JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 7, November-December 2009

Exploring the use of Package Templates for flexible re-use of Collections of related Classes

Stein Krogdahl, Department of Informatics, University of Oslo, Norway
Birger Møller-Pedersen, Department of Informatics, University of Oslo, Norway
Fredrik Sørensen, Department of Informatics, University of Oslo, Norway

Abstract

It has been recognized that there is a need for a language mechanism that support re-use of collections of related classes. Existing approaches either use an enclosing class for this purpose, or introduce a special purpose, new language construct. In this paper we explore the use of packages for the grouping of related classes. Ordinary packages are already grouping of classes, so we explore the combination of packages and templates, i.e. package templates. By instantiating package templates, the classes of the package templates are provided as if ordinary packages are produced and imported.

1 INTRODUCTION

Concepts of some complexity are often most naturally implemented by more than one class. A typical example is the concept of a graph, which normally will have classes for nodes and edges. Thus, it has for some time been recognized that there is a need for a language mechanism that support re-use of collections of related classes. And, because we want the collections to be as generally useful as possible, we want a mechanism that allows the classes of a collection to be tailored to specific use situations. Such a mechanism will obviously include some sort of language construct representing the collection of classes.

The following two examples are used to motivate properties of such a mechanism.

- Graph example: The collection contains two classes Node and Edge, with variables to represent a graph structure and methods to navigate in the graph. The edges are kept in a list associated with the node where they start, and the class Node therefore has a method "Edge firstEdge()" and the class Edge has a method

Stein Krogdahl, Birger Møller-Pedersen, Fredrik Sørensen: "Exploring the use of Package Templates for flexible re-use of Collections of related Classes", in *Journal of Object Technology*, vol. 8, no. 7, November - December 2005, pp. 59 - 85 http://www.jot.fm/issues/issue_2009_11/article1/

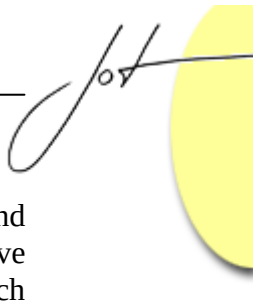
"Edge nextEdge()". In addition, Edge has the methods "Node from( )" and "Node to( )".

A typical use of this collection is to let it form the basis for a data structure that shall represent a phenomenon with a graph-like structure. It could e.g. be a structure of cities and roads, and we would then e.g. add the attribute "name" to Node and "length" to Edge.

- **Compiler example**: This is similar to an example used in [Nystrom et al., 2006]. The collection is here a number of classes that together makes up the front end part of a compiler for a given language. Among these classes we find the hierarchy of the classes for the nodes in the abstract syntax tree. Among these we could e.g. find the most general class TreeNode, with subclasses (or sub-subclasses) representing declarations, statements, expressions, etc. When this collection is used we typically want to add a code generator. This will probably include adding variables in the different classes to keep track of memory addresses etc, and an abstract, virtual method GenCode() in TreeNode that is implemented in all the subclasses.

The following is a list of properties that such a mechanism should have. For each property we first state it for a collection C containing classes A and B (and in the last property also a collection D with a class E), and we then illustrate it with one of the examples above.

- **Parallel extension**: When using the collection C in a certain setting we can add attributes to A and B. These additions should also have effect for the code of C, e.g. so that we by means of an A-variable defined in C can directly (without casting) access the attributes added to A.
- In the graph example, assume that we have added the int variable length to Edge, and that n is a Node-reference. With this property we can conveniently specify directly: "n.firstEdge().length = 5;", as firstEdge is typed with the extended Edge class.
- **Hierarchy preservation**: The mechanism should allow B to be a subclass of A, and if additional attributes are given to A and to B, then the B with additions should be a subclass of the A with additions. Note that this will not be the case if we just use the collection C with the classes A and B and then define subclasses A' and B' to A and B, respectively, with the additions we want in these subclasses. B' will then not be a subclass of A'.
- In the compiler example this is exactly what we need in order to be able to add attributes as explained in the example.
- **Renaming**: When C is used, we should be able to change the name of A and B, and of their attributes, so that they fit with the specific use situation.
- For the graph example, the renaming property makes it possible to rename the nodes and edges to cities and roads.



- **Multiple uses**: It should be possible to use the classes of C for different and independent purposes in the same program, and so that each purpose have different additions and renamings. The compiler should be able to check that each use implies a different set of classes as if they are defined in separate hierarchies.
- In a program we may need the basic graph structure for different purposes. In addition to using it for cities and roads, we could in the same program use it to form the structure of pipes and joints in a water distribution system.
- **Type parameterization**: It should be possible to write a collection of classes that assumes the existence of classes that have some required attributes, but are not yet completely defined. In each use of this collection, one can provide specific classes that have at least the required attributes.
- In the compiler example we assume that the front end shall always produce Java Bytecode, and we use some readymade mechanism for packing the code to a correctly formatted class file. However, a number of such packers may be available, and we do not want to choose which to use while implementing the front end class collection.
- **Class merging**: Assume we have the two collections C (with classes A and B) and D (with class E). When they are both used in the same program, we should be able to merge e.g. the classes A and E so that the resulting class gets the union of the attributes, and so that we via an E-variable defined within D can also directly see the A attributes (and similarly for an A-variable in C).
- Assume that we in addition to the graph collection have a collection Persons with a class Person. In a program handling personal relations we then want to use both collections together so that we obtain a new class, say PersonNode, which has all the attributes of Node and Person, and where a Person-variable p defined in Persons gives access directly to the Node attributes, e.g. “p.firstEdge”.

