Challenges in the Design of the Package Template Mechanism

Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl, and Birger Møller-Pedersen

University of Oslo, Gaustadalléen 23 B, N-0373 Oslo, Norway
{eyvinda, fredrso, steinkr, birger}@ifi.uio.no

Abstract. Package Templates is a mechanism for writing modules meant for reuse, where each module (template) consists of a collection of classes. A package template must be instantiated in a program at compile-time to form a set of ordinary classes, and during instantiation, the classes may be adapted by means of renaming, adding attributes (fields and methods), and supplying type parameters. Also, classes from two or more instantiations may be merged to form a new class which will have all the attributes from the merged classes and also the attributes added in the instantiating template. An approach like this naturally gives rise to two distinct dimensions along which classes can be extended. One is the ordinary subclass dimension, while the other is comprised of the ability to merge classes and to add attributes during instantiations. The latter dimension thus allows for a form of static multiple inheritance and adaptation, that is handled entirely at compile-time. This paper discusses how these dimensions play together in the different mechanisms that make up the Package Templates approach, and the design considerations involved. The paper also argues that the compromise in Package Templates between simplicity of the type system on the one hand and expressiveness on the other is, for most purposes, better than similar approaches based on virtual classes.

Keywords: Programming Languages, Language Constructs, Object-Oriented Programming, Modules, Packages, Modularization, Inheritance, Templates

1 Introduction

The basic concepts of object orientation, that is classes, subclasses including subtype polymorphism and virtual methods, were introduced with the Simula language in 1967 [1]. These concepts became important for later research concerning how languages should be designed to support specialization/generalization, separation of concerns, reuse of code, etc. Since 1967, a number of mechanisms have been added to this original repertoire of language mechanisms, e.g. multiple inheritance [2], generic classes [3], virtual classes [4, 5], aspects [6], traits [7], and mix-ins [8]. Some have focused on specialization/generalization, others on separation of concerns and re-use of code, and some have tried to combine these.

Package Templates, or PT for short, is another such mechanism, with emphasis on separation of concerns and re-use; its main ideas have been presented in [9], while some of the issues discussed in this paper have previously been presented in [10]. PT is intended to be viable for inclusion into various programming languages, so the mechanism is therefore more a set of ideas and concepts that need to be adapted to each host language, rather than a fully defined language. It is here (and in [9]) presented as an extension to Java, with syntax and solutions adapted to that. An adaption of PT for the dynamic language Groovy is explored in [11, 12].

When designing a mechanism such as PT, there are obviously many issues for which a choice between diverging design considerations needs to be taken, and where which alternative is the best is not obvious. The aim of this paper is thus both to discuss some of the design issues we had to deal with during the development of PT, and furthermore to provide a more thorough exposition of some of the issues that were described in less detail in [9, 10]. This entails that we will discuss issues that we believe we have solutions for, but also issues that we are still actively researching. In addition, we discuss some extensions to PT that are being considered.

The main motivation for PT is that concepts of some complexity are usually defined in terms of other concepts and relations between these concepts. A well-known example is the concept of a graph, which typically will be defined in terms of concepts like nodes and edges, and mechanisms for handling structures of these. Utilizing an object-oriented approach, such a complex concept can naturally be reified as a collection of classes that represent the constituent concepts (here *Node* and *Edge*), and a name for the collection as a whole (here *Graph*). The collection may e.g. be represented in a program by an enclosing class (if nested classes are supported), or by a Java-like package.

In order for a mechanism for handling such class-collections to be useful, it would be preferable if we could make suitable adaptions to them when they are used in a program. We may e.g. want to rename the classes of the collection and to add attributes (methods and fields) to the classes. We could e.g. use the Graph for making graphs of Cities and Roads. It would then be desirable to rename `Node` to `City`, and `Edge` to `Road`; in addition `Road` should e.g. have a `length` attribute and `City` a `name` attribute. One might think that this easily can be obtained by defining, in the program, new subclasses of the classes from the original class-collection. However, this would not give the effect we want in PT. Assume that the collection `Graph` has the following two classes:

```
class Node { ... }
class Edge {Node from, to; ...}
```

In addition there will be code in each of these classes that may include creation of objects of the classes `Node` and `Edge`, and the effect we want here is that when this class collection is used to form classes `City` and `Road`, a statement like `new Node()` would automatically be changed to `new City()`. However, if `City` and `Road` are simply subclasses, we will not get this effect. Furthermore, given a variable `Road someRoad` in the program, we would like the following assignment to compile without a cast: `String someName = someRoad.from.name`. However, if we use ordinary subclasses, a statement of the following form would be required: `String someName = ((City)someRoad.from).name`. Finally, if the classes `Node` and `Edge` had the common superclass `GraphElem`, and we also wanted to extend this to e.g. `GeographyElem`, then `City` and `Road` would not be subclasses of this class. To summarize the problem, we would like to preserve the hierarchy and other relationships between the classes, and at the same time be able to modify them.

One approach to class-collection mechanisms that remedies much of the above problems is based upon *virtual classes*, a mechanism pioneered by BETA [4]. This approach uses an enclosing class for representing the class-collection itself, and *inner* (nested) classes to represent the constituent concepts. The inner classes that should be subject to adaptions are defined as virtual classes. To get different adaptions of the basic set of inner classes one then defines subclasses of the enclosing class, and in these subclasses overrides (or rather *extends*) the inner virtual classes. Thus, within objects of the subclasses of the enclosing class, one will have different adaptions of the virtual inner classes. Typical representatives for such approaches are gbeta [13], Caesar [14], J& [15, 16], and Newspeak [17].

The virtual class mechanism entails that generation of objects of the virtual classes in the original outer class will imply generation of objects of the redefined classes, and explicit casts are thus not needed. Usually, it is possible to extend the virtual classes with new attributes and methods, but there are no means for renaming. A drawback with the virtual class mechanisms is that, at least in statically typed languages, they tend to require rather complicated type systems, with concepts such as prefixed types, exact types, dependent types, and intersection types [16, 18, 19]. However, such mechanisms provide a large degree of dynamic expressiveness, which can be valuable in certain situations. They also show a good "economy of concepts", in that they almost solely use the traditional concepts of object-orientation (including nested classes), only adding the concept of virtual classes and in some cases multiple inheritance. To some extent PT represents a move away from this economy principle in that it introduces a brand new mechanism for handling class-collections.

PT is based upon (Java-like) packages. However, the options for unintrusive adaption of Java packages to their use are limited. Therefore, we have added a "generic level" to the package mechanism, and the result is *Package Templates*. Thus, a package template (or just *template* for short) syntactically looks much like a Java package. While Java packages may only be imported as they are, a package template must be *instantiated* (at compile-time) in a program before the classes of the template become ordinary classes in that program.

A template may be instantiated multiple times in the same program, and each instantiation will produce a new, independent, set of ordinary classes. Also, each instantiation may employ adaptions that are required in order for the new classes to fit well with their use in the program. Adaptions may include adding attributes (fields or methods) to the classes, renaming of declarations in the template, overriding of abstract or virtual methods defined in the template classes, and providing actual types to the type parameters of the template.

Thus, a main idea with PT is that the manipulations of class-collections that can be described in PT should be taken completely care of during compilation. To form the resulting ordinary classes of an instantiation, a copy is first taken of the template classes, and then all occurences of the class names defined in the template (e.g. `Node` above) will be replaced by the name of the extended class (here `City`), thus obtaining the "virtual class effect" we want. This also has the effect that the classes described in the templates will not be present at run-time, and we will not have objects of these. This is the main reason why PT gets a simpler type system than what is typical for virtual class systems.

This way of doing things also has the effect that we can allow more drastic adaptions when templates are instantiated, since the compiler can check that the changes we prescribe end up in a consistent program. We can e.g. easily allow extension, renaming of classes, and template type parameters. However, also more elaborate transformations may be considered, for instance template-wide transformations in an AOP-like manner.

The templates of PT may seem similar to those of C++, as they both can be seen as advanced versions of traditional textual macros. However, templates in PT are fundamentally different from their C++ counterparts in the sense that PT templates can be fully type checked as independent units, without considering their subsequent usage in a program.

With PT, programmers have the option to use compile-time composition of template classes instead of subclassing for pure code re-use, and may thus to a large extent restrict their use of subclassing to classification purposes. As we shall see, PT also has a *merging* mechanism, by which we, during instantiations, can merge two or more classes from different templates to a new class that gets the union of the attributes, and a new name given by the programmer. Then, all types referring to these classes in the involved template instantiations are unified, under this name. And as before, the original names of the merged template classes will not exist at run-time. This mechanism may be seen as a form of "static multiple inheritance", and is meant for compile-time composition of code from different sources. Our hope is that this mechanism can take care of many of the situations where general multiple inheritance is used today. Thus, for pure code re-use there should be no reason to introduce multiple inheritance

The rest of this paper is organized as follows: Section 2 gives an overview of PT. Section 3 comprises the main content of this paper, and discusses a number issues involved in the design of PT, what problems arise and how they may be solved. Related work is discussed in Section 4, and Section 5 gives some concluding remarks.

2 An overview of PT

In this section we give a general overview of the PT mechanism. More details will be given where relevant in Section 3.

2.1 Basics

For definitions of package templates, we shall use the following syntax, where class definitions are enclosed by a `template` construct:

```
template T < ...compile-time type parameters... > {
   class A { B b; ... }  // b is referred to in the text below
   class B { ... }
}
```

We will return to the template type parameters in Section 3.7, and will hence not cover them in any detail in this section.

The classes inside the template `T` are referred to as *template classes*, and can essentially be written as ordinary Java classes. For the template classes `A` and `B` to become ordinary classes we have to instantiate the template `T` in a package `P`. This is done by the `inst` statement, as shown in the second line in the example below. This statement makes the template classes `A` and `B` available in `P`, but here under the names `AA` and `BB`, respectively, as specified by the arrows (`=>`) in the *with clause*.

