Dynamic Composition 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

We show how *package templates*, a mechanism for code modularization, can be extended with features for dynamic loading. We pose the question of whether or not this is may a useful mechanism with respect to software composition and dynamic configuration of software.

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, Dynamicity, Templates

1. INTRODUCTION

There are several challenges when working with a large software system, including modularization, separation of concerns and reuse. These challenges are large enough in themselves when writing, testing and maintaining a large system. However, even more challenges arise when there are many different variations of a system, when the environment changes over time and when new requirements and extensions may arrive after the initial system has started running and should not be taken down.

Given these challenges, it seems beneficial to have similar support for such dynamic features as one has for the static build of large systems. One mechanism that is useful in this respect is the use of interfaces and abstract classes in writing the system and the possibility of loading different implementations into a running system based on configurations and runtime events. These implementations may be written and separately compiled after the system they are loaded into has been started.

Although a set of such classes may be written, compiled and loaded together, they are still considered as separate entities and not checked as one coherent entity upon loading. There is no assurance that the individual classes belong to the same version of the extension. In this work, we are looking at a way to dynamically load an entity that represents a group of classes in a package or template. We believe it to be useful if one is allowed to take a group of statically checked classes, that are compiled and tested together, and adapt them in a coordinated fashion to an existing system.

A recent paper [15] describes a new language mechanism called PT (short for Package Templates), which is meant to be a useful tool for software composition. The mechanism is intended for object oriented languages in order to support better code modularization, where statically type-checked templates can be written independently and subsequently be instantiated in programs with such flexibility that classes may be *merged*, methods renamed and new classes added. Templates may contain hierarchies of classes (single inheritance) and these hierarchies are preserved when a template is used. Also, different instantiations of the same templates are independent of each other, creating unique types.

The basic PT mechanism allows flexibility in package reuse, program composition and support for readability and reusability in large systems. The way that multiple classes may be affected by instantiating a template gives the language an AOP-like quality. More AOP-specific extensions to PT have also been studied in [3].

In this work, we look at a possible extension to the basic package template mechanism that supports dynamic loading of package templates. We ask to what extent this will be a useful tool for dynamic configuration of software systems. Furthermore, we discuss some of the properties of this mechanism.

We introduce an extension to PT where templates are parameterized by templates. A template with template parameters must be instantiated with actual templates that "extend" the formal parameters' bounds. In standard PT, all instantiations of templates are done at compile-time. However, in this work we look at extending this concept so that templates may be loaded dynamically into a running sys-

tem, not unlike how classes may be loaded dynamically in languages like Java. We discuss how this can be achieved with template extensions and with parameterized templates.

Being able to load classes dynamically into a system is useful since it allows extensions to be written after the system has started running. It also allows one to configure and reconfigure a running systems and for a system to configure itself based on discovery of its environment.

Dynamically loading classes in a controlled type-checked way has advantages over other dynamic linking mechanisms. A compiler and loader will together have checked that the loaded class can be used in a type safe way. Doing this at the levels of templates, each containing several related classes, extends the reach of this checking.

The mechanism proposed here for dynamically loading templates in PT preserves the relation between the classes in the template and thereby supports family polymorphism [9]. It has the advantage of the flexible adaption and name change mechanisms of PT while still being dynamic. Since new types are created when a template is loaded, different versions of the same software package can safely be used simultaneously in the same runtime without name clashes. Thus, PT extended this way becomes more a mechanism of dynamic composition than a static mechanism of reuse and extension.

We will first present an overview of the basic package template mechanism. Then we will present templates that extend other templates and template parameterized templates. After that, we will present dynamic loading before a discussion and a survey of related work.

2. OVERVIEW OF THE PACKAGE TEMPLATE MECHANISM

We here give a brief and general overview of the package template mechanism as it is described in [15]. The mechanism is not in itself tied to any particular object-oriented language, but the examples will be presented in a Java-like syntax, and the forthcoming examples will be based on Java.

