UNIVERSITY OF OSLO

Department of informatics

Package Templates and programming in the large

Alexander Eismont

August 2014

Acknowledgements

I want to thank my supervisor Stein Krogdahl for his guidance and helpful feedback during the course of this thesis.

Table of Contents

Chapter 1

Introduction

The OMS-group at the department of Informatics at the University of Oslo has from 2006 to 2012 had a NFR-project called the SWAT-project (Semantics-preserving Weaving – Advancing the Technology). This project was working with new modularization mechanisms for modeling and programming in object oriented languages. One of results from that project, and what this thesis will be about, is a mechanism called Package Templates.

Package Templates (or PT for short), is a mechanism that was designed to make it possible to write and use modules containing multiple classes. Its main idea was that it should be possible to extend the classes of such a template in parallel (and do other additions) when such a template is used in a program.

As the name *template* indicates, PT is a mechanism where the content of the module is inserted, once or multiple times, into the program. To become real classes in program, a package template (or just template for short) has to be instantiated in that program. During an instantiation the classes can be adjusted with renaming, merge of classes and additions to a class. The name *Package Templates* for the mechanism discussed here might well describe what it is all about. However, it has the drawback that it uses the same name for both the mechanism as such, and or a language construct. Thus, to avoid misunderstandings, we will refer to the mechanism as such as PT, while we will call the corresponding construct in the language a *template*.

PT is meant as a mechanism that can be added to different object oriented languages. Currently there is a working compiler for a version of PT defined for the Java language (the compiler JPT). Though this version of the compiler can be used for some programs, they are still many features that could be implemented to make the compiler better. In my thesis I will study and evaluate the usefulness of PT and how easy it is to do "programming in the large" with it (for more information on what aspects we will look at in this thesis see chapter 3). I

will therefore write program-libraries and frameworks as separate reusable modules in PT, and evaluate how good the mechanism is for the chosen cases.

While writing a program with a number of classes, one could find oneself in a situation where the classes you are writing have many similarities with other classes defined by you or with classes that exist in the library for the programming languages. As is well known from the use of subclasses, adjusting or expanding a class can make the job easier, since we won't need to write a fully new class, but instead use methods that already exist in another class. In Java expanding a class can be done by making a subclass containing the add-ons we want. The well-known scheme for writing subclasses in Java is as follows:

```
class A { … }
class B extends A {  … }
class C extends A { … }
```

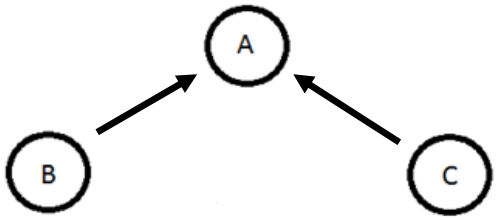


Figure 1 Visual representation of code above

However, subclasses do not always give what we want. As an example, assume that we want to expand the classes above, by adding a couple of variables/methods to each of A, B and C, so that the additions to A can be seen from the additions to B and C. We can try to do this by creating subclasses of A, B and C called A', B' and C' respectively, that define those new variables/methods.

Below is a figure of the resulting hierarchy, where the class A is at the top of the hierarchy tree:

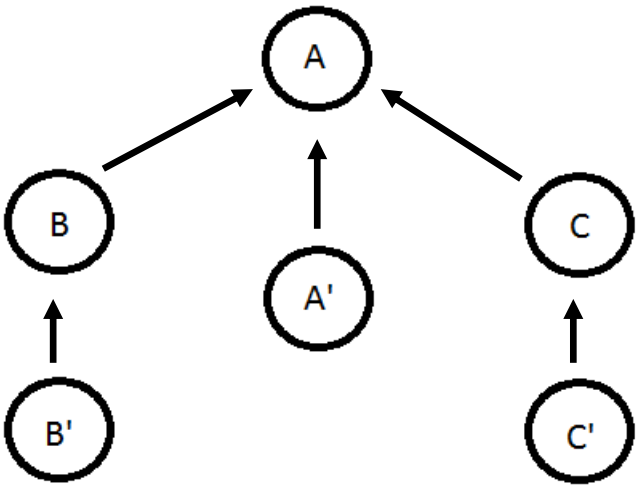


Figure 2 Added subclasses to Figure 1

The problem with this solution is that classes B' and C' can't make any reference to the class A', where the new variables/methods are. Thus, to obtain what we want it seems that we need a mechanism that can add parts to A itself, so that every subclass of A could refer directly to this method. PT introduces the ability to create a copy of a template class (at compile time) and at the same time make changes to that class. In the next chapter we will look on how this mechanism works.

1.1 Thesis structure

In chapter 2 we will describe PT in much more detail, focusing on the different mechanisms PT offers. Chapter 3 deals with how I will perform my research and discuss what aspects of PT we should be looking at. Chapters 4 through 6 describe the different cases that I worked at during my thesis. The final chapter will summarize what we found from the cases I have done and discuss these findings.

Chapter 2

An overview of the Package Templates mechanism

In this chapter we will describe how the PT mechanism works. We will present PT mechanisms as they are described in previous works, but a problem is that some of those mechanisms have not been implemented yet. Thus, we cannot compile and run PT programs that use these mechanisms, but we will describe how they are planned to be working.

2.1 Syntax

A template is similar to a standard Java package, since a template in PT is also a collection of classes. Templates are here written in a Java-like language and PT then works as an add-on to Java. The code below shows a simple template with two classes.

```
template T {
    class C1{ … }
    class C2{ … }
}
```

A *template* starts with the keyword *template* followed by the template name. Inside the template braces, we can include several classes. As in Java packages, the classes inside a template can include variables, methods and references to objects of classes in the same template, and some may be subclasses of others as in a package.

To use the classes of a template in a Java program we need to *instantiate* it in that program (that is, in a package). Inside the package an *inst*-statement is used to create such an instantiation of a declared template. To do that we write *inst* followed by the name of the

template. The code below creates such instantiation of the template T (note that we write Java packages in the same style as PT-templates):

```
package P{
    inst T ... ;  //a simple instantiation
    …
    class C3 { ... }
    class C4 { ... }
}
```

The package *P* has an instantiation of template *T* and the result is that copies of the classes of instantiated template are added to the package after some adjustments (that are discussed in sec 2.2.3). Since these classes are copies of original classes, it's also possible to make many instantiations of the same template in a single package (but we then need to do some renaming, which is discussed later). The package *P* might also include its own classes, like *C3* and *C4* above.

2.2 Graph structure example

Before going further in the PT mechanisms, we will look at an example with will be used for the next couple sections. The code below is an example of a template, which has two classes that are used to implement a graph-like structure, where the end-nodes of the outgoing edges of a node in the graph are stored in a linked list.

```
template Graph {
    class Node {
         int node_id;
         Edge first_edge;
         void add_edge(Node c) {…}
     }

    class Edge{
         Node from, to;
         Edge next_egde;
         void add_edge() {…}
    }
}
```

Every node in this graph will have two local variables and a method for adding edges for a graph node. One of the variables is an integer called *node_id*, which will have a unique number for all nodes in the graph. The other variable is a reference to the first object in a

linked list of outgoing edges. This variable is of type *Edge*, which is a class that is used for creating a linked list. The class *Edge* has a pointer to a node in the graph and a pointer to the next *Edge* object, which will have a pointer to the next edge in the list. Both *Node* and *Edge* have a method called *add_edge*, which takes a *Node* object as a parameter and adds that object to the list of edges.

2.3 Renaming and additions

The main idea of the PT mechanism is that a graph structure like the one described in the previous section can be used for any problems that need a graph-structure for its data. However in such a program one usually also need some additional variables or methods than those that are defined in the template classes.

Let's look at a chess program example that uses graph to store the possible states and situations it has looked at. This chess program is meant to be used for finding the best move in the current situation. Each state (or node) must have variables to store were the chess pieces are placed on the board. To make the data-structure for this chess problem we can adjust the name from the template *Graph*. For this example we may want to change the name of the class *Node* to a name that will be a more accurate description for the chess program. We can also rename local variables and methods. The name of the variable *node_id* will be changed to *chess_id*. The method *add_edge* in *Node* and *Edge* will be renamed to "add_chess_edge". Finally the second class in the template *Graph*, class *Edge* will be renamed to *Edge_Chess*.

The following code makes the changes described above:

```
package Chess_Simulation {
    inst Graph with Node => Chess_State
        (node_id -> chess_id, add_egde -> add_chess_egde ),
                    Edge => Edge_Chess
        (add_edge -> add_chess_edge);
}
```

Using an arrow "=>" after the "inst"-statement renames the specified class. The name of the class on the left side is a class from the instantiated template and on the right side is the new name of the class. For each "inst"-statement, the renaming process can be done on all classes inside a template. For each the template classes, renaming the variables and methods is done

inside parentheses following the “inst” statement, using “->”, with original name on the left side of the “->” and the new name on the right.

In class *Node* in template Graph, the method *add_edge* takes a *Node* object as a parameter. Since the class *Node* has been renamed, the parameter to *add_chess_edge* must now be an object of class *Chess*.

Continuing with the chess program example, we shall look at another important feature in PT. This feature is adding more variables and methods to a class defined in a template. Additions to a class can only be made during an instantiation of a template. For this chess program will need a variable that stores the state of the chess board. Therefore the class *Chess_State* (renamed from class *Node*) includes a two dimensional array called “state” to store information about the chess board. A number of other methods will also be needed for this program. One of them being a method called *check_board* that will try to find the best possible move in the current situation.

The package Chess_Simulation adds the changed described above:

```
package Chess_Simulation {
   inst Graph with Node => Chess_State
        (node_id -> chess_id, add_edge -> add_chess_edge);
   List_Node => List_Chess
        (add_edge -> add_chess_edge);

   class Chess_State adds {
      String[][] state;
      …
      void check_board(){
            … // finds best move
      }
   }
}
```

After an instantiation of the template *Graph* is made with the “inst”-statement, we can now add variables and methods to the class *Chess_State*. This is done by using adds-statement after the name of the class you want to add to. All the code that comes after the adds-statement will be included in the class *Chess_State* along with the variables and methods that already where defined previously in class Node.

2.4 Class hierarch's inside templates

Templates may also contain class hierarchies, and these will be preserved during an instantiation independent of addition and name changes. In the code below there is a class A with a subclass B:

```
template Sub{
      class A {
          int n;
          void print(){
              System.out.println(“ N: ” + n);
          }
      }

      class B extends A{
          int s;
          void print {
               super.print();
               System.out.println(“S: ” + s);
          }
      }
}
```

Note that both classes in this template have a print method. The one in class A prints the variable *n*. In class *B* the method *print* prints the variable *n* and variable *s* by using the super-statement to call on *print* method in the superclass *A*.

The code below renames both classes in the template *Sub*:

```
package Sub_Package{
      inst Sub with A => C, B => D
      class C adds {
          int c;
      }
      class D adds {
          int d;
      }
}
```

The class A and B will change names to C and D. After that, the class D will now be a subclass of the class C and we therefore have the same class hierarchy as before when B was a subclass of class A. Calling the *print* method in D will call the print method in D's

superclass, which in this case is C. If we only changed the name of class B to D and we didn't change A, we still keep the same class hierarchy, so that D will be a subclass of A.

2.5 Merging

Another interesting feature of Package Templates is that it is possible to merge classes when instantiating templates. The classes that are merged can come from the same or different instantiations. When classes are merged the resulting class will have all the variables and the methods of the original classes. PT does not allow name collisions between the merged classes (but this can easily be solved by renaming). A name collision comes from there being two or more members with the same name in the resulting merged class.

As an example of a merge, we look at the template bellow called *FigureInfo*. *FigureInfo* has a class *Figur*, which represents different figures. It has two variables called *form* and *pos*, and two methods *display_figure* and *print_form*. The string *form* says what form the figure has, like square, circle etc, while *pos* gives the position e.g. on a screen.

```
template  FigureInfo {
      class Figure{
          String form;
          int pos;
          void display_figure(){
                … // draws the figure on the screen
          }
          void print_form(){ … }
      }
}
```

The second template used in the merging process is called *ActionInfo*. This template keeps information about different actions that can be performed by the user: mouse clicks, play sound etc. These occur in a class Action with variables *name* and *id* and methods *do_action* and *print_form*.

```
template ActionInfo {
  class Action {
      String form;
      int id;
      void do_action () {
      …       // performs an action based on the form
              // example: if the name is play_sound, then do_action
              // will play a sound
      }
      void print_form() { … }
  }
}
```

We here want a class, which we will call *ActiveFigure*, which is a merge between the classes Figure from template *FigureInfo* and Action from template *ActionInfo*. The code below is an example of that merge, but note that it's not legal because of the name collations, both for f*orm* and *print_form*.

```
package P {
        inst FigureInfo with Figure => ActiveFigure;
        inst ActionInfo with Action => ActiveFigure;
}
```

In the package P we create one instantiation of *FigureInfo* template and rename the class Figure to name *ActiveFigure*. In the instantiation of the template *ActionInfo*, the class Action is also renamed to *ActiveFigure*. Since they are renamed to the same name, the rules of PT say that the classes will be merged to a new class *ActiveFigure* in package P. All the variables and methods from the original classes will be included in this new class.

In both Figure and Action classes we have a variable and a method with the same names. We therefore during the merging process, have to change names to avoid these name collisions. The resulting program could look like:

```
package P {
      inst FigureInfo with Figure => ActiveFigure
      (form -> figure_form, print_form -> print_figure_form);

      inst ActionInfo with Action => ActiveFigure
      (form -> action_form, print_form -> print_action_form);

      ActiveFigure adds {
            void do_action ( ) {
              ... // adds code for performing actions on figure: move
                  // figure with mouse,change color etc.
            }
      }
}
```

One could imagine that the methods and variables with the same name could be put together, but Package Templates doesn't support the merging of variable and methods with the same name. This is because two methods with the same name might not include the same code, or two variables with the same name will be used for the same purpose. Therefore when merging multiple classes, we have to change names, so that all names in the involved classes are different. If the classes are large, and many names are equal, the process of renaming could become quite long. But usually we will be merging classes that are quite different from each other, so that the probability for name collision is small.

2.6 Template hierarchies

So far we have seen instantiation of templates being made in packages, but will now we look at instantiations of templates inside other templates. The process of doing that is no different than using templates in packages.

A typical example of a template hierarchy:

```
template T1 {
     class A1 { … }
     class B1 extends A1{ … }
     class C1 extends B1{ … }
}

template T2 {
     inst T1 with A1 => A2, B1 => B2, C1 => C2;
     class A2 adds { … }
     class B2 adds { … }
     class C2 adds { … }
}

package P1 {
     inst T2 with A2 => A3, B2 => B3, C2 => C3;
     class A3 adds { … }
     class B3 adds { … }
     class C3 adds { … }
}
```

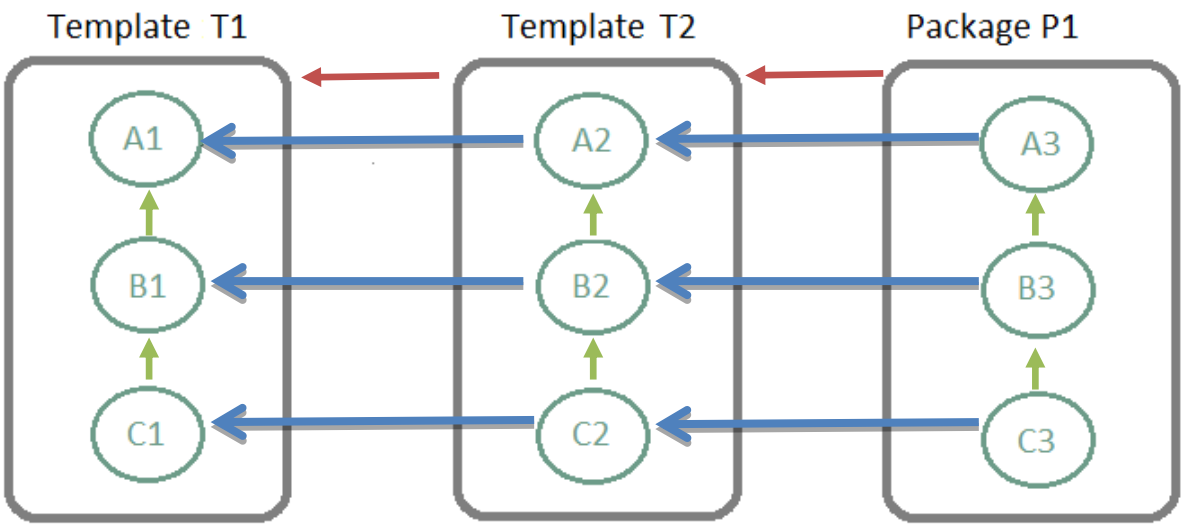


Figure 3 Visual representation of a template hierarchy from example above

This example can be visualized as in Figure 3. Here the arrows that point from one template to another represent instantiations. The arrows go from one class upwards to another represent the subclasses relation. From template T1 we can see that B1 is the subclass of A1 and C1 is the subclass of B1. Note that B2 will be a subclass of A2 (because B1 is a subclass of A1), even if this is not indicated in T2.

The horizontal arrows that go from a class in a template to a class in another template represent relation between the old and new version of a class in the instantiated template, with the arrow going from the new to the old version. The new version may be simply a copy of the old one, or there may be additions and/or renaming in the new one. Even if a template class is not mentioned in a "with" statement, we still get this type of arrow. The class that the blue arrow points to is in PT called its tsuperclass. The class on the opposite side will inherit variables and methods from that tsuperclass. During instantiation of a template, the class from that template that was used for the renaming process will be the tsuperclass. From the figure above we can see that the class A1 from template T1 is the tsuperclass of A2 in template T2. Since package P1 made an instantiation of the template T2 and renamed class A2 to A3, the class A2 will be the tsuperclass of A3.

As templates or packages that make an instantiation of a template will get copies of all classes from that instantiated template, thus T2 will get copies of classes A1, B1 and C1. Then class A2 in template T2 will consist of everything from A1 and whatever was added to A1. The same goes for the class A3, which will include copies of A1 and A2. This can be visualized as in Figure 4. Note that we e.g. in T3 can no longer talk about the classes A2 and A1. They are incorporated in renaming.

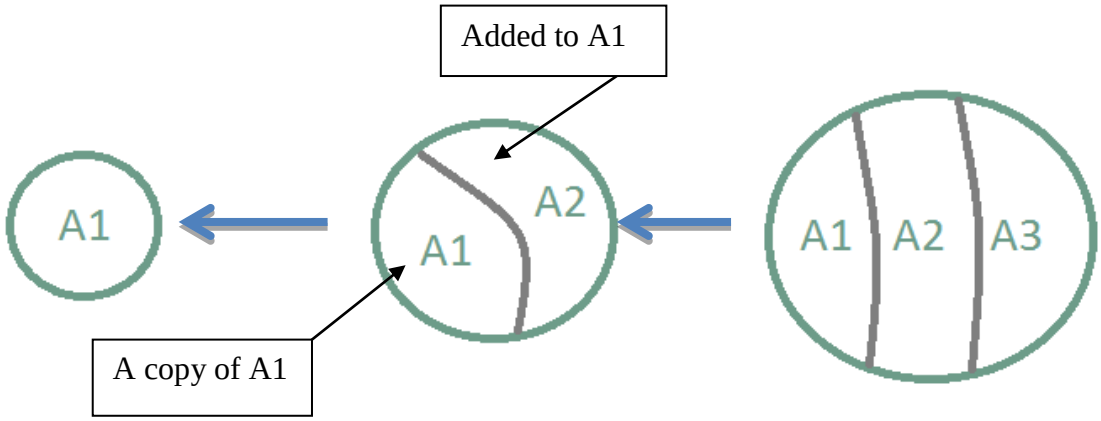


Figure 4 A Closer look at classes after instantiations

The figures 3 and 4 show the same instantiation of templates T1 and T2. The only difference between those two is that figure 4 gives us a closer look at what other template classes a class consists of. The final class in figure 4 consists of a copy of A1, a copy of what was added to A1 and what is added to A2. This is also the case for figure 3, but it shows a simpler version by just including the final name of a class.