In addition to these properties, it is important that such collections of classes can be separately type-checked. We also prefer a mechanism that allows only single inheritance, as the merge property described above to a large extent will take care of the need for combining code from different sources (for which purpose multiple inheritance is often used). Finally, the type system should be as simple and intuitive as possible.

In many current proposals for such a mechanism, the collection is also a class (enclosing the classes of the collection), and usually the inner classes are then considered virtual classes [Madsen & Møller-Pedersen, 1989] in some sense. By utilizing subclassing of both the collection class and the inner classes, different varieties of collection-handling mechanisms can be obtained.

Typical representatives for such approaches are Caesar [Mezini & Ostermann, 2003], J&, ([Nystrom et al., 2004], and [Nystrom et al., 2006]). To which extent these approaches satisfy the requirements stated above will be discussed later (section 4). As far as we know, there is no mechanism that fulfills all these requirements.

A drawback with classes with inner virtual classes is that the redefinitions of the virtual classes usually get the same names as the virtual classes. Another problem is that such a mechanism easily implies rather complex type systems.

In this paper we present a proposal for a mechanism that to a large extent fulfills all the properties above. The main idea behind the proposal is to use (Java-like) packages as the collections, and in order to obtain tailorability we combine packages with the notion of templates. Our approach is therefore called *PT* for *Package Templates*.

We shall describe our proposal by first sketching parts of a language with the basic PT elements. This will be done through simple examples, without too many technical details. Then, in Section 3, we look at some more intricate examples, and in Section 4 we will look at some similar languages/mechanisms, and discuss how these relate to our ideas.

2 BASIC PROPERTIES OF PACKAGE TEMPLATES

We shall use the following syntax for ordinary packages:

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

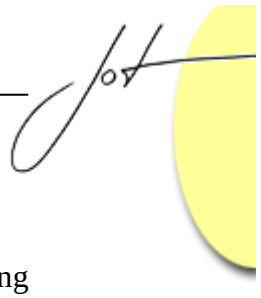
For package templates we shall use the following syntax:

```
template PT <formal type parameters>{
    class C {...}
    class D {...}
}
```

Definitions of package templates have the same form as definition of packages, but we do not get any classes in the traditional sense before the package templates are instantiated. Instantiation is done at compile time. As part of such an instantiation, the classes of the package template become directly available as if the produced ordinary package is imported (in the Java sense). The templates may be instantiated multiple times in the same program. Each instantiation will result in a new set of all the template classes, and these are independent of classes from other instantiations. If a class of a package template has static variables, each instantiation produces a class with its own static variables.

Tailorability is obtained by allowing certain adaptations to be made for each instantiation. The most important of these are:

1. In each instantiation the classes within the package template may be given additions: variables and methods may be added and virtual methods may be redefined.
2. Elements of the package template may be renamed.
3. Formal package template type parameters may be given actual types.



Graph example

To demonstrate the first two of these adaption mechanisms, consider the following package template with Graph concepts:

```
template Graph {
  class Node{
    Edge firstEdge;
    Edge insertEdgeTo(Node to){ ... }
     void display(){ ... }
     ...
   }

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

An instantiation of a template is done as part of a program or as part of another package or template (in general in some scope), and the package produced by the instantiation will implicitly be imported to this scope so that the names of this package are visible throughout the scope.

In the example below the template Graph is instantiated as part of the package RoadAndCityGraph, in order to use objects of class Node for representing cities and objects of class Edge for representing roads. To this purpose, Node is renamed to City, Edge to Road, and both get additional properties:

```
package RoadAndCityGraph{
  inst Graph with Node => City, Edge => Road;
  class City adds{
    String name;
    void someMethod( ){
         ...
         int n = firstEdge.length;  // 1
         City c = ... ;
         Road r = insertEdgeTo(c);  // 2
         ...
    }
    ...
  }
  class Road adds{
   int length;
   void someMethod(){
      ...
      String s = to.name;           // 3
      ...
      int n = nextEdge.length;      // 4
      ...
```

```
    }

    ...
}
```

Note that an instantiation includes the inst-clause itself (with possible renamings) and additional definitions of the (possibly renamed) classes that come from the instantiated package. As indicated by the with-clause, the classes Node and Edge will get new names City and Road, respectively, and all occurrences of the names Node and Edge in the Graph template will be renamed to City and Road, respectively. The classes City and Road will also have additional attributes, which are specified in the definitions of City and Road.

Thus, as part of the instantiation of Graph above, Node and Edge will disappear as types, and the typing in the new package will be as if it was made especially for cities and roads. The statements marked with 1, 2, 3, and 4 above will therefore all be type correct. No objects of the classes Node and Edge will be generated (also "new Node()" is changed to "new City()"), and there will be no variables, parameters or method results typed with these classes.

Note that in statement 2 there will be no co- or contra-variance problems, as the method insertEdgeTo has the following signature within RoadAndCityGraph:

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

Virtual methods defined in a class within a template can be overridden by a method defined in an addition as part of an instantiation. That is, if the addition to class City has a definition of the method insertEdgeTo typed as above, then this will override the method insertEdgeTo in Node.

All class definitions from a template become definitions of classes as part of an instantiation. If not renamed and if no additions are given, then the classes are defined as in the template.

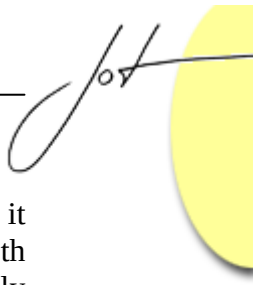
Renaming is not restricted to the names of classes instantiated from a template. We may e.g. change insertEdgeTo to insertRoadTo. The syntax for that is:

```
inst Graph with
  Node => City (insertEdgeTo -> insertRoadTo),
  ...
```

Note that it is only legal to rename declarations made in the template itself (including the templates that it has instantiated, etc.). This excludes names that stem from external interfaces that are implemented by template classes.

Multiple instantiations

We may instantiate the same template more than once in the same scope. We can e.g. use the Graph template for producing the classes City and Road as above, and in the same scope use it e.g. to represent the structure of pipes and connections in a water distribution system.



When instantiating the same template package more than once in the same scope, it is required that instantiated classes are renamed so that the scope will not get classes with the same names. Renaming only Edge from two instantiations of Graph will not imply that there is a common Node class, but rather an error in that the scope will get two classes with the name Node.

Instantiation of a template within a template

A template can be instantiated within another template. When the latter template is instantiated, the contained instantiation is also performed.

As an example of use we can assume that we want the package RoadAndCityGraph to be a template, so that further additions can be made to the classes City and Road. The only change necessary is then to use the keyword **template** instead of **package** when defining RoadAndCityGraph.

Cyclic structures of templates instantiating templates are not allowed.

Hierarchies of classes within a template

We allow ordinary subclass hierarchies to be defined inside a template. As part of an instantiation of the template, all classes in such hierarchies, not only the leaf classes may be given additions. All instantiated classes may also be renamed.

As an example consider the following sketch of a template:

```
template TrafficSimulation {
  class Automobile{
    int speed;
    Automobile next; ...;
  }
  class Car extends Automobile{
    int numOfSeats; ...;
  }
  class Lorry extends Automobile{
    int length;...;
  }
}
```

and a use of it:

```
package BusinessTrafficSimulation {
  inst Traffic with
    Automobile => Vehicle, Car => PrivateCar,
    Lorry => BusinessCar;
  class Vehicle adds{
    string brand; ...;
  }
  class PrivateCar adds{
    int luggageVolume; ...;
  }
  class BusinessCar adds{
    int loadCapacity; ...;
```

```
    }
    ...
}
```

The result of this is roughly the following class hierarchy within BusinessTrafficSimulation:

```
class Vehicle{
  // attributes from Automobile with renamings:
  int speed;
  Vehicle next; ...;
  // additions in Vehicle:
  string brand; ...;
}
class PrivateCar extends Vehicle{
  // attributes from Car with renamings:
  int numOfSeats; ...;
  // additions in PrivateCar:
  int luggageVolume; ...;
}
class BusinessCar extends Vehicle{
  // attributes from Lorry with renamings:
  int length; ...;
  // additions done in BusinessCar:
  int loadCapacity; ...;
}
```

In the above code, all occurrences of the names Automobile, Car, and Lorry within code from the template are renamed to Vehicle, PrivateCar and BusinessCar, respectively. An implication of this is that the 'next' variable typed with Automobile in TrafficSimulation is typed with Vehicle in BusinessTrafficSimulation.

Except for virtual methods, we do not allow declarations with the same name in a template class and in an addition to it. A virtual method defined in Automobile may be redefined within Vehicle. If so, calls to this method (from the code of both Automobile and Car) will be calls to the one defined in Vehicle, provided it is not also redefined in Car. If redefined in Car then the same calls will be to this redefined method. A corresponding rule is used e.g. in J& [Nystrom et al., 2006].

Templates with type parameters

Templates may have type parameters. The following is a sketch of a template defining a linked list where each list will have a list object and a number of list elements, each represented by an object (of class AuxElem) that references the real list element (object of class E). This means that a given E-object can be in any number of lists (and even more than once in the same list).

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

A similar construct can obviously also be made with generic classes (classes with type parameters). The benefit with templates with type parameters is that after an instantiation of ListsOf, e.g. with parameter Person, we need not mention this parameter for each generation of a new List, as we would have to do in case of generic classes. Also, with generic classes, both of the classes List and AuxElem must have a type parameter, and they must get the same actual parameter to make a consistent list. This situation, where the same type parameters apply to a number of classes, calls for defining the type parameters at the package level.

Type parameters can be constrained, and these constraints are specified inside the body of the template. As an example, assume that each object of class E shall keep track of how many times it is a member of some list or another (an element might appear twice in the same list). To implement this we can e.g. require that class E has two methods "void incNo()" and "void decNo()", that are called in the methods insertAsLast and removeFirst, respectively. This can look as follows:

```
template ListsOf<E>{
  constraint E has{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 constrained E by simply listing a number of methods it must implement after the keyword has. One can also constrain a type parameter by listing a number of interfaces it must implement, or a class it must be a subclass of.

By giving the constraints inside the template we can naturally constrain a parameter by a class defined in the template itself, which can be convenient in certain cases.

Obviously, inside the template we are allowed to use any property of a type parameter that follows from the constraints. But also, any use of these parameters that cannot be justified from the constraints constitutes an error.

A type parameter can not be used as a superclass of a class defined in the template. The reason for this will shortly be explained.

Merging template classes during instantiations

It turns out to be very useful to combine instantiations of two or more templates, so that classes in one template can be merged with classes in other templates. In this way classes in independently defined templates can be made to work together, with a typing of the resulting classes that works as if all classes were written especially for the actual case.