A package template looks much like a regular Java package, but the classes are always written in one file and a syntax is used where curly braces enclose the contents of the template, e.g. as follows:

```
template T {
  class A { ... }
  class B extends A { ... }
}
```

Valid contents of a template (with a few exceptions) are also valid as plain Java programs. As such, templates may also be type checked independently of their potential usage(s).

In PT, a template is instantiated at compile time with an `inst` statement like below. This has some significant differences from Java's `import`. Most notably, an instantiation will create a local copy of the template classes, potentially with specified modifications, within the package where the `inst` statement occurs.

```
package P {
  inst T with A => C, B => D;
  class C adds { ... }
  class D adds { ... } // D extends C, see text
}
```

In this example, a unique instance of the contents of the package template `T` will be created and imported into the package `P`. In its simplest form, the `inst` statement just names the template to be instantiated, e.g. `inst T`, without any other clauses. In the example above the template classes `A` and `B` are also renamed to `C` and `D`, respectively, and expansions are made to these classes. Expansions are written in `adds`-clauses, and may add variables and methods, and also override virtual or implement abstract methods from the template class.

An important property of PT is that everything in the instantiated template that was typed with classes from this template (`A` and `B`) is *re-typed* to the corresponding expansions (`C` and `D`) at the time of instantiation (PT rules guarantee that this is type-safe). Any sub/super-type relations within the template is preserved in the package where it is instantiated. Therefore, implicitly `D` extends `C` since `B` extends `A`.

Another important property is that classes from different, possibly unrelated, templates may also be *merged* to form one new class, upon instantiation. Consider the simple example below.

```
template T {
  class A { int i; A m1(A a) { ... } }
}
template U {
  class B { int j; B m2(B b) { ... } }
}
```

Consider now the following usage of these templates:

```
inst T with A => MergeAB;
inst U with B => MergeAB;

class MergeAB adds {
   int k;
   MergeAB m2(MergeAB ab) { return ab.m1(this); }
}
```

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`.[1] Note that both `m1` and `m2` now have signatures of the form `MergeAB` → `MergeAB`.

Summing up, some of the useful properties of PT are: It supports writing reusable templates of interdependent, cooperating classes which may be statically type checked without any information about their usage. Upon instantiation, a template class may be customized, and merged with other template classes. References within a template to a template class will be re-typed according to the instantiation. After all the instantiations, a PT program will have a set of regular classes with single inheritance, like an ordinary Java program.

[1]The handling of potential name clashes resulting from a merge is beyond the scope of this article, but PT has rules and the rename mechanism discussed to deal with this

3. TEMPLATE EXTENSIONS AND TEMPLATE PARAMETERS

In this section we propose an extension to PT where templates may have bounded template parameters. Dynamic loading of templates will be presented in the next section.

To be able to enforce a bound on template parameters, we also introduce the concept of a template *extending* another, and thus also of *sub-templates*. A skeleton of a parameterized template may look as follows.

```
template W <template S extends U, template T extends V>
{ ... }
```

`S` and `T` are template parameters and `U` and `V` are statically known templates that serve as parameter bounds for the respective parameters.

When a parameterized template is instantiated, one must provide an actual template for each parameter, which must be an extension of the bound of the formal parameter.

Below is a template called `Ext`. This template could be part of a framework and programmers would be supposed to write extensions to it in order for their code to use the functionality of the framework. Other templates in the framework (like `Use` below) may have parameters bounded by the template and may hence be instantiated with a programmer's sub-templates of `Ext` as parameters. The template `Ext` and a sub-template `ExtOne` may look as follows.

```
template Ext {
  class A { void m1(B b) { ... } }
  class B { void m2(A a) { ... } }
}
template ExtOne extends Ext {
  class A adds { ... }
  class B adds {
    void m2(A a) { ... } // redefines m2
    void m3(B b) { ... } // new method m3
  }
  class C { ... }        // new class C
}
```

There may be an open ended number of extensions to a template, written separately and without knowledge of each other. The extensions may override method implementations and add methods and properties to classes, and they may also add new classes and instantiate other templates. Extension templates can only extend one template, and there is an implicit instantiation (`inst`) of the extended template within the extension. For now, we will not consider the possibility of allowing name changes in extension templates.

