Reuse and Combination with Package Templates

Fredrik Sørensen
University of Oslo
Department of Informatics
P.O. Box 1080, Blindern
N-0316 Oslo, Norway
fredrso@ifi.uio.no

Eyvind W. Axelsen
University of Oslo
Department of Informatics
P.O. Box 1080, Blindern
N-0316 Oslo, Norway
eyvinda@ifi.uio.no

Stein Krogdahl
University of Oslo
Department of Informatics
P.O. Box 1080, Blindern
N-0316 Oslo, Norway
steinkr@ifi.uio.no

ABSTRACT
Package Templates (PT) is a mechanism for writing modules meant for reuse, where each module (template) consists of a collection of classes. Such a template must be instantiated in a program (at compile time) to form a set of ordinary classes, and during instantiation the classes may be adjusted with renaming and additional attributes. Package templates can be instantiated multiple times in the same program, each time with different adjustments and each time resulting in a fully independent set of classes. During instantiations, classes from two or more templates may be combined so that they get a new shared type with the properties from all the classes. This paper presents and discusses two proposed extensions to PT. The first has to do with the fact that PT naturally gets two variants of the “super” concept, where one is for ordinary superclasses, and the other is for the additions made to classes during instantiation. The second extension has to do with allowing templates to instantiate templates that are later to be specified.

Categories and Subject Descriptors
D.3 [Software]: Programming Languages; D.3.3 [Programming Languages]: Language Constructs and Features—Classes and objects; D.3.3 [Programming Languages]: Language Constructs and Features—Modules, packages

General Terms
Languages, Design

Keywords
OOP, Modularization, Inheritance, Templates

1. INTRODUCTION
The basic concepts of object orientation, that is, classes, subclasses including polymorphism, and virtual methods were introduced with the Simula language in 1967 [7]. These concepts became important for most later research concerning how languages should be designed to support separation of concerns, reuse of code etc. However, during the years after 1967, a number of mechanisms have been added to this original repertoire, e.g., multiple inheritance [22], generic classes [4], virtual classes [14], aspects [11], traits [19], mix-ins [5] and many more.

Package Templates [12], or PT for short, is another such mechanism, and it is especially aimed at supporting code reusability. A template is a kind of package (that is, a set of classes called template classes), but it must be instantiated as part of a program before the template classes become ordinary classes in the program, and such instantiations are done at compile time. Each instantiation of a template will produce a new independent set of ordinary classes, and, most importantly, in each instantiation one may specify a number of adjustments so that the resulting classes become best suited for their role in the program.

These adjustments may include adding attributes (fields or methods) to the classes and renaming of declarations in the template, and one may also (re)define abstract or virtual methods defined in the template classes. One should note that package templates are semantic units that can be fully type checked as separate entities. Bindings made during such a type check will not be invalidated by later name changes or additions.

The main theme of this paper is to discuss in further detail some extensions to the basic PT mechanism that were presented briefly in [21]. These additions center around two ideas. One is that PT naturally gets two kinds of inheritance: the regular kind related to subclasses, and another one related to adding attributes to classes during instantiation. Thus, two meanings of “super” and “abstract” connected with these two kinds of inheritance can also be introduced. In the original definition of PT [12] it was not possible to navigate explicitly along these different “dimensions”. However, we have observed that this, in some cases, might be useful for the programmer.

The other idea is to introduce template parameters to templates and that templates can provide other templates. Below, we will first describe the basic concepts of PT. Then we discuss in more detail the two ideas sketched above and show how they can be used through examples. Finally, we discuss to what extent these ideas are different from comparable proposals.

2. PACKAGE TEMPLATES
This section gives a brief overview of the PT constructs described in the introduction. As in [12], we use Java as the
underlying language. A package template is a set of classes, and these are syntactically enclosed by curly braces. As an example, this is a template `Graph` with two classes `Node` and `Edge`:

```java
template Graph {
    class Node {
        Edge[] outEdges;
        Edge insertEdgeTo(Node other){ ... } }
    class Edge {
        Node from, to;
        void delete(){ ... } }
}
```