This is obtained by allowing two or more classes from different template instantiations to share a common addition class, and thus end up as one class, with the name of the addition class. We shall then say that these template classes are merged. The new class gets the union of all the attributes of the instantiated classes (all with the given renamings), together with the attributes of the addition class. During the instantiations all occurrences of the names of the merged template classes are changed to the name of the shared addition class. The following example illustrates how this mechanism works, and how it can be useful.

As in the example in section 2.1, we now assume that we have a template Graph, and want to use this as a basis for forming classes City and Road. However, instead of adding the extra City and Road attributes in an addition class, we this time assume that we have another template GeographyData with classes CityData and RoadData where these extra attributes are defined:

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

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

```
package RoadAndCityGraph{
  inst Graph
       with Node => City, Edge => Road;
  inst GeographyData
       with CityData => City, RoadData => Road;

  class City adds{...}
  class Road adds{...}

  ...
}
```

The resulting classes City and Road will have the combined properties of Node and CityData, and of Edge and RoadData, respectively, and the classes Node, CityData, Edge, and RoadData will not appear at all in the new context. They are, in both instantiations, replaced by the types City and Road.

The syntax for merging classes from package templates may appear somewhat implicit. The reason for it being this way is that instantiations include inst-clauses and all class additions in the same scope, i.e. in order to find the definition of the instantiated class City above, one shall consider all inst-clauses with renamings to City and a City class addition.

Avoiding indirect multiple inheritance

As discussed in the introduction, we want a language construct that does not imply multiple inheritance. However, because of the merging feature discussed above, we can indirectly get multiple superclasses to a class, even if we do not write it explicitly. As an example, assume that the template classes Node and CityData in the example above had superclasses A and B, respectively. Then the addition class City will end up with the two superclasses A and B. We want to avoid this, and we therefore have to make some restrictions.

The first restriction is simply that we forbid template classes to have superclasses defined outside the template. Thus, as mentioned above, we also have to forbid template classes to have type parameters (classes) as superclasses. This may seem drastic, but one should remember that template classes can still freely implement interfaces defined outside the template, and this will to a large extent make up for the inconvenience introduced by the above superclass rule.

However, we do not want such a drastic restriction inside templates, as this would disallow hierarchies of template classes, which we consider very valuable. Thus, we here introduce the following rule:

If, in a set of instantiations, two or more template classes are merged, then the superclasses they have (which, if they exist, are also template classes) must also be merged in the same instantiations.

We also have to add the requirement that a merge must not result in a cyclic superclass-structure.

Merging and constructors

Assume that two or more classes from different instantiations are merged, e.g. as in the following situation:

```
template T {
  class A{...;
          A(C c, D d){...init. code ...}
          ...;
  }
}
template U {
  class B{...;
          B(F f){...init. code ...}
          ...;
  }
}
```

T and U are instantiated, and the classes A and B are merged under the name X, as follows:

```
package P {
  inst T with A => X;
  inst U with B => X;

  class X adds{ ...see below ... }
}
```

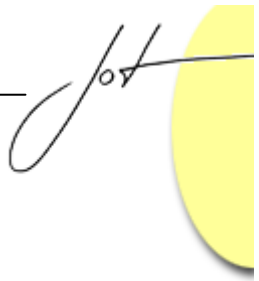
Here, class X has to get constructors corresponding to both the ones given in A and the ones given in B, but this is obviously difficult to do automatically. Thus, we should probably in the merging case require that all necessary constructors must be given explicitly in the addition X. However, inside these new constructors we must be able to call the constructors of A and B. The ordinary super will be ambiguous, in addition, the names A and B do not exist after the instantiations, as they are both replaced by X wherever they occurred.

The solution is to allow the names A and B to be used in the addition X, but only in very special situations: super[A] and super[B]. The necessary constructors of X can then be written as follows:

```
class X adds{
  ...;
  X(C c, D d){
    ...;
    super[A](c1, d1);
    super[B](f1); ...;
  }
  X(F f){
    ...;
    super[B](f2); ...;
    super[A](c2, d2); ...;
  }
  ...;
}
```

Merging and virtual methods

We get a similar problem as above when virtual methods with the same name and parameter types are defined in more than one of a number of merged classes. We therefore require that in such situations the virtual method must be redefined in the addition class. We then need a way to call the versions in the merged template classes and (in the same style as above) we can do this as follows (where “vm” is the name of the virtual method):



```
package P {
 inst T with A => X; //A has virtual int vm(c,d)
 inst U with B => X; //B has virtual int vm(c,d)

 class X adds{
    void vm(C c, D d){...;
        int a = super[A].vm(c1, d1);
        int b = super[B].vm(c2, d2)); ...;
    }
 }
}
```

Final comments on basic PT mechanisms

In this subsection we shall look briefly at some aspects of PT that have not been discussed above, but that need to be taken care of in a final version.

Visibility

PT obviously needs a way of specifying the visibility of names, but we currently have not worked out the details. However, we envision something like the following:

Within a template we should at least allow the same visibility modifiers as in a normal Java package: protected, private, public, and the default package. However, we also have to decide which attributes of a template class that are visible from inside an addition class. Do we need a new modifier?

One could also have public and private instantiations, which will determine whether the public attributes of the template classes will be considered public or private relative to the package/template in which the instantiation occurs. The names of the template classes themselves are replaced by the names of addition classes, and can thereby be regulated by modifiers on the addition classes.

