I have become unsure if using JastAddJ as a base for the JPT compiler was the right decision. JastAdd is a complicated system in itself, so before one can begin to understand the code of the JastAddJ and write JPT additions to it, one must spend a lot of time learning JastAdd. If the JPT compiler had been made as an extension of an open source compiler, without the use of a system like JastAdd, the process of understanding how the system is built would probably have been much quicker.

However, the JastAddJ compiler is well designed. The implementation closely follows the Java language specification [9], so when one has an understanding of the JastAdd concepts, it is easy to read and understand the JastAddJ code. With such an understanding from the start of the project, the choice to use JastAdd would probably be the right one.

Another problem with JastAdd and JastAddJ is that it was not as finalized as we assumed. As mentioned above, when I upgraded the JastAddJ version used by the JPT compiler, I discovered that many of the JPT additions to JastAddJ would not work anymore because of changes to basic functionality both in JastAdd and JastAddJ. These were well-reasoned changes seen from JastAdd and JastAddJ, but still this kind of instability somehow defeats the purpose of having an extensible compiler as a base. If JastAdd and JastAddJ had been more mature, the chances of such large changes being made would be smaller, and would not have been such a problem. The effects of this upgrade are still causing problems in the JPT compiler, and this will also be discussed further later in this chapter.

The upgrade process would probably have been easier if it had been done more often, as there would then be a smaller number of changes in each upgrade. With less changes it would be easier to find out exactly what had changed since the last upgrade, and thus made it easier to make the necessary changes to the JPT part of the compiler. When I started the upgrade, several thousand lines of code were changed in JastAdd and JastAddJ, so it was not possible to get a complete understanding of all these changes in a reasonable amount of time.

6.2.1 The reason for the upgrade

JastAddJ comes complete with a pre-compiled version of JastAdd. The JastAdd package that came with the version of JastAddJ used in the original JPT compiler turned out to have a bug which led to an error in the code generated from JastAdd for the JPT compiler. The following shows the problematic part of the generated code.

```
public class ASTNode<T extends ASTNode> extends beaver.Symbol
                      implements Cloneable, Iterable<T> {
  ...
  private int childIndex;
```

```
  ...
  public void setChild(T node, int i) {
    ...
    node.childIndex = i;
  }
  ...
}
```

When compiling this code, newer versions of the OpenJDK Java compiler, failed with the error message "childIndex has private access in ASTNode", while older versions allowed this method to compile.

I suspected that this was a bug in JastAdd and not in the JastAddJ part of the compiler. To verify this I first tried to replace the JastAdd version that came with the original JastAddJ with a new version. This did fix the problem, but I did not consider it a satisfactory solution because JastAddJ is distributed with a specific version of JastAdd, and it is probably better that these two together work correctly as a basis for the JPT compiler. The solution was then to upgrade the JPT compiler to use a newer version of JastAddJ, which also included a newer version of JastAdd.

However, when upgrading the JPT compiler to the new version of JastAddJ, there were some problems due to the aforementioned changes in both JastAdd and JastAddJ. Almost all the test programs failed after the upgrade, so I had to make changes to the JPT part of the compiler to make it work as it did before. These changes will be discussed in the next few subsections.

6.2.2 Omitted return statements

In the old version of JastAddJ, when omitting the return statement from an inherited attribute calculation, the generated code would keep traversing the AST upwards, and doing the attribute calculation until it found a return statement. This was a bug in JastAddJ, and was consequently removed from the new version.

However, the JPT compiler had exploited this bug, as the previous developers had believed this to be expected behaviour from JastAddJ. Thus, the inherited attributes that exploited this had to be changed before the new version of JastAddJ would work. I fixed this by writing the AST traversal code in the affected calculation manually.

6.2.3 Handling of primitive types

The next problem that arose was that the handling of primitive types had changed in the new version of JastAddJ. Previously, primitive types were added to each compilation unit by calling a method called `addPrimitiveTypes()`. The new version had removed this method, and replaced it with a method to lookup primitive types, called `lookupLibType()`. This method is then called from the `lookupType()` method when it fails to find the type it's looking for in its local scope. Fixing the JPT com-

piler to work with this was a simple matter of removing the call to the `addPrimitiveTypes()` method and add the `lookupLibType()` method.

There are still some remnants of the old `lookupType()` implementation left in the JPT compiler. When I tried to remove these, a few of the tests that used to pass started failing, so I reverted the change. More work is needed to fix this this part of the JPT compiler, but for this thesis I chose not to prioritize this work since the tests pass with the old code still there.