Additions to the template classes are given in class-like constructs called *addition classes*. Such addition classes are provided for `AA` and `BB` as explained below, while `C` and `D` are new classes in the package `P`. Note that we can refer to `C` and `D` in the additions to `AA` and `BB`. Throughout this paper, we shall write packages in the same style as we write templates, i.e. with their contents enclosed by curly braces, as shown below:

```
package P {
   inst T with A => AA, B => BB;
   class AA adds { ... additional attributes in AA ... }
   class BB adds { ... additional attributes in BB ... }
   class C { ... }
   class D { ... }
}
```

When a template is instantiated, its contents can be adapted. The most important forms of such adaptions are the following:

- Elements of the package template may be renamed. This is done in the with-clause of the `inst`-statement, and is here only shown for class names. For renaming of class attributes another arrow is used (`->`), so that if we also want to change the name `b` to `bb`, the line must read: `inst T with A => AA (b -> bb), B => BB;`. Note that all renaming in PT is done in a "semantic way". That is, renaming is done based on the name bindings from the static analysis.
- In each instantiation the classes in the template may be given additions: fields and methods may be added and virtual methods may be redefined. This is done in `adds`-clauses as shown above. The order of the `adds`-clauses and other declarations in the package is not significant.

– We may “merge” classes from different template instantiations.
– If the template has formal type parameters, actual type parameters must be supplied.

Note that the additions are made in a “static virtual” manner. That is, if we in the `adds`-clause of `BB` declare `int i`, we may in the `adds`-clause of `AA`, without casting, write `b.i = 1;` even if the variable `i` was not known when `b` was declared in `A`.

As an example we again consider a template for defining graphs:

```
template Graph {
   class Node {
      Edge[] outEdges;
      Edge insertEdgeTo(Node to){ ... }
      ...
   }

   class Edge {
      Node from, to;
      void deleteMe(){ ... }
      ...
} }
```

Below, the template `Graph` is instantiated in the package `RoadAndCityGraph`, in order to use objects of class `Node` for representing cities and objects of class `Edge` for representing roads (and we disregard, for simplicity, the fact that template `Graph` defines directed graphs). For this purpose, `Node` is renamed to `City`, `Edge` to `Road`, and both get additional attributes. We also rename `insertEdgeTo` to `insertRoadTo`. When renaming a method, its parameter types must be given, since there (in general) may be definitions of method overloads or a variable with the same name.

```
package RoadAndCityGraph {
   inst Graph with Node => City (insertEdgeTo(Node) -> insertRoadTo),
                   Edge => Road;
   class City adds {
      String name;
      void someMethod(){
         int n = outEdges[3].length; // 1, see text below
         City c = ... ;
         Road r = insertRoadTo(c);   // 2, see text below
   } }

   class Road adds{
      int length;
      void someOtherMethod(){
         String s = to.name;         // 3, see text below
 ...
} } }
```

Here, a copy is first made of the Graph classes, then the specified renamings are done in these, and finally each class gets the additions given in the corresponding `adds`-parts. The `inst` statement and these `adds`-parts are then removed, and replaced by the result of the above transformations. Thus the names `Node`, `Edge`, and `insertEdgeTo` will not exist at all in the package RoadAndCityGraph, and the statements marked 1, 2, and 3 above will all be legal, and this can be determined statically. No objects of classes `Node` and `Edge` will ever be generated in the package `RoadAndCityGraph`, as occurrences of e.g. `new Node()` in the template are changed to `new City()`. The method `insertEdgeTo` in `Node` will have the following signature within `RoadAndCityGraph`:

```
  Road insertRoadTo(City to){ ... }
```

Virtual or abstract methods defined in a class within a template can be overridden by methods defined in an `adds`-clause as part of an instantiation. That is, if the addition to class `City` has a definition of the method `insertRoadTo` with the same signature as above, then this will override the method `insertEdgeTo` in `Node`.

2.2 Subclass Hierarchies Within Templates

PT allows ordinary subclass hierarchies to be defined inside templates. As part of an instantiation of a template, all classes in such hierarchies (and not only the leaf classes) may be given additions. As an example, consider the following sketch of a template:

```
template Vehicles {
   class Vehicle { float maxSpeed; ... ; }
   class Car extends Vehicle { int numOfSeats; ... ; }
   class Truck extends Vehicle { int length; ... ; }
}
```

and a use of it:

```
package TrafficSimulation {
   inst Vehicles with Vehicle=>SimVehicle, Car=>SimCar, Truck=>SimTruck;

   class SimVehicle adds{ PosType position; float curSpeed; ... ; }
   class SimCar adds{ int luggageVolume; ... ; }
   class SimTruck adds{ int loadCapacity; ... ; }
   ...
}
```

Note here that `SimCar` and `SimTruck` automatically will be subclasses of `SimVehicle`, as this is true for the corresponding classes in `Vehicles`. It is perfectly legal here to e.g. add an abstract method in `SimVehicle`, and give concrete versions of it in `SimCar` and `SimTruck`. But what if an abstract method is defined in `Vehicle` and is given concrete versions in both `Car` and `SimVehicle`? Which version will then be used if it is called from `SimCar`? These and similar questions will be discussed in Sections 3.2 and 3.4.

2.3 Multiple Instantiations

A basic idea in PT is that one can do two or more instantiations of templates in the same scope. One can even instantiate the same template more than once in the same scope so that one e.g. can use the `Graph` template for producing the classes `City` and `Road` as above, and in the same scope use it to represent e.g. the structure of pipes and connections in a water distribution system.

2.4 Instantiation of Templates Within Templates

Templates are always written as independent entities, and are instantiated at the outermost level of packages or other templates. This makes it possible to build hierarchies of instantiations, which, for some tasks, turns out to be very useful. As an example, assume that we also want the package `RoadAndCityGraph` from the beginning of Section 2 to be a template, so that further additions can be made to the classes `City` and `Road` before we use them at a later stage. The only change needed is to use the keyword `template` instead of `package` when defining `RoadAndCityGraph`. Then this template can be instantiated in a package e.g. for drawing maps where also positions are needed for cities.

When templates are instantiated within templates, as for the `RoadAndCity-Graph` template outlined above, this should work as follows: Whenever `RoadAnd-CityGraph` is instantiated, instantiations specified inside of this template should also be carried out. Cyclic structures of templates instantiating templates are not allowed.

2.5 Merging Template Classes as Part of Instantiations

In order to get optimal reuse of code, it is important to be able to merge independently written code. A challenge here is to merge the independent types of the code components to one such that they can cooperate in a type safe manner. PT's answer to this is to offer a mechanism where a class from one instantiation is merged with a class from another to form one new class. Code from the different templates can only access the attributes of objects of the new class that stem from the respective templates, while in the scope with the instantiations and the merge, all attributes of the new class can be accessed.

Syntactically, merging is obtained by allowing classes from two or more template instantiations to share a common addition class, and they thereby end up as one class, with the name of the addition class. The new class gets all the attributes of the instantiated classes, together with the attributes of the addition class.

The following example illustrates how this mechanism works: As in the example in Section 2.3, we assume that we have the template `Graph`, and that we want to use this as a basis for forming classes `City` and `Road`. However, instead of adding the extra attributes of `City` and `Road` in the `adds`-clauses, we this time assume that we also have another template `GeographyData` with classes `CityData` and `RoadData` where the extra attributes to form the classes `City` and `Road` are defined:

```
template GeographyData {
   class CityData{ String name; ... ; }
   class RoadData{ int length; ... ; }
}
```

We can now define the classes `City` and `Road` as merges of `Node` and `CityData`, and `Edge` and `RoadData`, respectively, with addition classes `City` and `Road`. We do this by instantiating the templates as follows:

```
package RoadAndCityGraph2 {
   inst Graph with Node => City, Edge => Road;
   inst GeographyData with CityData => City, RoadData => Road;
   class City adds{ ... }
   class Road adds{ ... }
   ...
}
```

As described above, classes that are renamed to the same name, such as `Node` and `CityData`, will be merged to one class, under the new name. The resulting class `City` will then have all of the attributes of `Node` and `CityData`, plus those given in the shared addition class, and likewise for the new class `Road`. Objects of these classes can be accessed in exactly the same way as with our earlier definitions of `City` and `Road`.

Related to this construct, there are obviously many issues that we need to resolve. We can for instance easily get name collisions, and what about abstract or virtual methods defined in two or more of the merged classes? Similarly, what about constructors? These and other issues are discussed in more detail in Sections 3.2, 3.3 and 3.5.

2.6 Multiple Inheritance

Through merging, PT gets a form of static multiple inheritance that is handled entirely at compile-time (that is, the classes that are merged to form the new class are not accessible nor present at runtime in the final program). However, we do not want this static form of multiple inheritance to also imply traditional multiple inheritance. We can easily forbid explicit multiple inheritance in templates and programs. However, even if we do this, we can indirectly get multiple superclasses to a class through merging, and we therefore have to introduce further restrictions. These issues are addressed in Section 3.6.

2.7 Interfaces and Inner Classes

Interfaces are mentioned a few places in this paper where they are relevant, but in general the full role of interfaces in PT is an important topic for future work, and not discussed in detail in this paper. The same is true for inner (nested) classes.

3 Specific Issues

3.1 Template Classes can be Extended Along Two Dimensions

Standard object-orientation provides a single "dimension of extension" for classes, which is traditional subclassing. By introducing templates and `adds`-clauses as parts of instantiations, template classes get a second dimension of extension. This dimension has other properties, that represent both new possibilities and new challenges.

An important design consideration is whether both of these dimensions can be referred to in the code, i.e. if the addition dimension is reified through the available language primitives, or if it is only present on a more conceptual level. For instance, providing the ability to refer to overridden method definitions in a template class from a package class would require specific syntax and semantics.

Thus, an immediate observation is that the *flattening property* as employed by traits [7] cannot coexist with a reified two-dimensional structure; the flattening property entails that the semantics of the composed unit is exactly the same as if the contents of every constituent was written directly in the composed unit itself.