Below, the template `Use` is defined with a template parameter `E` bounded by `Ext`. The template parameter can be used in an `inst` statement in `Use`, but the actual instantiation is postponed until an actual parameter is provided.

Within `Use`, it is known from the bound `Ext` and the `inst` statement that the template will have at least the classes `A` and `B` from `Ext`, and they may be used in `Use` in the same way as classes from a regular `inst` statement.

Thus, through the use of `adds`-clauses (such as `A` and `B` below), the parameterized template may add fields and methods to the classes defined in the parameter bounds, as well as override methods from these classes. Furthermore, it may instantiate other templates using a normal `inst` statement, and add its own classes (such as `C` in the example below).

```
template Use<template E extends Ext> {
  inst E;
  class A adds { ... }
  class B adds {
    void m2(A a) { ... } // redefines m2
    void m3(B b) { ... } // new method m3
  }
  class C {              // new class C
    void foo(B b) {
      new A().m1(b);
    }
  }
}
```

In a program, the template `Use` can be instantiated with `ExtOne` as its actual parameter, as shown in the example below. In the package[2] `Program`, the contents of the parameterized template and of the actual parameter are statically known, and make up the available classes (and interfaces) accessible from the instance of `Use<ExtOne>`. For these classes, additions may be supplied in the normal PT manner, as shown below for `A`, `B`, `C` and `D`. Other templates may be instantiated as well, and merging may be performed as for normal PT instantiations.

```
package Program {
  inst Use<ExtOne> with Use.C=>C, ExtOne.C=>D;
  rename ExtOne.B.m3=>m4;
  class A adds { ... }
  class B adds { ... }
  class C adds { ... }
  class D adds { ... }
  class E { ... }
}
```

Both the actual parameter (here `ExtOne`) and the parameterized template (here `Use`) may have a class that overrides a method in a class defined in the template bound `Ext` (like `m2` above). In that case, the general rule is that changes from the parameterized template (`Use`) override changes from the extending template (`ExtOne`). This rule fits into a programming pattern where it is the programmer of the parameterized template who is in charge and wants to use the method in the parameterized template regardless of the extension to the base template. However, we envision that there might be a need for users to change this precedence, but we explicitly leave that topic open for future work.

A similar issue is the question of what should happen when the extension (actual parameter) and the parameterized template both have defined new classes with the same name (like `C` above). In such situations, these new classes are considered to be separate classes, and must be renamed in the package `Program` in the regular PT fashion to avoid ambiguity (as in the example above). The same goes for methods within an existing class (like `m3` above). A regular package may also add classes (like `E`).

Below is an example that illustrates some of the different situations that might occur with regards to method overrides.

[2]Like templates, packages are written within curly brackets in PT.

```
template U {
  class A { void m(){ ... } }
  class B extends A { void m(){ ... } }
}
template V extends U {
  class A adds { void m(){ ... } }
  class B { void m(){ ... } } // extends A implicitly
}
template X <template UU extends U> {
  class A adds { void m(){ ... } }
  class B { void m(){ ... } } // extends A implicitly
}
package P {
  inst X<V>;
}
```

Package `P` will have classes `A` and `B`, both with a method `m`. The methods will be the ones from template `X`, since they override the others. As with regular single inheritance a call to `super` in `B` will invoke the method defined in `A`. If, on the other hand, `A` in `X` did not define an override of `m`, a call to `super.m` in `X.B.m` would call the implementation in `V.A`.