When we in the class A3 want to use a variable or a method that is defined in the class A1, we can access it directly (with the new name if it is renamed). When a variable or a method is not declared in a *adds* part, PT will first search for a declaration in the tsuperclasses before superclasses (and its tsuperclasses). So if we in B3 want to use a variable that is declared in B2 and A3, the variable that will be used is the one that is declared in B2.

2.7 Super- and tsuper calls

Calling methods defined in superclasses in Java is normally done by a direct call to that method. If the class and its superclass have methods with the same name, the class doing a call on one of those methods will not be a call on a method from the superclass, but instead its own. Therefore we have to use the super-statement, which lets us explicitly call on methods from the superclass.

Below is an example of a standard Java super call with classes from template T1.

```
class A1 {
       int a1;
       void print () {
            System.out.print(a1);
       }
}

class B1 extends A1 {
      int b1;
      void print () { //overrides print method in A1
           super.print();
           System.out.print(b1);
      }
}
```

We will next look at how we can call methods defined in a tsuperclasses, when we, in an addition class have a method declaration with the same name and parameters. Similar to Javas "super"-statement, PT has a "tsuper"-statement to call methods in tsuperclasse. This is done by using keyword "tsuper" followed by the name of the method with the necessary parameters you want to pass. This type of call is possible because the renamed class is just a copy of a class from an instantiated template.

Example below uses the "tsuper"-statement:

```
template T2 {
    inst T1 A1 => A2, B1 => B2, C1 => C2;
    class A2 adds {
        int a2;
        void print () { //overrides print() from T1's class A1
           tsuper.print();   // call print in A1
           System.out.print(a2);
        }
    }
    class B2 adds {
         int b2;
         void print() {
            super.print();          // calls print in A2
            System.out.print(b2);
         }
    }
    class C2 adds { … }
}
```

Here, after the class B1 from template T1 is renamed to B2, we add a method *print* which will override print method in B1. Since B1 is the subclass of A1, then B2 can make a super call on print method in A2. The class A2 also has a print method, which overrides the one from class A1 in template T1 and calls print method in A1 with tsuper.print(). If we try to run the *print* method in B2, it will print out the variables a1, a2 and b2 respectively.

2.8 Multiple tsuperclasses

In the two previous chapters we have been looking at classes in templates and packages having a single tsuperclass. But unlike superclasses in Java, where a class can have only one superclass, a class in Package Template can have multiple tsuperclasses. This is because of a merging where the classes that were merged are the tsuperclasses of the merged class.

Below is a typical example of a class having multiple tsuperclasses:

```
template T1 {
    class A1 { … }
    class B1 extends A1 { … }
}

template T2 {
    class A2 { … }
    class B2 extends A2 { … }
}

template T3 {
    inst T1 with A1 => A3, B2 => B3;
    inst T2 with A2 => A3, B2 => B3;
    class C3 extends B3 { … }
}

package P1 {
    inst T3 with A3 => A4, B3 => B4, C3 => C4;
    class A4 adds { … }
    class B4 adds { … }
    class C4 adds { … }
}
```

This program can be visualized as in Figure 5:

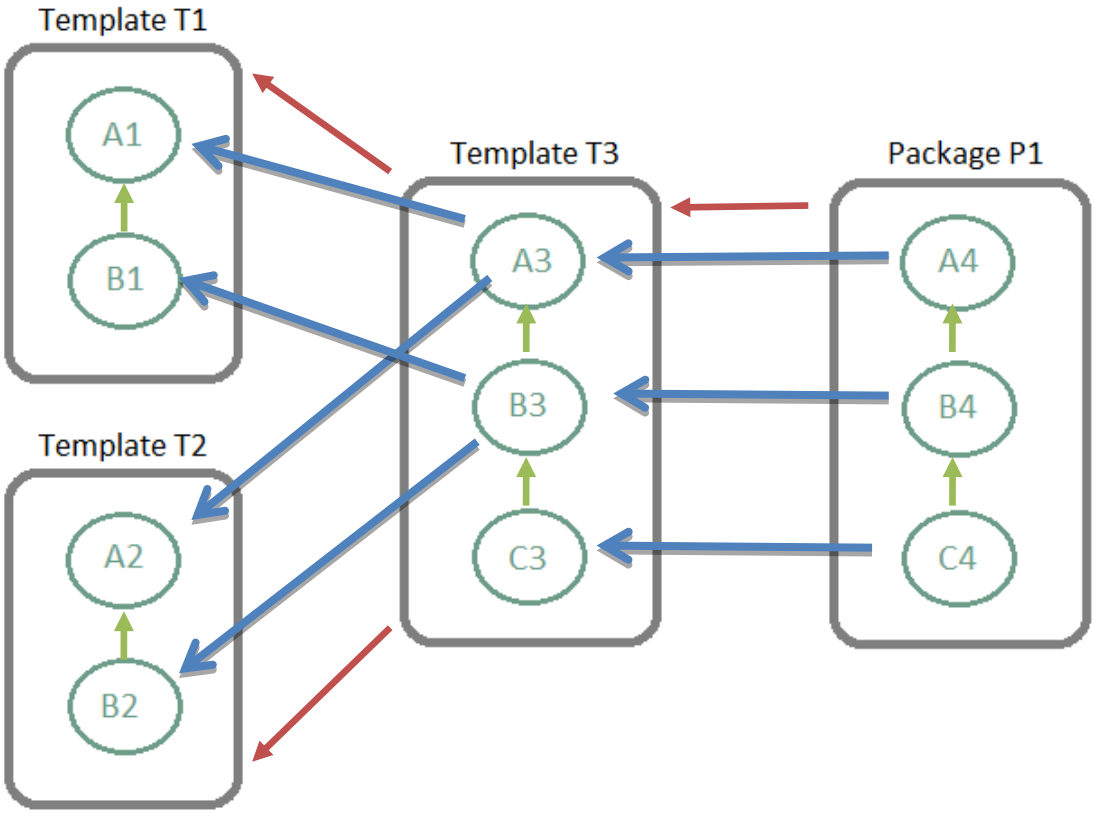


Figure 5 Example of multiple tsuppeclasses

Figure 5 shows three templates and one package. The package P1 makes an instantiation of T3, and T3 makes an instantiation of templates T1 and T2. The classes A1 and A2 are renamed to A3, which makes A1 and A2 the *tsuperclasses* of A3. This is shown in the figure above, where the class A3 has a blue arrow to A1 in template T1 and a similar arrow to A2 in template T2. The classes B1 and B2 in templates T1 and T2 were both renamed to B3 and therefore are B3's *tsuperclasses*.

In this situation the *tsuper* calls described in the previous chapter will no longer work, since PT can't know what *tsuperclass* the call is meant for. Therefore the *tsuper* call has to be changed so that we can tell PT what class it should be searching in. To take care of such cases the tsuper call can be done with "tsuper[classname].m()". The class name in the call is the name of the class that has the method we want to call on. In the special cases when multiple classes have the same name, we have to specify what template the class that we are calling comes from. This is done by adding template name followed by the name of the class in that template: "tsuper[T.C].m()", where T is the template name and C is the class. An alternate way of solving this problem is by giving an instantiation a name:

```
t1: inst T with …
```

"t1" is the name of an instantiation of template T1. When making a *tsuper* call to a class in T, instead of writing "tsuper[T.C].m()", we can use "tsuper[t1.C].m()". Since no instantiation can have the same name, PT will know exactly what *tsuperclass* the program is trying to call.

Below is extended code for the classes A1 and A2 from templates T1 and T2:

```
class A1 {
      int a1;
      void print () {
            System.out.print(a1);
      }
}

class A2 {
       int a2;
       void print () {
            System.out.print(a2);
       }
}
```

Both classes were extended with print methods that print their local variables. The next we want to do is to merge these classes and try calling each print method from classes A1 and A2 from the merged class:

```
template T3 {
      inst T1 with A1 => A3 (print -> print_A1);
      inst T2 with A2 => A3 (print -> print_A2);

      class A3 adds {
            int a3;
            void print () {
                  tsuper[A1].print();
                  tsuper[A2].print();
                  System.out.print(a3);
            }
      }
}
```

An alternative to the "tsuper" calls in the code above is to call on "print_A1" and "print_A2". But if either "print_A1" or "print_A3" is overridden, then we have to use the "tsuper[…](…)" notation to call on the original "print" method in A1 or A2.

When working with multiple inheritances in Package Templates, we have to make sure that a class won't get multiple superclasses. If we only merge two or more subclasses and not merge their superclasses, the merge class will inherit the class hierarchy from the templates that were used. But since several classes might have different superclasses, the merged class will end up with multiple superclasses. Since a class in Java can't have multiple superclasses, we therefore have to merge all the superclasses of the subclasses that are merged.

2.9 Constructors in Package Templates

Classes in templates can have constructors, and they are called so that, when an object is generated, exactly one constructor is called in each template class and each adds-class (if we disregard this-calls). Calling a constructor in Java is done with "this(parameter list)" (if we want to call on a constructor in the same class) or with "super(parameter list)" (if we want to call on a constructor in the superclass). When working with template hierarchies, we can call the constructors of a tsuperclass. The syntax for calling constructors is similar to superclasses. Instead of calling "super(…)" with parameters inside the parentheses, we need to call tsuper().

Since a class after a merge can have multiple tsupperclasses we have to specify what class the call is made for with tsuper[classname]().

The code below is an example of tsuper calls on constructors:

```
template T1 {
    class A1 {
        int a1;
        A1(){
            a1 = 1;
        }
    }
}

template T2 {
     class A2{
        int a2;
        A2(){
           a2 = 2;
        }
     }
}

package P1 {
     inst T1 with A1 => A3;
     inst T2 with A2 => A3;

     class A3 adds {
         A3(int p){…}

         void test_constructor(){
            tsuper[A1]();
            tsuper[A2]();
         }
      }
}
```

The rules for declaring constructors in template classes are that every constructor of a template class must make calls to a constructor in each of its tsupperclasses (and if there are multiple, one must always use the notation "tsuper [....](...)"). But you're not allowed to use "super (...)" in template classes. The package-classes works in a similar way, but it also need to call a constructor of its superclass with "super(...)". Then we need, if it is neccessary, call "tsuper (...)" or "tsuper[...](…)" for all tsuper-classes, in any order we want. In the end, a constructor will be called only once in each package and each template class.

2.10 Required types

Required types in PT are types that can, during an instantiation, be given a concrete type that will at least provide the same methods that are included in the definition of the *required type [1]*. In this paper we will refer to this process as concretization, where the required type can be concretized with a template class. For the concretization to be legal the methods in the class and *required type* need to have the same name and number of parameters. The return type and the types of parameters for each method also need to match. With this mechanism a template can create objects of a type and call on its methods without these methods being yet implemented. Since these *required* types can be concretized with other classes, every method that we call on in that type can have many different implementations. The benefit that comes with *required types* is that we can reuse the same code for many different classes, instead of writing a piece of code that only uses a single class. Below is a notation of how to define a required type:

```
template T1 {

	required type R{
		void print();
	}

	class B {
		void test(){
			new R().print();
		}
	}
}
```

Inside a template we can define a required type with the keywords “required type”, followed by the type’s name. Inside of its brackets we can include multiple method declaration without an implementation (similar to Javas interfaces). Other classes in the same template as the *required type* can make objects of R and call any of its methods, knowing that this type will later be given an actual class type with the same methods.

Below is an example of how to merge a *required type* with a class (for this we use the example from above):

```
template T2 {
      inst T1 with R <= A;

      class A {
            int a;
            void print() { System.out.println(a); }
      }
}
```

After the “with” keyword in an instantiation we can do a merge. This is done by writing the name of the *require type*, follow by “<=” and the class that you want to concretize the type with.

Like with Java and template classes, required types can also extend other classes and implement interfaces. The superclasses and interfaces need to be visible in the template that the required type was declared in. A class that is concretized with a required type that extends another class needs to also have the same superclass, else the merge won’t be legal. Required type can also be merged with other required types which are done within the same way we merge template classes. Like with template classes there can be no name collision, so we therefore have to do some renaming if there is a case of name collision. Adding to a required type can also be done with “adds” statement. The class that is concretized with a required type will need to implement the added methods as well.

Java includes a similar mechanism known as generics, that doesn’t restrict a piece of code to just one type, but instead each time you create an object of a generic class you have to send a type as an actual parameter. Generics also are present in PT. Below is an example of generics and templates.

```
template T < E > { … }
```

In the example in the code above, *E* is a type parameter and during the instantiations you can give a class as an argument. Even though both generics and *require types* are similar, we will use *required types*, since they can be expanded after instantiation.

2.11 Subtemplates and templates as parameters

Templates in PT can have other templates as parameters, where during an instantiation we can send another template as an argument, which is also instantiated during the initial

instantiation. Even though template parameters allow us to send different templates as arguments, we still want formal parameter to have some specific classes that we can add to or call on their methods. Therefore we are able to restrict what template can be given as an argument. The syntax for template parameters is "template T < P bound T2> {…}", where P is the name of the formal template parameter. Following P is a restriction that limits P to be the template T2 or any other template that instantiates T2.

To find what template makes an instantiation of another, PT doesn't look at what other templates instantiate inside the template clauses, but instead make one template to be a *subtemplate* of another template (that we will call a *supertemplate*). When instantiating a template that is a subtemplate of another the *supertemplate* gets instantiated as well. A *subtemplate* instantiation is made by writing the "subof" statement after the template name, followed by the name of the *supertemplate*. Below is an example [2] of a *subtemplate*:

```
template Vehicles {
      class Vehicle { … }
      class Car extends Vehicle { … }
      class Truck extends Vehicle { … }
}

template VehiclesWithWeight subof Vehicles{
      class Vehicle adds {
            int weight;
            void print(){ System.out.println(weight); }
      }
      class Car adds { … }
      class Truck adds { … }
}
```

VehicleWithWeight makes a subtemplate of the template Vehicles and is able to add to the template classes of Vehicles. The next template that we will look at is *RentalVehicles* [2] that takes in a template parameter:

```
template RentalVehicles <template E bound Vehicles>{
      ...
}

package VehiclePackage{
      inst RentalVehicles<VehiclesWithWeight>;

      class Vehicle adds { ... }
      ...
}
```

RentalVehicles takes a formal template parameter E and instantiates it. In this case the template parameter *E* must either be *Vehicles* or a template that is a subtemplate of *Vehicles*. In package *VehiclePackage* we make an instantiation of *RentalVehicles* and send *VehiclesWithWeight* as argument. Since *VehiclesWithWeight* is a subtemplate of *Vehicles*, this instantiation is legal.

Template parameters can also be useful in cases when we need to instantiate a group of templates, where at least two of them instantiating the same template. This will be a problem, since the same template is instantiated more than once and we therefore get multiple copies of the same classes and its members. The paper "Challenges in the Design of the Package Template Mechanism" [2] describes a case where templates *PrintExpressions*, *MultExpressions* and *ValueExpressions* make their own instantiation of the template Expressions. Each of these template make their own changes to the template classes from Expressions and in the end we want to make to combine all templates together in a package *CombinedExpression*.

Below is an example that solves the problem of multiple instantiations of the same template:

```
template Expressions {}

template PrintExpressions <template E bound Expressions> subof E {...}

template ValueExpressions <template E bound Expressions> subof E {...}

template MultExpressions <template E bound Expressions> subof E {...}

package CombinedExpressions {
      inst MultExpressions<ValueExpressions
            <PrintExpressions<Expressions>>>;
}
```

The instantiation in the package *CombineExpressions* will start by giving *MultExpressions* the template *ValueExpressions* as an argument and with the "subof" statement becomes a subtemplate of *ValueExpressions*. *ValueExpressions* becomes a subtemplate of *PrintExpressions* and *PrinExpressions* of *Expressions*. Since *Expressions* is instantiated other templates that need its classes can now add to *Expressions* without having to instantiate it themselves. The order of how the templates are combined together determents where a tsuper call will go.

2.12 Visibility

Java includes multiple options for modifying access ability for variables, methods and classes. These options include *public*, *private*, *protected* or no modifier. Currently the PT-compiler hasn't implemented any visibility regulation for the template mechanism, but after certain templates have been instantiated in a package, the final classes and their members will have the same visibility as in Java. We will here look into some of the possible rules for visibility regulation in PT that are sketched in the paper Challenges in the Design of the Package Template Mechanism [2].

The *private* modifier in Java reduces the visibility of a variable or method to only that class. This means that this member can't be accessed from any other Package, not even from subclasses in the same package. Something similar was proposed for PT as well. Any member of a class inside a template that has a private modifier would only be visible inside that template.

For classes or members of a class in a template that have a *public* modifier, we could make them visible to any template or package that instantiate them. When it comes to classes or

members that have no visibility modification, we could make them only visible inside the template they were declared in.

The *protected* modifier could have the same rules as in Java with packages, and make protected members in classes not visible outside the template that is was declared in, with the exception of subclasses (of the class that has this member) in any template or package. Additionally we could also include the “adds”-mechanism to one of the exceptions, so that a protected member can also be accessed from a “adds” part in another template. Since protected modifier will increase visibility to both subclasses and “adds” parts, it was proposed that we could create additional modifiers that would allow “adds” part to have access to a member on not subclasses, and vice versa. One of these modifiers is “aprotected”, which makes a member only visible inside the template were it was declared and the “adds” part in other templates. The second modifier is “eprotected”, which gives access to a member to subclasses, but not “adds” parts.

Another way we could regulate visibility, is to use a public or private modifier on an instantiation of a template. During a private instantiation the classes and their members could then be visible inside the template that makes that instantiation, but not visible in further instantiations. A public instantiation of a template will make the classes and its members visible to all templates that make further instantiations.

Chapter 3

General consideration for evaluation

This master thesis will be about evaluating how helpful the PT mechanisms can be in larger scale programming. In this chapter we will look, more generally, at what such an evaluation will include. We will describe different aspects of what is meant by saying that including PT in an OO-language is an advantage, how we will approach each test case in terms of programming, what PT mechanism we should look at and finally what we wish PT can help us achieve. The three cases that we will look at later will be discussed in chapters 4, 5 and 6. Each chapter will give a short description of the purpose of the program, before going into how it was programmed.

3.1 The programming process

In each of the cases we will look at will try to incorporate as many different PT mechanics as possible (but only where it makes sense to use them), so that we get to test and evaluate all the important parts of PT. Some PT mechanisms will be used in multiple cases, since a single case might not be enough to try to determine wherever a mechanism works as it supposed to and how useful it can be.

Since we are using Java with PT for all of the cases, all code will be placed inside either a template or a package. The different parts of the code will be split into multiple templates so that we, hopefully, obtain better separation of the code. We will try to use PT so that each template can be instantiated and also used in other programs that need the same functionalities. Like with all of the examples from the previous chapter, all the code will be written in Java extended with PT. For most of the examples, they will also be written in regular Java (without the use of any of PT's mechanisms) to be able to compare with the structure and flexibility of the two versions obtained with PT.