Designing a version of PT supporting the flattening property could in itself be an interesting endeavor, and a simpler, yet still useful, subset of what we will discuss in the rest of this paper would be worth exploring. However, supporting this property would also seemingly preclude constructors in templates (see Section 3.5) as well as calls to overridden template class methods etc.

For PT we have found, through the examples we have studied, that e.g. being able to explicitly refer to overridden methods defined in a template class from a package class does indeed seem very useful, and we have thus opted for introducing explicit ways of referring to each dimension of extension.

General Overview To see the general scheme, we look at the following sketch of a program with two templates and a package; for simplicity we keep merging out of the picture for now.

```
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{   }          // Subclass of A2, as B1 is subclass of A1
   class C2 adds{   }          // Subclass of B2, as C1 is subclass of B1
}

package P {
   inst T2 with A2 => A3, B2 => B3, C2 => C3;
   class A3 adds{   }
   class B3 adds{   }          // Subclass of A3
   class C3 adds{   }          // Subclass of B3
}
```

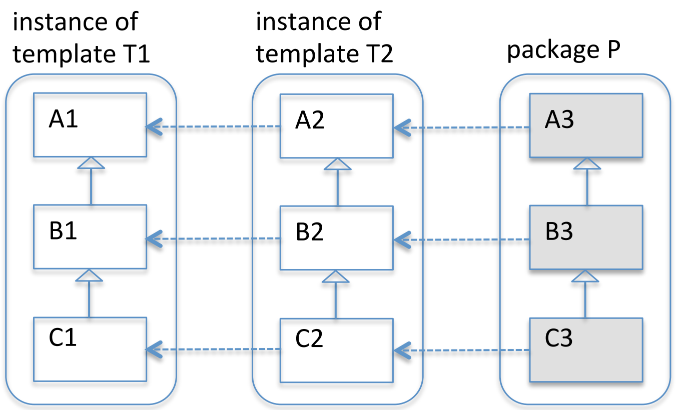

Fig. 1. Template classes can be extended along two dimensions, the *subclass dimension* and the *addition dimension*. Note that the figure depicts instances of the templates `T1` and `T2`, respectively. `P` is an ordinary Java package.

Figure 1 shows a graphical representation of the relationships between the different parts of the program sketch above. Extensions in the traditional dimension of sub/superclasses are drawn along a vertical axis (with arrows from subclass to superclass), while extensions in the new dimension of adding attributes during instantiations are drawn along a horizontal axis (with a dotted arrow from the `adds`-clause to the template class). We shall call these dimensions the *subclass dimension* and the *addition dimension* respectively, and a class at the head of the arrow is called the superclass and the *tsuperclass*, respectively, of the class in the other end. Thus, in Figure 1, we can for instance see that `C1` is a tsuperclass of `C2`, and `B2` is a superclass of `C2`. The contents in `A2`, `B2`, `C2`, `A3`, `B3` and `C3` will (as seen in the code above) occur syntactically as `adds`-clauses, while `A1`, `B1` and `C1` will look like normal classes or subclasses.

(A note on syntax: we have chosen not to write e.g. `class B2` ***extends A2*** `adds { ... }`, and thus the subclass relationship between `A2` and `B2` in `T2` is implicitly given only by the relationship between `A1` and `B1` in `T1`. We do acknowledge, however, that there are valid arguments for both options here.)

There are certain essential differences between the two dimensions:

– The most obvious difference is the following: The subclass dimension is "dynamic" in the sense that the `A`, `B`, and `C` levels will exist at runtime so that we can generate objects of all three of the classes `A3`, `B3` and `C3` (in gray above), and that these classes are traditional subclasses of each other. The addition dimension, on the other hand, is more static in the sense that it is taken care of entirely at compile-time. Classes `A1`, `A2`, `B1`, `B2`, `C1`, and `C2` will not exist as separate entities at runtime, but rather as parts of `A3`, `B3` and `C3`, respectively; thus, even if we may specify `new B2(...)` in a method of a class in template `T2`, it will be transformed to `new B3(...)` in the package `P` by the compiler.
– Another important difference is that the subclass dimension only has single inheritance, while the addition dimension enjoys a sort of multiple inheritance through merging. Thus, everything that has to do with this special form of multiple inheritance is treated at compile-time, and at runtime we will only have straight-forward single inheritance. We think this is a promising compromise between the need to combine code from different sources and adjust it for reuse purposes, and the complexities of traditional multiple inheritance.

Thus, these dimensions give us a number of possibilities, but they indeed also give rise to problems for the design of a consistent system. Some of these problems have rather straightforward solutions, while others (e.g. those concerning constructors) are more challenging. These problems will be discussed in the subsections below.

3.2 Virtual Methods and `super` Calls

With two dimensions, method call resolution is not as straightforward as for a single dimension. Assume that `A1` from Figure 1 defines an abstract method `m()`, and that an implementation is provided both in the subclass `B1` and in the addition class `A2` (and nowhere else). Given a call to `m()` e.g. in `B2`, which version will be chosen in an object of the class `B3`? (Note that there cannot be any objects of any classes except `A3`, `B3` and `C3` in this program.) As a guide in this and similar cases, we emphasize the fact that the addition dimension represents compile-time composition to a single class, while the different levels of the subclass dimension still exist at runtime. This entails that the addition dimension should bind strongest, so that the version in B1 should be chosen.

Now, what if the method is also overridden in C2 and B3, and the call is done in a C3-object (but still from the program text of B2)? As for calls to (virtual) methods in most languages, the version chosen should only depend on the class of the object, and according to the binding priority indicated above, we should look for the correct version by first searching along the addition dimension from the class of the object, and if it is not found there, move one step up in the subclass dimension, and repeat the process. Thus, even if the call is made from the text of B2, the version in C2 will be chosen.

Merging and Complex Instantiations The preceding example possesses a somewhat artificial regularity in that it contains an equal number of classes (3) in each template/package. In the general case this is obviously not necessarily so. Figure 2 shows an example of a more complex instantiation that includes merging and templates that have different numbers of classes (a sketch of the program is omitted, but code similar to the previous example is hopefully easy to envision).

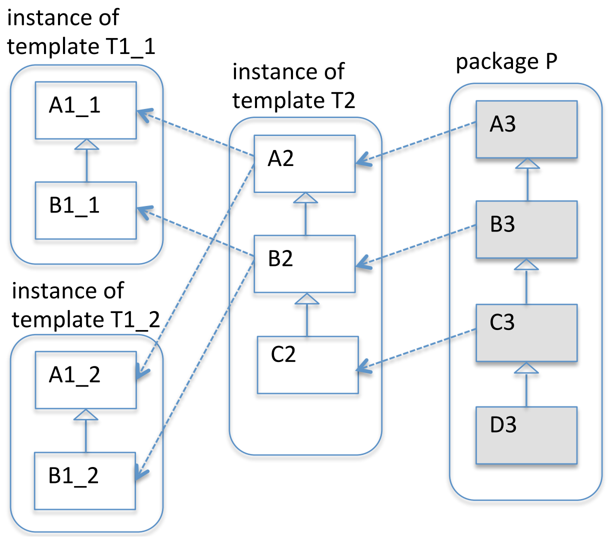

Fig. 2. An example of a more complex instantiation.

When merging is involved, a given class will have two or more tsuperclasses, and this is something that we need to take into consideration when resolving method calls. Note that if the classes that are merged have superclasses, such as e.g. `B1_1` and `B1_2` in the figure, these superclasses (i.e. `A1_1` and `A1_2`) will also have to be merged; more on this in Section 3.6.

A pseudo-code algorithm for method body lookup for a virtual method call is shown below. At the start of the algorithm, `C` is the class of the object in which the call occurs, and `m` is the signature at the call site. As we will get back to in Section 3.3, all name conflicts must be resolved for the classes that are to be merged, and thus the algorithm will never have to choose between several compatible method implementations at the same level, even when there are multiple tsuperclasses. Furthermore, all renaming must be performed in the templates before running the algorithm; when the algorithm searches for a method, it will always use the names as they appear in the package.

```
MethodDefinition virtualLookup(MethodCallSignature m, Class C) {
  if(there is a local definition mdef matching m in C)
    return mdef;

  // A class will have more than one tsuperclass if a merge is involved, thus
  // foreach is used below. The order in which the tsuperclasses are processed is
  // not significant, and thus left as an implementation detail.
  foreach(direct tsuperclass AC of C along the addition dimension) {
    mdef = virtualLookup(m, AC)
    if(mdef != null)
      return mdef;
  }

  if(C is defined directly in a package &&
       C has a direct superclass EC along the subclass dimension) {
     return virtualLookup(m, EC);
  }
  return null;
}
```

Calls to super and tsuper Since PT has two dimensions of extension, the existing `super` keyword in Java is not sufficient to be able to reach all method definitions in a controlled manner. Thus, a mechanism for super calls along the addition dimension is needed. We will employ Java's traditional `super` keyword for reaching superclass methods in the normal subclass dimension, and introduce the new keyword `tsuper` for the new dimension.

When a template has regular inheritance inside it, methods can be overridden both in the `adds`-clauses and in subclasses. Below, we see an example of this (which is a slightly modified version of the example from Section 2.2).

```
template Vehicles {
  class Vehicle { String model; void display(){ out(model); } }
  class Car extends Vehicle {
    int passengers;
    void display(){ super.display(); out(passengers); }
  }
  ...
}

package RentalVehicles {
  inst Vehicles with Vehicle => RentalVehicle,
                     Car     => RentalCar;

  class RentalVehicle adds {
    int price;
    void display(){ tsuper.display(); out(price); }
  }

  class RentalCar adds {
    boolean nosmoking;
    void display(){ tsuper.display(); out(nosmoking); }
  }
  ...

    // In some method in some class:
    RentalCar rc = new RentalCar();
    rc.model = "Ford"; rc.price=120;
    rc.passengers=4; rc.nosmoking=true;
    rc.display(); // Prints "Ford 120 4 TRUE"
  ...
}
```

Inside the template `Vehicles`, the class `Vehicle` contains a virtual method `display`. This method is overridden in a normal Java manner in `Car`, in which the original implementation is called through the use of `super`. If the `Vehicles` template had been instantiated with no additions, it would work just as any ordinary Java package with regards to virtual methods and `super` calls.