In this paper, for simplicity, ordinary packages are also written in a similar syntax, as will be demonstrated shortly. When we want to use the above template in a program, we must, as is mentioned above, instantiate it in that program. As an example, assume that we want to represent sets of cities and roads, including which roads go between which cities, and so that each city and each road have some extra data. For representing sets of cities and roads we want to use the graph structure already implemented in the `Graph` template, with `Edge` objects representing roads and `Node` objects representing cities. The extra data for each city is `name` and `size` and for road it is `length` and `boolean isDirtRoad`. Suitable classes `City` and `Road` can then be obtained by the following instantiation of the template `Graph` (in a program represented by an ordinary package with name `Program`):

```java
package Program {
    inst Graph with Node => City, Edge => Road;
    class City adds { String name; int size; } 
    class Road adds { int length; boolean isDirtRoad; }
    ...
}
```

The instantiation is specified by the keyword `inst`, which is followed by the template name and a `with` clause specifying the new names of the template classes. Additions to the classes are given in class-like constructs, where the additions are given after the keyword `adds`. As stressed in the introduction, we can now e.g. access the added field `name` of `City` through references defined in template `Graph` with type `Node`, without any casting (so that e.g. “`to.size`” is legal and type safe in a `Road` object). Note that the classes in the template may well form an inheritance hierarchy and such a hierarchy will be preserved during an instantiation.

One should note that the classes of the instantiated template (or rather the adjusted instantiations of these) will become directly visible at the outermost level of the `Program` package. Thus it is not correct to say that the instantiation of a package template simply results in an ordinary package. The main reason for this is to make merging simple. Obviously, also other names than those of the template classes can be changed during an instantiation (one could e.g. want to change `insertEdgeTo` to `insertRoadTo`) but we do not show the syntax for this.

We now move on to merging of template classes, that is, how we can specify that certain classes should be merged when two or more instantiations are made in the same scope. As an example, assume that we again want to form classes `City` and `Road` as above, but this time, we also have another template `Geography` with classes `CityData` and `RoadData` containing most of the attributes that we wanted in addition to those from `Node` and `Edge` in `Graph`. Template `Geography` can look like this:

```java
template Geography {
    class CityData { String name; int size; }
    class RoadData { int length; } isDirtRoad is missing }
}
```

We can now form the class `City` by merging the classes `Node` and `CityData`, and likewise for `Road`. In addition, we have to add the missing `boolean isDirtRoad` to class `Road`. This can be written as follows:

```java
package Program {
    inst Graph with Node => City, Edge => Road;
    inst Geography with CityData => City, RoadData => Road;
    class City adds { }
        // Empty (and can be skipped)
    class Road adds { boolean isDirtRoad; // Adding the
                     // missing field }
}
```

We can see that the classes `Node` from `Graph` and `CityData` from `Geography` should be merged from the fact that the name of both are changed to the same name `City`, and likewise for `Edge`, `RoadData` and `Road`. The above two instantiations will result in exactly the same classes `City` and `Road` as in the first example, and consequently one can also in the latter write “`to.size`” without casts in a `Road` object.

We should finally note that even if we use single inheritance within each template, we may indirectly get multiple inheritance if we merge two classes that both (in their respective templates) have superclasses. However, as we do not want to force languages that introduce PT to also introduce multiple inheritance, we adopt the rule that if two classes that are merged both have superclasses, then these superclasses must also be merged.

### 3. INHERITANCE AND ADDITIONS

In addition to allowing methods to be added and overridden in a subclass in the usual way, one may add and override methods in the `adds`-clauses. 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. Note that since `Truck` extends `Vehicle` in the template `Vehicles`, `TrafficTruck` extends `TrafficVehicle` in the template `Traffic`.

```java
template Vehicles {
    class Vehicle { void move(int dist){ ... } }
    class Truck extends Vehicle {
        void move(int dist){ ... super.move(dist); ... } }
}
```

```java
template Traffic {
    class TrafficVehicle {
        void move(int dist){ ... super.move(dist); ... } }
    class TrafficTruck {
        void move(int dist){ ... super.move(dist); ... } }
}
```

The method `move(...)` in `Truck` overrides the one in `Vehicle` in the usual way for virtual methods. A call to the `super-method`, like in `Truck`, will look to the regular subclass (originally `Vehicle`) for the method. However, in `Traffic`, where `TrafficVehicle` is the superclass of `Truck` (now `TrafficTruck`) that call to `super.move(...)` will invoke the `move` method in `TrafficVehicle` and the same is true for a supercall in the `adds-part` of `TrafficTruck`. To be able to invoke the overridden method in `Vehicle`, we add a new keyword to PT: `tsuper`. It has a similar meaning from
additions to classes as super has from subclasses to classes. The call to tsuper.move(..) in TrafficVehicle will call the one in Vehicle and tsuper.move(..) in TrafficTruck will call the one in Truck. This way, all the overridden methods can be reached.