Templates not at the outermost level

Until now, we have only considered templates that are defined and instantiated at the outermost level of a program. They are all instantiated before the execution starts, and this makes things reasonably simple. This solution may in fact be good enough, but one may also consider templates at inner levels, e.g. inside classes, methods or other templates. This is, however, a theme for further research.

Interfaces

We have so far not said very much about interfaces, but we should obviously also be allowed to define interfaces inside templates, and these interfaces may depend on the type parameters.

One should note that being able to rename attributes of classes and of interfaces independently can make inconsistent situations. Probably a restriction on the renaming mechanism is necessary.

3 EXAMPLES

Above we have sketched the basic mechanisms of PT, and we have seen a few simple examples. In this section we shall look at some more involved examples so that the reader can get a more realistic impression of the mechanism, and what it can accomplish.

The expression problem

We first look at the well known expression problem (see e.g. Torgersen 2004] and [Nystrom et al., 2006]). The starting point is that we have given a hierarchy of classes representing the nodes in an abstract syntax tree for simple expressions, e.g. number as the only kind of operand and with addition as the only kind of operation. We then want to add a new operation (multiplication) to the expression language, and we want to add the capability of printing an expression in prefix order. The challenge is to make a new hierarchy of classes based upon the class hierarchy for simple expressions, without changing anything in the text describing existing classes (non-invasiveness), and by only adding what is needed for the new kind of expressions. In this case we only need to add a new expression node class for handling multiplication, and we need to add a method preOut to all expression node classes.

To get this effect the basic node class hierarchy must be given in a template, and we assume that the simple version of the tree node hierarchy is as follows (SExpression for Simple Expression):

```
template SExpression{
  class SExp{
    int eval();
  }

  class SAdd extends SExp{
    SExp e1, e2;
    int eval(){return e1.eval()+e2.eval();}
  }

  class SNumber extends SExp{
    int value;
    int eval(){return value;}
  }
}
```

We can add a multiplication node and a method preOut() in all the classes as follows:

```
template Expression{
  inst SExpression with SExp    => Exp,
                        SAdd    => Add,
                        SNumber => Number;
  class Exp adds{void preout();}
  class Add adds{
    void preOut(){
      print('+'); e1.preout(); e2.preout();
    }
  }
  class Number adds{
```

```
    void preout(){print(''+value);}
  }
  class Multiply extends Exp{
    Exp e1, e2;
    int eval(){return e1.eval()*e2.eval();}
    void preout(){
      print('*'); e1.preout(); e2.preout();
    }
  }
}
```

Note that we for the purpose of illustration have used different names, but we could just as well have named the classes Exp, Add and Number in both templates. In that case there would have been no with-clause after 'inst SExpression'. The names of the templates must, however, still be different.

Now, instantiating Expression instead of SExpression in our program will give us the extended hierarchy that we want. If instantiated with renamings, the resulting code will be as if it was made specifically with the use of the names Exp, Add, Number, and Multiply. If there is no renaming, no changes are necessary in the rest of the program, and we can now use the new class Multiply and the method preout.

Extendable compiler

Assume that we for a given programming language already have a front end compiler written as a template, and within this a class hierarchy is defined for the abstract syntax tree nodes. We then want to extend these node classes so that they fit the following adjustments: One is to add a new constructs to the language (e.g. a repeat-until statement), and the other is to add a back-end that produces code for a certain machine. The first extension will usually require a new node class for the syntax tree, and probably some new methods in a recursive descent parser, while the second can easily be accommodated by adding a code generating method and usually some variables (for e.g. memory addresses) to the node objects.

Thus, as the front end compiler is written as a template, this problem is very similar to the expression problem discussed above and can thus be solved in the same way. A similar example is also discussed in [Nystrom et al., 2006].

F-bounded type parameters

F-bounded type parameters are a well known mechanism for constraining the type parameters to generic classes, and are typically used as follows:

```
class GC<P extends GC<P>>{
  P binOp(P p){
    // a binary operation that
    // computes the answer of '(this P) binOp p'
  }
}

GC can now be used as follows:
```

```
class A extends GC<A>{...}
  // in some method:
  A x, y, z;
  ...
  x = y.binOp(z);
  ...
```

The important point here is that we want to constrain P so that the only legal parameters to the binary operation are objects of the class that GC is instantiated to. In the above case this is A. We obtain this by using the parameter name P in the constraint of P itself.

It turns out that we very easily can get the above effect in PT, in a way that seems simpler to grasp. We define GC in a template as follows:

```
template T{
  class GC{
     GC binOp(GC p){...}
  }
}

To obtain what we want we use this template as follows:
package Q{
  inst T with GC => A;
  class A adds{...}

  // in some method:
  A x, y, z;
  ...
  x = y.binOp(z);
}
```

The parameter to binOp must now be an A-object. Thus, at least for cases like the one above, the sometimes rather mind-boggling construct of F-bounded type parameters turns out to be a quite natural concept in PT.

Linked lists

We shall here look at some examples using another version of a linked list than we did earlier in this paper. This time there is no AuxElem class, and the elements themselves have a reference variable “Elem nextElem” and a method “Elem next()”, so that we for an element “e” may specify “e.next()”. With this approach user-defined elements can only reside in one list at a time. A template for such lists could be like this:

```
template LinkedList{
  class List{
    Elem first, last;
    void addLast(Elem e){...}
    Elem removeFirst(){...}
  }
  class Elem{
    Elem nextElem;
    Elem next(){ return nextElem;}
```

```
  }
}
```

This template can now be used to create a number of different kinds of lists by instantiating it with different addition classes for Elem, with the extra attributes needed for each type of element. The compiler will then check that each list only receives objects of the right type. Also the class List can be given additional attributes, e.g. the name of the set of elements in that list.