3.2 Special aspects of PT

In this section we will look at certain aspects of PT that can make things problematic. We will therefore describe and discuss my experience from the use of the different implementations, and try to indicate where, and in what sense the PT mechanisms made the implementation better or easier (or the opposite). In the cases where I found that PT could make the job easier, we will try to describe in what sort of cases this is true. In cases where PT does not help (or, in fact get in the way) we will look for improvements to PT so that we avoid these type of problems. We will also try to pint out some possible ways to work around these problems in cases where there is no good improvement or fix available. If it was difficult to develop, the reasons for that might include bugs in compiler or maybe a PT mechanism that might need a certain feature that can help make the developing process easier.

Some mechanisms described in chapter 2 (like template parameters), are planned, but haven't been implemented yet in the compiler. However, we will generally assume that these mechanisms are fully implemented and that they work as described in the previous chapter.

If a mechanism doesn't work as well as we had hoped, we will also discuss some possible improvements or changes that could potentially make this mechanism better. Some of the mechanisms obviously might be more useful in some situations than in others, so these situations should be pointed out and discussed.

3.2.1 Visibility regulation

Currently the PT-compiler doesn't have an implemented visibility regulation for templates. The type of regulations that we will describe will include the protected, private and public modifiers and for each of them discuss if the already proposed regulation for these modifiers are good enough for larger programs. If not, I will try to create rules for visibility regulation that, at least, fill the needs of the cases that we look into in the next three chapters. Then for each rule try point out advantages and disadvantages for that rule. As part of this we will discuss if the is any need to distinguish between the two modifiers *aprotected* and *eprotected* or if the old *protected* modifier is enough. In the end of the visibility discussion we will talk about the *private* and *public* instantiations, and if they have any use as additional visibility regulations.

3.2.2 Issue with superclasses

Another problem that we might run into is when classes that you want to merge, both have their own superclass. When classes are merged the resulting class has all the code from all of its tsuperclasses. Additionally then the resulting class will also end up with more than one superclass. Since the resulting class is not legal in Java, the superclasses will have to be merged as well. If the superclasses can be merged without any conflict, then the resulting class will be legal in Java. In some situations the merge of superclasses won't be possible and therefore the resulting class will not be legal. We will try to point out these situations where this merge problem occurs. Another aspect of merge of superclasses that we will discuss is what happens to the subclasses and their functionalities after that merge.

3.2.3 Java with PT

We will also be looking into what Java mechanisms work well in PT and what don't. There are certain aspects of Java that is currently not introduced in PT, e.g. nested classes. We will therefore look at nested classes and other Java mechanisms, and check if they are translated well into PT. For mechanism like nested classes we discuss if they should they be implemented properly or just ignored because we really don't have any need for them in our programs.

3.3 What makes PT good

Since we are looking at different mechanism to see if they make PT better or not, we should also take a look into the different aspects of what can make PT more useful when programming larger programs.

For one, the program that we are developing with PT should first of all be correct. It should be fully functional and work the way it was intended to. If there are any aspects of PT that prevent us from achieving this, then we should definitely be looking into how PT can be improved.

When writing code for different programs, we would like the process of doing so be as easy as possible. This should also be true when programming using PT. Though a steep learning curve will always be expected when starting to use a new mechanisms/language like PT, it should still be accessible and easy to use. For example the problem that occurs with superclasses mentioned earlier could cause some issues with the coding process. If we have problems with merging of superclasses, we might have problem with completing the program

or have a hard time with finding other ways to get around this problem. The same problem can also occur with missing mechanism like template parameters.

The introduction of a new concept into a language will always in one sense make using the language more complex. This might also be the case for PT. Introducing PT could e.g. be good for writing flexible separate modules, but it might be bad in the sense that it introducing yet another concept into the set of OO-concepts, making it harder to choose the right concept to use.

Another aspect of PT that important for the quality of a module system is reusability. Since we with PT can make instantiations of multiple different templates, we are therefore able to construct many different programs using the same code multiple times through instantiations, renaming and additions. In cases where none of the existing PT mechanisms will help us with reusability or maybe even make it harder. We will examine an option of using regular Java subclasses instead. If using subclasses instead of PT might give the same flexibility as with the use PT, then would definitely need to do some more work on PT for it be an par or better than just using subclasses.

Efficiency can be an important factor that we should be looking into, since efficiency can also make PT be useful to program with. One of the things that we will discuss when it comes to efficiency is runtime and if or how PT can make the runtime faster than with just the use of regular subclasses.

Chapter 4

A simulation package in PT

As our first case we shall look at a framework for discrete event simulation in PT, with a suitable modularization of the framework into templates. The idea is to write a framework based on a discrete event simulation framework in the Simula language called *Simulation*.

The concept that corresponds to a Java package in the Simula language is a class with local (nested) classes. The outer class then corresponds to the package as such, while the inner classes correspond to the classes of the package. To be able to use the inner classes of a Simula package, the program should have the form: "package_name begin . . . end", where the inner class of package_name now become directly accessible from within the "begin . . . end" part. The Simula version of the simulation framework uses two such Simula packages: *Simset* and *Simulation*. In addition Simula has a coroutine mechanism (more about coroutines in chapter 4.2.2) built into the language. This is not the case for Java (with or without PT), so that we will have to provide such mechanism, preferably as a template in the PT version.

However, it turns out that the danish professor, Keld Helsgaun, has already written a "copy" of Simulas simulation package [3] in pure Java, including a Java package for the coroutine mechanism. We will build our PT version upon Helsgauns version, and will thus make it easier to see whether we can get any benefits from using PT instead of pure Java. .

The next sections will describe what a discrete event simulation is, Helsgaun's implementation of the *Simulation* framework in Java, PT version of *Simulation* and visibility regulation for templates.

4.1 Discrete event simulation in Java

Discrete event simulation typically contains a group of entities (usually processes) that have some sequence of actions they want to perform, and where the next action for one process

may depend on the state if the rest of the system. Every action in a simulation will take different or the same amount of time to complete. Because in this type of simulation where only one action can be executed at a time, processes have to wait in a set/list until another process is done executing its action or is interrupted. When a process is next to execute its action, this process is removed from the list and gets resumed. After that process is done, it will either we terminated (because the process has completed all of its actions) or suspended and put back into the list, waiting to be able to perform the next action.

4.2 Helsgaun's version

4.2.1 Simset, a two way circular list

Simula has a *package* called *Simset* that implements a two-way linked list where each element has a pointer to the next and previous element in the list. The package *Simset* contains a class *Linkage* and the subclasses *Link* and *Head*. This is meant as a utility mechanism to be used in programs needing a list. This mechanism is used twice in the Simula simulation framework (more about this in chapter 4.2.3). Helsgaun has a package *Simset* in Java with the same classes and methods.

Objects of the class *Linkage* have pointers to the next and the previous element in the linked list. These pointers are called *SUC* and *PRED*, and are of the type *Linkage* and are not accessible from outside *Linkage*. The methods offered by *Linkage*, *Link* and *Head* will ensure that we, at runtime, always have a number of circular double linked lists, each with exactly one Head object and the rest are Link objects (more about *Link* and *Head* classes later). An empty list consists of a single *Head* object, with *SUC* and *PRED* pointing to itself.

```
class Linkage {
      Linkage SUC, PRED;
      public final Link pred() {
            return PRED instanceof Link ? (Link) PRED : null;
      }

      public final Link suc(){
            return SUC instanceof Link ? (Link) SUC : null;
      }
}
```

Creation or deletion of a connection between two elements is done in the class *Link*, which is a subclass of *Linkage*. *Link* includes methods *out*, *follow*, *precede* and *into* (as shown below).

```
class Link extends Linkage {
      public final void out(){…}
      public final void follow(Linkage ptr){...}
      public final void precede(Linkage ptr) {...}
      public final void into(Head s) {...}
}
```

Method “out” takes an element from the linked list by creating a link between an element’s predecessor and successor. The method *follow* will insert this object as successor to the Linkage pointer that is given as a parameter, while the method *precede* will insert this object as a predecessor. The method *into* will insert this Linkage object in the end of a list.

All versions of *Simset* also have a class *Head* that is a subclass of *Linkage*. *Head* has pointers to the first and last element in the list, which are the *SUC* and *PRED* pointers in *Linkage*.

```
class Head extends Linkage {

      public Head() {
            PRED = SUC = this;
      }

      public final Link first() {
            return suc();
      }
      public final Link last() {
            return pred();
      }
      public final boolean empty() {…}
      public final int cardinal() {…}

      public final void clear() {
            while (first() != null)
                  first().out();
      }
}
```

The methods *first* and *last* return the first and last element in the list (and *null* if the list is empty). The first element is *Head’s* successor while the last node is its predecessor. The method *empty* returns true if the list is empty. The last two methods are *cardinal*, which finds the number of nodes in the list, and *clear* that removes all Link-objects from the list and set their *SUC* and *PRED* pointers to *null*. To create a circular list we need to create a *Head* object, so that the method *into* in Link will connect the first element and last element in the list.

When the list is empty (no Link objects in the list) the methods *suc* and *pred* in Linkage will return null if the pointers are null or of type Head.

4.2.2 The Coroutines package

A coroutine mechanism allows us to suspend execution of a task and give control over to another task. When the control is then passed to a task that was suspended, the execution of that task is resumed from where it left off. This type of execution is known as quasiparallel execution (of a set of coroutines). Below is a class *Coroutine* (from Helsgauns *Coroutines* package) that includes code for suspension and resumption.

```
abstract class Coroutine implements Runnable {

      private Coroutine caller, callee;
      private static Coroutine current, main;
      protected boolean terminated;
      private Thread myThread;

      final public void run() {
            body();
            if (!terminated) {
                  terminated = true;
                  detach();
            }
      }

      abstract protected void body();
      public static final void resume(Coroutine next) {…}
      public static final void detach() {…}
      public static final Coroutine currentCoroutine() {…}
      public static final Coroutine mainCoroutine() {…}
      private void enter() {…}
}
```

The class *Coroutine* implements the interface *Runnable* so that each object of *Coroutine* can, at the outset, run as a thread in parallel to other *Coroutine* objects. However, these threads are controlled so that only one of the threads can run at a time. To start or to resume a coroutine we call the method *resume* and give it a pointer (as an argument) to the coroutine we want to start or resume. Inside that object, the method *resume* we will call on the method *enter* to either wake up a coroutine. To make the coroutine mechanism work with threads, we will use the methods *start* and *notify*. These are methods from the class *Thread*. The method *start* will start the execution of a thread by calling the threads *run* method. The *notify* method is also

from class *Thread* and is used to wake up a waiting thread. A thread will go into a waiting state when the *Thread's wait* method is called.

The method "body" should be overridden with a method that describes what this coroutine should do during its execution. While in the *body* method, a coroutine can be suspended and when it is resumed by another *Coroutine* object (or the main program) it continues from where it left off. Since the class *Coroutine* implements *Runnable*, it needs to implement a *run* method as well, where it calls the method *body*.

4.2.3 The Simulation package

The final class in Helsgaun's simulation framework is the class *Process*. This class includes all the fields and methods needed for managing event time, which is needed to decide when a process needs to be resumed. The *Process* class in both versions has the class *Link* as its superclass. The objects of the class *Process* in both Simula's and Helsgaun's versions can be inserted into a *Simset* list and removed from the same list. This is meant as a utility feature, so the writers of simulation programs can keep track of their different processes, by having them in sets/lists.

In Simula's version there is an additional class called *Event*. This class extends the class *Link* and there is one *Event* object for each *active* process object. Each *Event* object holds information of the event time for a single process. By using the *Simset* mechanism, the *Event* objects form an event list sorted on event time and inform the *Process* class when to resume or suspend its execution. Since the *Process* class also extends *Link*, the *Process* objects can form a secondary list. In Helsgaun's version (see code below for more information) there is no *Event* class, but instead he has additional members in the *Process* class to create an event list. For this purpose he had to write almost the same code as found in *Linkage, Link* and *Head* classes.

```
public abstract class Process extends Link {
    private final Coroutine myCoroutine = new Coroutine() {
        protected void body() {...}
    };

    private Process PRED, SUC; // previous and next element in a event list
    private final static Process SQS; // head of a event list

    public final Process nextEv() { //
            ... // identical to suc() from Linkage class
    }

    private final void cancel(){
            ... // identical to out() method from the Link class
    }

    ... // additional methods
}
```

Helsgaun's version of the *Process* class has an anonymous inner class (a nameless local class that has access to members of the outer class) of *Coroutine.* Simula's version of the simulation program doesn't include a *Coroutine* class, since coroutines are built in to the Simula language.

4.3 Simulation with PT

In this section we will present a PT version of Helsgauns simulation framework. The way we are going to split the classes into template is by taking every package in Helsgaun's version and create similar classes from those packages in templates. The *Simset* package with classes *Linkage*, *Link* and *Head* will be placed in the template *Simset*, while the class *Coroutine* will be placed in the template *Coroutines*. With this approach we will keep these utility mechanisms separate from each other so that they can be instantiated without instantiating the other templates. So when we need the coroutine mechanism, then we can just instantiate the *Coroutines* template without the *Simset* list.

In our case we would like to instantiate both of these templates in a new template called *Simulation* where we will define the *Process* class that will use both the list and coroutines. Our implementation of all of these classes is very similar to Helsgaun's version, but with some essential differences.

Though creating an anonymous inner class of *Coroutine* works in Helsgaun's program, using these anonymous classes does have some drawbacks. The outer class that the anonymous

class is located in cannot access any of the members defined in the anonymous class. Therefore in certain situations it is better to create a proper subclass instead, since we get the ability to define additional methods in the subclass that can be called by other classes. In the PT version we can't use anonymous inner classes as this mechanism has not been implemented yet in PT compiler. One way to get around this is for the *Process* class to have *Coroutine* class as a superclass. Using this approach, the *Process* class will inherit the coroutine mechanism and can directly call the methods from the *Coroutine* class.

Using this subclass approach can reduce the code we need to write, since we won't need to create objects of another class and manage these pointers. Thus, we get a proper communication between the class *Process* and its superclass *Coroutine*.

Programming a similar framework where the *Process* inherits, in the subclass sense, from both *Link* and *Coroutine* classes is not possible with Java. However, in the PT version we can instantiate the templates *Coroutines* and *Simset* in the template *Simulation*. During the instantiations we can merge the classes *Link* and *Coroutine* into one single class with the merging mechanism. This way the class *Process* only needs to extend a single class instead of two.

An alternative solution (and what we will be using in this case) is to merge the classes *Link* and *Coroutine*, and then use the *adds* statement to add to the merged class all the methods and variables that the *Process* class requires. The additions that are made to the *Process* class are similar the members of the class from Helsgaun's version.

The figure and the code below shows the instantiations of the different templates and the changes made to their classes. Note that in this PT version we will use the instantiated *Simset* list as an event list and the second utility list in PT will be described in the next section. This way we can easier describe how the main parts of the *Process* class work together, before moving how we will implement into additional utility lists.

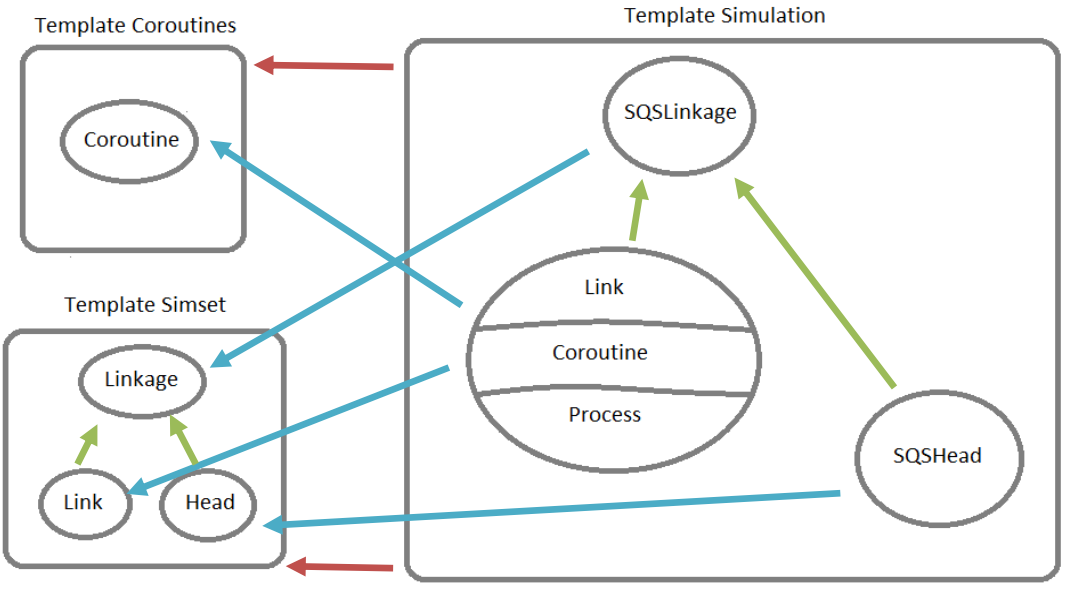


Figure 6 A visual representation of Java Simulation program with templates

```
template Simulation {
      inst Simset with Link => Process,
                       Head => SQSHead, Linkage => SQSLinkage;
      inst Coroutines with Coroutine => Process;

      class Process adds{...} //see the code below
}
```

The names classes *Head* and *Linkage* were changed to *SQSHead* and *SQSLinkage*, so that we can ensure that our instantiation of *Simset* won't interfere (caused by a name collision) with another instantiation of *Simset* in either *Simulation* or other templates/packages. All the pointers of type *Link* and *Coroutine* in template *Simulation* will be changed to type *Process*. This change will also be made in other classes in the *Simulation* template and not just in the *Process* class. The methods *suc* and *pred* in Linkage that returned pointers *SUC* and *PRED* of type *Link* will now return pointers of type *Process*.

After the merge, we should add variables and methods for managing processes:

```
class Process adds {
      private final static SQSHead SQS = new SQSHead();
      private double EVTIME;
      private boolean TERMINATED;
      private static Process MAIN;

      protected void body() {...}
      abstract protected void actions();
      public static final void hold(double t) {...}
      public static final Process current() {
            return SQS.suc(); // returns first element in the event list
      }
      static final void activat(boolean reac, Process x,
                                      double t, Process y) {}
      public static final void activate(Process p) {
            activat(false, p, 0, null);
      }
      final static void resumeCurrent() {
            resume(SQS.suc());
      }
}
```

The process class will include a pointer to a *SQSHead* object called *SQS*. This variable is static so that all processes use the same *SQSHead* object. Each process has a variable *EVTIME* which is the event-time and *TERMINATED* which is state of the process. The value is set to "true" when a process has executed its code.

The method "actions" includes the code for what this process needs to do and should be overridden by a subclass of the class *Process*. To start execution of a process-object we need to call the method *activate* with the Process object that we want to start. Helsgaun had in his code multiple versions of "activate" were each takes in multiple parameters which let us decide from the beginning of the life cycle of a process when it should start to execute its code. In our implementation we decided to make things simpler and use only one version, where we set the EVTIME variable to the current time and the process is inserted into the right place in the list based on the event-time. *activat* is the method that inserts the a process into the Simset list and is called by *activate*. Below is an implementation of the method *activat*:

```
static final void activat(boolean reac, Process x,
                                    double t, Process y) {
      Process current = SQS.suc(), p = null;
      double now = time();
      if (x == current)         // a Process can't activate itself
            return;
      t = now; p = SQS.suc();
      if (x.suc() != null)      // if a process is not in the list,
            x.out();            // take it out
      if (p != null) {
            for (p = SQS.pred(); p.EVTIME > t; p = p.pred())
                  ;// finds the right spot in the list
            x.EVTIME = t;
            x.follow(p);
      }else{
            x.EVTIME = t;
            x.into(SQS);
      }
      if (SQS.suc() != current)
            resumeCurrent();
}
```

Before we can insert a process into our list, it first needs to be removed from it. This is done in the second "if" statement. Since the Process class gets a copy of all members from Link, we can call the method "out" to take a process out of the list by using a *Process* pointer. The same goes for all other methods that *Process* inherits through addition. If the list is empty (aside from the main program) we set the EVTIME variable to current time and set this process into the beginning of the list by calling "x.into(SQS)". If the list is not empty, then we insert the process at the right spot in the list by calling "x.follow(p)".