We also introduce the keyword tabstract and it has a similar relation to abstract. A method that is abstract must be implemented in a subclass. A method that is tabstract must be implemented in an adds-clause. However, there is a subtle and important difference between the two. 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 it is known that they will become concrete classes when the template is instantiated in a package (program). Such tabstract methods can, for example, be used to write the implementation of the Observer Protocol [9], below.

```java
package ObserverProtocol {
    class Observer {
        tabstract void notify(Subject changee); }
    class Subject {
        List<Observer> observers = new List<Observer>();
        void addObserver(Observer o) { observers.add(o); }
        void changed() {
            for each(Observer o in observers) o.notify(this);
        }
    }
}
```

Here, the class Observer has the tabstract method notify. A class that is created from Observer when an instance of this template is made, must implement this method. The method notify is used as usual in changed.

The template ObserverProtocol can be used together with the template Vehicles above to make a package TrafficObservation, shown below.

```java
package TrafficObservation {
    class Vehicle {
        void move(int dist){
            tsuper.move(dist); // Reuse from Vehicle
        }
    }
}
```

Here Vehicle is merged with Subject to become the new class TrafficVehicle. The method move in the adds-clause of Vehicle in the package overrides the one in Vehicle in the template. To make sure that the method also does what it used to do, a tsuper call is made. Since Vehicle now has the method changed from the class Subject, that method can be called as if it was defined in Vehicle. Note that calls to super.move(..) in Truck in the template Vehicles will invoke the method move defined in Vehicle in the package above and correctly result in a call to the changed method.

4. TEMPLATE PARAMETERS

In PT, one may also instantiate templates in templates. In basic PT, the exact templates used in the inst-clauses of a template are known when writing the template. We propose adding template parameters to templates that can be used in the inst-clauses. Thus, the choice of the actual templates can be delayed to the place where the template is used (instantiated). In order to give some bounds on the formal parameters (for static checking), the concept of one template providing another is introduced. Using this, the template Traffic from the previous section can be written as shown below (this time without renaming for simplicity, even though renaming would cause no problems as discussed above).

```java
template Traffic provides Vehicles with T {
    T inst Vehicles with Vehicle=>Vehicle, Truck=>Truck;
    class Vehicle adds { /* Same as TrafficVehicle */ }
    class Truck adds { /* Same as TrafficTruck */ }
}
```

That template Traffic provides Vehicles means that it must have an inst of Vehicles or of another template that provides Vehicles. It can have more than one instance of Vehicles, but only one of those instances (here T) can be named as the provided one using the with keyword. Traffic can then take the place of Vehicles as an actual parameter to a template as we will see below.

The template TrafficObservation from the previous section can now be rewritten as below, using Vehicles as a parameter bound for the parameter V. We change the first line of the template code to the inst-clause below and keep the rest of the template as it was in the previous section. We now have a simulation template that can be used with any template that provides the template Vehicles. In the version of Traffic from this section, the classes from Vehicles are not renamed. This could have been done and yet the instantiating template could use the original names just as it is done below. This is because they would still bind to the same classes that come from Vehicles according to the extended PT rules.

```java
template TrafficObservation <template V provides Vehicles> {
    inst V with Vehicle => Vehicle, Truck => Truck;
    /* The rest is the same as in the previous section */
}
```

Inside a template, any number of instantiations can be made using the parameters and the classes from them can be merged with classes from instances made from other parameters or with classes from instances of templates that are fully known at that point, like ObserverProtocol. Renaming and all the kinds of overriding and reuse that are allowed for a normal inst-clause are allowed for one that uses a parameter.