For another similar use of this LinkedList template, we assume that we have a second template ED, with a class ElemData containing the data and methods that the elements need. We can then use merging to get the type of elements we want:

```
package Program{
  inst ED with ElemData => Element;
  inst LinkedLists with Elem => Element;
  class Element{ ... anything more? ... }
}
```

Now the list elements have all the attributes of the ElemData class, which is what we wanted. With an ElemData variable vED defined in ED we can have "vED.next()" without casting.

In our third example, we shall see how we can define a class whose objects can be member of two given list types at the same time, and still be able to access the next element in each list. This can be obtained as follows:

```
package Program{
  inst LinkedLists with
    List => Club (first-> Cfirst, last-> Clast),
    Elem => Person (next-> nextInClub);
  class Club adds{String streetAddress;...;}
  inst LinkedLists with
    List => Family,
    Elem => Person;
   class Family adds{String familyName;...:}
      // No renaming here
   ...
   class Person adds{int age; String name;}
}
```

Here an object of class Person can be an element of both a Club and a Family. Inside Person we can now have nextInClub() which will give you the next in the person's Club, and next(), which will give you the next in the person's family. Person objects will have two next-pointers, one for each list-type. The list objects (of classes Club and Family) can be administrated independently. Note that we do renaming in one of the instantiations to avoid name collisions.

As yet another example we look at how we could use the LinkedList template as a basis for forming the list of edges of a node in a Graph template similar to the one we have discussed earlier. The Graph template could then look as follows:

```
template Graph{
   inst LinkedLists with
     List => Node (next -> NextEdge),
     Elem => Edge (first -> firstEdge);
   class Node adds{...More attr. in Node?...}
   class Edge adds{ Node from, to; ...}
}
```

Finally, we look at how we can instantiate the LinkedList template twice so that we get lists of lists. We then use merging and renaming as follows:

```
package Program{
  inst LinkedLists
    with List => ListOfLists, Elem => ElemList;
  class ListOfLists adds{...}
  class Elem adds{...}

  inst LinkedLists
    with List => ElemList, Elem => Person;
  class Person adds{...}
}
```

Here an object of class ElemList is both an element in a ListOfLists and in a list of persons. This structure could e.g. be used to store sparse matrices.

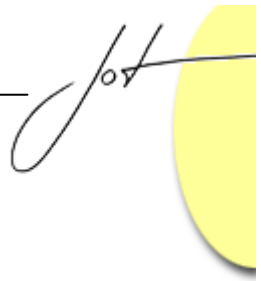
Implementing mixins and traits

A mixin [Bracha & Cook, 1990] is a class-like construct that contains functionality that other classes might want to include. It is most useful in languages with single inheritance; otherwise multiple inheritance can cover at least some of the same needs. It should be possible to write mixins so that they can be used in as many different situations as possible where their functionality is wanted.

The situation could typically be that we want to define a class B as a subclass of A, but at the same time want to add the functionality of a mixin M to B. In general a mixin will have a list of attributes that it requires from the superclass of B, in this case A. However, the superclass can be any class that has these attributes.

The simplest version of a mixin is one where it requires nothing from the class above it, which means that the mixin only contains new and self-contained functionality. This version can easily be implemented in PT, by straight-forward use of merging. Assume the classes A and M are defined in the templates TA and TM, respectively, and we want to make a template (or package) TB with a class B as described above.

In PT this can mean two different things: In the template TB we may either merge the classes A and M to one class, with B as the addition class, or we can let A become an ordinary class, which becomes the superclass of M with B as an addition class. The two alternatives would be like this:



```
template TB
  inst TA with A => B;
  inst TM with M => B;
  class B adds {...}
}
template TB{
  inst TA with A => AA;
  inst TM with M => B;
  class B extends AA adds {...}
}
```

For clarity we have renamed A to AA in the second alternative.

Note here that it is OK to make B a subclass of AA, as AA is a template class within TB, and M has no superclass.

Now, let us look at the more complex (and normal) case where M specify some attributes that it requires from the superclass A of the class with the mixin. If parameter classes could legally be used as superclasses for classes in the template, this would have been a good mechanism for implementing this, but, alas, we have forbidden this in PT for reasons that have been explained above.

However, virtual methods provide a solution, and below we sketch what corresponds to the latter case above (where A remains a separate class, and is renamed to AA). In TM we also declare a superclass SM of M, and in this we declare (as virtual methods) the methods that M requires from A. As an example, assume that M requires "A reqFromA(A a)" from A. TM will then be like:

```
template TM {
  class SM{SM reqFromSM(SM s);}
  class M extends SM{
    ...; SM sm = reqFromSM(...); ...;
  }
}
```

Note that SM is used to type various elements. If class A in template TA has the required method, then template TB can be as follows:

```
template TB {
  inst TA with A => AA;
  inst TM with M => B, SM => AA;
  class AA adds{
    AA reqFromSM(AA s){return reqFromA(s);}
  }
  class B adds{...} // Is a subclass of AA
}
```

Here A and SM are merged to AA, and in AA we simply redefine the virtual method reqFromSM in SM so that it becomes a call to reqFromA in A (which is possible, since after the merge both result and parameters of these methods are typed with AA). The reason why B becomes a subclass of AA is that M is a subclass of SM.