The class *Process* will override the method *body* from *Coroutine*. Therefore when a new thread is created, the *run* method will call *body* which contains the code that each process needs to execute. The overridden *body* will call the method *actions* and then remove itself from the list and resume another process by calling *resume* (which Process inherited from *Coroutine*) with the next process that waits to start or continue executing. A process can also suspend itself by calling the method *hold* with the amount of time it wants to wait before continuing execution. This process is taken out of the list with *out* and puts itself another place with the method *follow*. Finally it will resume the next process that is waiting in the list.

4.3.1 Second utility list

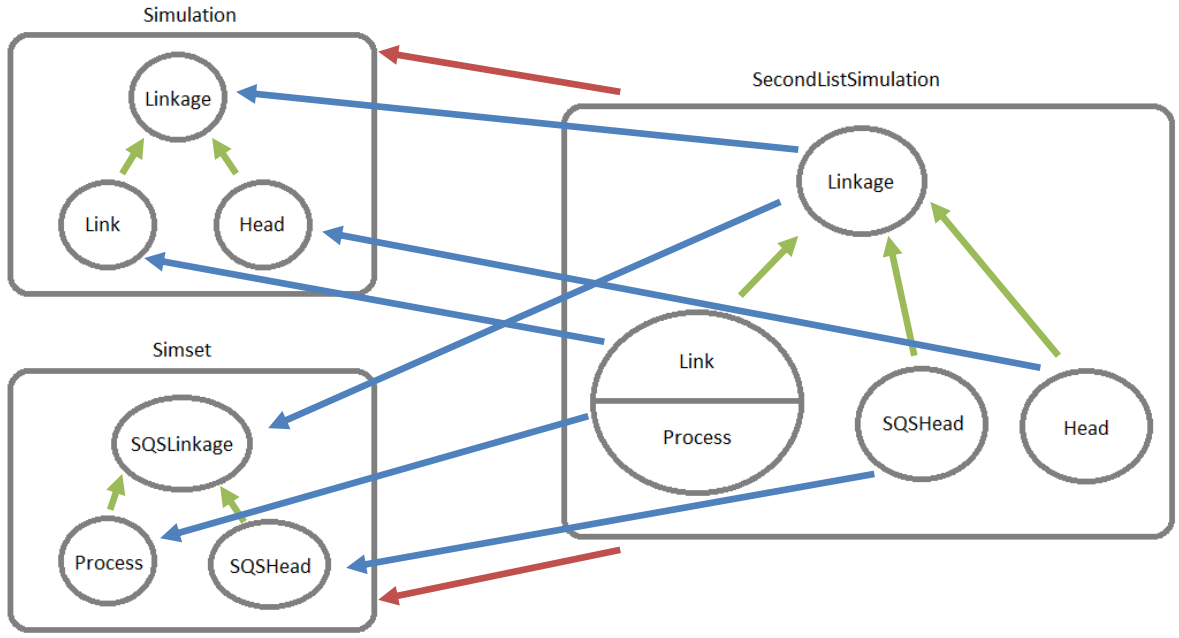
So far we have written a simulation framework where processes can be inserted into an event list, but we would also like for our PT version to have a secondary list that processes that be inserted into. Using Simula's or Helsgaun's approaches would require us to write code for a list that we already have the code for (which is the *Simset* list). The secondary list could have been much easier to code if the *Process* class could inherit from the *Simset* list a second time. That is of course not possible, since *Process* class can't extend two classes, not even if they are the same class. PT on the other hand allows us to rename classes (therefore creating new copies of the original class), merging of superclasses and being able to have several copies of the same class. We will now try to use all of these mechanisms to create a version of the *Process* class that is able to inherit from two *Simset* lists.

To create the secondary list, we need to instantiate the template *Simset* once more. We will do this instantiation in a new template. Since we want to add the second *Simset* list to our simulation program, we will instantiate the *Simulation* template as well. What we want to do next is for the *Process* class to inherit from *SQSLinkage* and Linkage (from the newly instantiated *Simset* template). For the *Process* class to only have one superclass, we have to merge the classes *SQSLinkage* and *Linkage*. This merge will cause a name collsion, since all the members from both classes have the same name. To resolve this name conflict, one of copies of the variables and methods need to be renamed. So if we change the name of the variables *SUC* and *PRED* during the instantiate of *Simulation* and rename them to *SQSSUC* and SQS*PRED*, then these variable we help us create an event list, while *SUC* and *PRED* from *Simset* will work as a secondary utility list.

So that a process can create a link between itself and other processes, we will merge the class *Link* with the class *Process*. All the members from *Link* will also need to be renamed, which we will need to do so that we can avoid a name collision and have separate methods for both lists. The new copy of the *Head* class doesn't need to be merged with any other class. An object of *SQSHead* will be the head of the event list, while an object of the class *Head* works as the head of the secondary list.

The code and figure below shows what we want to achieve from the instantiations:

```
template UtilitySimulation{
      inst Simulation with SQSLinkage => Linkage (SUC->SQSSUC,
                              PRED ->SQSPRED, ...)
                  Process => Process (follow(*) -> SQSfollow, ...);
      inst Simset with Link => Process;
}
```



When compiling a template that does renaming there is a bug that results to a compiler error. This is because renaming of members is not done in all classes in the template. But when fixed, the instantiations above should be legal. From the code above we can see that writing the second utility *Simset* list in PT doesn't require a lot of new code. Instead, using PT's renaming and merging mechanisms allows us to use the *Simset* list twice. This benefits us, since PT makes our code more reusable.

4.4 Visibility regulation

In our simulation framework, many of the variables and methods have some kind of visibility modifier. Since no visibility regulation for templates is defined in PT, what members a template can access from other templates is determined by visibility regulation in the Java language. However, this turns out to not be sufficient for a good visibility regulation in a system like the PT version of *Simulation*.

Once we merge the classes *Coroutine* and *Link*, all the private variables from the *Coroutine* class will be visible in the *Process* class. In the "adds" part of our implementation we can now

access any of the *private* members that were previously located in Coroutine. In case we don't want the *Process* class to have access to *Coroutine's* private members, we would need those members stay only visible to the *Coroutine* part of the code. So just like with Java packages, where private members are "package-private", we could make private members inside template classes to not visible outside the template they were declared in.

The *protected* modifier in Java reduces the access level to only the class and it subclasses. Any class from other package cannot access these protected members. In our example we add to *Process* class, code for overriding the method *body*. This method is set as protected in *Coroutine*, so that only *Coroutine* or it subclasses have access to it. But since we now can override methods from *adds* classes, we should also have a visibility regulation for *adds* classes as well.

Using the proposed visibly regulation from [2], we could increase the access level for the *protected* modifier to include *adds* in other templates. With this change the *protected* modifier in templates will work similar to how they work in packages, with the exception that we are now able to access *protected* members during addition to a class. In this case we could have used *aprotected* ("additions protected") modifier on the *body* method, since it's not used/overridden by any subclasses. This would make the method *body* only visible inside a template and the *adds* part of other templates. But since we would like the Coroutine class to be instantiated and used in other programs, we want subclasses from other templates to have access to *Coroutine's* protected members.

For members of a class that are set as *public* should work as with packages, so that they be visible from any template or class.

Class members with no modifier in Java are by default package-*private*. So members of a class or a class itself are visible to all other classes in the same package, but not visible from any other package. When choosing how *no-modifier* will change the visibility for template we should look at what other visibility regulations we already have. Making *no-modifier* work as any of the other modifiers will have little use for us, since we could just use that modifier. Making the *no-modifier* work in a similar way as how it works in Java will give us an additional way to regulate our classes and their members. We can therefore make *no-modifier* to be *template-private*, so that a class or its members are visible in the template they are declared in, but not outside that template.

Next we will discuss other possible use of public and private instantiations in our Simulation program. The templates *Coroutines* and *Simset* are the templates that we could make a private instantiate on. This way we can protect all the classes and their members from being accessed from any other template that will instantiate *Simulation*. If there are any classes in *Couroutines* or *Simset* that are set as *public*, the visibility of that class would still be limited to the template that made the *private* instantiation. The same goes for *protected* modifier and obviously *private* modifier as well. So having this visibility regulation could be helpful in some cases.

4.5 Evaluation

During the programming part of this example I came across very few challenges. Since I used Keld Helsgaun's code for coroutines and *Simset* list for Java, the biggest challenge was to make his code work with PT. Helsgaun used multiple nested classes in his implementation, but since PT doesn't yet allow nested classes, I had to change those classes by placing them outside the class in which they were declared in and add multiple pointers to the former outer class. Helsgaun's code also included a static initialization block that created a single *Head* object for the *Simset* list. When compiling the code, PT looked at this block as a nested class and therefore resulted in a compiler error. But the fix for this problem was quite simple, since all I had to do was to replace this block with a static method. None of this was really a big issue, since the code could easily be tweaked to make it work with PT.

Aside from a few changes to the code, everything else worked really well with PT. Since this was the first time I wrote a program using PT, I wasn't really sure how well the compiler actually worked. In this example we get to use several main PT features, like renaming, addition and merging, and all of them worked like they were supposed to. What I was most interested in was the resulting class of the merge of *Link* and *Coroutine*. Since *Link* was a subclass of Linkage and the class Coroutine implemented an interface *Runnable*, the resulting class Process ended up with implementing *Runnable* as well as having Linkage as its superclass. So overall the transition from Java to PT was quite simple, since most of the code we used for the Java version could also be used in the PT version as well.

4.5.1 Multiple superclasses

After studying this example, it seems that not having a superclass is beneficial when using the merging mechanism. If we made *Coroutine* a subclass of *Thread* (from Java's library) instead of implementing the interface *Runnable*, we wouldn't be able to merge the classes *Link* and

Coroutine. This is because both *Link* and *Coroutine* now have superclasses and to make the merge possible we would need to merge the superclasses as well. Since the class *Thread* doesn't exist in any template, but instead is part of Java package, we wouldn't be able to merge this class with any other class.

There is a particular problem that we can get when merging two superclasses. If we in this example could merge Thread with Linkage, this could create another problem, because Head would now inherit from Thread as well as Linkage, which is not what we want in this example.

A possible solution (but that currently wouldn't work with PT) for using Thread (and possibly other superclasses for other cases) is to make a nested class inside Coroutine that extends Thread. All variables and methods that have to do with suspension and resuming will be included in the Coroutine class, while the Thread methods will be moved to Coroutine's inner class. Below is an example of Coroutine with such an inner class:

```
public class Coroutine {
      //same variables as before
      private Runner myRunner;
      static Runner firstFree;

      protected void body(){...}
      public static final void resume(Coroutine next) {...}
      ...
      private void enter() {...}

      class Runner extends external Thread {
            Coroutine myCoroutine;  // The target coroutine
            Runner nextFree;        // Next Runner in the free list
            Runner() {…}
            void setCoroutine(Coroutine c){
                  myCoroutine = c;
            }
            ...
            public synchronized void run() {...}
            synchronized void go() {...}
      }
}
```

The method "enter" in *Coroutine* will now create a new object of class *Runner* and set the *myRunner* variable to point to that object. Every time a *Coroutine* object needs to start, resume or suspend, it can use this pointer to call on the method *go* in *Runner*. The class

Runner has the method run, which like previously, calls on *body* and then be removed by calling *detach* in the *Coroutine* class.

Though the approach above could help us with merging of superclasses, it wouldn't work with our PT version because inner classes have not been implemented yet. So an alternative solution and one that works with PT is to move the *Runner* class outside the *Coroutine* class (but inside the template *Coroutines*). The code for *Runner* needs to be changes since it can no longer access Coroutine's variables and methods without using additional pointers.

Although this solution works with PT, it creates another problem. *Coroutine* has multiple variables that are set to *private* or *protected*. When *Runner* was inside the class Coroutine, it could access Coroutine's *private* members. But when we move Runner outside Coroutine (but still in the same template), Runner will no longer have access to those variables. Therefore we would need to remove the *private*" and *protected* keywords from all variables that are used by Runner. Now any class that makes an object of *Coroutine* is able to change the variables that were previously private and protected, which ruins the purpose of restricting the access level in the first place. This makes the nested solution better, since we can keep the same access level as the original file.

Since nested classes were not implemented yet into PT, we will try to come up with another solution that will help achieve some of same results as with the use of nested classes. For one we want to keep the same visibility regulation for the class Coroutine and at the same time make all the members that are used by the *Runner* class accessible. If we decide to make the *no-modifier* to be *template-private*, we can use this modifier on the methods and variables (that were previously set as *private*) in class *Coroutine* that is used by *Runner*. With this we would achieve the same result as with nested classes. Members with *no-modifier* will not be visible for any other template, the same as with their previous *private* modifier. At the same time the *Runner* class will have access to all of *Coroutine's* members, since they are no longer private and are visible in the template they were declared in.

A Java package might include several other classes (in addition to classes and have nested classes). The class with the nested class still keeps the same visibility regulation with other classes present and the same should also be true for template classes as well. We will continue with the same solutions for alternative for nested classes from previous paragraph. Let's say that that we have other classes in the same template as Coroutine and Runner. Because of the

modifiers that we used, the members of the classes *Runner* and *Coroutine* would no longer work in the same way as a nested class. This is because the variables previously defined as private in these classes have now no longer a modifier, so other classes have now access to these members. We would still like for these classes not to have access to these members. We can achieve this by having the additional classes in a separate template. Then we can instantiate that template and *Coroutines* in a new template. This way all the members with no modifier will work as private for any other class in the new template, the same way as we would in a regular Java package and nested classes.

4.5.2 Java library as templates

At this moment we are able to get access to the Java library by importing the package in our program. But we could imagine that the entire Java library will be located in separate templates. If we needed to a class like *Thread* we could, instead of importing that package, make an instantiation of the template that has the class *Thread*. Being able to instantiate a class from the Java library, means that we could we could do renaming on its members, as well as making additions to that class (which can replace the use of subclasses). Most import benefit that we get to use for this example, we would be able to merge the class *Thread* with the class *Linkage* (since it's not possible in the current version).

Aside from getting the ability to do more with the classes from Java's library, nothing else really changes. The programmer can still use these classes as with the import option. Therefore having Java's library in templates doesn't have any drawbacks and we get plenty of benefits by using this solution.

Chapter 5

Compiler

For my second case I will look at how we can write a typical compiler program using PT. The purpose of this example is to try to make a working compiler and find possible benefits writing it with PT rather than with pure Java. The source language for this compiler is rather simple, in that it was based on the mandatory assignment for the course Inf5110 at University of Oslo and was called *Oblila[4]* the year I took the course, but in this thesis we will not go further into this language. The next section describes how this compiler will be implemented and what parts we will focus on.

5.1 Structure of our compiler

To be able to compile a program in a certain language we need a scanner and a parser that can handle this language. The scanner will read from a source file and create tokens for every string that was read. Tokens in a compiler are generally names of classes, types, variables and more, or some kind of symbol like a plus sign or a bracket. The parser will take these tokens and create objects of a corresponding class. If for example the scanner reads the keywords "void method_name" from the source file, then it will create a *token* for that method name. The parser then takes this token and creates an object of a class that is used to represent and store information about a method. To differentiate the different methods from each other, that class could take a name as a formal parameter. Since a method can include formal parameters and instructions in the method body, the method class can also include pointers to classes that represent these parts.

While making these objects, we might find that one object has one or more pointers to other objects created by the parser. In our language, variables are declared in a class or a method, so that an object that represents a method will have pointers to objects that represent variables. Using this approach we create a type of tree structure also known as an *abstract syntax tree*,

where the root of the tree is the program itself and its children (leave nodes) are the classes and methods. This is helpful when, after all nodes are created, we can traverse all objects to do some additional work on the source code. An example of this is that after the *abstract syntax tree* is built, a compiler can create bytecode for the source code and make a semantic analysis of the source code represented by the *abstract syntax tree*. Through this semantic analysis the compiler will find the correctness of a program. This could include type checking, no double declaration of variables in the same scope. For this case we will focus on the *abstract syntax tree*, semantic analysis and bytecode generator. The next three sections describe the implementation of each of these parts with the use of PT. To get flexibility with this implementation, each part will have a separate template.

5.2 Additional tools

For the scanner and parser, we have decided to use the tools called JFlex and Cup respectively. For our compiler to work, we will need to create additional Java classes that are not part of the *abstract syntax tree*. These classes will use the parser and scanner to read the tokens in the source file and create the necessary object in the *syntax tree*. In this thesis will focus on the *abstract syntax tree* and not the parser, and scanner classes. Although we have decided to use regular Java versions of the parser and scanner classes in our PT implementation of the compiler, one could also write them using PT.

5.3 The template Syntax

The first template that we will talk about is the template *Syntax*. This syntax tree is represented by nodes which are objects of classes created for each. All the classes that are used for nodes in the syntax tree are present in the template *Syntax*.

Below is the implementation of template *Syntax* with some of the classes needed for the syntax tree.

```
template Syntax{
	class Program{
		List<Decl> decls;
		public Program(List<Decl> decls){
			this.decls = decls;
		}
	}

	abstract class Decl{
		public Decl(){}
	}

	class ClassDecl extends Decl{
		String name;
		List<Decl> decls;
		Type tType = new Type();
		public ClassDecl(List<Decl> decls, String name){
			this.decls = decls;
			this.name = name;
			tType.setType(name);
		}
	}
	...
}
```

The class *Decl* is declared abstract, because in later templates in our case it will include methods that we want other similar classes to implement. *ClassDecl* is one of these classes and is therefore a subclass of *Decl*. Other subclasses include *ProcDecl* (a class that has the name of the procedure and information on all operations inside that procedure) and *VarDecl* (a class that represents a variable in the source language). *Program's* constructor takes in a list of *Decl* pointers, where each pointer will either be of type *ClassDecl*, *ProcDecl* or *VarDecl*. *ClassDecl* also takes in a list of *Decl* pointers as a parameter, since a class in our language can include multiple variables.

When our compiler reads a piece of code, it will (depending on the symbol) create an object of one the classes include in the template. For example, if the compiler reads finds the symbol "class" in the code, then the next symbol must be the name of the class. Therefore our compiler creates an object of *ClassDecl* and sends the name of the class and all the declarations inside that class as an argument to the constructor.

5.4 The template Semantic

This section describes the template called *Semantic* that does semantic analysis on the source code. To make this analysis easier we will need a symbol table, which is a data structure that

keeps information on all scopes in the program. All declared variables and their scope must be added to this symbol table. When a variable is used in a procedure, the compiler has to a lookup operation on this variable, to check if this variable is declared and is visible from the scope that it is used in. The symbol table data structure will be defined in a class *SymbolTable*.