6.2.4 Changes to the `fullCopy()` method

In JastAdd each AST node is given methods to allow the node to be copied. One of these methods, called `fullCopy()`, copies the entire tree below the node it is called from. This method is used several places in the JPT compiler. In the new JastAddJ version, a subtle change was made to this method: the (new) root node of the copy, had its parent node reference set to **`null`** to sever its ties to the tree it was copied from. In the old version it pointed to the parent of the old root node. For some of the usages in the JPT compiler, this link to the old parent was used after the copy was made. This was resolved by creating a new method, called `fullCopyWithParent()` which would save the parent of the AST node before calling the `fullCopy()` method, then setting the parent of the copy to the saved parent before returning the copy.

This change affected every use of the `fullCopy()` method, but the link to the old parent was not used at every point where a copy was made using this method. Before replacing each call to the `fullCopy()` method, I first verified that the link was actually needed. For those calls where it was not needed, I did not change the method call.

The problem with keeping a reference to the parent is that the AST's generated by JastAdd keeps references both ways, i.e the parent keeps references to its children, and a child node keeps a reference to its parent. When the reference to the parent node is set after the `fullCopy()` method, the copied child node will have a reference to the parent node, but the parent node will still only reference the original node. This inconsistency could typically lead to errors in future changes.

6.2.5 Performance issues after the upgrade

When upgrading the JPT compiler to the newer version of JastAddJ, some major performance issues surfaced. It is not yet known from which part of the JPT code these issues stem from. Performance profiling did not show that any part of the compiler used more time than other parts, but it did show that some methods were called many times. Two of the methods that are called many times are the `lookupType` method and the `getChild` method.

The performance problem does not occur on the test programs which are part of the current test suite for the compiler. It seems to only happen on larger programs, so trying to figure out the source of the problem is made harder because the program where it occurs is too large for it to be feasible to run through the compilation process with stepwise debugging.

The size of the program which triggers the performance issue also makes it hard to know if the methods that are called many times are called *too* many times, or if all these method calls are parts of expected behaviour.

6.3 Method naming conflicts resolved by overriding

The JPT compiler used to allow method name conflicts in merged classes to be resolved by overriding the method in an addition class. So the following would be legal:

```
template T {
  class A {
    void f() { ... }
  }
}
template U {
  class B {
    void f() { ... }
  }
}
package P {
  inst T with A => AB;
  inst U with B => AB;
  class AB adds {
    void f() { ... }
  }
}
```

Here we see that the addition class in the package `P` overrides the `f()` method. This overriding method would then replace both the `f()` method from class `A` and the `f()` method from class `B`. This, however would cause problems with ambiguous **`tsuper`** statements, as discussed in [6]. Because of these problems(and some others), resolving naming conflicts by overloading in an adds class was disallowed, so this had to be removed from the compiler. To do this I added a new method to check for these naming collisions and produce an error message when they occured. This was rather straight-forward to do, when I understood what was going on.

6.4 Qualified names for Packages and Templates

When I started working on the compiler, it would only accept names for templates and packages that consisted of alphanumeric characters and underscores. Because of this it was not possible to have the hierarchical structure that is normally used for organising source code in a Java project, due to the fact that names could not contain dots. Adding the possibility for package hierarchies was less complicated than adding template hierarchies, so that was done first.

In the parser, the terminal used for the name was "IDENTIFIER". This terminal can only consist of alphanumeric characters and underscores, so a package or template name containing dots would result in a compilation error. As JastAddJ already contained the non-terminal "name_decl", which allowed for names with dots (called *qualified names*), changing the parser to allow qualified names, was simple.

It was also necessary to change the way the final output from the compiler was written. The standard way of organizing the source code in Java is to have the directory structure reflect the package name. So if a package is called "a.b.c", it should be stored in the directory "a/b/c". There was already a method for determining the path of an output file to be written, and it was being used by the Java code generator in the compiler. This method would not replace dots with path separators, so it would create a single directory for the package which would be named with the package's qualified name. The solution to this was simply to write a specialized method for generating package paths that replaced all dots with path separators, and change the Java code generator to use this new method.

6.4.1 Qualified template names

The template names also used the aforementioned "IDENTIFIER" terminal. However, unlike package names which only appears in package declarations, template names also occur in the instantiation statements. Therefore the parser needed more changes to make qualified template names work than it needed to make qualified package names work.

In addition to the parser changes, allowing qualified template names caused an error with the **`tsuper`** statements. The **`tsuper`** statements are not translated to standard Java code until right before the code is written to file.

The compiler handles the **`tsuper`** methods by creating a unique name for them. This name is constructed using the word "tsuper", the original name of the class qualified by the the original template name, and the method name itself. In these generated names, there are some characters, like brackets and dots, which are not legal characters for method names in Java. These characters are substituted textually from the code just before it is written to the output file. This string substitution did not allow dots in template names, and the way the string substitution was implemented made it hard to add support for them without changing the whole approach.

At first I tried to fix this with a string search operation that recognized the string "tsuper", and then did the substitutions on the string that followed it, until it reached the method name. However, there were some special cases where this approach would do string substitutions where they were not needed, so I had to develop another approach.