If we want to merge both A and M to one class B (as in the former solution for the simple case above), things are much simpler, as we don't need the class SM in TM.

The notion of Traits was first proposed in [Curry et al., 1982] and later[Schärli et al., 2003] introduced as a lightweight mechanism for composing classes. A trait is simply a set of methods. Composing a class from traits imply that the class gets the methods of the traits. In addition to the methods that a trait provides, a trait will also have a list of methods that it requires from one of the other traits in a composition, or from the class itself. Traits can be composed to form larger traits, and then not all the required methods need to be provided by other traits in the composition, but will instead be included in the required list of the new larger trait. However, when traits are composed to form a class, then all requirements must be satisfied. Originally traits did not have variables, but in later versions this possibility is included.

With the above implementation of mixins in mind, traits (with variables) are very simple to implement. We simply define a template with one class inside it, and the methods and variables of this class are the methods and variables of the trait. When the trait wants to refer to the class it will be composed into, it simply refers to the name of the class in the template. The set of required methods is given as virtual methods. If a trait wants to access required variables in other traits, this must be done by defining virtual get -and set methods.

To use such traits we instantiate the set of trait templates we want to compose, and merge all the classes in these templates to one class. The corresponding addition class is given the name we want for the composed class or trait, and in this we write simple glue-code to connect the required with the provided methods, exactly as for mixins. We can here also add methods and variables that are not defined in any of the trait templates.

4 RELATED WORK

In the introduction we put up the following list of desired properties of our language mechanism:

- Parallel extension
- Hierarchy preservation
- Renaming
- Multiple uses
- Type parameterization
- Class merging

One may always discuss to which degree a language mechanism fulfills a set of properties set up for it. However, from the preceding sections it seems fair to say that PT to a large degree fulfills the above requirements. In this section we shall look at other mechanisms and languages that aim at fulfilling similar sets of requirements, and compare these with PT. Many of these are built on some kind of virtual classes. This section is based on earlier work reported in [Sørensen & Krogdahl, 2007].

BETA [Madsen et al., 1993] is the language that pioneered virtual classes [Madsen & Møller-Pedersen, 1989]. BETA allows nested classes to any depth, and objects of inner classes are always local to an object of the class enclosing them. Also, objects that are local in different objects of the enclosing class are of different types.

BETA can be used for building PT-like templates in the following way: We use an enclosing class to represent the template itself, and inner virtual classes to represent the classes of the template.

We can then obtain an instantiation with extensions by first defining a subclass S of the enclosing class E where we extend the virtual classes, and then generate an object of the class S. In this object, the extended versions of the virtual classes will (according to the semantics of virtual classes) have effect also in the code stemming from the enclosing class E. Thus, programming within this class S we get many of the possibilities offered by PT. However: We are not able to change names of the inner classes during the extension, and we cannot get something similar to merging, as BETA has no form of multiple inheritance.

It is possible in BETA to obtain more than one instantiation of the same template. This can be done simply by defining multiple subclasses of the original enclosing class, and generate an object of each of these. The inner classes from these instantiations will be kept strictly apart by the compiler, as they are different types. The inner classes of these objects can be accessed by remote access into an object of the outer class, or directly from inside the enclosing class.

As in many appraoches based on virtual classes, it is in BETA sometimes not possible to statically decide whether certain constructs involving virtual classes will be correctly typed at run time. This necessitates run-time type tests in certain situations.

BETA does not allow virtual superclasses, and we are therefore not able to describe a hierarchy of extendable classes within a container class. However, one may have two unrelated virtual inner classes, and these can be extended in parallel in a subclass of the enclosing class.

BETA (and also gbeta, see next subsection) has two ways of declaring virtual classes. One is the traditional kind where you define a virtual class in an enclosing class, and can add more attributes to it in a subclass of the enclosing class.

The other version is to declare a virtual class simply as a name, say A, inside an enclosing class, and in the same construct bind this name to some other class U defined independently of (but visible from) the declaration of the name A. Then, in a subclass of the outer class, we can rebind the name A to another class V, the only restriction being that V must be a subclass of U.

This mechanism can be used to obtain a restricted form of type parameters of the enclosing class (corresponding to type parameters of templates in PT). In the scheme above, A will represent the formal type parameter, and U will represent the constraint of this parameter. Then the actual parameter V can be provided by defining a subclass of the outer one, and rebinding A to V. Note that the constraint on A has to be a subclass-constraint, meaning that the constraint class U must be visible from the definition of V as

well as from the container class that has U as a type parameter. This may in some situations be less flexible than we would like.

Summing up, BETA has the following relation to the list of desired properties:

- Parallel extension: Yes, but only for classes that are not in sub- or superclass relation
- Hierarchy preservation: As virtual superclasses are forbidden, no hierarchies can be built.
- Renaming: Not possible, as we cannot change names of virtual classes when they are extended.
- Multiple uses: Possible by using objects of different subclasses of the outer (container) class.
- Type parameterization: Yes, but only with subclass constraints [Madsen & Møller-Pedersen, 1989].
- Class merging: Not possible, as BETA does not have multiple inheritance.

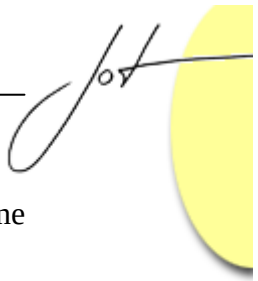
gbeta [Ernst, 1999] is a generalized version of BETA, which also allows multiple inheritance and virtual superclasses, but which (at least in its strict form) also has some restrictions to the type system relative to BETA.