However, here the method `display()` is also overridden in the addition part `RentalVehicle`, and in this method we want to call the original version in `Vehicle`. This can obviously not be done by means of `super` as `Vehicle` is not a superclass of `RentalVehicle`. Thus, the only way to reach a virtual method defined in a template class, but that has been overridden in an `adds`-clause to that class, is to use a `tsuper` call.

A pseudo-code algorithm for the lookup of a `tsuper.m(...)` call is shown below. When the algorithm is called, `C` is the class in which the call *syntactically* occurs (and thus not necessarily the runtime class of the object, as opposed to for the previous algorithm), and `m` is the signature at the call site.

```
MethodDefinition tsuperLookup(MethodCallSignature m, Class C) {
 // A class will have more than one tsuperclass if a merge is involved,
 // thus foreach is used. The order in which the tsuperclasses are processed is
 // not significant, and thus left as an implementation detail.
 foreach(direct tsuperclass AC of C along the addition dimension) {
   if(AC has an accessible method definition mdef matching m)
     return mdef;
   else {
     mdef = tsuperLookup(m, AC);
     if(mdef != null)
       return mdef;
   }

 return null;
}
```

In the algorithm above we see how a call to `tsuper` will not result in any lookups along the subclass dimension. A missing method definition in the addition dimension will result in a compile-time error.

The lookup for a `super.m(...)` call is similar to the virtual method lookup presented earlier in this section. However, for `super` calls we do not start the lookup at the actual class of the object, but instead one step up from `C` in the `super`-direction. Thus, if a call to `super.m(...)` for instance occurs in `C2` in Figure 2, the search for `m` will start in `B3`.

An important thing to note is that these algorithms follow the general principle that an instantiation represents an actual substitution with semantic bindings. Each `tsuper` call will be transformed to an ordinary method call at compile-time. Of `tsuper` and `super`, only ordinary `super` calls are left in the final program, with the exact same semantics as a normal Java `super` call.

3.3 Name Conflicts

When classes are merged, there is a possibility that (accessible) attributes with the same signature are defined in more than one of the source classes, which gives rise to a conflict in the target class. One way to resolve such issues is to rely on ordering to resolve conflicts, such as e.g. giving the class from the template listed first in the instantiation precedence. Relying on order for conflict resolution is the approach taken with Mixin-based inheritance [8]. While this might have benefits in terms of ease of use etc, it might also lead to unwanted/unexpected behavior and problems with breaking changes in new versions of a given piece of code. Thus, for PT, we have opted for an approach that requires explicit conflict resolution. This is in many ways similar to the way conflicts are resolved for traits [7]. Note, however, that as opposed to what is the case for traits, method exclusion is not supported by PT. Supporting this (in a safe manner) would require analyzing call graphs and potentially excluding/deleting all methods that refer to the method originally intended for exclusion. We believe that the extra complications involved with this outweigh the benefits.

This leaves two primary options for method conflict resolution: (1) to rename the conflicting method, or (2) to provide a new method that overrides all of the conflicting methods for a given signature.

Renaming (1) is the only resolution strategy that is currently supported in the implemented version of PT, and is done as a part of the instantiation as an optional clause to the `inst` statement, as explained in Section 2.1. Note that as opposed to method name aliasing in traits and similar systems, the method renaming in PT is "deep" in the sense that all references to the method in question will be renamed for the current instantiation.

For the latter option (2), an override must be provided in the addition class of the merged classes. Typically, in such an override, one would want to call one or both of the existing methods. This can be achieved through `tsuper` calls, however, a mechanism for qualifying the original class names would be needed, e.g. as in `tsuper[A].m()`, where `A` is the name of the original class containing the desired implementation of `m`. However, for more complex scenarios, e.g. when the same template is instantiated multiple times, or different templates have classes with the same name, this simple qualification scheme is not sufficient to differentiate between the different classes. A more elaborate scheme could then be allowed, utilizing either template name or instantiation name (the latter is not treated explicitly in this paper, but would be relevant if the same template was instantiated twice) in addition to the class name.

The main reason for not supporting option 2 is the added complexity it brings, as discussed above, paired with the fact that there is not much to be gained by having this option available. For instance, if we really want one new method to be called whenever either of two existing methods was called prior to a merge, we could opt for the following solution:

```
inst T with A => AB (m() -> m_a); // T has a class A with a method named m
inst U with B => AB (m() -> m_b); // U has a class B with a method named m

class AB adds {
   void m_a() { // Overrides original definition from A
      m();
   }

   void m_b() { // Overrides original definition from B
      m();
   }

   void m() {
     // perform desired computations for calls to both A.m() and B.m()
     // here, for instance call tsuper.m_a() and/or tsuper.m_b().
   }
}
```

For fields, providing an override is not an option, so the only solution for field conflicts is to rename one or more fields until there are no more conflicts.

Template classes may also extend ordinary Java classes and implement ordinary Java interfaces. In such cases, the rules for renaming are more restrictive, see Section 3.6 for details.

3.4 Abstract Methods in Template Classes

For a normal Java class, methods can be marked as `abstract` to specify that they need to be given a concrete implementation by subclasses. In PT, abstract classes can be defined both in packages and in package templates.

Along the addition dimension in PT, a similar concept can be defined for methods that are meant to be implemented in an addition class at a later stage. This is useful in order to allow template developers to "promise" that there will be an implementation of a given method, even if no sensible implementation can be provided when writing the template (e.g. because it will depend on its usage scenario). We shall mark such methods with the modifier `tabstract` (for *template abstract*); `tabstract` methods must be given an implementation in an addition class at the latest when the enclosing template is instantiated in a package. Failure to provide such an implementation will result in a compile-time error.

An important difference between `abstract` and `tabstract` is thus that while one is not allowed to make objects (with `new`) of a class with `abstract` methods, one may do so with classes with `tabstract` methods. This is allowed since every such method will get an implementation at the latest when the template is instantiated in a package.

Below is an example using a combination of `abstract` and `tabstract` methods. `Figure` is an `abstract` class with an `abstract` method. No objects can be made from that class. The two subclasses are not abstract, and objects can be made from them since it is guaranteed that in a package using this template they will have implementations of the `draw` methods and thus be concrete classes. Thus, the requirement imposed by the `abstract` modifier can in fact be fulfilled by a `tabstract` declaration.

```
template Figures {
  abstract class Figure { abstract void draw(); }
  class Circle extends Figure { tabstract void draw(); }
  class Square extends Figure { tabstract void draw(); }
  ...
    Circle c = new Circle();
    Figure f = c;
    f.draw();
  ...
}
```

If a template containing classes with `tabstract` methods is used in another template, one may choose to implement the `tabstract` methods there or the implementation may be postponed until that template is instantiated in a package. Overriding a method that implements a `tabstract` method is allowed in the same way normal method overrides are allowed.

Since the mechanisms related to the `adds`-clauses of the templates are orthogonal to the mechanisms related to subclasses, they may also apply to `static` methods. This means that in PT one can declare a `static` method to be `tabstract`, and subsequently override a static method in an `adds`-clause and also call a static method defined in a template class with `tsuper`.

3.5 Constructors

Constructors are a convenient (and, in some languages, necessary) construct for initializing objects at runtime. In Java, all instance and static variables are initialized to neutral default values independently of the constructors. Thus, constructors are mainly a convenience for the programmer rather than a necessary construct to enforce type safety. Consequently, when designing a scheme for constructors in PT for Java, our main focus has been usefulness and comprehensibility for the programmer.

Classes in Java must have at least one (explicit or implicit) constructor, and (except for the effect of `this(...)` calls) exactly one constructor is invoked at each subclass level when an object is created. Constructors may use its parameters to initialize its own subclass level and/or pass them to a constructor of the superclass. The call to the constructor of the superclass must always come at the very start of any constructor so that all superclasses are initialized before the class itself. These rules create a regularity that as far as possible should be preserved in PT. An apparent idea is then to also allow template classes/`adds`-parts to have constructors, and in addition to using `super(...)` for calling constructors in the superclass introduce `tsuper(...)` for calling a constructor in the tsuperclass.

However, an obvious issue is that if constructors are called along both dimensions, a given constructor might generally be called a number of times in each class/`adds`-part. This is clearly not a desirable situation, and would would probably be very difficult to use in a controlled way if implemented.

One drastic alternative going in the opposite direction is to disallow constructors in template classes/`adds`-parts (and thus also the use of `tsuper(...)` calls) altogether, and say that initialization is the responsibility of the proper package classes/`adds`-parts. Only at that point can the developer have a complete picture of the workings of the fully composed classes. Note here that one can define a suitable set of ordinary methods (e.g. in template classes) that are designed to be called by proper constructors for initialization purposes, alleviating at least some of the problems with such an approach.

It would, however, be preferable if we could find a scheme where we could allow a constructor in each class/`adds`-part also in the templates, and where these are called in a systematic way so that exactly one constructor is called for each such part in a new object (again, except for the effect of `this(...)`). If we envision the general structure of programs like the one shown in Figure 1, such a strategy is not difficult to find. We might call it the "backwards E strategy", and it entails making `super` calls only in the rightmost column of the figure, and from there (after the `super(...)` call) make appropriate `tsuper(...)` calls towards the left (thus following the form of a backwards or mirrored E). Classes/`adds`-parts in templates should not have `super(...)` calls. Note that the compiler can syntactically recognize the rightmost column of such a program, as this will always be a package and not a template. Thus, the compiler can check that the constructors are called according to the rules. Parameters for constructors can be used freely, and we can easily make a scheme where default constructors and constructor calls are inserted if missing.