However, if one wants to reuse the methods that are overridden by the template mechanism as opposed to by ordinary class inheritance, the keyword `tsuper` may be used, and a call to `tsuper` in `A` in template `X` will call the method defined in `A` in the template that is given as the actual parameter (in `P` this is `V`). A call to `tsuper` in `A` in template `V` will invoke the one in `A` in template `U`. Combining `super` and `tsuper` yields a useful and flexible mechanism for reuse.

A package can be used as a regular Java package in other compilation units and its classes are regular classes. A regular Java class may, for example, refer to (and import) the class `P.A`. We will see later that regular classes can also have template parameters, and these work in a different way.

We have not worked out rules for visibility or access restriction at the package level yet. Hence, there is no mention of public or private classes or methods.

There are obviously many other questions around templates with template parameters that are not fully answered in the text above, but to keep this exposition fairly short, we will not pursue all of these questions here.

4. DYNAMIC INSTANTIATIONS

We saw in Section 2 how to instantiate templates, and how to merge classes from different templates by using the compile-time `inst` construct. In section 3 we saw how templates can have template parameters and how to write sub-templates. In this section we introduce *dynamic* instantiation and adaption of templates. We believe this is very useful, as which features (in the form of templates) should be used is often not known until after the execution has started. For simplicity and to keep this short we limit our detailed discussion here to instantiating templates dynamically without any name changes or merges.

The approach we use is based on the hierarchies of templates formed through the *extends*-relation, and on using this somewhat like the hierarchies of subclasses in traditional object-oriented programming. Thus, we can type e.g. a variable with a *template based type*, and it can thereby refer to instances of that template, or to instances of a sub-template.

However, we shall also use template types in a way that is somewhat unusual, by saying that each *template instance* has a type of it own which includes both the template of which it is an instance, and the identity of the instance. Thus, two instances of the same template have different types. This is done to make it easier to handle the fact that the "same" local class in different instances of a template are indeed different classes. To form a consistent model, we also say that the full type of an instance is a subtype of the template it is an instance of.

In the following discussion, we will use as an example the three templates below. Templates `U1` and `U2` can be written after a program referring to `U` has started running. They can be separately compiled and neither of `U1` and `U2` need any knowledge of the other (nor does `U` need any knowledge of its descendants, obviously).

```
template U { class A {...} class B {...}  }
template U1 extends U {
  class A adds {...}   class B adds {...} }
template U2 extends U {
  class A adds {...}   class B adds {...} }
```

In this context, `U` can often be seen as a sort of "template interface", providing mostly abstract classes (in a template sense), and `U1` and `U2` can then be different implementations of this interface. At runtime, a program referring to U may load one of the templates `U1` or `U2` (or further sub-templates of these) and create one or more instances of it. Such instances can be kept track of by template-typed variables and can be passed around by assignments etc. according to normal object-oriented polymorphism rules, e.g as the following code.

```
Instance<instance ? of U> u = /* A dynamically
            generated instance of U, U1 or U2 */;
Instance<instance ? of U1> u1 = /* A dynamically
            generated instance of U1 */;
u = u1;
```

Here, `Instance<T>` is a class much like the class `Class` in Java. It is parameterized by an instance type `T` and has the signature `class Instance <instance T>`. It represents a template *instance* (and not a template) in the same way that `Class` represents a class. The reason that we use a class that represents the instance and not the template is that every instance generation results in the creation of a new instance type and new types for all the classes in the instantiated template. The concrete mechanism for performing a dynamic load and instantiation will be explained shortly, but the result is an object of the class `Instance<T>`. The special syntax `<instance ?  of U>` tells the compiler that the exact instance is not known statically, but that it is a sub-template of `U`. The parameter `T` of `Instance` will be bound to the type of the instance. As is shown in the last line above, template instance references may be assigned to a template variable having a more general type. Also, an obvious form of casting can be used for the opposite case.