The new approach does the string substitutions in the same way as the previous approach, but instead of running on the entire output string of the compiler, it only operates on AST nodes which may contain **`tsuper`** method names.

6.5 Duplications of implemented interfaces in merged classes

In this section we will look at how the problem described in the presentation of the second test in section 6.1.1 was fixed.

The JPT compiler uses *type descriptors* to store information about types (i.e classes and interfaces). These type descriptors contain, among other things, all the information needed to compare two types to each other.

When classes are merged the compiler needs to make sure that the list of implemented interfaces for the new (merged) class contains all the implemented interfaces of the original classes, and that there is only one mention of each interface. This is done by creating type descriptors for all the implemented interfaces of each class, and add these type descriptor to a set. Since a set can only contain unique values, there will not be one element in the set that is equal to another element in the set of type descriptors.

However, the compiler would still add the same interface several times to the implements list of a merged class if it was implemented by more than one of the original classes. This was caused by a subtle bug in the type descriptor class; it did not overload the `equals` method correctly. The `equals` method in the type descriptor class took a reference to a type descriptor as its formal parameter. However, to overload the `equals` method from the `Object` class correctly, the formal parameter must be of type `Object`.

Because of this wrong overload, the set used the `equals` method from the `Object` class when comparing two type descriptors. Thus they were not compared correctly, and type descriptors that described the same interfaces were not considered equal to each other.

I fixed this bug by writing a correctly overloaded `equals` method. This equals method checks if the actual parameter is a type descriptor, and if it is, it calls the incorrectly overloaded `equals` method.

6.6 Future work

In this last section, we will go through some topics that are not directly related to those discussed above, but that could be of interest for the further development of the compiler.

6.6.1 Refactor the rewrite phase

The different transformations in [5] provide a good base for how the JPT compiler could work. As discussed in section 5.2.5 the compiler does not perform all the steps in these transformations, and that is the cause of some bugs. These bugs have been known for a while, but I was not able to give them priority, as I had to spend much time on other issues, like the JastAddJ upgrade and the performance issues.

These bugs should be quite easy to fix, but I think that fixing them should be part of a larger refactoring of the rewrite phase, to make the compiler follow the transformations in [5] more closely. This would be

a larger amount of work, but the result would be that the rewrite phase would have a cleaner structure than it currently has. This would make the compiler's code more comprehensible and easier to work with.

6.6.2 Template parameters

Template parameters are currently not fully implemented the JPT compiler; the parser accepts the code, but the compiler will throw an exception and stop at a later stage. The details of how template parameters should be implemented were not finished in time for me to be able to implement them fully in the compiler, so currently there is only this partial implementation.

6.6.3 Improved error messages

The JPT compiler should always produce a correct Java program or it should issue an error message. Thus the final pass of the JPT compiler (to translate the Java version to bytecode) should not cause any problems during normal compilations. However, we may indeed get runtime errors when the resulting program is executed, and in order to make it easier to use JPT, it would be desirable to have these error messages refer to which template, and which line in the template, the error originated from.

There are some ways this could be done. One of them is to make the JPT compiler output bytecode rather than Java code (see below).

Another way to do it would be to have the JPT compiler generate a file which contains some kind of mapping from the generated Java code to the corresponding part of the JPT code. It would then be possible to create a wrapper program for the Java Virtual Machine (JVM) which would inspect the messages from it and replace the error messages. It could also keep the original error messages and add new messages explaining where the Java code originated from.

Such a wrapper program would have to "understand" the error messages from the JVM, so if the messages are changed from one version of the JVM to the next, the wrapper program would have to be updated. Thus, a problem with such a wrapper program is that it would probably need a lot of maintenance.

6.6.4 Compile directly to bytecode

We would get several benefits if we could generate Java bytecode directly from the JPT compiler. One of them would be that the JPT compiler would have more control over error messages generated by the compiler, which would make it easier to give improved error messages at runtime, as the correct line references etc. could be inserted into the bytecode.

Another benefit would be the reduced overhead during compilation. As it is now, the JPT compiler writes the files containing Java code, and this code then has to be compiled by running a Java compilation.

The fact that every JPT specific part of the code must be changed into regular Java code also introduces overhead. If the AST nodes which are

JPT specific could be translated directly to bytecode, this overhead would be reduced quite a bit, because less AST transformations would have to take place.

JastAddJ is a complete Java compiler, so it already has support for generating bytecode. The JPT compiler only needs to extend these parts of the JastAddJ compiler to support the JPT specific parts of the AST. As I have not been studying the code generation part of JastAddJ, I am not able to estimate how big a job it would be to do this.

Bibliography

[1] Apache ant - welcome. http://ant.apache.org/.