If we have merging, we must, in the `adds`-clause of the merged class, do one `tsuper(...)` call to a constructor in each of the merged classes, and here any order can be allowed. In these situations each `tsuper(...)` call should be qualified by a `[...]` construct as discussed for name conflicts in Section 3.3, e.g. like this: `tsuper[T.C](...)`, where `C` is a class that is being merged and `T` the instantiated template. Note, however, that as opposed to what is the case with name conflicts, the need for additional qualification in `[...]` cannot here be avoided by renaming or any similar scheme (unless we allow renaming of constructors).

To exemplify the "backwards E strategy", we look back to Figure 1, and now assume that a variable `a1` is defined in `A1`, `b1` is defined `B1` etc. To initialize these variables with constructors according to the scheme above, we could, in the different classes/`adds`-parts, have constructors as sketched below:

```
A1(a1){this.a1 = a1;}
A2(a1, a2){tsuper(a1); this.a2 = a2;}
A3(a1, a2, a3){tsuper(a1, a2); this.a3 = a3;}
... B1 and B2 like A1 and A2 above ...
B3(a1, a2, a3, b1, b2, b3)
  {super(a1, a2, a3); tsuper(b1, b2); this.b3 = b3;}
... C1 and C2 like A1 and A2 above ...
C3(a1, a2, a3, b1, b2, b3, c1, c2 ,c3)
  {super(a1, a2, a3, b1, b2, b3); tsuper(c1, c2); this.c3 = c3;}
```

We can see that this will work out nicely for objects of the classes `A3`, `B3`, and `C3`. It is not difficult to set up a similar scheme for the more complicated case in Figure 2, but we leave that to the reader.

For this scheme to be complete, we must also consider object generation inside templates. In a template with a template class `C`, we are allowed to write `new C(...)`. However, allowing this mandates that the final package class `PC` supplies suitable constructors `PC(...)` for every parameterization of `C(...)`. Thus, a requirement to implement the necessary constructors is put upon the user of a template. However, we have not yet found the right balance between freedom of expression in templates and convenience for the user when instantiating templates, and our current implementation therefore only allows parameterless template class constructors to be called from within the template.

Except for the latter problem, the backwards E strategy might seem fine. It does, however, have one important drawback: When writing a class/`adds`-part in a template (e.g. in `B2` of Figure 1) one cannot directly control how the constructor of the superclass (here `A2`) is called. One is not even allowed to use `super(...)`, so the call to a constructor in the superclass will come indirectly "in from the right", through templates or packages that probably are not yet written. Thus, this prevents the templates from taking control over their own initialization.

An alternative approach could be labeled the "lying E strategy" (with "arms" and "legs" up), entailing that the constituents from leftmost template of Figure 1) is initialized before continuing to the right. This has the advantage that each template can allow its classes to use `super(...)` calls to initialize their superclasses, and thus making each template conceptually more self-contained. However, it turns out that also this strategy has its problems. First of all, it will necessarily involve breaking the principle that a superclass should be fully initialized before the class itself is. The second problem is that we now cannot determine syntactically where the "lowest row" is (corresponding to the rightmost column for the backwards E strategy), as this will depend on what sort of object is generated at runtime. If we e.g. look at the example from Figure 2, the lowest row (from right to left) corresponding to the different sorts of objects are as shown below. When merging is involved the merged classes (including their potential tsuperclasses) are listed one after the other, in no specific order.

```
In A3 objects:  A3, A2, A1_1, A1_2
In B3 objects:  B3, B2, B1_1, B1_2
In C3 objects:  C3, C2, B1_1, B1_2
In D3 objects:  D3, C2, B1_1, B1_2
```

It is not difficult to come up with a scheme where the constructors of all the classes in lowest row for the class of the created object are called. However, passing parameters to constructors in classes from this row is not entirely trivial, since there might not be a direct relationship between the runtime class of the object and the template classes that in the lowest row (as is the case for instance between `D3` and `C2` in Figure 2).

Thus, none of the proposals above seem to fully satisfy our needs, and we are continuing our research in pursuit of better solutions.

3.6 Avoiding Multiple Inheritance

PT supports merging of independently written template classes, and the class resulting from such a merge can thus indirectly get multiple superclasses if two or more of the merged classes have superclasses (other than `Object`).

Since we want PT to be a viable mechanism for languages that employ single inheritance, we need to introduce some additional rules, and consider a rather restrictive rule set to begin with. The first rule is that we forbid template classes to have superclasses (other than `Object`) that are defined outside the template itself and outside instantiations made in this template. Thus classes defined in an ordinary package cannot be used as superclasses. Note, however, that there are no such restrictions on implementing interfaces that are defined in this way.

However, we do not want such drastic restrictions for superclasses defined inside the same template (or inside instantiations made in this template), since this would disallow class hierarchies inside templates altogether, and we consider these very valuable. Thus, we also introduce the following rule:

If, in a set of instantiations, two or more template classes are merged, then the superclasses they might have (which, if they exist, are also template classes according to the first rule) must also be merged in the same instantiations. In addition it is required that such a merge must not result in a cyclic superclass-structure.

External Classes There are cases where the first rule (that template classes cannot have external superclasses) is obviously a nuisance, e.g. when we want to introduce our own exception classes in a template, that (in Java) have to be subclasses of those defined by the system. For these cases we introduce a special syntax to make the external relationship explicit and to ensure that multiple inheritance will not occur. When such an external superclass is used, the programmer must specify this explicitly with the keyword `external`, as in the following example:

```
package R{class RC{...} }

template T{
   class A extends external R.RC{...}
}
```

This also has the consequence that the `adds`-clause to `A` in an instantiation of `T` (and transitively in `adds`-clauses of further instantiations) must also be marked as having an external superclass as follows:

```
template U{
   inst T with A => B;
   class B extends external R.RC adds{...}
}

template V{
   inst U with B => C;
   class C extends external R.RC adds{...}
}
```

Syntactically we can here omit `R.RC` (thus only indicating that `B` or `C` has *some* external, anonymous, superclass), but this will imply stronger restrictions in the following step. Now assume that we in a package `P` instantiate `V` and another template `W` containing a class `D` as follows:

```
package P{
  inst V with C => CD;
  inst W with D => CD;
  class CD adds{ ... }
}
```

This merge will now be legal if class `D` in `W` has no external clause at all, or if the external clauses of `C` and `D` refer to the same external class. If one of the merged classes has an anonymous external clause the rule is stricter. Then no other of the merged classes can have any external clause at all. The `adds`-clause of `CD` does not need any external clause as a package class will not participate in any further merges.

Name Changes For instantiations involving classes that are subclasses of external classes, names stemming from an external superclass cannot be changed. This will never be a problem with respect to the use of renaming to obtain unique names in merged classes (since multiple inheritance is prohibited). However, when implementing external interfaces, name conflicts might be an issue. Thus, we propose that for instantiations involving classes that implement external interfaces, the names may be changed in the instantiation, but the resulting class will then no longer fulfill the requirements placed on the class by the interface. Unless the addition class subsequently defines the missing (due to the renaming) interface methods, there will be a compile-time error.

3.7 Templates with Parameters

Templates may have parameters, and in the paper describing basic PT [9] one kind of such parameters is discussed, which are type parameters working in much the same way as generic parameters to classes in e.g. Java or C#. However, recently we have also considered parameters to templates that are themselves templates, and this seems very useful. Below we first discuss type parameters about as they are presented in [9], and then the "formal template" kind (which were introduced in [10]).

Type Parameters As an example of type parameters, consider a template that implements a kind of list library, where each list will have a head object and a number of elements of parameter type `E`. Each element of the list is represented by an object of the internal class `AuxElem` that has a reference to the real list element of type `E`. This means that a given `E`-object can reside in any number of lists (and more than once in the same list). Type parameters can be constrained, and we include this in the example by assuming that each object of class `E` should be able to keep track of how many times it currently occurs in some list. For this purpose we require that the element class has two methods: `void incNo()` and `void decNo()` will increase or decrease the value of an internal counter, respectively, and these will be called at appropriate points in the methods `insertAsLast` and `removeFirst`, as seen in the example below:

```
template ListsOf<E implements {void incNo(); void decNo();}> {
   class List{
      AuxElem first, last;
      void insertAsLast(E e){e.incNo(); ... }
      E removeFirst(){first.decNo; ... }
   }
   class AuxElem{
      AuxElem next;
      E e; // Reference to the real element
} }
```

We have here constrained E by an "anonymous interface" given as part of the parameter specification, so that an actual parameter for E has to implement the given methods. One can also constrain a type parameter by listing a number of ordinary interfaces that an actual parameter for E must implement, or by a class of which an actual class for E must be a subclass. If we as constraint use a class C defined in an ordinary package, this class C will be considered the same in all instantiations of this template and of other templates using the same bound. Obviously, inside the template we are allowed to use any property of a type parameter that follows from the constraints in a type-safe manner.

If we are to use a type parameter T with a bound B to create objects of T, then the following restrictions apply:

– T must be concrete.
– T must provide the same constructors as B, and B must thus be a class. Alternatively, one could envision a scheme similar to that of C#, where required constructors are specified explicitly as part of the parameter's bound. In the latter case, B could well be an interface (even an anonymous one).

In some cases it can be useful to use a type parameter T with a bound B as superclass for a class inside the template. In order to allow this, one must provide restrictions on T so that T is not more restrictive with regards to what its subclasses can contain than B itself is. This implies the following restrictions:

– If B is concrete, then T must also be concrete.
– T must provide the same constructors as B.
– B must thus be a class.
– T cannot use covariant return types in overrides of methods from B.
– T cannot introduce final overrides of methods from B.

These restrictions should ideally be reflected by the bound B, but we currently have no specific syntax for this.

We do not allow template type parameters to also cover basic types (int, byte, etc.), e.g. like in C#. However, by introducing similar mechanisms for generics as in C#, this could also be allowed in PT, though with an additional restriction that parameters that are value types (primitives, structs, or enums) cannot be inherited from.