We can now use the version of TrafficObservation from this section with the version of Traffic from this section.

```java
package Program {
    inst TrafficObservation<Traffic>;
    class Vehicle adds { 
    }
    class Truck adds { /* Same as TrafficVehicle */ }
}
```

Note that calls in the adds-clauses of Program using tsuper will go to the TrafficObservation template, calls to tsuper there will go to Traffic and calls to tsuper there will go to Vehicles.

Templates parameterized with templates can also be used to combine different extensions to a shared base template. This can be used to implement a solution to what is known as the “expression problem” [23] in a way that allows one to choose and combine different extensions as needed. Below is a template with an abstract class representing an expression and some subclasses for different kinds of expressions.

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

4. TEMPLATE PARAMETERS

In PT, one may also instantiate templates in templates. In basic PT, the exact templates used in the inst-clauses of a template are known when writing the template. We propose adding template parameters to templates that can be used in the inst-clauses. Thus, the choice of the actual templates can be delayed to the place where the template is used (instantiated). In order to give some bounds on the formal parameters (for static checking), the concept of one template providing another is introduced. Using this, the template Traffic from the previous section can be written as shown below (this time without renaming for simplicity, even though renaming would cause no problems as discussed above).

```java
template Traffic provides Vehicles with T {
    T inst Vehicles with Vehicle=>Vehicle, Truck=>Truck;
    class Vehicle adds { /* Same as TrafficVehicle */ }
    class Truck adds { /* Same as TrafficTruck */ }
}
```

That template Traffic provides Vehicles means that it must have an inst of Vehicles or of another template that provides Vehicles. It can have more than one instance of Vehicles, but only one of those instances (here T) can be named as the provided one using the with keyword. Traffic can then take the place of Vehicles as an actual parameter to a template as we will see below.

The template TrafficObservation from the previous section can now be rewritten as below, using Vehicles as a parameter bound for the parameter V. We change the first line of the template code to the inst-clause below and keep the rest of the template as it was in the previous section. We now have a simulation template that can be used with any template that provides the template Vehicles. In the version of Traffic from this section, the classes from Vehicles are not renamed. This could have been done and yet the instantiating template could use the original names just as it is done below. This is because they would still bind to the same classes that come from Vehicles according to the extended PT rules.

```java
template TrafficObservation <template V provides Vehicles> {
    inst V with Vehicle => Vehicle, Truck => Truck;
    /* The rest is the same as in the previous section */
}
```

Inside a template, any number of instantiations can be made using the parameters and the classes from them can be merged with classes from instances made from other parameters or with classes from instances of templates that are fully known at that point, like ObserverProtocol. Renaming and all the kinds of overriding and reuse that are allowed for a normal inst-clause are allowed for one that uses a parameter.

We can now use the version of TrafficObservation from this section with the version of Traffic from this section.

```java
package Program {
    inst TrafficObservation<Traffic>;
    class Vehicle adds { 
    }
    class Truck adds { /* Same as TrafficVehicle */ }
}
```

Note that calls in the adds-clauses of Program using tsuper will go to the TrafficObservation template, calls to tsuper there will go to Traffic and calls to tsuper there will go to Vehicles.

Templates parameterized with templates can also be used to combine different extensions to a shared base template. This can be used to implement a solution to what is known as the “expression problem” [23] in a way that allows one to choose and combine different extensions as needed. Below is a template with an abstract class representing an expression and some subclasses for different kinds of expressions.

```java
abstract class Exp {
    class Plus extends Exp { Exp left, right; }
    class Num extends Exp { int value; }
}
```
Below are three examples of extensions to Expressions. The first adds a method to print the expression, the second adds a method to calculate the value of the expression, and the third adds a new kind of expression. In addition to adding methods and classes, the templates could also have added variables to the classes. Note the two first add an abstract method in the abstract class and then implement it in the subclasses.

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

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

template MoreExpressions <template E provides
Expressions> provides Expressions with T { 
T: inst E with Exp>>Exp, Plus>>Plus, Num>>Num;
class Minus adds { return left.value() -
right.value(); } }
class Num adds { } // extends Exp
class Plus adds { } // extends Exp
class Exp adds { abstract int value(); } // abstract
}
```

The templates can be combined as follows.