For the template *Semantics* to get access to the syntax tree we have to instantiate this template. Most classes from *Syntax* would also need to be expanded to include this analysis. Below is the implementation of template *Semantic* with some of the classes needed for the semantic analysis.

```
template Semantics {
      inst Syntax;

      class Decl adds{
            abstract protected boolean checkCorrect();
            abstract protected Type getType();
            abstract protected String getName();
            abstract protected void setTable(SymbolTable st);
      }

      class SymbolTable{...}
}
```

Since all subclasses of *Decl* require some sort of semantic check and update of symbol table, we want to expand *Decl* by adding additional methods. These methods are declared abstract so that at least the leaf subclasses of *Decl* will have to implement them.

Below is an implantation of *ClassDecl* in template Semantic:

```
class ClassDecl adds{
	SymbolTable SYMBOL_TABLE;

	public ClassDecl(List<Decl> decls, String name){
		tsuper(st, name);
	}

	public void setTable(SymbolTable st){
		SYMBOL_TABLE = st;
	}

	protected boolean checkCorrect(){
		boolean ok = addToTable();
		// checks correctness on all of its variables
	}

	// adds all declarations into a symbol table
	protected boolean addToTable(){
		if(decls!= null){
			for (Decl d : decls){
				boolean ok = SYMBOL_TABLE.insert(d);
				if(!ok) return false;
			}
		}
		return true;
	}
}
```

Since *ClassDecl* is a subclass of *Decl* it would need to implement all of the abstract methods if *Decl*. By using the *adds* statement we give *ClassDecl* a pointer to a *SymbolTable* object and the method *setTable* will set that pointer. The added methods *checkCorrect* and *addToTable* are used for semantic check.

5.5 The template GenerateCode

For the last part of this case we want our compiler to generate a sort of bytecode for a source program. The bytecode that we will be using is a version that was written and is used in the compiler course Inf5110 at University of Oslo. For generating bytecode we will use the syntax tree with all the additions done by the template *Semantic*. After the tree has been generated and the semantic analysis found the program to be semantically correct, the compiler can start to generate bytecode.

For this part we will create an additional template called *GenerateCode*. Each class from *Semantic* template will be further expanded with methods and variables that will be used for generating bytecode.

Below is a figure of all instantiations made to complete this compiler and implementation of some of the classes in *CodeGenerate*.

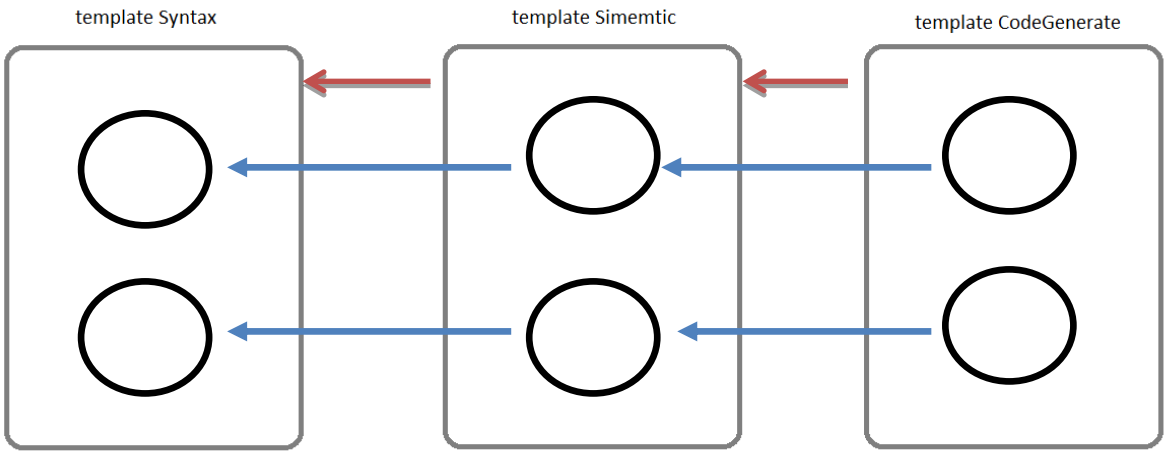


Figure 7 Visual representation of CodeGenerate

```
template GenerateCode{
	inst Semantic;

	class Decl adds{
		abstract protected void generateCode();
	}

	class VarDecl adds{
		public VarDecl(String name){
			tsuper(name);
		}

		protected void generateCode(CodeFile sp){
			// adds bytecode for variables with name and type
			if((tType.t).equals("float"))
				... // update variable
			else if((tType.t).equals("int"))
			...
		}
	}
}
```

We add to *Decl* an abstract method *generateCode* so that all subclasses of *Decl* will have their own implementation for bytecode generation. Then for each class that requires bytecode generation we override *generateCode* method from their superclass and give the proper implementation.

5.6 Evaluation

Since this compiler was first devolved in a master course, we didn't use PT during our programming process. The entire *Oblila* compiler was therefore written in regular Java and primarily used subclasses and no splitting of the syntax tree, semantic analysis etc into different classes. Rewriting the code so that it works with PT wasn't particularly difficult. In the beginning of the coding with PT I could just copy the entire java code of the compiler and place it in a template with any major problem. After that the code needed to be split into different templates and with the use of instantiation, combined to form a working program. All of the PT mechanics need to achieve that, like instantiation and additions, worked like they were supposed to, and I didn't have any problem accessing the classes and their members. So, all the basic mechanisms of PT seem to work really well for this particular case

So, for building our compiler as three separate and more or less independent parts, PT works very well. However, we will now assume that we want to extend or change the compiler in different ways, and discuss to what extent PT can help us with the expansion.

5.6.1 Making changes to the compiler

Since we split our compiler into three different templates, we would like that each of these can be expanded separately. If we were to use our syntax tree, but wanted to add an additional class or change already existing class, we could make an instantiation of the template *Syntax* and then make the necessary changes. Or if we wanted to use the same structure for the syntax tree and semantic analysis, but use a different generator for bytecode, then we could we make an instantiation of *Semantic* in a new template for generation of bytecode.

While trying to find how we can further expand our compiler through new templates and instantiations, I experienced some challenges with PT that, in certain cases, makes it harder for us to reuse template classes. To show these challenges and how we might solve them we will use the previous three declared templates *Syntax*, *Semantic* and *GenerateCode*.

If we want to add additional grammar to the source language of our compiler, we could make and instantiation of template *Syntax* and add the necessary classes. We make this instantiation in a new template *Syntax_A*.

Since we introduced a new grammar to our language, we would also need to do a semantic analysis for this part. To not have to rewrite the entire *Semantic* template, we would like to use *Semantic* and expand the semantic analysis on the new classes or create additional analysis on the old classes. To do so, we want to expand the template Semantic by making an instantiation of it in a new template called *Semantic_A*. Since we want to do semantic analysis on the new syntax tree, *Semantic_A* needs also to instantiate *Syntax_A*. We will do a similar thing for generation of bytecode as well. A new template *GenerateCode_A* will make an instantiation of templates *GenerateCode* and *Semantic_A* and add the necessary code for generating bytecode.

The code and figure below shows all the new templates and instantiations.

```
template Syntax_A{
      inst Syntax;
      ...
}

template Semantic_A{
      inst Semantic;
      inst Syntax_A;
      ...
}

template GenerateCode_A{
      inst GenerateCode;
      inst Semantic_A;
      ...
}
```

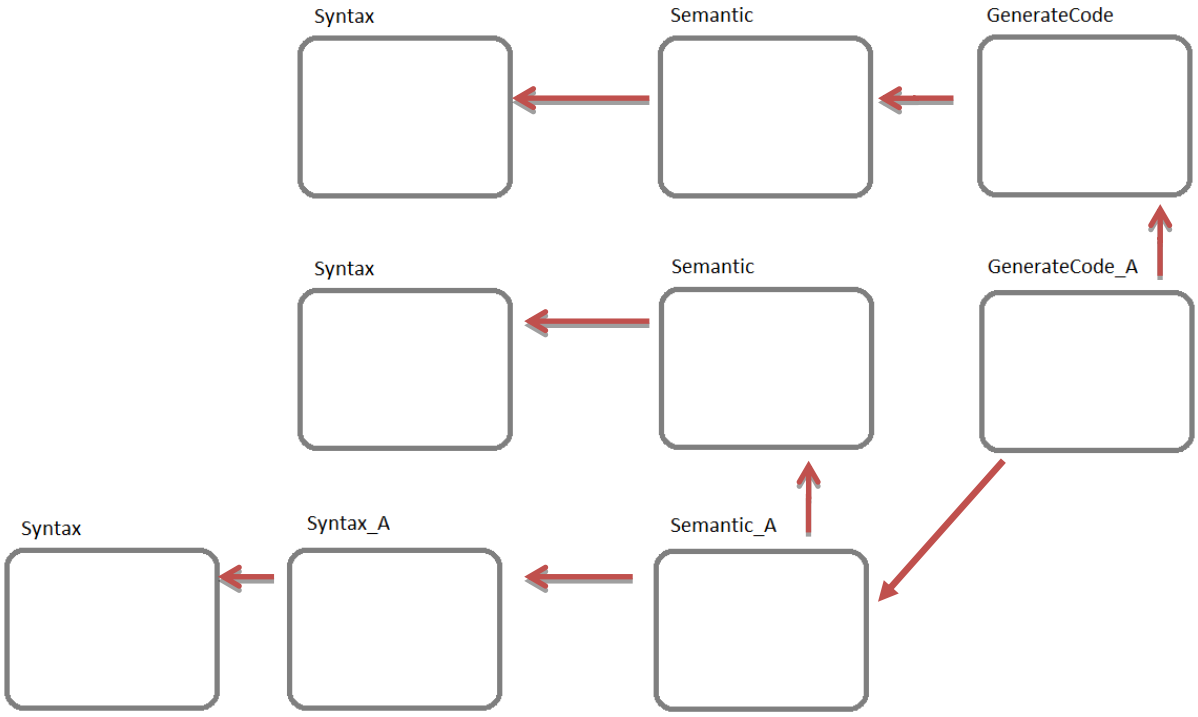


Figure 8 Visual representation of our the code above

The problem with this implementation is that we get multiple instantiations of the same template. If we look at the template *Syntax*, this template gets instantiated more than once. One of these instantiations is made by the template *Semantic*, which is again instantiated by template *Semantic_A*. Another instantiation is made by *Syntax_A*, which is also instantiated by *Semantic_A*. This creates a name collision between both instantiations of the template *Syntax*, because the template *Semantic_A* gets two copies of *Syntax* with exact same variables and methods. To make our compiler program work with PT we would need to do a lot of renaming. But the result of renaming would be that we would get two different syntax trees, which is not what we want for this case.

Ways to work around this problem is to instantiate *Syntax_A* in a new template (that doesn't instantiates Semantics) and write a completely new semantic analysis in that template. The problem with this solution is that we won't get to use the already written semantic analysis in the Semantics template. In this case generating bytecode would also need to be rewritten again.

Another solution would be not to make the additions after instantiating *GenerateCode*. This way we could create new classes for the syntax tree, semantic analysis and generating bytecode in one instantiation. But this solution is also not ideal. The addition to the compiler

is done on the last of our main three templates. That would make it hard for us to use just one part of the compiler. So if we wanted to just use the syntax tree and, not semantic analysis and bytecode from the new compiler, we wouldn't be able to do that. This particular problem occurs when we instantiate multiple templates which at some point instantiate the same template and we therefore get name collision and instances of the same class.

The next section will try to use template parameters to solve our problem with multiple copies of the same class.

5.6.2 Template parameters

A way to solve our problem with multiple instantiations of the template is to use template parameters (see chapter 2.11). We can do this by changing some of our initial templates and adding additional code for template parameters. What we want to achieve with template parameters is for our main three templates to be able to make instantiations of other templates that themselves insatiate the main three templates. So that if the template *Semantic*_A instantiates both the templates *Syntax*_A and *Syntax*, it will only get one copy of the classes from the template *Syntax*.

The template *Semantic* expands on the template *Syntax*, so we can make the template *Semantic* be a subtemplate of *Syntax*. This will also let the template *Semantic* be passed as a argument to other template which formal parameter requires an instantiation of the template *Syntax*. For the template *GenerateCode* we will do a similar change, where we make this template a subtemplate of *Semantic*. Below are the changes described in this paragraph and a package that will combine all the parts:

```
template Syntax {...}

template Semantic subof Syntax {...}

template GenerateCode subof Semantic{...}

package Compiler{
      inst GenerateCode;
}
```

Below are some of the changes that are made to the new templates:

```
template Syntax_A subof Syntax{...}

template Semantic_A <template S bound Syntax> subof S{...}

template GenerateCode_A <template S bound Semantic> subof S{...}
```

The template *Semantic_A* takes a formal template parameter that is either the template *Syntax* or a subtemplate of *Syntax*. In our case we can pass the template *Syntax_A* as an argument to *Semantic_A*. Instantiation of all six templates is done like this:

```
package Compiler_A{
      inst CodegenerateCode_A<GenerateCode
                  <Semantic_A<Semantic<Syntax_A<Syntax>>>>>;
}
```

With this instantiation we will only get one copy of each class from the template *Syntax*. This is because the only template that instantiates *Syntax* is the template *Syntax_A*. The template *Semantic_A* will in this case just instantiate the template *Syntax_A*. The instantiations above is visualized in the figure below:

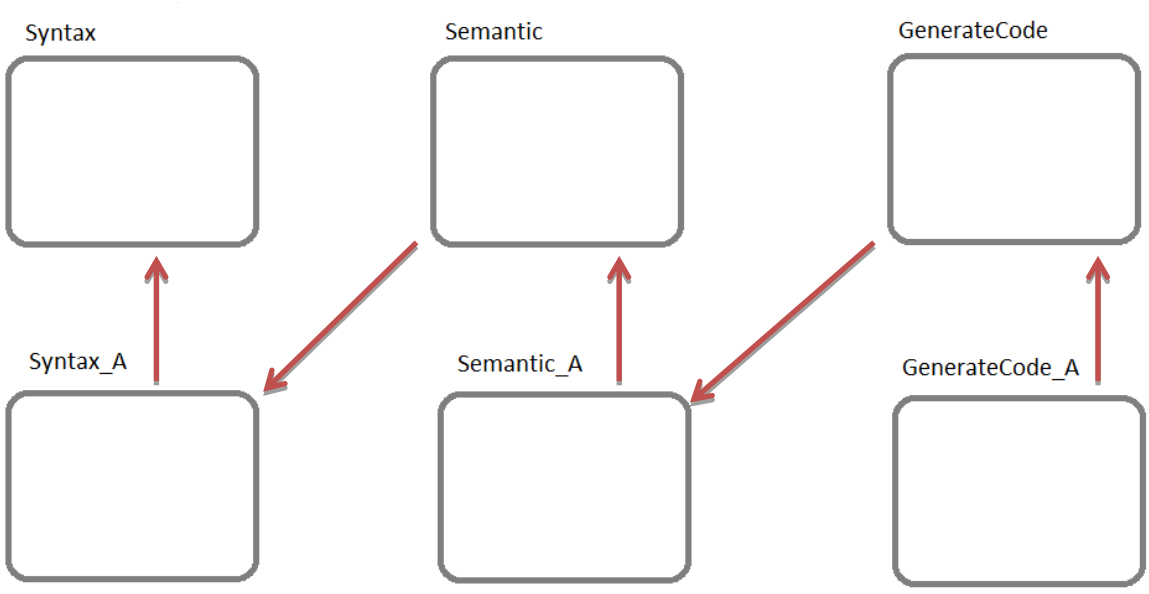


Figure 9 A visual representation of our expanded compiler using template parameters

Though using template parameters fixes our problem, we wouldn't be able to do any type of renaming in templates that are sent as arguments. This could be problematic since in programs that we need to do a merge of classes we will no longer be able to do so.

5.6.3 Tabstarct template instantiation

While I was working with this thesis an alternative mechanism for PT was being developed. This mechanism is called *tabstract template instantiation* and works in a similar way as *required types*. It will allow us to concretize the template given as "default" in *tabstract instantiation* with a subtemplate of the instantiated template. This could simplify the compiler templates as follows: The template *Semantic* could instantiate the template *Syntax* and when another template instantiates *Semantic* it could concretize the instantiation of the template *Syntax* with instantiation of a subtemplate of *Syntax*. The code below shows a version of our main three templates using *tabstract templates*:

```
template Syntax {
      // classes for the abstract syntax tree
}

template Semantic{
      tabstract inst Syn default Syntax;
      //additions to the classes from the template Syntax
}

template GenerateCode {
      tabstract inst Sem default Semantic;
      //additions to the classes from the template Semantic
}

package Compiler{
      inst GenerateCode;
}
```

We can see that, while the template *Syntax* will stay the same, the other templates will have some changes. For the template *Semantic* it will now make a *tabstract* instantiation of the template *Syntax* by adding the keyword *tabstact* before the *inst* statement. Then the instantiation is given a name, which in this case is *Syn* which will eventually be set to the default value *Syntax* if nothing more specific is said in later instantiations. Similar changes are made to the template *GenerateCode* as well.

When we want to expand the existing compiler like we did in the previous section, we can give a new default, which must be a subtemplate of the previous default. Below is an implementation of such expansions.

```
template Syntax_A subof Syntax {...}

template Semantic_A subof Semantic {
      tabstract inst Syn default Syntax_A;
      ...
}

template Codegenerate_A subof CodeGenerate {
      tabstract inst Sem default Semantic_A;
      ...
}

package Compiler_A{
      inst GenerateCode_A;
}
```

In the additions the three main templates are as in the case with template parameters are subtemplate. In this case they will again make a *tabstact* instantiation, but the only difference from the previous code example is that they will now replace the previous *tabstract* instantiation with a new default template that they will instantiate. The template *Semantic_A* will now instantiate the template *Syntax_A* (the new default template) instead of the template *Syntax*. Suing the abstract template mechanism will also avoid getting multiple copies of the same classes from the template *Syntax*, since it is only instantiated by the template *Syntax_A*.

Another way we can use *abstract templates* is to concretize them like we do with *required types* (see code below). In the second tier of additions to the compiler, we have the template *Semantic_B* which is a subtemplate of the template *Semantic_A*. Later in a package *Compiler_B* we can instantite GenerateCode_B and concretize the instantiation with the name *Sem* with the template *Semnatic_B*. The template *Semantic_A* is still instantiated, but the instantiation is only done by *Semantic_B*.

```
template Semantic_B subof Semantic_A{...}

package Compiler_B{
      inst GenerateCode_A with Sem <= Semantic_B;
}
```

5.6.4 Subclasses as a option

Now we will look at how similar additions can be done with regular Java subclasses and which of the methods is easier to use. With subclasses we have multiple ways for us to write a

compiler. One way we can do this, is to create classes for each unique templates class in all of our main templates. So that for example, *ClassDecl* in this version will contain all the code for syntax, semantic check and bytecode generator in one single Java class. If we then want to add something to the compiler, we can make a subclass of the class we want to add to.

This subclass will then include all the additions. Since a class can have many subclasses, we can make many different versions of a specific compiler. The disadvantage with this solution is that every class consists of all three parts (syntax, semantic and code generation), so that it would be difficult for us to implement a compiler where we for example want to use the semantic part of one of *ClassDecl* subclasses and bytecode generation from another. If we for example want to add to the syntax part of one version of a class and then add to the semantic part of another version, then we wouldn't be able to combine these two since the combined subclass will have multiple superclasses, which we don't want to merge.

Another way we can represent our compiler is for every template class in our Syntax template to have their own local class. Then each of these classes will have a subclass that includes the code for semantic analysis. The same is done for the generation of bytecode. Even though this lets us add to different part of a compiler (syntax, semantic or generating code), but we get a problem with multiple superclasses. This is demonstrated in the figure below with an example of additions with subclasses.