Templates with Template Parameters Letting templates abstract over regular type parameters provides a certain degree of flexibility that seems useful in many cases. However, a natural generalization of that mechanism would be to allow templates to abstract over templates, through the use of formal template parameters.

To be able to use templates as parameters in a meaningful way, we need a way to define bounds for them. Using a template `U` as a bound, a possibility would be to say that any template that instantiates `U` could be used in its place as an actual template parameter. However, relying on internal instantiations within a template would appear to break the principle of encapsulation, and we thus propose that a template can explicitly declare its instantiation of another template outside the template body, as shown below. We will refer to such instantiations as *explicit instantiations*, and correspondingly say that the clause specifying such an instantiation is an *explicit instantiation clause*. To illustrate this, we return to the `Vehicles` example from previous sections. Below, we see how a template `WeightVehicles` has an explicit instantiation of the `Vehicles` template:

```
template WeightVehicles inst Vehicles {
   class Vehicle adds {
      int weight;
      void display(){ tsuper.display(); out(weight); } }
   class Truck adds { ... }
   class Car adds { ... }
}
```

Note that `WeightVehicles` might make additions to the classes from `Vehicles` in the normal manner. A design decision here is whether one should be allowed to change the names of the elements from an explicit instantiation. This can probably be useful in some cases, but it will also make the structure of the program more difficult to follow. Thus, we assume for the rest of the paper that this is not allowed.

We can now write the following parameterized template, utilizing the `Vehicles` template as bound:

```
template RentalVehicles <template V inst Vehicles> {
   /* The rest is the same as in Section 3.2 */
}
```

The `RentalVehicles` template can subsequently be instantiated with an actual parameter that is `Vehicles` or a template that has an explicit instantiation clause that (transitively) includes the `Vehicles` template, such as e.g. `WeightVehicles` above.

```
package Program {
   inst RentalVehicles<WeightVehicles>;
   class Vehicle adds { ... }
   class Truck adds { ... }
   class Car adds { ... }
}
```

The instantiation of `RentalVehicles` will now instantiate `WeightVehicles`, which will instantiate `Vehicles`. The classes from `Vehicles` will form the base classes, `WeightVehicles` will then add its `adds`-clauses to these and then the package `Program` will make its additions through its `adds`-clauses, as shown above. Thus, calls in the `adds`-clauses of `Program` using `tsuper` will go to the (instance of the) `RentalVehicles` template, calls to `tsuper` there will go to `WeightVehicles` and calls to `tsuper` there will go to `Vehicles`. Note that the `tsuper` calls happen in exactly the same order as if `RentalVehicles` would have a normal (non-parameterized) instantiation of `WeightVehicles`, `WeightVehicles` would have a normal (non-explicit) instantiation of `Vehicles`, and `Program` would have a normal instantiation of `RentalVehicles`. Thus, the lookup algorithm for `tsuper` is still the same as the one presented in Section 3.2.

A template with an explicit instantiation clause can also be parameterized, and an important detail here is that the template that is explicitly instantiated can depend upon, or be one of, the template parameters, e.g. as shown below, where `U` is a formal parameter name, while `T` and `V` are actual templates:

```
template T <template U inst V> inst U { ... }
```

Interestingly enough, the construct above can be used to combine different variations of a shared base template. This can be used to solve the "expression problem" [20, 21] in a way that allows one to choose and combine different variations of a base version of expressions as needed, and in the order one wants them. The expression problem is an example showing the limitations of single inheritance. If one has a set of expressions (implemented as subclasses of a common class, like `Exp` below) and a set of operations (implemented as virtual/abstract methods of that class) one may easily add new kinds of expressions by writing new subclasses (of e.g `Exp`), but adding new operations (virtual methods) requires changes to the existing classes. A technique that allows one to easily add new operations is the Visitor Pattern in [22], but that requires changes to the existing code to add new kinds of expressions. A solution to the expression problem allows one to add both new kinds of expressions (subclasses) and new operations (methods) without changing the existing code.

So in PT, the base template for expressions can e.g. look like this:

```
template Expressions {
   abstract class Exp { }
   class Plus extends Exp { Exp left, right; }
   class Num extends Exp { int value; }
}
```

Different variations of this template may add fields and methods to the classes, override methods, and add new classes.

Below are three examples of such variations of `Expressions`, written as templates that explicitly instantiate their parameter, in order to prepare them for subsequent composition. The first template adds a method to print the expression, the second one adds a method to calculate the value of the expression, and the third adds a new kind of expression node.

```
template PrintExpressions <template E inst Expressions> inst E {
   class Exp adds { abstract void print(); } // abstract
   class Plus adds {                         // extends Exp
      void print() { out("("); left.print(); out("+");
                     right.print(); out(")")}
   }
   class Num adds { // extends Exp
      void print(){ out(value);}
   }
}

template ValueExpressions <template E inst Expressions> inst E {
   class Exp adds { abstract int value();} // abstract
   class Plus adds {                       // extends Exp
      int value() { return left.value() + right.value(); }
   }
   class Num adds { // extends Exp
      int value(){return value;}
   }
}

template MultExpressions <template E inst Expressions> inst E {
   class Mult extends Exp { }
}
```

Now, we might want a version of expression that has all three of these variations, and the solution is to write this as follows:

```
package CombinedExpressions {
   inst MultExpressions<ValueExpressions<PrintExpressions<Expressions>>>;
   class Exp adds { }        // abstract
   class Plus adds { }       // extends Exp
   class Num adds { }        // extends Exp
   class Mult adds {         // extends Exp
      void print() { out("("); left.print(); out("*");
                     right.print(); out(")")}
      int value() { return left.value() * right.value(); }
   }
}
```

The code above works because all of the templates can take the place of the template `Expressions` as a template parameter. Furthermore, since they can all be instantiated with a template that explicitly instantiates `Expressions`, they can be combined in any order as parameters to each other and we can choose only the ones that are needed. The choice of order in such cases is usually not very important, but it defines in what order the `adds`-clauses are added and which method definition is reached using `tsuper`-calls from the different `adds`-clauses. Note how the original template `Expressions` is itself used as the parameter to the template `PrintExpressions` to form the base that the other templates successively add to. Note also that the class `Mult` has neither the `print` nor `value` method when originally defined in `MultExpressions`. Those methods are required in the program as `Mult` is a subclass of the abstract class `Exp`. The two required methods can conveniently be implemented in the `adds`-clause of `Mult` in the package `CombinedExpressions`.

There are a couple of complications that arise with template parameters that are not covered satisfactorily by bounds specifications. For templates with explicit instantiations (that can thus subsequently be used as actual template parameters), we have to introduce the following additional restrictions:

– additions to classes from an explicit instantiation cannot introduce method overrides with a covariant return type,
– method overloads cannot be introduced in additions to classes from an explicit instantiation.

3.8 Access Modifiers in PT

We have so far said nothing about access modifiers for PT, but they are obviously as important in PT as in any modern language, and perhaps even more so in PT since it is a system targeted directly at modularization of programs. When incorporating PT into a new language, there are at least two important questions with regards to access modifiers that need to be resolved. One is how to best adapt the access modifier system of the underlying language to the PT extension (or vice versa), and the other is to find the new interactions that turn up with PT that also should be regulated by some type of access protection. We will discuss these questions below, with the assumption that the underlying language is Java.

We first state the fact that *templates are effectively public*, just like packages in Java. That is, there are no access modifiers that are applicable to the template definitions themselves; access modifiers are applied only to the elements within a template, i.e. classes and interfaces, and their respective attributes.

Java Access Modifiers Used in Package Templates The Java access modifiers are: private, default (none specified, which means internal to the declaring package, also referred to as "package private"), protected and public. In a Java package, classes and interfaces can have either the default accessibility, or be public. It makes sense to allow the same modifiers for class or interface definitions in templates, and thus default means that the class or interface is accessible only within the declaring template, and e.g. not to templates or packages that instantiates this template. The default accessibility for definitions inside templates can as such be referred to as "template private". Correspondingly, a public template class will be accessible from everywhere within the declaring template, and also from everywhere within an instantiating template or package.

The same scheme that is described for classes in the previous paragraph can also be applied to the modifiers private, default (package private), and public when used for attributes in classes or interfaces. (Methods in Java interfaces are implicitly public, and the same should be true for template interfaces.) However, the modifier protected turns out to have some interesting aspects when applied to PT, and we will get back to this modifier shortly. Another interesting issue concerns the modifier public, and the question of whether public elements in a template will always also be public in the template/package that instantiates them. We will get back to that later in this section.

The `protected` Modifier, and Potential New Access Modifiers In Java a `protected` attribute of a class is accessible from anywhere in the enclosing package, and also from subclasses of that class defined outside the package. However, in PT a new element similar to subclasses has emerged, namely the addition class. We may then ask whether a `protected` attribute should also automatically be accessible from an addition class, or whether it sometimes is convenient to say that an attribute is accessible from a subclass, but not from an addition class (or the other way around). From the programming and sketching we have done, it often seems natural to consider the `adds`-part as being closer to the template class than a subclass is. This should indicate that we need a modifier that expresses that the attribute is accessible from an addition class, but not from a subclass. However, for symmetry, we should then probably also have the opposite one, and we could e.g. call them `aprotected` and `eprotected` ('a' for *adds* and 'e' for *extends*). We could then let the traditional `protected` modifier mean that the attribute is accessible from both dimensions.

One could also discuss whether we need a modifier that says that an attribute is accessible from an addition class, but *not* from a subclass nor from the rest of the template. In C#, protected has this more restricted meaning for subclasses (i.e. it does not, as opposed to Java, provide access to the entire "package"). However, we feel that this is more a discussion about Java versus other languages than about PT in itself, and we will thus not pursue that question in this paper.

Access Modifiers for Instantiations In the previous subsections, we discussed the use of access modifiers within templates. However, when a template is instantiated in a scope (package or template) there is also a need to regulate the accessibility outside that scope of the definitions received from the instantiated template. The most natural place to control this is in, or somehow connected to, the `inst` statement. We therefore propose the concepts of `private` and `public` instantiations. In a `private` instantiation everything that was made accessible to the instantiating scope, will in this scope be considered "package private" (or "template private"). This might typically be used in situations where an instantiated template is used for the internal implementation of some functionality, and where one does not want to expose such implementation details to subsequent users of this functionality.