In the program, a method can have template parameters bounded by U in the following way:

```
<instance T of U> void method(){
  T.A a = new T.A();
  a.doStuff();
}
```

Within such a parameterized method, elements can be typed with classes from the template using the template type as a prefix, like `T.A` above. Code like this makes it possible to use classes and invoke methods in dynamically instantiated templates in a type-safe way. Type safety depends on the fact that, in the scope, `T` is bound to an instance and `T` (a type parameter) does not change like object variables.

We are considering whether a syntax like the following should be allowed:

```
Instance<instance ? of U> u = /* A dynamically
               generated instance of U1 or U2 */;
...
method<u.TYPE>();
```

Here, the formal parameter `T` in the method is bound to the current type of `u` (which is identical with the instance identity), and will remain so throughout the body of the method. This will fail if `u` is a `null` value.

A class can be written with the same kind of template parameter. This can look as follows:

```
class P <instance T of U> {
  public T.A a;
  public P(){
    ...
    a = new T.A();
    ...
  }
}
```

This class can be statically and separately checked in the same way that a generic class can be type checked, with the difference being the *dot-named* classes, like `T.A`. `T` will be the same type in this scope and `T` will be an instance of `U` or of a subtype of `U`. Thus, the class `T.A` can be seen as a type just like any type and it has all the properties of `A` in `U`. Note that if `A` has a method `void m(B b)` it can be invoked with `m(new T.B())` here. `T.A` and `T.B` will, since `T` is the same, be from the same instance and at runtime the actual type of `B` will match up with the method signature.

The class `P` can then be used e.g. as follows:

```
Instance<instance ? of U> u = /* Instance of U or of a
                        sub-template */;
P<?> pu  = new P<u.TYPE>();

...   // Maybe another assignment to u

P<?> pu1 = new P<u.TYPE>();

pu = pu1;       // OK
pu.a = pu1.a;   // COMPILE TIME ERROR !
```

Here, `P<?>` is a type where `?` is similar to `?` in Java generics in that `pu` can point to any object of `P`. Similarly, `u` can point to an instance class for any instance of `U` or instance of a sub-template of `U`. Since it is not known statically in this scope if `pu` and `pu1` point to an object created with the same template instance (because of the question mark), the last assignment is not legal.

Objects of the classes of a template instance may be passed around like the template instance itself. This can be illustrated by the following two methods:

```
<instance T of U> void method_1() {
  T.A a1 = new T.A();
  T.B a2 = new T.B();
  method_2<T>(a1, a2);  // <T> may be omitted as it
}                       // can be inferred

<instance Z extends U> void method_2(Z.A a1, Z.B b) {
  ...; a1.m(b); ...;
}
```

When `method_1` is invoked, `T` is bound to the type of some template instance `u`. The clause `T.A` is bound to the type of `A` for that particular instance. The second method takes two formal parameters that are of types called `Z.A` and `Z.B` where `Z` is a the type of the template instance (a sub-type of `U`). The invocation of this method in the first method can be type checked since both the actual parameters are typed with class `A` from the same template which is also a sub-template of `U`. Inside the second method, the methods (for example `m`) defined in template `U` can be called.

In a scope, there is often only one known template that is being used and it would be nice not to have to write the instance type parameter `T` all the time. Therefore, we propose the shorthand notation shown below. Within the `with`-block, class names can be written without the type prefix.

```
<instance T of U> void method(){
  ...
  with(T) do {
    ...; .A a = new .A(); ...;
  }
  ...;
}
```

Other times, one may want to work with two (or more) different instances of a template (or more likely, two instances of different sub-templates) at the same time. Below we assumes that `A` in `Ext` has a field `b` of type `B` and that `B` has a field `x` of type `int`.

```
<instance T of Ext, instance U of ExtOne>
                    U.A[] method(T.A tas[]) {
  U.A uas[] = new U.A[tas.length];
  for (int i = 0; i<tas.length; i++){
    uas[i] = new U.A();
    uas[i].b = new U.B();
    uas[i].b.x = tas.[i].b.x; // A and B from different
  }                           // instances are never
  return uas;                 // mixed up
}
```