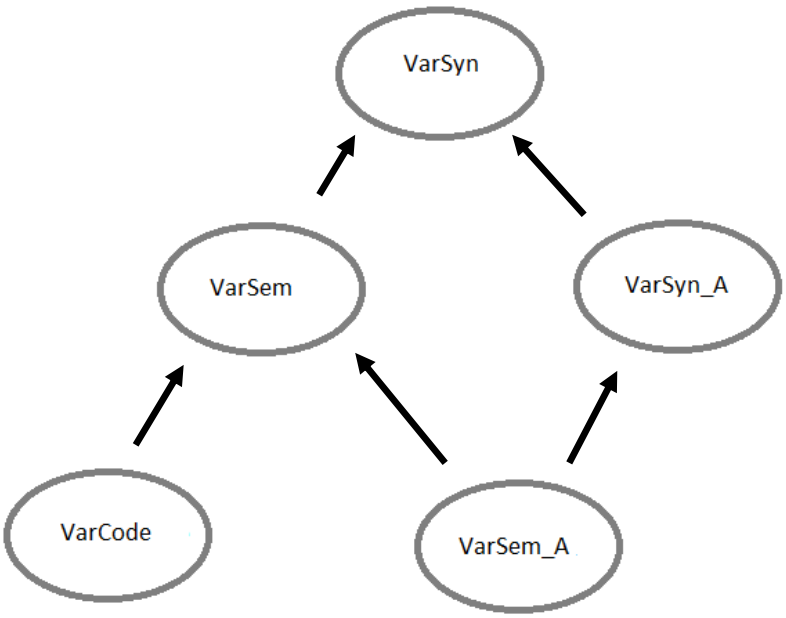


Figure 10 Expanding a compiler using subclasses

Here *VarSyn_A* and *VarSem_A* are additions to *VarSyn* and *VarSem* respectively. Since *VarSem_A* also wants to use the new syntax from *VarSyn_A*, it will end up with two superclasses.

For this example, it seems that using PT is easier than just using Java's subclasses. Using template parameters we are able to avoid the problem of having multiple superclasses like we did with Java's subclass solution.

5.6.5 Visibility regulation

Finally we will also look at visibility regulation. Since our compiler has many subclasses, we decided to use the protected modifier on many of the methods and variables, so that these members in Java classes can only be seen from the package in which the class was declared and from subclasses of that class inside other packages. Using *aprotected* modifier has not much use in this case, since certain parts of our program require subclasses. For example, the *Program* class has a list of different declarations in the source program and is of type *Decl*. So that objects of classes *ProgDecl* and *ClassDecl* can be inserted into that list, these classes would need to be subclasses of the class *Decl*.

The *eprotected* modifier can be used in cases where one doesn't want certain members to be accessible from add parts and therefore limit the additions that can be made to a compiler. In our compiler we have little use *eprotected*, since we want it to be added to from other templates. So the *protected* modifier that works for both the subclasses and the *adds* classes seems more appropriate in our case, because we want that "adds" class to have accessibility to the protected members.

Chapter 6

Graphical User Interface

In this chapter we shall look at the use of PT in two cases connected with GUI programming. Both of them will center on the Java GUI libraries Swing and AWT. In the first case we will assume that Swing and AWT are as they are now (as collections of Java packages), and look at whether PT can be of any help in implementing a specific GUI design.

The other case will look at what would happen if we decide to rewrite the whole Java GUI library so that it, at least to the user, appears to consist of a set of templates. And try to do this in a way that will make the library more easy and flexible to use than the current version. We will then discuss whether, and in what sense, we have obtained this.

But before we start looking at these cases, we will give an overview of the parts of Swing/AWT that we will use.

6.1 AWT and Swing

Our aim in the first study is to write a program that uses the existing version of Java's AWT (Abstract Window Toolkit) and Swing libraries. AWT is a collection of classes that provides us with some basic GUI mechanisms, which includes operations for creating buttons, 2D shapes and much more. Like AWT, Swing is also used for GUI programming. The Swing library builds on AWT, but provides additional features that are not present in AWT. For this reason Swing components can also use operations from the AWT's library and in our study we will mostly be using Swing with a few classes from AWT.

The historical relation between Swing and AWT is that AWT was the first GUI library for Java. However, with experience the creators saw that they needed more powerful and flexible components and a library that would have the same look no matter what platform you run it on (which was not necessarily the case for AWT). Thus, they created a new library, Swing, that included these ideas. However the relationship between the two became somewhat confusing as the old classes from AWT are still there in Swing, and partly has duplicate classes of those in AWT (with the same names, only preceded by a "J").

All of these Swing classes are extensions if similar classes in AWT. For example, *JFrame* is an extension to the AWT class *Frame*. Among the classes that Swing provides, there are several top-level containers[5]. Such container classes are essential in a Java GUI program, since all other GUI components need to be added (through a method) to the object of that class. Therefore every GUI program needs to have at least one top-level container, because without them no component will show up on the screen. For the cases that are described in this chapter we will use the top-level container class *JFrame*. Objects of the class *JFrame* appears on the screen as a graphical window that will show all the other components that we inserted into it. Components can be added to the window with the method *add(...)*. This class also has several methods (that we will use in the first study) to set up or change the appearance of a window (e.g. by changing the size of the window, background color). A typical GUI program using Swing is shown in the figure below.

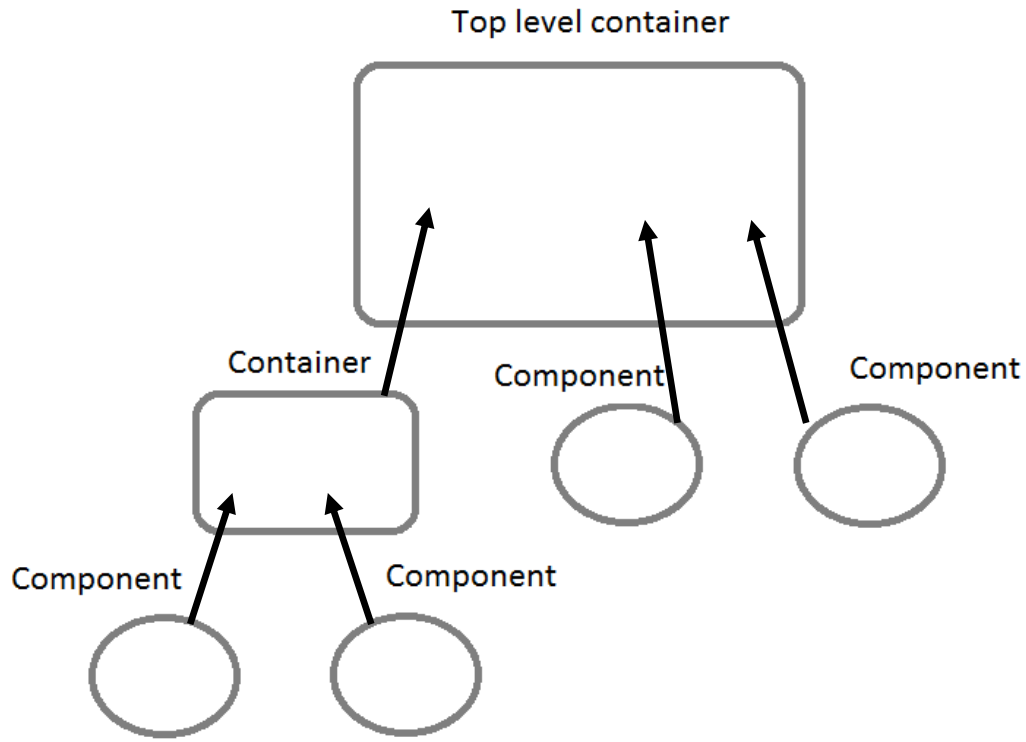


Figure 11 How a typical GUI program looks like

The component and container elements in the figure are object of the subclasses of the component and container classes respectively. By adding component to containers or container to containers, we get a tree like structure to our GUI program where the top-level container is the root element.

One of the main component classes in Swing is the abstract class *JComponent[6]*. Subclass of this class will inherit different functionalities like painting (with the method *paintComponent*), resizing, binding a component to a key and much more. One important class that extends *JComponent* is *JPanel*. Unlike many other classes in Swing, *JPanel* works both as a container and a component[7], and can be added as a component to container. One of the benefits of using other containers like *JPanel*, is that we can place multiple components into that container and then if we need to change e.g. the position of all of these component we only are need to move the container, instead of moving each component individually.

The class *JButton* is also extends *JComponent*. The *JButton* class includes methods for creating GUI buttons[8]. To make sure that a button actually performes an action when pressed, a class (that typically creates *JButton* objects) will need to implement the interface *ActionListener* and its method *actionPerformed(...).* Since a button needs to be added to a container, the container class can create a listener for each of its buttons. What this listener does is that it listens to a specific button (one can create multiple listener, one for each button) and if the button was pressed, the method *actionPerformed* is called. In *actionPerformed* we can determine (with the help of e.g. formal parameters) what button was pressed, and perform the appropriate actions.

From the *AWT* library we shall use the class *BorderLayout[9]*. This class helps a container with organize components by offering five ways of placing the components in the window.

6.2 Design of a specific GUI

For a first case we will implement a GUI program using PT. But before going into an implementation part of the program, we will first describe the look of a specific GUI. When the program is run we want it to display a graphical window with several other graphical components that are displayed in that window. One of the components that we want to show in the window is a button that we have two functions when it’s pressed. One these functions will be to change shape of a geometric figure and the other is to increase a numeric value. Both the figure and the numeric value will be displayed in the window. Then we would like

for the figure to change color based on user input. So for this we would to have several buttons, where each buttons is assigned a specific color and will change the figure to that color. Below is a sketch of how we want our GUI to look:

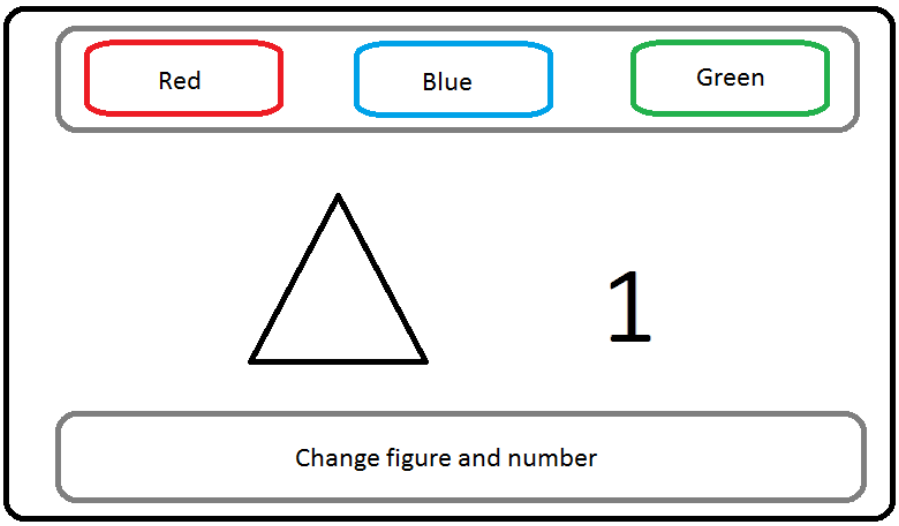


Figure 12 An example of the GUI program we want to program in the first case

6.3 Implementing a specific GUI

In this section we will continue with the GUI presented in previous section and discuss how we will implement this GUI. The classes from Swing that we will use in this program are *JFrame*, *JButton*, *JPanel* and *JComponent*. For our GUI window we will need to use a top-level container and for that we will use the class *JFrame*, so that we are able to display graphical window and add components to that window. The geometric figure and the numeric value will both extend the Swing class *JComponent*. All the buttons that are displayed in the window are subclasses of *JButton*. Since we want all the color buttons to be placed near each other, we can put all the buttons inside a container, since it makes it easier to move them all at once, painting borders etc.

Before starting with the implementation of the GUI program with PT, I made a Java version of the same program. This is so that I had a working version that will be the basis for my PT implementation and also have something to compare the PT version to. When deciding how this program should be split up in template to achieve better flexibility (for this and similar GUI-design), I tried looking at what parts of the program that naturally belong together and place them in their own template. For example, the part of the program that draws the different figures could be placed inside its own template. In the next section we will describe the PT implementation of the same GUI program.

6.3.1 The template Frames

This section describes the template *Frames* and the class *MainFrame*, and how this class creates our main graphical window (se the program below). Since we decided that our program shall include a menu, we could already in this template place that component somewhere in the window. Since we want our program to be as flexible as possible, we won't include the code for the menu in this template. Instead we will use a *required type* called *Menu* that will be concretized with a class during an instantiation of *Frames* and provide us with a menu that satisfies a criteria specified in the *required type*.

A class that can be concretized with the *required type Menu* must extend *JPanel* (so that the menu works as a container for buttons) and implement the interface *ActionListerner* as well as having a constructor that takes a *Mainframe* object as a formal parameter. This ensures that the menu class will be able to access the other component (when they eventually will be added to the *MainFrame*) by using the *MainFrame* pointer. Below is the code of the template *Frames*:

```
template Frames{

	required type Menu extends JPanel implements ActionListener{
		Menu(MainFrame mf);
	}


	public class MainFrame extends external JFrame{

		Menu m1 = new Menu(this);
		CenterComponent c1;
		MainFrame(){
			...
			add(m1, BorderLayout.NORTH); //place on top of the frame
		}
	}
}
```

Since *MainFrame* can make an object of any visible *required type*, we can already make an object of *Menu* and place the object at the top of the window (and this will be an object if the final version of *Menu*, not yet fully defined).

6.3.2 The template Menu

For the menu part of our program we will create a separate template called *ColorMenu* (se the program below). Here we create a class *ColorOptions* that extends the class *JPanel*. Since we have decided that the menu is a container that is to be placed inside a window, the class *ColorOptions* will then be used together with the *MainFrame* class. We will therefore instantiate the *Frames* template.

We wrote the template *Menu* to be able to change color of a component in the window. We therefore need the class that represents those figures to have a method that can change the color of components. Since that class is not yet visible from the template *ColorMenu*, we can again use *required types*. The *required type FigureComponent* can set a color for its figure, we write a constructor that takes a *Color* object as a formal parameter (initial color) and a method "setColor(Color r)" that sets the color of that figure.

The code below shows some the parts of *ColorMenu* template:

```
template ColorMenu {
      inst Frames with Menu <= ColorButtons;

      required type FigureComponent extends JComponent{
                  FigureComponent(Color c);

                  public void setColor(Color c);
                  public void paintComponent(Graphics g);
      }

      public class ColorButtons extends external JPanel implements
                                                      ActionListener{
            ...
      }

      class MainFrame adds{
            ...
            void changeColor(Color r){...}
      }
}
```

6.3.3 The templates PaintFigures and PaintNumbers

The templates *PaintFigures* will include a class *RandomFigures* that can display random geometric figures (based on an integer generated by a random number generator). Objects of this class will need to be inserted into a window. Therefore the class *RandomFigures* is a

subclass of the class *JComponent* and includes the methods *setColor* (to set current color of the figure) and *paintComponent*. We will also add to the class *MainFrame* a method that creates a *RandomFigures* object, as well as insert it into our window and set the size of the figure.

For the template that includes the class that paints a numeric number in a window, we will do a similar thing as we did with *PaintFigures*. The template *PaintNumbers* make an instantiation of Frames and includes the class *IncNumbers*. As with the class *RandomFigures*, we add to MainFrame a method that creates an object of IncNumbers and put it into the window. To change the number in the window, we will also add a button to our window, that when pressed will increase the number and repaint the window to show that new number.

Below is a part of the implementation of *PaintFigure* (*PaintNumbers* looks very similar to this):

```
template PaintFigures{

      inst Frames;

      class RandomFigures extends external JComponent{
            // draws random figure
      }

      MainFrame adds{
            MainFrame(){tsuper; addToCenter();}

            void addToCenter(){...}
      }
}
```

6.3.4 template MainGUI

In the end we want to combine all templates together in a template to create a functional GUI program. We will place this program in a template instead of a package, so that the joined program can be further added to. The person who wants to run the program, can instantiate *MainGUI* in a package. Below is the code for the template MainGUI and the figure that shows all the final instantiations. However, these instantiations will result in a compiler error, because the templates *PaintFigures*, *PaintNumbers* and *ColorOptions* each make their own instantiation of the template *Frames*. This will result in us getting three copies (as

instantiations) of class *MainFrame* and a name collision between all the members of the different copies of *MainFrame.*

```
template MainGUI{

      inst NumberFigures;
      inst PaintFigures;
      inst ColorOptions with FigureComponent <= RandomFigures;

      class MainFrame adds{
            MainFrame(){
                  tsuper();
                  super("GUI example for master thesis");
            }
      }
}
```

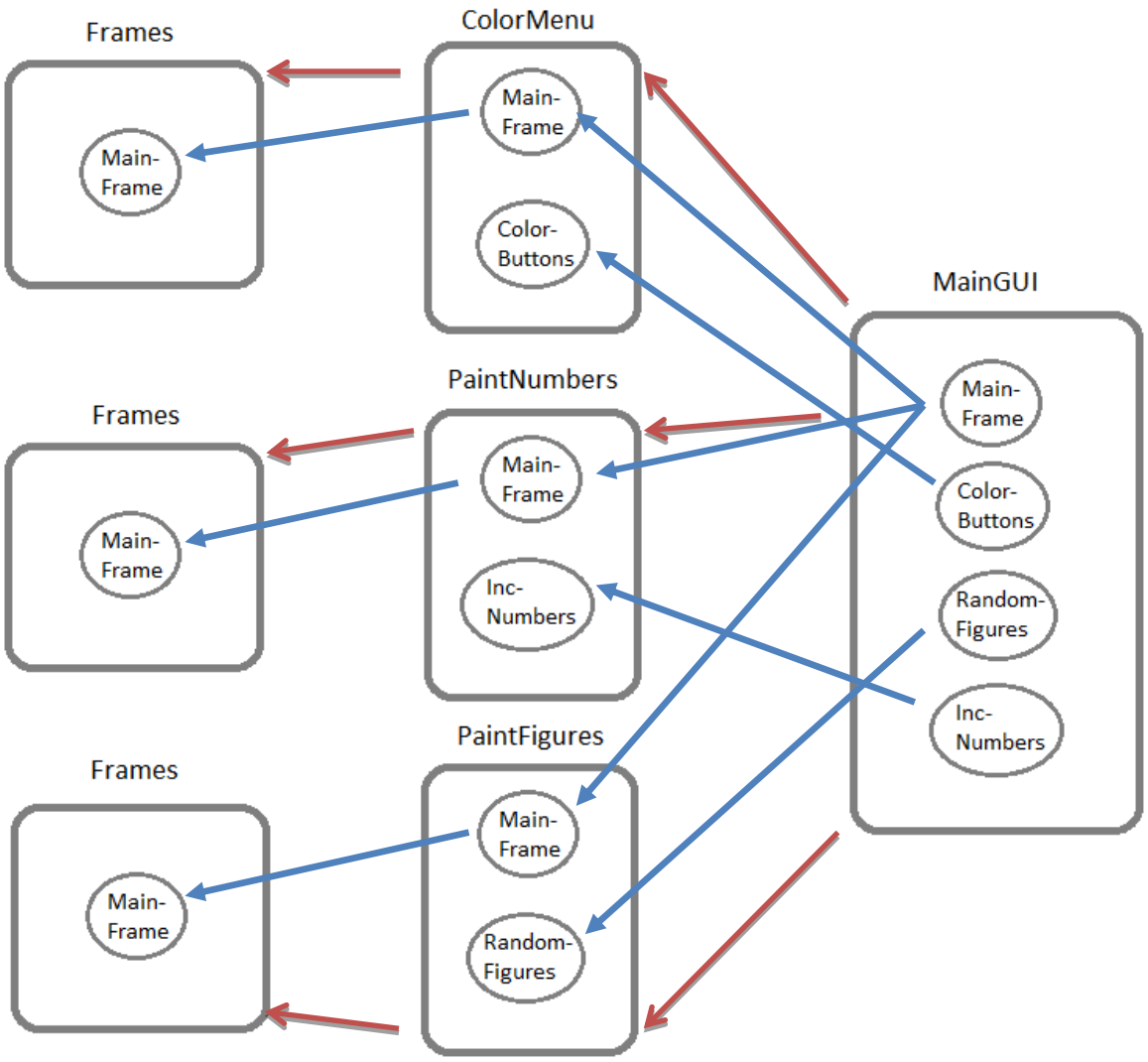


Figure 13 Visual representation of our GUI-program

6.3.5 GUI program with templates parameters

To try to get around the instantiation problem from last section we can again try to use PT's template parameters. Like in the previous case and chapter 2, we will try to use template parameters to reduce the number of instantiations we make of a specific template.