On the other hand, in a `public` instantiation everything that was made accessible to the instantiating package/template, will get the same accessibility in this scope. This could typically be used when the instantiated template is some type of framework, and one in the instantiated scope only wants to add some final definitions. Note that it should also be possible to have modifiers on the addition classes in the instantiation scope, and a reasonable rule is that if a modifier occurs it overrides the one from the template.

We think that the accessibility system described above is promising, but based on our limited programming experience with PT in a large scale setting, it seems to be too early to conclude. Thus, as we gain experience and hopefully get external input, we hope to revisit this topic as part of our future work.

3.9 Implementation

A mechanism like PT can basically be implemented in two different ways. One is a so-called heterogeneous implementation, where each instantiation of a template results in the insertion of the relevant program text (or e.g. byte code) into the instantiating package or template. The resulting package can then be compiled as a whole. This resembles how templates are implemented in C++. A potential problem with this strategy is that we might end up with a lot of code at runtime (a problem often referred to as "code bloat"). However, the problems associated with this are probably less pronounced now compared to some years ago, as the amount of available memory has been growing steeply in recent years.

The implementation we are currently working on is of the heterogenous type, and it is built upon the JastAdd system [23, 24]. In addition to its extensibility, JastAdd also provides a good compiler for traditional Java "for free".

The source code for the prototype compiler can be downloaded from the following url: `http://www.ifi.uio.no/swat/software/`. This compiler currently implements the main concepts of this paper, but not some of the more recent additions to the PT mechanism such as the `tabstract` modifier and template parameters.

The output from the prototype compiler is plain Java source code, which can then subsequently be compiled using the standard `javac` compiler. When the compiler generates the Java code, it mangles overridden method names from template classes, and subsequently transforms `tsuper` calls to ordinary calls to the mangled names. Template class constructors are implemented in a similar manner.

Since all substitutions and additions in PT are done semantically, one might also try to make a homogenous compiler, which means that only one version of the code for a template is produced and stored during execution, and tables and other mechanisms are used to keep track of the different instantiations and their adaptions to the stored code. The problem here is the speed of the execution, e.g. since merging will lead to many of the same problems as multiple inheritance does. However, we have some ideas on how this can be done in an efficient manner, and hope to be able to test them out in the near future.

4 Related Work

The trait mechanism [7] approaches composition from the angle that the unit of composition is a trait and that a trait or a class can be composed of traits. A trait is a stateless[1] collection of methods. The methods of the traits become methods of the new trait or class that is composed of traits. In addition to the methods that contribute to the composed trait or class, a trait may also specify required methods, i.e. methods that it requires the composed trait or class to have, either from other traits or from the class definition itself. The composition of traits is said to be *flattened*. This means that (1) the trait ordering in each composition is irrelevant, and (2) that a class composed from traits is semantically equal to a class in which all the methods are defined directly in the class. Traits were originally developed for the dynamic language Squeak, and supports method aliasing and exclusion upon composition. A statically typed version also exists [26].

PT is similar to traits in the following way: When classes are merged, all the methods from the merged classes are included in the resulting class. PT does not have a mechanism like required methods, and while the `tabstract` modifier might seem similar, there are quite a few differences. When a class with a `tabstract` method is merged with other classes, the `tabstract` method must be implemented in the addition class (or remain `tabstract`). If one of the other classes has a method that matches the signature of the `tabstract` method it will only be a name collision that has to be resolved. Another difference between traits and PT is that in PT composition is performed at the level of a package (template) and thus across more than one class, while trait composition applies to single traits or classes.

Mixins [8] are similar in scope to traits in that they target the reuse of smaller units that are composed into a class. Mixins also define provided and required functionality, and the main difference between them and traits is the method of composition. Mixins traditionally rely on inheritance, by defining a subclass with an as-of-yet undefined (virtual) superclass, and the result is that mixins are linearly composed. Mixins allow super calls to methods in that superclass. The actual superclass is specified when the mixin class is used in a program.

Calling methods in an as-of-yet undefined (virtual) superclass can also be achieved in PT by using templates as template parameters. This is what is done in the example with the expressions in section 3.7 where the different additions to the same class (like `Exp`) are layered; one adding to and overriding another in the order specified in the `inst` statement in the program and where a `tsuper` call would call the overridden method in the class from the "previous" template in the sequence. As with traits, another difference is that in PT composition is

[1] Traits were originally defined to be stateless, although a more recent paper [25] has shown how a stateful variant may be designed and formalized.

performed at the level of a package (template) and thus across more than one class at once.

Virtual classes were introduced in the BETA language [4, 5], and have subsequently formed the basis for a number of languages, such as e.g. gbeta [13], J& [16] (pronounced “jet”), and Caesar [14] (the latter also contains AOP features, see below). The main idea is that a class encloses one or more inner classes that are virtual in the sense that they can be overridden in subclasses of the enclosing class. Except for BETA, the mechanisms also allow some kind of merge (or multiple inheritance) of enclosing classes, with merging between inner classes following from that.

Mechanisms based upon virtual classes provide some of the same flexibility for unanticipated adaption as PT’s addition classes. The main difference is that PT does not rely on nested classes, there will thus be no objects of an enclosing class and no types dependent on such runtime objects. Also, there is no multiple inheritance in PT since combination of classes from different templates is handled with the merge construct. Since the different packages (or templates) that instantiate a template are not subtypes of the template they instantiate, this gives PT some flexibility in what one can do at merge time, but it does not allow full family polymorphism [27].

Lightweight family polymorphism [28] makes very much the same trade-off that PT does in that it represents families by classes instead of objects, whereas PT represents them by templates and packages. Thus, according to the authors, one gets a simpler type system but with some restrictions. They do not discuss merging classes from different families which is one of the main features of PT. Variant path types [29], which is an extension of Lightweight family polymorphism also introduces inheritance between classes within a family. The type system is still somewhat simpler than those of languages like gbeta and J&, but also still not as expressive. Variant path types also makes much the same trade-offs as PT, but with its exact types and inexact types and partially inexact types with their inexact qualification and exact qualification one could argue that PT is even simpler. They do allow methods that work uniformly over different families.

Classboxes [30] were designed to allow unanticipated and unintrusive changes, like adding fields and methods to existing class hierarchies and make the expanded classes available to new clients without affecting existing clients. Classboxes do not have a mechanism for merging classes like PT has, nor does it have any form of multiple inheritance, but it is similar to PT in that different additions to a base hierarchy can be combined in a layered fashion. Method overriding and lookup are very similar, with separate keywords for `super` and `tsuper` (the latter called `original` in Classboxes).

MultiJava [31] also allows methods to be added to existing classes and also allows such added methods to be overridden by methods added to subclasses, but it does not allow methods in the existing classes to be overridden by the so-called external methods. MultiJava also has symmetric multiple dispatch which neither classboxes nor PT has.

Expanders [32] is another mechanism for adding methods and fields to existing classes that only affect a well defined scope. With Expanders methods and fields can be added to interfaces and interfaces can also be added to classes. The mechanism has modular type safety and existing objects can be updated, even with new state. Methods in expanders can override methods in other expanders, but not methods in the expanded classes. Like the other mentioned mechanisms for expanding classes it does not have a merge construct.

Difference-Based Modules in MixJuice [33] are similar to PT in that the modules define or adapt more than one class, that modules can extend other modules, and that they can be combined to form new modules. Like PT, MixJuice also makes a clear distinction between subclassing and adapting a class and uses the keywords `super` and `original` where PT uses `super` and `tsuper`. It does not have a renaming or merge construct for adapting classes and combining classes that were independently written.

DeepFJig [34] uses nested static classes as modules and, like PT, it is designed for flexibility in combining modules. When (outer) classes are combined, inner classes are merged recursively when they have the same name. More than one abstract method or field with the same name is joined into one while a non-abstract one replaces or implements the abstract one. DeepFJig has a series of composition operators so that renaming, hiding and overriding can be done in much the same way as in PT. However, subtyping within a module (outer class) is different in that in DeepFJig one only declares a class to implement another and thus all elements must be (re)declared in that class, which is unlike regular inheritance which one can use within a package template, where elements are inherited from the supertype. DeepFJig does not have the separate `super` and `tsuper` calls that PT has and composition operators must be used to rename methods to achieve the same. Unlike PT, there does not seem to be any way to refer to a shared base when writing and combining different extensions in DeepFJig, like in the solution to the expression problem in [34]. PT does not allow classes, methods or fields to be removed or hidden from templates.

Newspeak [17] is a dynamically typed class based language descending from Smalltalk and Self, with no global state or namespace. All classes, including superclasses, are virtual and they can be nested arbitrarily. Newspeak is similar to PT in that there can be more than one version of a module (class in Newspeak and template in PT) at runtime and that the "actual imports" (instance parameters in Newspeak and templates as template parameters in PT) are decided by the client. Unlike Newspeak, PT has a global namespace and, like BETA, PT is based on further binding and not on completely replacing a class like replacing a virtual method. Furthermore, in Newspeak, module instances are per instance (object), while in PT they are per class or package.

Ada originally (in 1983, [35]) had no mechanisms supporting object-orientation, but it had a mechanism called generic packages with some of the same aims as package templates, in that generic packages can contain type definitions and that each instantiation of a generic package gives rise to a new set of these types.

In Ada 95 [36] a mechanism for object-orientation was introduced (further elaborated in Ada 2005). As far as the authors understand it, there is nothing similar to virtual classes (at compile-time or at runtime) in the language, and the mechanisms for adapting a package to its use are not very advanced.