Dynamically generated instances of templates are produced by a special loader, with a method `instantiate` that has a template parameter, and a normal `String` parameter. The first parameter should be a statically known template, and the second should be a filename (or net-address, etc.) where a sub-template of the template parameter can be found. The loader will check that this is the case, and maybe also compile the template if necessary. Thus, a dynamic instantiation may look as below. The exact details of the loader and its implementation are not worked out, but at runtime it can be checked that it will only return an instance of the given template or a sub-template.

```
Instance<instance ? of U> u =
                TLoader.instantiate<U>("-file-");
```

Just as metods and classes in regular classes can be parameterized with regular template instances and used with any

instance of any sub-template, they can also be parameterized with a parameterized template. The example below is a class that uses the template `Use` from earlier as a parameter bound.

```
class StartOff<instance T of Use>{
  run(){ ...
     new T.C().foo(new T.B());
     ...  }
}
```

An object can be created of this class using any instance of `Use` instantiated with any sub-template of `Ext`. All the classes known in `Ext` can be used within this class, prefixed by `T`. Below is an example of instantiating an instance of `Use` with `ExtOne` and using `StartOff`.

```
Instance<instance ? of Use> u =
          TLoader.instantiate<Use>("-Use<ExtOne>-");
StartOff<?> s = new StartOff<u.TYPE>();
s.run();
```

Note that the use of the name `C` in `StartOff` refers to the one originating in `Ext` and that the one from `ExtOne` is not visible in `StartOff`.

All these examples of regular classes and methods parameterized by templates are mainly there to be starting points for the code within the templates. The interesting code will probably be inside templates `Ext` and `Use` and classes like `StartOff` will usually just set this off.

There are more details about dynamic instantiations of package templates that have not been discussed here and open questions obviously still exist, but we nevertheless believe that the mechanism should be useful as a starting point for developing a language mechanism for dynamically configured and composed systems.

5. DISCUSSION AND FURTHER WORK

The aim of this work is to develop tools that allow parts of larger software systems to be written as independent pieces and that can be merged in a flexible way. This should provide flexibility in both organization of a system and reuse of components. To be truly flexible, merging and composition of independently written parts of software should even be allowed at runtime. However, one would like to do this with some sort of static checking to avoid some of the often occurring errors with uncoupled code. Also, there is value in the ability to run different versions of a library or framework in the same runtime without having to deal with conflicting types. We try to solve this with an extension of the package template (PT) mechanism.

PT and the extensions proposed here should provide some of the apparatus not only for merging and composing unrelated templates of cooperating classes, but also for doing so in a running system.

One feature of the PT approach is that the classes within the templates form a whole and that the relations and inheritance between the classes are preserved during instantiation.

New types are created whenever a template is instantiated. Each instance is kept separate, and in addition to allowing flexibility in merging and renaming, this can be used to make sure that objects created from different template instances have their own type. This enables a form of family polymorphism [9].

The dynamic loading and composition proposed is not as dynamic as some other systems. Requiring that the loaded template be a sub-template of some bound, the program can be type checked and if the loading itself does not fail, the system will not cause type errors during execution.

Loading single classes, as is usual in Java, means that one can only depend on a single interface with named methods that have parameters that are of statically known and unchangeable types. Loading a complete template not only allows one to view several classes as a whole, but in PT the types of the parameters of methods and variables in the template itself are adapted to new types. This is a useful new feature of package templates and dynamic package templates in particular – which does this composition at runtime.

PT is more flexible in renaming, etc, than is discussed here. There is also more flexibility in using template parameters than what has been discussed. There is work going on looking at merging and other type safe mechanisms for creating instances from multiple dynamically loaded templates and on abstract methods in templates.

We have not discussed how this proposal would work together with the aspect oriented extensions discussed in [3]. A consistent combination of the two extensions should be worked out. For an even more dynamic approach, there is also a study of a similar mechanism for the dynamically typed language Groovy [2].