By letting templates that need to instantiate *Frames* have a formal template parameter that is a subtemplate of *Frames*, we can instantiate these templates together and make just a single instantiation of *MainFrame*. Each of these templates should also be a subtemplate of the formal template parameter, so to make sure that they can also be sent as arguments to other templates. The code for the new templates is below, as well as a figure that shows how each resulting instantiation will look like. The code only shows *ColorOptions* and *MainGUI*, since *PaintFigures* and *PaintNumbers* have the same changes done to them as *ColorOptions*.

```
template ColorOptions <template F bound Frames> subof F{
      ...
}

template MainGUI{
      inst ColorMenu<PaintFigures<PaintNumbers<Frames>>>;
}
```

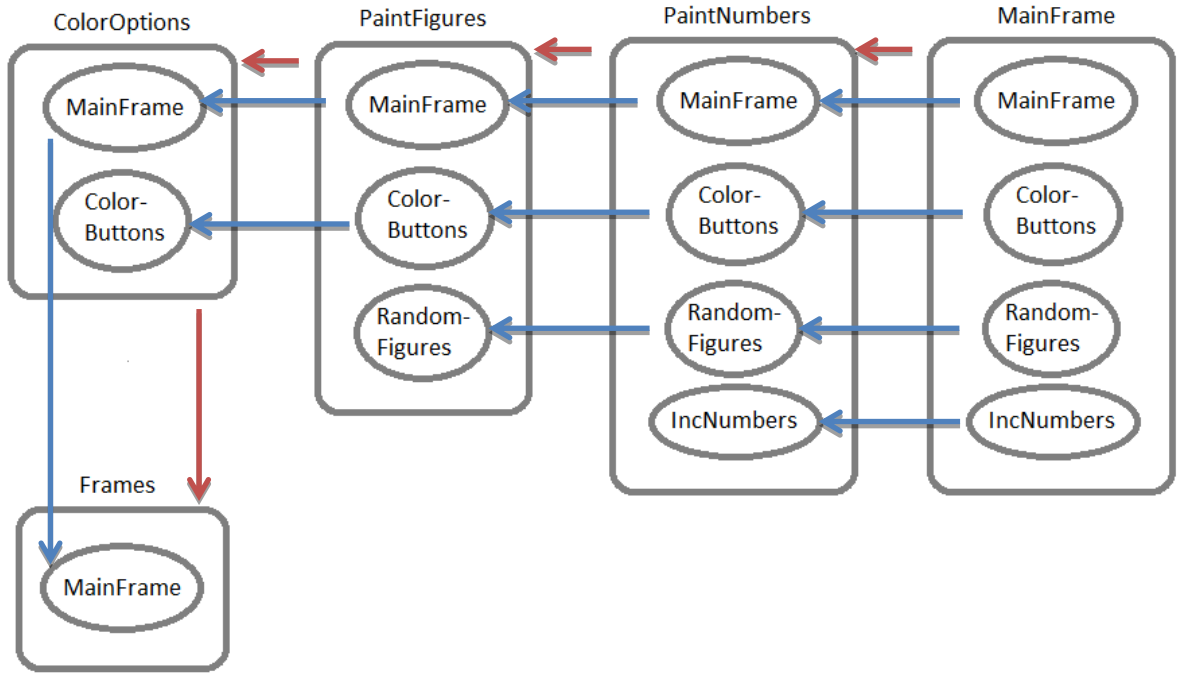


Figure 14 Visual representation of our GUI-program using template parameters

Even though we can resolve the problem with multiple unwonted instantiation of a single template, we will instead get a new problem if we use template parameters for this case. When a template is instantiated, through subtemplate instantiations or when it's sent as an argument, we aren't able to rename any of the classes and their members. Since we can't do any renaming, then we wouldn't be able to concretize the *required types* with a class in most of the templates. This will result to that we won't be able to compile the program, since our implementation has multiple *required types*.

To get around this problem we could avoid using *required types*. So instead of adding a method that changes the color of the figure to *MainFrame* in *ColorOptions* template, we could wait with that until we actually combine the templates together (since the template *PaintFigures* hasn't been instantiated yet). Then in the template *MainGUI*, we can add code for the class *MainFrame* that calls on the method *setColor* and changes the color of a figure. The problem with this implementation is that we have to postpone writing implementation for certain parts of the code and leave it all to the programmer that makes the final instantiation of these templates. This can make it more difficult for the programmers, since they may have to spend time to get themselves familiarized with the original classes to finish the code for that class.

6.4 Evaluation

After working with this example I found *required types* to be quite useful. For one, we obtain the ability to write code when a certain class hasn't been defined yet, which is what we did with the *Frames* template. There we could start writing code for placing a menu in a window, before we had a class with an implementation. This is helpful, because if the original programmer of a template uses required types, the programmer can already there write an implementation of the program using these required types. A new programmer that makes an instantiation of that template would only need to concretize the required types with the new classes and avoid spending time to understate every line of code in the instantiated template to complete the it's code so that it works with the new classes.

Another benefit that we get with this case is that we can concretize a *required type* with many different classes. So for the *ColorOptions* template, which requires a class with specific methods, it isn't bound to just a single class that draws figures. Instead *required types* allow us to create many different classes that have their own unique implementation of painting figures and still use the same menu, without having to write a new frame/menu class for every

new figure painting class. Both of these benefits can be useful to make a good flexible and reusable program, and therefore we should try to make them available for template parameters as well.

6.4.1 Solutions for *required types*

Since the problem we have with *required types* has to do with not being able to rename them, we will try to look into different ways to either get around this problem or propose a change to some of the mechanics that could help us with renaming.

We can start by looking at what we can do with template parameters so that we are able to concretize the *required types* with a class. Since a template can take another template as an argument, an instantiation of the template is made. Depending on the future implementation of template parameters, instantiation is made implicitly when a parameter is passed or instantiation of the argument has to be made inside the template body. If the instantiation has to be made inside the templates body, then we could make renaming the same way we would do renaming during regular instantiations. If the case is that instantiation is made when a template is sent as a parameter, we could do renaming in the same statement as the definition of the template parameter. Templates can also make subtemplate instantiations that is not made inside the template body, but the same statement as the template name. We could therefore allow renaming to be done after *subof* statement.

Below is a proposed renaming with formal parameter and explicit instantiation:

```
template M {
      required type R{...}
}

template T < template n bound M with R <= F > {
      n: with R <= F;
      class F {...}
}

template T2 subof M with R <= G {
      class G {...}
}

package P {
      inst T2;
      inst T<M>;
}
```

Here we have three templates M, T and T2, where only M has a required type R. In the template *T* we concretize the class F (defined in the template body) with the *required type R* in the angle brackets of the template parameter statement. The template *T2* makes the same type of renaming during the explicit instantiation of *M*.

6.4.2 Challenges with renaming using template parameters

Though allowing renaming with template parameters or the redirecting mechanism is really helpful to us, but implementing renaming will also bring some challenges. To show this we shall look at our GUI example that is using template parameters. When *ColorOptions* gets *Frames* as a formal template parameter, it might want to rename a couple of variables and methods. Then *PaintFigures* gets *ColorOptions* as formal template parameter. Since *PaintFigures* is doing an addition to *MainFrame*, this template expects that class to have variables and methods with specific name.

If the template *ColorOptions* has renamed one of *MainFrame's* members that *PaintFigures* requires, we will get an error while compiling the PT program since the necessary member no longer exists. Avoid or fixing this problem will not be patricianly easy. We can't exactly know what kind of renaming the templates have done before the template that is currently trying to do its own instantiations. For now we could consider looking into allowing merging of *required types* for template parameter instantiations and explicit instantiations, and try to find a way to avoid the program described in this paragraph. If this is possible, we can make

sure that the templates can expect the correct classes and their members and at the same time fix the problem we had with *required types*.

The problem with letting renaming to be done by templates is that they will be instantiated later, and the renaming might cause problems in other templates. Unlike templates, packages in PT on the other hand don't instantiated. So letting packages do renaming won't be as problematic as doing the same for template, since this renaming will be the last one that is done for this program. No other template or package will require this package to have any of the classes, *required types* etc. that they need. So letting package de some renaming could also help us with the problem of not being able to rename *require types* and the same time not cause any problems for other templates and packages.

6.4.3 Alternative instantiations

So far we have looked at how we can expand other existing and new mechanism to allow renaming to be done in different situations. In this section we will discuss how we can avoid this problem all together by using the mechanisms that are currently available, as well as different ways we could instantiate templates.

In the implementation of our GUI program, multiple templates needed to instantiate the template *Frames*, since *Frames* is the template that included the top-level component and most other templates in our program do additions to *MainFrame* to add their component to the window.

One example of instantiation we can make is always instantiate GUI program in a specific order. An example of this type of instantiation is after the template *ColorOptions* has instantiated Frames, the template *PaintFigures* will instantiate *ColorOptions* and *PaintNumbers* will instantiate *PaintFigures*. This creates a specific order of how the different parts are added together and therefore only have a single instantiation of the template *Frames*. The disadvantage with this solution is that it makes it difficult to add to the different templates, since they might instantiate other templates that we don't want to use. Therefore if we wanted to instantiate *PaintNumbers*, we would also get the classes from *ColorOptions*, even though we wanted to use a different template for our menu. This is because we are instantiating a specific template, unlike with template parameter, where we can instantiate different templates in any order we like. Therefore having some templates be separate from each other makes them easier to expand upon later.

Another way we can instantiate our GUI program is by not having templates with component classes, instantiate the template *Frames*. The figure below shows a different way the different templates can be instantiated.

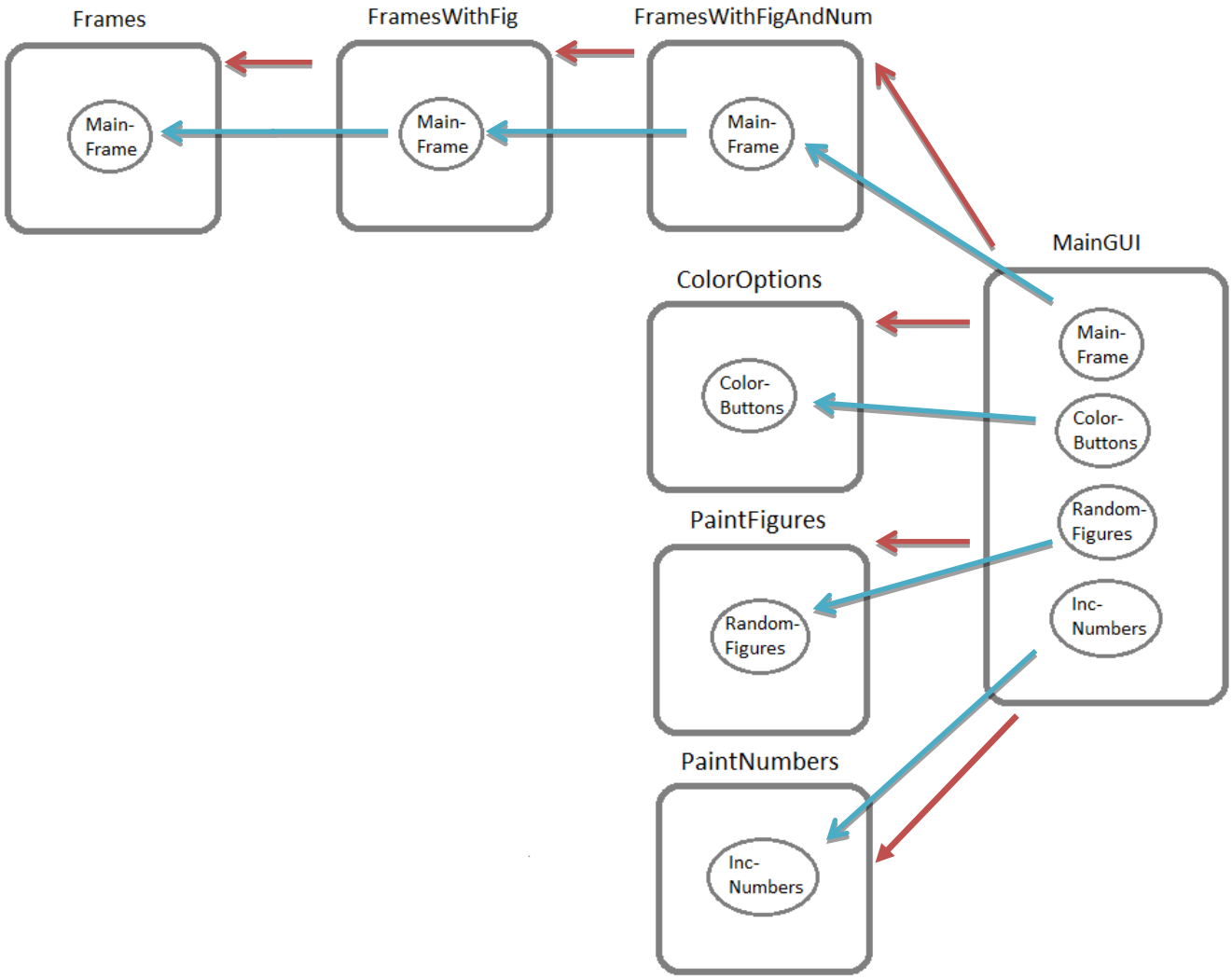


Figure 15 Visual representation of our GUI-program with an alternate instantiation

Where previously the templates *ColorOptions*, *PaintNumbers* and *PaintFigures* each instantiated the template *Frames*, in this version these templates won't be instantiating anything. Instead, the *PaintNumbers* and *PaintFigures* will just include the classes needed for painting their components. Both of these templates don't instantiate Frames, because we will instead make additions to the class *MainFrame* in another template. The template *ColorOptions* on the other hand need to have a pointer to a *MainFrame* object, since it will need to change the color of the figures and therefore call on of *MainFrame's* methods. To not instantiate Frames, it will instead include a *required type*, that has the same methods as *MainFrame* that are needed by the classes in *ColorOptions*.

While *Frames* template will stay the same, we will create two additional templates that will expand on the *MainFrame* class. The first of the new templates is *FramesWithFig*, which instantiates Frames. The additions that it makes to *MainFrame*, are the same addition that we made in the previous version of the template *PaintFigures*, which is to create a component (in this case random figures) and add it to the frame. The next template is *FramesWithFigAndNum*. Here we instantiate FramsWithFig and create a new *required type* that will be concretized with the class *IncNumbers* and includes the methods that are needed by *MainFrame*. In the end, the templates *FramesWithFigAndNum*, *PaintFigures*, *PaintNumbers* and *ColorOptions* are instantiated in a package/template and then all the *required types* are concretized with the compatible classes.

In this type of instantiation of templates, we will try to avoid instantiated the same templates more times than we need to. For the classes that we need to create objects of or call on their methods, we instead create *required types*. In cases where we need to add to a class (like in a previous example of *PaintFigures* where we added to *MainFrame),* we could create a new template (the template *FramesWithFig*) that instantiate the template that we want to add to (in this case *Frames*) and make the additions in that template.

Using *required type* will in a way replace the necessity of instantiating certain templates, since that type has all the methods that we need. This also reduces the need to use template parameters for prevention of multiple instantiations of the same templates.

6.4.4 Subclasses and type parameters as an option

Now we will look at how we could program the same GUI program with similar additions without using PT. The additions to any of the parts in our GUI program can be done by making a subclass. The class *MainFrame* needs to be changed so that it would be able to take in different components and add them to the window. For adding component, we can create several methods that each takes in different objects as formal parameters. For example a method "*addPanel(JPanel p)"* will take in an JPanel object at a formal parameter and add it to the window. This allows us to send any object that is a subclass of *JPanel* as an argument. The same thing can also be done for classes that extend *JComponent*. So that a method works with many different objects, we can use type parameters. A method "void <T extends JComponent> addToFrame(T t)" can take any object that extends *JComponent*, which in our example could be the menu that extends JPanel (which is a subclass of JComponent) or even the class RandomFigure (which extends *JComponent*).

The benefits we get from using *required types*, is that we can concretize them with many different classes that meet the criteria of that type. As seen in the previous paragraph the same benefits can be achieved by using type parameters. By using type parameters we can also create additional restrictions to the type parameter by making that parameter extend a class or implement an interface. What we can do with *required type*, but not with type parameter is expanding the restrictions for what class can be concretized with a *required type*. Below is an example of code that tries to add additional restrictions to a type parameter, but fails because it breaks the rules that are set for overriding methods. In this example *A*, *B* and *C* are classes, while *D* is an interface:

```
class A{

      public <T extends C> void print(T t){...}

}

class B extends A{
      ...
      public <T extends C & D> void print(T t){
                    super.print(t);
                    ...
      }
}
```

In chapter 2 we discussed what *Java* requires from a method that supposed to override one from a superclass. In addition to that, the type parameters must also have the same bounds. This is not the case in the example above. Class B tries to override "print" and at the same time add an additional bound to type parameter T. This is not possible in Java, since the type parameter needs to have the same bound in both versions.

With *required types* we don't have this type of restriction. Instead we can add to *required types* additional methods and interfaces, and extend classes (if the *required type* doesn't have a superclass or the superclasses can be merging with another class). This gives us a lot of flexibility which can be helpful when doing additions to a class, since the methods and classes will change over time and will require additional properties from other classes and types (parameter types for Java and *required types* for PT).

A lot of the benefits we want to achieve in GUI example by using PT can also be achieved with regular subclasses and type parameters. Most of the additions we have done in PT for

this example can also be done with subclasses. Type parameters, like *required types*, can represent multiple different classes. But there are also some additional benefits from using PT. There are some types of additions that are just not possible with using just subclasses. Like the cases where we want to use code from many different classes, but since a class can't have multiple superclasses, the addition can't be made. This is similar to the problem we had in the compiler example.

6.4.5 Visibility regulation for required types

Since we allow classes to have public or private visibility regulation, we can also look at *required types* and if they need a similar modifier. Since *required types* need be concretized with other classes, giving them a private modifier, so that the *required type* will not be visible outside the template that it was declared in would be pointless. Putting private modifier on a *required type* will only make that type visible inside the template it was declared in and can therefore never be concretized with other classes. If we have no real use for private modifier, do we then need a public modifier? Since the *required types* need to be seen for any other template for it to be concretized, then we wouldn't really have any need for a public modifier, but instead make *required types* public by default.