Aspect-oriented programming [6, 37, 14] (AOP) is an approach to separation of concerns and code reuse where an important notion is that of crosscutting concerns, i.e. concerns that are not easily captured in just one class of an OO application. In [38] the authors state that AOP in essence is *"quantification and obliviousness"*, indicating that, according to their view, AOP involves quantification of program locations to affect and that code should be unaware of the aspect code that will affect it. In that sense, aspects can be seen as a special case of meta-level entities that examine and manipulate programs. The pointcut-mechanism has received some criticism for its fragility with respect to changes in the involved classes, known as the *fragile pointcut problem* [39].

Package templates are oblivious to both their potential usage and changes made to them by the `inst` statement or corresponding addition classes. The `inst` statement can also be seen as a limited way of quantifying program elements, by naming templates to instantiate and by specifying class merges to obtain the desired end result. So-called intertype declarations play an important role in many AOP solutions. In PT, such additions can be made to classes either through merging, or through the use of addition classes.

However, while PT as presented in this paper can solve a certain category of AOP problems, there are quite a few things that seem to better be approached through a mechanism that includes some notion of pointcut and advice. Thus, we have also performed some experiments with adding more explicit AOP support to PT, see e.g. [40]. In that paper, we strive to find a middle-way between the more restricted inst mechanism and some of the power of expression inherent to e.g. AspectJ, and show how a restricted version of pointcut and advice definitions can be incorporated into template classes and how this can be utilized to create a reusable implementation of the Observer design pattern [22, 41, 42]). The gist of the paper [40] lies in the fact that reusable entities like design patterns can be merged with other template classes in order to add the functionality of the pattern directly to them, and that abstract pointcut specifications can be concretized in addition classes, utilizing both definitions from the design pattern and from the code with which the pattern is merged.

5 Concluding Remarks

In the introduction we stated that we want PT to represent a compromise between (1) simplicity of the type system, compared with type systems for approaches built on virtual classes, and (2) expressiveness, especially for large-scale programming. This overall goal has thus been used as a guide throughout the design process in face of diverging design considerations.

Concerning (1), with a simple type system we mean a type system that is mainly easy to understand and use, but also relatively easy to implement. Below we list a few points that we think support the view that the type system of PT generally is simpler than those of systems built on virtual classes.

- The main thing that complicates type systems for approaches based upon virtual classes is that the types corresponding to inner, virtual classes depend on the object of the enclosing class. There may be several objects of this enclosing class, each of these will have the same type (and may therefore potentially be denoted by the same reference), while types according to the inner, virtual classes are different for different objects. Template classes of PT are not ordinary classes. Instantiation of a template is performed at compile-time, and it results in a set of ordinary classes, not in the scope of an enclosing object, but in the scope that instantiates it. Several instantiations of the same template will result in independent sets of classes.
- Instantiation of templates and especially merging of template classes can be described as program compositions. An implication of this is that the template classes form constituent parts of the resulting ordinary classes, so that they do not exist as separate classes after the instantiation, and there will not be objects of the template classes themselves. This is generally not the case for virtual class systems, where a class and an extension of it will often both exist and even have the same name, so some mechanisms must be introduced to distinguish them.

Concerning (2), we have demonstrated that package templates solve some well–known problems, like e.g. the expression problem. With respect to large scale programming we still need some more evidence. Hopefully, for some of the more contested design issues, such as constructor call schemes or access modifiers for template classes, the right solution will become more evident as our experience with programming with the PT mechanism grows.

With respect to reuse it is worthwhile to compare with e.g. gbeta [13] and the paper on Family Polymorphism [27] by E. Ernst. Here it is claimed that for an approach to really support reuse, it must be possible to dynamically generate any number of specific class collections (which in his setting are objects of an outer class) from a general class collection. We agree that it is important to be able to generate multiple instantiations of the collections, but we think that for most purposes it is enough to generate them statically. Combined with the mechanism of adaptions, package templates support that specific collections suit specific needs. Approaches based on virtual classes do not (by nature of the underlying mechanism) support the flexibility of renamings and merges.

The added benefit of using package templates that can be instantiated and adapted multiple times during compilation will represent a significant step forward compared to e.g. ordinary packages, and we believe that the final step towards dynamic instantiations are important only in very special cases.

Acknowledgements

We would like to thank Eivind Gard Lund and Ivar Refsdal for their dedicated work on the PT compiler. Furthermore, we would also like to extend our gratitude to the anonymous reviewers for their insightful and valuable comments on an earlier version of this paper.

The work reported in this paper has been done within the context of the SWAT project (The Research Council of Norway, grant no. 167172/V30).

References

1. Dahl, O.J., Myhrhaug, B., Nygaard, K.: Simula 67 common base language. Technical Report Publication No. S-22 (Revised edition of publication S-2), Norwegian Computing Center (1970)
2. Stroustrup, B.: Multiple inheritance for C++. Computing Systems **2** (1989) 367–395
3. Bracha, G.: Generics in the Java programming language. Technical report, Sun Microsystems, Santa Clara, CA (2004) `java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.
4. Madsen, O.L., Møller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications, New York, NY, USA, ACM (1989) 397–406
5. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-oriented programming in the BETA programming language. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1993)
6. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: LNCS. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
7. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: ECOOP 2003. Volume 2743 of LNCS., Springer Berlin / Heidelberg (2003) 327–339
8. Bracha, G., Cook, W.: Mixin-based inheritance. In Meyrowitz, N., ed.: OOPSLA/ECOOP, Ottawa, Canada, ACM Press (1990) 303–311
9. Krogdahl, S., Møller-Pedersen, B., Sørensen, F.: Exploring the use of package templates for flexible re-use of collections of related classes. Journal of Object Technology **8** (2009) 59–85
10. Sørensen, F., Axelsen, E.W., Krogdahl, S.: Reuse and combination with package templates. In: Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance. MASPEGHI '10, New York, NY, USA, ACM (2010) 3:1–3:5
11. Axelsen, E.W., Krogdahl, S.: Groovy package templates: supporting reuse and runtime adaption of class hierarchies. In: DLS '09: Proceedings of the 5th symposium on Dynamic languages, New York, NY, USA, ACM (2009) 15–26
12. Axelsen, E.W., Krogdahl, S., Møller-Pedersen, B.: Controlling dynamic module composition through an extensible meta-level API. In: DLS 2010: Proceedings of the 6th symposium on Dynamic languages, New York, NY, USA, ACM (2010)
13. Ernst, E.: gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance (1999)
14. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An overview of CaesarJ. In: Trans. AOSD I. Volume 3880 of LNCS., Springer Berlin / Heidelberg (2006) 135–173

15. Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2004) 99–115
16. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM (2006) 21–36
17. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in newspeak. In D'Hondt, T., ed.: ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings. Volume 6183 of Lecture Notes in Computer Science., Springer (2010) 405–428
18. Compagnoni, A.B., Pierce, B.C.: Higher-order intersection types and multiple inheritance. Mathematical Structures in Computer Science **6** (1996) 469–501
19. Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '06, New York, NY, USA, ACM (2006) 270–282
20. Torgersen, M.: The expression problem revisited. In: ECOOP 2004 - Object-Oriented Programming, 18th European Conference, June 14-18, 2004, Proceedings. Volume 3086 of Lecture Notes in Computer Science., Springer (2004) 123–143
21. Ernst, E.: The expression problem, scandinavian style. In Lahire, P., Arévalo, G., Astudillo, H., Black, A.P., Ernst, E., Huchard, M., Sakkinen, M., Valtchev, P., eds.: MASPEGHI 2004. (2004)
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns -Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
23. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. SIGPLAN Not. **42** (2007) 1–18
24. Ekman, T., Hedin, G.: The JastAdd system — modular extensible compiler construction. Sci. Comput. Program. **69** (2007) 14–26
25. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits and their formalization. Computer Languages, Systems & Structures **34** (2008) 83–108
26. Nierstrasz, O., Ducasse, S., Reichhart, S., Schärli, N.: Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland (2005)
27. Ernst, E.: Family polymorphism. In Knudsen, J.L., ed.: ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings. Volume 2072 of Lecture Notes in Computer Science., Springer (2001) 303–326
28. Saito, C., Igarashi, A., Viroli, M.: Lightweight family polymorphism. J. Funct. Program. **18** (2008) 285–331
29. Igarashi, A., Viroli, M.: Variant path types for scalable extensibility. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. OOPSLA '07, New York, NY, USA, ACM (2007) 113–132
30. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: controlling the scope of change in Java. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. OOPSLA '05, New York, NY, USA, ACM (2005) 177–189

31. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design rationale, compiler implementation, and applications. ACM Trans. Program. Lang. Syst. **28** (2006) 517–575
32. Warth, A., Stanojević, M., Millstein, T.: Statically scoped object adaptation with expanders. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. OOPSLA '06, New York, NY, USA, ACM (2006) 37–56
33. Ichisugi, Y., Tanaka, A.: Difference-based modules: A class-independent module mechanism. In: Proceedings of the 16th European Conference on Object-Oriented Programming. ECOOP '02, London, UK, Springer-Verlag (2002) 62–88
34. Corradi, A., Servetto, M., Zucca, E.: DeepFJig - modular composition of nested classes. In: 2010 International Workshop on Foundations of Object-Oriented Languages (FOOL '10). (2010)
35. Ledgard, H.: Reference Manual for the ADA Programming Language. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1983)
36. Barnes, J.: Programming in Ada95. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1995)
37. Colyer, A.: AspectJ. In: Aspect-Oriented Software Development. Addison-Wesley (2005) 123–143
38. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: AOSD. Addison-Wesley (2005) 21–31
39. Störzer, M., Koppen, C.: Pcdiff: Attacking the fragile pointcut problem, abstract. In: EIWAS, Berlin, Germany (2004)
40. Axelsen, E.W., Sørensen, F., Krogdahl, S.: A reusable observer pattern implementation using package templates. In: ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software, New York, NY, USA, ACM (2009) 37–42
41. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. SIGPLAN Not. **37** (2002) 161–173
42. Mezini, M., Ostermann, K.: Conquering aspects with Caesar. In: AOSD '03, New York, ACM (2003) 90–99