[2] JastAdd.org. http://jastadd.org/web/documentation/reference-manual.php#jadd.

[3] Eyvind W. Axelsen and Stein Krogdahl. Groovy package templates: supporting reuse and runtime adaption of class hierarchies. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, page 15–26, New York, NY, USA, 2009. ACM.

[4] Eyvind W. Axelsen and Stein Krogdahl. Adaptable generic programming with required type specifications and package templates. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, page 83–94, New York, NY, USA, 2012. ACM.

[5] Eyvind W. Axelsen and Stein Krogdahl. Package templates: a definition by semantics-preserving source-to-source transformations to efficient java code. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, page 50–59, New York, NY, USA, 2012. ACM.

[6] Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl, and Birger Møller-Pedersen. Challenges in the design of the package template mechanism. *Transactions on Aspect-Oriented Programming [to appear]*, 2012.

[7] Torbjörn Ekman and Görel Hedin. Modular name analysis for java using JastAdd. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, number 4143 in Lecture Notes in Computer Science, pages 422–436. Springer Berlin Heidelberg, January 2006.

[8] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, 42(10):1–18, October 2007.

[9] James Gosling. *The Java language specification*. Addison-Wesley, Upper Saddle River, N.J., 2005.

[10] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering*

III, number 6491 in Lecture Notes in Computer Science, pages 166–200. Springer Berlin Heidelberg, January 2011.

[11] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.

[12] Håkon Stordahl. BooPT: implementasjon av package templates for boo. April 2012.

Appendix A

JPT AST additions

PTCompilationUnit : CompilationUnit A compilation unit that contains PT declarations, i.e template declarations or package declarations using the PT package declaration syntax.

abstract PTDecl A PT declaration, i.e the base node for templates and packages.

PTTemplate : PTDecl A template declaration and its body.

PTPackage : PTDecl A package declaration and its body.

PTDeclContext Context information for PT declaration. Currently this node is not used for anything, although every PT declaration gets its own `PTDeclContext` object.

abstract SimpleClass : ClassDecl Superclass for class declarations inside a PT declaration and addition classes.

PTClassDecl : SimpleClass A class declaration inside a PT declaration.

PTClassAddsDecl : SimpleClass An addition class declaration.

PTInterfaceDecl : InterfaceDecl An interface declaration inside a PT declaration.

PTGenericInterfaceDecl : GenericInterfaceDecl A generic interface declaration inside a PT declaration.

PTInterfaceAddsDecl : PTInterfaceDecl An addition interface declaration.

PTGenericInterfaceAddsDecl : PTGenericInterfaceDecl A generic addition interface declaration.

PTEnumDecl : EnumDecl An enum declaration inside a PT declaration.

TemplateClassIdentifier An identifier for templates used in e.g **`tsuper`** calls. The `TemplateClassIdentifier` node is the what is put in brackets when qualifying a **`tsuper`** call, e.g **`tsuper`**`[T.A]()`.

PTInstDecl A template instantiation clause.

RequiredTypeInstantiation A concretization of a required type.

PTInstTuple A rename clause.

PTTSuperConstructorCall Call to a **`tsuper`** constructor.

PTConstructorDecl : ConstructorDecl A constructor declaration inside the body of a PT class declaration.

PTConstructorPromise : BodyDecl An **`assumed`** constructor declaration.

abstract PTDummyRename Base node for the method and field rename clauses.

PTMethodRename : PTDummyRename A rename clause for a method.

PTMethodRenameAll : PTDummyRename A rename clause for several methods using the wildcard parenthesis mechanism.

PTFieldRename : PTDummyRename A rename clause for a field.

abstract PTMethodAccess : MethodAccess Base node type for accesses to methods in an instantiated template.

TemplateMethodAccess : PTMethodAccess Access to a method using the **`tsuper`**`[<templateName>.<className>].<methodName>()` form.

TemplateMethodAccessShort : TemplateMethodAccess Short form of `TemplateMethodAccess`, i.e without the template name inside the brackets.

TemplateConstructorAccess : PTMethodAccess Access to a constructor in an instantiated template.

TemplateConstructor : MethodDecl A constructor declaration inside a template class.

PackageConstructor : MethodDecl A constructor declaration inside a class in a PT package.

TabstractMethodDecl : MethodDecl A method declaration with the **`tabstract`** keyword.

RequiredType : ReferenceType A required type declaration.

RequiredClass : RequiredType A required class declaration.

RequiredInterface : RequiredType A required interface declaration.

RequiredTypeAdds : RequiredType Addition to a required type.

RequiredClassAdds : RequiredClass Addition to a required class.

RequiredInterfaceAdds : RequiredInterface Addition to a required interface.

PTAbstractConstructor : MethodDecl In Java there is no such thing as abstract constructors, but with required types, there is. Therefore this is needed.