We are also looking at doing dynamic updates to running code, that is updating already loaded template instance, by for example redirecting calls to new methods. Allowing this in some restricted way, could create even more useful aspect oriented features in the language.

The most important future work is to settle the open questions concerning the rules of the language, prove its type safety and building a compiler.

6. RELATED WORK

The authors of the trait mechanism [22] approach the problem of composition from the angle that the primary unit to be composed is the class. A trait is as such a construct that encloses a stateless[3] collection of provided and required methods. Traits may subsequently be used to compose new traits or as part of a class definition. The composition of traits is then said to be *flattened*. This entails 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. When used to compose a class, all requirements must be satisfied by the final composition. Traits were originally developed for the dynamic language Squeak, and supports method aliasing and exclusion upon composition. A statically typed version also exists [20]. Still, neither the static

[3]Traits were originally defined to be stateless, although a more recent paper [5] has shown how a stateful variant may be designed and formalized.

nor the dynamic version have explicit support for *runtime* selection of which features that should be composed.

Mixins [6] are similar in scope to traits, in that they target the reuse of small units of code. Mixins also define provided and required functionality, and the main difference between them and traits is arguably 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.

Functionality similar to that of traits and mixins can quite easily be mimicked with PT. For instance, to create a reusable collection of methods (with or without accompanying state), one might simply define a template with a single class, consisting of the methods that are subject to reuse. This class may then be merged with other classes where the functionality is needed. When it comes to specifying required methods, PT provides no such concept out-of-the-box, but a solution might be to define abstract and/or virtual methods in the template class. As is the case with traits, merge/composition order is not significant in PT.

Perhaps the biggest conceptual difference between mixins/traits and PT comes in form of intended scope, in the sense that PT is targeted towards reusing and specializing larger chunks of code as one coherent unit. In that regard, the former two can be seen as a special case of what can be accomplished with PT, admittedly with a slightly more involved syntax and some 'glue code'.

Aspect-oriented programming (AOP) [14] involves several concepts related to PT. For instance, intertype declarations in AspectJ [7] may (statically) add new members to existing classes, and may as such be used to compose previously unrelated features. An example of this is exemplified through the Observer design pattern [10] in [11]. However, this implementation, and on a higher level the general approach employed to composition, is arguably less than optimal, given that it suffers from the fact that the aspect itself entangles several conceptual roles within a single aspect, and that this aspect also exists as a unit at runtime, lacking a clear mapping to objects from the problem domain. The Caesar language [1, 19] supports both aspect-oriented programming constructs and code reuse and specialization through the use of virtual classes. It also supports wrappers for defining additional behavior for a class, and dynamic deployment of aspects at runtime (through use of the `deploy` keyword). Dynamically deployed aspects are in effect from all calls propagating down the call stack with respect to the lexical scope of a `deploy` construct. Expanding on the notion of dynamic deployment, Tanter [24] describes a mechanism for controlling the scope of dynamically deployed aspects (including propagation down the call stack and to new objects). Note, however, that these aspects may affect behavior only, and not class structure or hierarchy, as opposed to dynamic instantiations in PT.

Context-oriented programming (COP) [8] provides a way to activate and deactivate *layers* of a class definition at runtime. Layer activation can be nested, and propagate down the call stack (for the current thread).

Like PT, Mixin layers [23] is a mechanism for writing an addition with affect accross multiple entities like classes. Mixin layers can be composed by instantiating a layer with another as its parameter and thus mixin layers are both reusable and interchangeable. They are also nested. However, there does not seem to be a way to build hierarchies withing a mixin layer.

BETA [18, 17], gbeta [9] and J& [21] (pronounced "jet") are systems that in many ways are similar to each other and in many respects can achieve similar end results to those of PT. A common property of all of them (except PT, that is) is that they utilize virtual classes (as introduced by BETA) to enable specialization 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. Neither BETA, gbeta nor J& support concepts similar to runtime template instantiations.