Another way we could use the *private* and *public* modifiers is for them to work as additional restriction for what classes can concretize with a *required type*. For example, if a *required type* has a *private* modifier, then it can only be concretized with a class that has all the same methods as the *required type* and the class has the *private* modifier. The same case can be made for the *public* modifier.

If we allow *private* and *public* instantiations to be made, we will also need to discuss what this can do for the *required types*. All the classes we get through a *private* instantiation of templates will make these classes only visible in the template that made the private instantiation. The other templates will have no access to any of these classes. But we also need to choose what happens to *required types* in that type of instantiation. If we choose to make these *required types* private to a template, this will mean that no other template will able to concretize it with another class. Therefore that *required type* need to be concretized in that template and not in other template. Else it will never be concretized with a class, since no other template is able to see it.

To make sure that a *required type* gets concretized with a class, we can put a requirement that says that if a *private* instantiation is made then a *required type* must be concretized with a class in the same template. If the programmer doesn't do the concretization on a *required type* that is used in another class, then this should then lead to a compiler error, since the class won't have the right requirements set by the compiler.

For a *public* instantiation of templates, the *required types* (like template classes) should be visible from all templates that get copies of that type from instantiation. In this case we won't get the same issue as with *private* instantiations, since the content of the publicly instantiated template is visible to other templates. Another solution could is letting *required types* ignore the *private* and *public* instantiations all together. Even though a *private* instantiation has been made, all the *required types* unlike in previous paragraph will still be visible to other templates. In this approach *required types* will always be public.

When *required types* get concretized with class, the class might have a visibility modifier. The default public modifier for that *required type* should then be ignored, so that if the class is set as *private* that still will be the case of the concretized with the *required type*. This is also the case for the *public* and *no modifier*.

6.4.6 Visibility regulation for template classes and their members

Now we will look at regular template classes and what effect will the different visibility modifier have on them. We will start by looking at the *public* modifier first. Letting the public modifier for template classes make them visible for all templates that get these classes as copies still seems reasonable for this example as well. This is especially true when we look at what we would like for other modifiers to do. Since none of the other modifiers grant full visibility, having one that gives us this option can be useful. An example of where we can use the *public* modifier at is to make the class *MainFrame* public. Since it is own main class (top level container), we would like to have this class to visible from many different templates so that they can add themselves to the window.

The next modifier that we shall look at is the *private* modifier. Since the *public* modifier makes the class and their members to all templates, it's only logical to have something similar for keeping the visibility away from other templates. For example, the *MainFrame* object in *IncNumbers* could be set as private so that no other class in other packages will be able to modify it. Having the same mechanism for templates, so that the same object won't be

accessed by any other template inside the same template and template classes in other templates can be beneficial.

Since having both *public* and *private* modifiers can have their use in PT, and then we might also need to have a mechanism in the middle of these two visibility regulations. Having the *protected* modifier work for templates as well as regular Java packages might be useful for cases where want to expand the visibility for the updated sub- and adds classes. Again we could take a look at the different versions of *protected* to see if they are useful in this particular example. Since most of our additions are made through the *adds* classes, we could have used the a*protected* modifier instead of the regular *protected* modifier. This *modifier* is useful in this example, since we want to avoid doing any additions with subclasses, but instead focus on the visibility for the PT mechanism. Though not in this example, *eprotected* can have a similar use. In cases where we don't want to use PT's *adds* mechanism and only want the additions to be made by creating subclasses, then we could use *eprotected*. The situations where these two special modifiers might be needed might not come as often, since one modifier will prevent the additions for being made in a different way than the modifier allows.

Since the different component in our GUI program a quite small and meant to be expanded upon, then making private instantiations of any of the components will have little use. But in the template MainGUI, where we make the instantiations of different parts of the program to combine them into a working program, we could make *private* instantiations of some of the templates. This way no further modifications can be done to any of the parts of the GUI, since this template is supposed to be an almost complete version of our program. We can still make instantiations of MainGUI, but the additions and the modifications of all variable of the private components will no longer be allowed.

We could also use a *public* instantiation on *Frames* template, so that this part is visible for further additions to our programs. If we have both *public* and *private* instantiation, we need to look at the default case, which is a default instantiation without any modifiers. For this case, there aren't many different options available. If the default case is to make the instantiation public, then that would make the *public* instantiation with a *public* modifier be pointless.

If in the default case the instantiations of templates are *private*, then there is no need for a *private* modifier. Since our choices are quite limited for visibility regulation, we could just go

with making the normal instantiation public. This is because the template they biggest benefit for using templates is the reusability, so if someone wants to a different instantiation they could just add a *private* modifier. But whatever we choose the default case to be, we can easily use a modifier that achieves the opposite effect. Overall, *private* and *public* instantiations gives us another way we can regular visibility that can't be done with other modifier discussed in this section.

6.4.7 Merging of classes

In this section we will be looking, by using this GUI case, into the multiple superclasses issue described in chapter 3. We will try to use the merging mechanism and try to find out what kind of benefits and problem we can encountered during our tests. For this we are going to use the classes *RandomFigures* and *IncFigures*. Originally the class *RandomFigures* was painting a random figure from the click of the button. Now we want that class to paint a specific figure depending on how many times the button was clicked. We could create an entirely new class that has a method that is called by the class *MainFrame* when the button is pressed or we can use some of our existing code.

The class *IncNumbers* already has all of the methods and variables need to increment its counter for number of click on the button from *MainFrame*. So we going to try to merge the classes *IncNumbers* and *RandomFigures*, and use the counter variable from *IncNumbers* to determine what figure we should be painting in the window.

The first thing we can start with is instantiating the needed templates and merge the two classes. The class *RandomFigures* has a method "paint(Graphics g)" that calls on the method "getNumber()" to get a random number. When the template *PaintFigures* instantiated, we are going to override "getNumber()" (from the class *RandomFigures*) method to now return the variable *curNum* (from the class *IncNumbers*) that is increased when the certain button is pressed. Though this piece of code could be solved with creating an object of the class *IncNumbers* and get the variable *curNum* with the help of a pointer, we going to continue with the merging solution to check what kind of problems we could encounter when we want to merge classes.

Since both the classes *RandomFigures* and *IncNumbers* have superclasses, when they are merged, the resulting class will end up with two superclasses. Typically we aren't able to merge superclasses from the external Java library, but in this case the merge is possible. If

both superclasses are the same class, then they can be merged to create a single superclass. In this example both superclasses are the class *JComponent*. This means that when merging *RandomFigures* and *IncNumbers*, the resulting class will have only one superclass, which is *JComponent*. Being able to make this type of instantiation gives us flexibility in how we want to expand or change the different classes in our program.

Another thing we could consider when it comes to merging is that the classes from the Java library might have superclasses themselves. Let's look at an example where we want to merge two classes that both have external superclasses. Unlike previous paragraph, in this example the superclasses are different. The figure below shows an example of a merge with what we want the resulting class to look as in the template on the right side of the figure, while the templates that are merged are on the left.

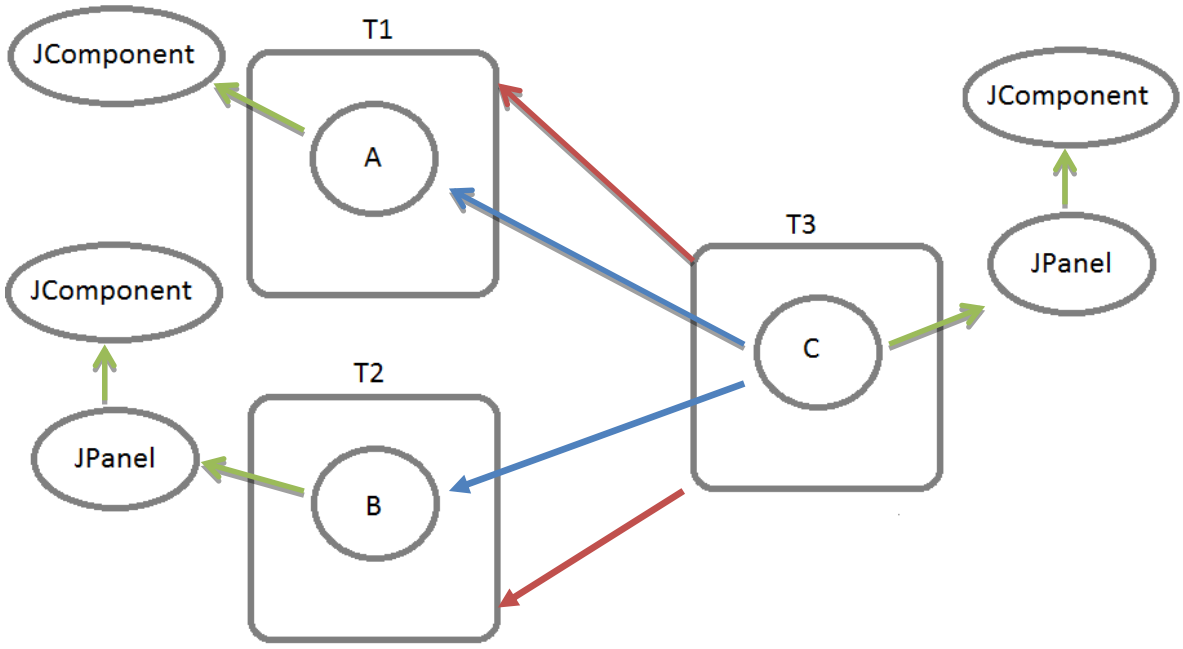


Figure 16 An example of a possible merge of superclasses

In the figure above we have three templates. Template *T1* includes a class *A* that has a superclass *JComponent*. Another template *T2* has a class *B* that extends the class *JPanel* from Java's library. Templates *T1* and *T2* are instantiated and the classes *A* and *B* are merged. This type of merge is not legal, since the resulting class *C* will end up with not one but two superclasses. Unlike previous example, these superclasses are not the same class and therefore can't be merged. Both of these superclasses come from Swing's library and in fact

JComponent is the superclass of *JPanel*. Therefore *JPanel* already inherited all the methods and variables from *JComponent*.

Since *JPanel's* superclass is the same class that we would like to merge JPanel with, we could, like in the previous example, allow the merger between the classes *A* and *B*. Of the issue here is how we get information on the class hierarchy of either class *A* or *B*. If the PT compiler allows this, we could traverse the superclass hierarchy until we find a match on either of the classes we want to merge. Since this type of cases might not occur too often, there might not need to implement this type of merge. But having another way to extend and combine classes gives PT an additional advantage over superclasses, since this type of addition is not possible with just the use of superclasses.

6.5 A new version of Swing/AWT using PT

So far in this thesis, we have looked at cases where we used classes from Java's library. To get access to these classes we need to import the package with library. Now we will take a look at a case where Java's library is programmed as templates and what kind benefits we can get through that.

We will start by looking into how to program these libraries (presented as Java packages) with templates. One way we can do this is to create template classes of the regular Java classes we find the different packages. Since a lot of these packages require classes/interfaces from other packages, they would need for this case, to instantiate other templates that have classes or interfaces that it needs for its own classes.

For example, a template that includes the class *JPanel* will need to instantiate the template that has the class *JComponent*, because *JPanel* would need to extend *JComponent* and therefore need this class to be visible for *JPanel*. Because in this case all libraries use PT, programmers who want to use these classes could take advantage of the many benefits that comes from using PT. Any of these classes could be added to or used in a merge unlike the current state of PT where what additional things we can do with external classes are limited.

Another way we could put the Java's library into templates is to use the *adds* mechanic on all classes. Each class will be programmed and placed in a template. Since some classes, like *JPanel*, requires a superclass, the template that includes *JPanel* needs to instantiate the template that includes *JComponent*. But unlike in previous paragraph, where *JPanel* extended *JComponent*, extending a class will not be needed. Instead we will, during the instantiation,

change the name *JComponent* to *JPanel* and then add to the new *JPanel* class all the necessary additions to create exactly the same class as that you will find Java's library. With this approach *JPanel* will no longer have a superclass. This will be done on all classes in Java's library, so that no or few of these classes have a superclass.

In our case from chapter 6.2 we had classes *RandomFigures* and *IncNumbers* extends other classes. By using the approach described in the paragraph above, we can make those classes even more flexible. For this we will use the *adds* keyword to create our classes. The template *PaintFigures* will instantiate the template that includes the class *JPanel* and rename it to *RandomFigures*. Then we add to *RandomFigures* the same members as it had in the previous cases. After this process the class *RandomFigures* will still behave in the same way as before. The only difference in this case is that *RandomFigures* no longer has a superclass. This approach can be very beneficial for a class, since it no longer needs to extend any of Java's library classes and can now extend another class. This will in a way let a single class extend multiple classes and therefore achieve more reusability than just using Java's subclasses.

Using the same approach as above can also lead to some drawbacks. With subclasses we get the benefits of subtyping. If we have three classes *A,B* and *C*, where *B* and *C* are subclasses of A, then a variable of type *A* can be made to point to an object of type *B* or *C*. If you use the approach from the previous two paragraphs, the class *A* will no longer exist, since it is now part of classes *B* and *C*. We will no longer we able to substitute a variable with either *B* or *C*, because they no longer have a common superclass. This approach is therefore not always ideal. In some cases we might need for superclasses to be their own classes, while in other cases we would like to avoid having superclasses.

Chapter 7

Conclusion

In this chapter we will go over the evaluation made in the previous three chapters and discuss whether PT is a useful mechanism or not based on these cases. When deciding the overall usefulness of PT, we will base the discussion around some of the topic brought up in chapter 3.

7.1 Did we get a better implementation with PT?

An important part of mechanism is that it actually works. In case for PT, it should create a working Java program. For all of the cases in this thesis that were coded using PT, the PT compiler generated a working Java program. All the changes that were made to the different classes that were made through additions, merge etc. were present in the new generated Java classes (after the template classes were instantiated in a package). Therefore when it comes to correctness of the generated program, PT does what it is supposed to do.

Next we will discuss whether PT can be useful in larger scale program, specially focusing on reusability. Since most of the Java code works the PT compiler, this means that all the reusability that we get for subclasses (and other Java mechanisms) can also be achieved using PT. If we didn't want to use any of PT mechanisms, we could still create reusable classes with the help of subclasses. If we wanted, we could also use the "adds" mechanism to achieve the reusability as with subclasses.

It is also important to look at if PT-specific mechanics can make our implementation better. As we have seen from the first three cases is that whenever we want for a class to inherit properties from more than one class, we can't achieve this with only subclasses. In our PT version Java Simulation program, we had the class *Process* to inherit from classes *Link* and *Coroutine*. With the use of PT we ended up with the program we initially wanted to create. We could of course have one superclass and a pointer to an object to another, but that might

not always lead to the desired result and might require more code than we would have with the PT version. Using the merge mechanism simplifies the coding process, as the programmer won't need to go through many hoops to get the desired program.

In the case of the compiler program from chapter 5, we wanted to do smaller additions to the different parts of the compiler. Here we again needed a class to inherit for two different classes. But unlike the Java simulation case, in this case we couldn't fix our problem with a simple merge. This was because the instantiations that were needed caused a template to have two copies of the same class, which resulted in a name collision. To avoid doing instantiations on the same template more than once, we instead tried to template parameters for our implementation.

When trying to do similar additions in chapter 6 with the first GUI case, we ended up with the same limitations. Trying to make more additions to our GUI program will, like in the previous paragraph, result in multiple instantiations of the same template. And just like in the previous paragraph the problem can be avoided with the use of template parameters, where only one single instantiation will be made of the template that we had a problem with. Since template parameters have not been implemented yet, it does limit how we currently can instantiate templates and thus might prevent the programmer from reusing code in cases like those described in chapter 5 and 6. Therefore there could be some incentive to have a mechanism like template parameters present in PT, since it can make PT better when it comes to making the programming process easier.

Another useful feature is *required types*. As we have seen in chapter 6, as with type parameters in regular Java, *required types* can be useful for programs that we want to work with many different classes. Since we can also make additions to and merge *required types*, we can further change the requirements for what classes that can be concretized with the *required type*. This can be useful, since our programs can get expanded further and over time the programmer might need these additional restrictions to make the program that they want.

With the GUI case, using required types can also make certain instantiation easier to make. Instead of instantiating a template with certain classes, we create required types that would have the same methods as these classes. With this approach we can avoid certain issues that can occur during an instantiation, like the case with multiple instantiations of the same template. Required types can therefore be very useful for certain programs.

Even though there are some benefits to using PT, there are still some limitations when it comes to reusability. Since no renaming can be done when using template parameters, we wouldn't be able to do renaming and concretize *required types*. This will prevent us from being able to compiler the PT program if a template is using both template parameters and *required types*, and because we can no longer do renaming, we won't be able to merge classes.

But the lack of renaming and concretization with template parameters doesn't necessary mean that PT is any less useful than previously discussed. I one of the mechanism can't be used with another, we can still get all the benefits of the mechanism that we can use. If we require merging to be done on certain templates, then those templates can't have template parameters. So we therefore have to choose what is most important for us in each case.

The process of writing PT code is quite simple. Programmers who are familiar with Java should have little problem getting into PT, since using the different PT mechanisms require little amount of additional code. Moving a program written in pure Java over to templates in most cases will be easy, since majority of the code will be understood by the PT compiler. But in certain cases, like if the Java code includes multiple inner classes, we wouldn't be able to recreate the same code. Introducing inner classes and other missing Java mechanism to PT will make PT even easier to use.

After working with the cases presented in this thesis, I found the rules proposed in previous work regarding visibility regulation really good. These rules will cover a lot of different ways a programmer might want to regulate the visibility of the code.

Overall, I found that programming with PT gives some useful advantages when it comes to reusability of our code. Though PT is not without some flaws, a programmer will still get very few disadvantages when using PT. The disadvantages that it has (like lack of inner classes), can be fixed by further developing PT and introduce the missing Java mechanics, making PT even easier to use and port your code to. One important PT mechanism that we should focus on is template parameters or *tabstract template instantiation*, since they can provide the programmer with a lot of flexibility when it comes to instantiation of templates.

Bibliography

1. Axelsen, E.W. and S. Krogdahl. *Adaptable generic programming with required type specifications and package templates*. in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. 2012. ACM.
2. Axelsen, E.W., et al., *Challenges in the design of the package template mechanism.* Transactions on Aspect-Oriented Programming, 2012.
3. Helsgaun, K., *Discrete event simulation in java*. 2000: Roskilde Universitetscenter, Datalogisk afdeling.
4. *Beskrivelse av programmeringsspråket Oblila* 2013; Available from: http://www.uio.no/studier/emner/matnat/ifi/INF5110/v13/oblig1/inf5110-9110_oblila.pdf.
5. Oracle. *Using Top-Level Containers*. Available from: http://docs.oracle.com/javase/tutorial/uiswing/components/toplevel.html.
6. Oracle. *The JComponent Class*. Available from: http://docs.oracle.com/javase/tutorial/uiswing/components/jcomponent.html.
7. Oracle. *How to Use Panels*. Available from: http://docs.oracle.com/javase/tutorial/uiswing/components/panel.html.
8. Oracle. *How to Use Buttons, Check Boxes, and Radio Buttons*. Available from: http://docs.oracle.com/javase/tutorial/uiswing/components/button.html.
9. Brunnland, A., et al., *Rett på Java: Inføring i objektorientert programmering*. 2005: Universitetsforlaget.

Appendix A List of figures