```cpp
program CombinedExpressions {
inst MoreExpressions<ValueExpressions<
PrintExpressions<Expressions>>>
with Exp>>Exp, Plus>>Plus, Num>>Num;
class Exp adds { } // abstract
class Plus adds { } // extends Exp
class Num adds { } // extends Exp
void print() { } // adds missing methods
int value(){ } // } } // end template
}
```

This works since all the templates can take the place of the template Expressions, and since they can all be instantiated with a template that provides 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 defines in what order the adds-clauses are added and which method is reached using tsuper-calls, if there are any. Note how the original template Expressions is itself used as the parameter to the template PrintExpressions to form the basis that the other templates successively add to and override. Note that the class Minus has neither the print nor value method when originally defined in More-Expressions. Those methods are required in the program as it is there a subclass of the abstract class Ext. The two methods, though, can simply be added in the adds-clause of Minus in CombinedExpressions.

### 5. RELATED WORK

Traits [19] are stateless collections of provided and required methods that are composed into classes. Traits were originally developed for the dynamic language Squeak, but a statically typed version also exists [17]. The composition of traits is said to be flattened in 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.

Mixins [5] are similar to traits, in that they enable the reuse of small units of code. 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 as-of-yet undefined parent, and thereby requiring that mixins are linearly composed.

PT can mimic trait-like composition (flattened) at the level of merging classes from different templates and a mixin-like combination with an unknown superclass (linearization) using template parameters as in the last example. The biggest conceptual differences between mixins/traits and PT is that PT is targeted towards reusing and specializing several classes as one coherent unit and that the types from the units are re-typed (and can be renamed).

Like PT, Mixin layers [20] is a mechanism for writing an addition with affect across multiple entities like classes. Mixin layers can be composed by instantiating one layer with another as its parameter and thus mixin layers are both reusable and interchangeable. They can also be nested. However, there does not seem to be a way to build hierarchies within a mixin layer.

BETA [15, 14], gbeta [8] and J& [18] are systems that in many ways are similar to each other and to a certain extent can achieve similar end results to those of PT. A common property of all of them (except PT, that is) is that they use virtual classes (as introduced by BETA) to enable specialisation and adaption of hierarchies of related classes. gbeta and J& support multiple inheritance, and this may to a certain extent be used to "merge" (in the PT sense of the word) independent classes. Note that, compared to the full family polymorphism obtained in gbeta, PT has given up on this so that one is freer to make name changes, etc, and to get a simpler type system.


Caesar [1, 16] supports both aspect-oriented programming constructs and code reuse and specialization through the use of virtual classes.

In a subject-oriented [10] programming (SOP) system, different subjects may have different views of the (shared) objects of an application. There is no global concept of a class; each subject defines ‘partial classes’ that model that subject’s world view. What is called a merge in SOP, is somewhat different from a merge in PT. SOP targets a broader scope, with entire (possibly distributed) systems (that may even be written in different languages) being composed.

Ada originally (in 1983, [13]) had no mechanisms supporting object-orientation, but it had a mechanism called generic packages with some of the same aims as templates in PT; in that such packages can contain type definitions and that you get a new set of these each time the generic package is instantiated. In Ada 95 [2] a mechanism for object-orientation was introduced (further elaborated in Ada 2005). Thus, the potential for PT-like mechanisms should be there, but as far as the authors understand it, there is nothing similar to vir-
tual classes (at compile-time or at runtime) in the language, and the mechanisms for adapting a package to its use are not very advanced.

6. CONCLUSION

Package Templates with template parameters provide flexible ways for writing code that combines and extends classes from different templates that can then deal with different concerns. A whole group of classes is encapsulated in a template and the inheritance hierarchy in a template is preserved when the template is used. The classes in a template are re-typed simultaneously and each instance of a template is completely independent of others.

By using a template parameter in the inst-clauses, flexible merging and adding is possible even when the actual template is not specified until the parameterized template is used. This works well together with the overriding and reuse with tauper of the adds-clauses that run orthogonally to the regular inheritance within templates. With the flexibility of merging and renaming it is also possible to take code that has been written completely separately and merge it and adapt it into a system.

Many details remain to be resolved regarding the language syntax and semantics, and the compiler for the language is far from completed. It also needs to be verified that such a mechanism is type safe.

7. ACKNOWLEDGEMENTS

Thanks to David Lievens and Bill Harrison for productive discussions about templates and inheritance and a lot more at Trinity College Dublin.

8. REFERENCES


[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns -Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