As we now introduce dynamicity and more free-standing template instances, the mechanism we present will become more similar to a solution with virtual classes and family polymorphism, as e.g. in gbeta [9]. However, the rules and restrictions used to keep the system consistent will be different in our version.

In a subject-oriented [13] programming (SOP) system, different subjects may have differing 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. In SOP, a merge is an example of a composition strategy (and there may be many of them), that tells the system how to compose separate subjects with overlapping methods and/or state. Like with mixins and traits, there seems to be a difference in intended scope when comparing SOP with PT; SOP targets a broader scope, with entire (possibly distributed) systems (that may even be written in different languages) being composed. One could, however, picture an extended PT-like mechanism as a basis for an implementation of SOP.

Our approach to typing classes and methods with instance types to keep different instances of a template apart at runtime is based on a clever idea for keeping different implementations of a single API apart at runtime using Java generics and tying the classes of an implementation together using a type parameter. This idea is found in [12].

Ada originally (in 1983, [16]) had no mechanisms supporting object-orientation, but it had a mechanism called generic packages with some of the same aims as PT, in that packages can contain type definitions and that you get a new set of these each time the generic package is instantiated. Generic packages also have type parameters.

In Ada 95 [4] a slightly untraditional mechanism for object-orientation was introduced, and it was 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 virtual classes (at compile-time or at runtime) in the language, and the mechanims for adapt-

ing a package to its use during instantiation are not very advanced.

7. CONCLUSION

We have proposed an extension to the package template mechanism that will allow dynamic loading and instantiations of templates. We have discussed some of the properties of the proposed language. It is in some crucial ways different from other mechanisms that try to solve similar challenges. The practical consequences of these differences need to be worked out.

Further studies need to be done to find out if the language is really useful for dynamic software composition, and details of the language need to be worked out based on a study of what makes most sense in a practical language.

Although our initial work on finding a translation of the mechanism to Java generics suggests that the language is type safe, a proof of this needs to be worked out and a compiler must be built.

8. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In Trans. AOSD I, volume 3880 of LNCS, pages 135–173. Springer Berlin / Heidelberg, 2006.

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

[3] E. W. Axelsen, F. Sørensen, and S. Krogdahl. A reusable observer pattern implementation using package templates. In ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software, pages 37–42, New York, NY, USA, 2009. ACM.

[4] J. Barnes. Programming in Ada95. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[5] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. Computer Languages, Systems & Structures, 34(2-3):83–108, 2008.

[6] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, Proc. Conf. O-O. Prog.: Syst., Lang., and Appl. / Proc. ECOOP, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[7] A. Colyer. AspectJ. In Aspect-Oriented Software Development, pages 123–143. Addison-Wesley, 2005.

[8] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In DLS '05: Proceedings of the 2005 symposium on Dynamic languages, pages 1–10, New York, NY, USA, 2005. ACM.

[9] E. Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

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

[11] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. SIGPLAN Not., 37(11):161–173, 2002.

[12] W. Harrison, D. Lievens, and F. Simeoni. Safer typing of complex api usage through java generics. In PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pages 67–75, New York, NY, USA, 2009. ACM.

[13] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, pages 411–428, New York, NY, USA, 1993. ACM.

[14] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

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

[16] H. Ledgard. Reference Manual for the ADA Programming Language. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[17] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications, pages 397–406, New York, NY, USA, 1989. ACM.

[18] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, 1993.

[19] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In AOSD '03, pages 90–99, New York, 2003. ACM.

[20] O. Nierstrasz, S. Ducasse, S. Reichhart, and N. Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[21] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In OOPSLA '06, pages 21–36, New York, 2006. ACM.

[22] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In ECOOP 2003, volume 2743 of LNCS, pages 327–339. Springer Berlin / Heidelberg, 2003.

[23] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol., 11(2):215–255, 2002.

[24] É. Tanter. Expressive scoping of dynamically-deployed aspects. In AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development, pages 168–179, New York, 2008. ACM.