As gbeta allows virtual superclasses one may define hierarchies of virtual inner classes in an enclosing class. This is like defining hierarchies of template classes in PT. In gbeta, when these hierarchy classes are extended in parallel in a subclass of the enclosing class, we get the same effect as in PT: Each class is extended as indicated, and the hierarchy is preserved. By using this, it is straight-forward to write a solution to the expression problem in gbeta, similar to the one given in PT in the previous section.

However, gbeta has a feature that PT currently lacks. As an example we can again look at the Expression Problem. Assume that we in gbeta has a container class Expression that contains the virtual class Exp, and its virtual subclasses Add and Number. As possible extensions we make two subclasses of the Expression class: one that adds a class Multiply, and one that adds the preOut method to existing classes. In an application we may want one or both of these extensions. If we want both, we can use multiple inheritance at the container level to combine the extensions, and the only code we have to add is a description of how the method preOut should work for the Multiply class.

It is here important for the functionality that the expression classes Exp, Add and Number will exist in only one version after the merge. And this is in fact what we get in gbeta, as the gbeta multiple inheritance mechanism only gives one shared version of a superclass visible along multiple inheritence paths.

PT, on the other hand, would in the corresponding situation give two separate versions of these classes, as the two extensions would each need a template of their own, and both of these would have to be instantiated in the application program. Even if this is not what we want for the expression problem, we claim that in many similar situations we would rather want the PT solution, and gbeta does not seem to provide this possibility. A



typical situation where the PT solution is wanted is one where an application needs one list of cars and one of persons, and you want to use the same list-template for both.

Thus, ideally one should have both. In fact, we currently consider introducing a gbeta-like mechanism in PT, and as far as we have looked at it, this seems straight-forward to do.

If we look at the earlier example where Cities and Roads are made from a composition of the classes Node and Edge on the one hand and the classes CityData and RoadData on the other (section 2.6), we observe that this can also easily be done in gbeta, and in this case the PT and gbeta solutions will come out the same way, as there are no common parts/superclasses of the classes that are merged.

In the same way as in BETA, we can in gbeta have multiple independent uses of a container class, each with different extensions. However, to avoid run time tests, (strict) gbeta requires that these instantiations must be represented by objects (of the container class) that to a large extent is known by the compiler, and this restricts the dynamic flexibility of instantiations so that it is more like the one in PT. Type parameters can be introduced in gbeta exactly as in BETA, with the same restrictions.

Using virtual classes and multiple inheritance, one can also have mixin-style solutions in gbeta. However, traits are not that easy to imitate, as the order of the superclasses in multiple inheritance is significant in gbeta. We cannot easily get the flattening property required for traits. We have also tried to obtain something like the list of lists example given for PT in the previous section, but have not been able to produce that in gbeta.

In gbeta we cannot rename classes and attributes during an instantiation (that is, in a further binding of a virtual class). However, gbeta can define alias names, so that classes and attributes can be given additional names.

Thus, summing up relative to our list of desired properties, gbeta can in fact do most of things that PT can do. However, certain important choices concerning superclasses and merging are solved differently in gbeta and PT, and we cannot in gbeta do fully controlled name changes during instantiations (which in general is difficult in approaches based on virtual classes). Also, the constraints of type parameters has the same limitations as in BETA.

In [Ernst, 2001] Ernst discusses the concept of *Family Polymorphism* which is based on concepts in the gbeta language. He here stresses the importance of being able to dynamically produce an unlimited number of *class families* (corresponding to instantiations in PT) for the language to support reuse in a proper way. While this is possible in gbeta, it is not possible in PT. However, we claim that in practical programming this is not so important as Ernst indicates.

J& [Nystrom et al., 2004] (pronounced ‘jet’) is a Java-based language that in many ways is conceptually similar to gbeta, with virtual classes and multiple inheritance, however with a different syntax. Thus, parallel extensions and multiple uses can be obtained in much the same way as in gbeta. However, there are also some crucial conceptual differences between J& and gbeta.

First, inner (or nested) classes are local to the class that encloses them, not to an object of this class. Thus, they work much like static inner classes in Java, and can thereby not access object-variables of the class enclosing them, which might be inconvenient.

Secondly, it is in J& possible to refer to both the current binding of a virtual type and to a previous binding, and one may thereby see objects of (and variables typed by) these two classes simultaneously. This might be useful when old and new code has to run in parallel e.g. during maintenance, but it gives a rather complex type system, with concepts like prefix types, dependent classes, and final paths.

Thirdly, J& does not have the special feature of virtual classes just being names that can be bound to independently defined classes. Thus, one can not with J& implement even the restricted form of type parameters that BETA and gbeta can.

J& has multiple inheritance, and like in gbeta, the properties of a common superclass are shared. One will then always merge classes that have a common origin in common container classes, but it is impossible to merge independent classes from different container classes. For example one can write a solution to the expression example with multiple extensions as described for gbeta and as in [Nystrom et al., 2006]. However, it does not work in situations similar to what has been shown for Node, Edge, CityData, and RoadData above.

In summary J& supports most of the same things as gbeta, but unlike PT it does not have type parameters and renaming.

CaesarJ [Aracic et al., 2006] is a language with aspect oriented programming capabilities, but it also has a mechanism for reuse and extension of code based on virtual classes. This mechanism is something in between the one in gbeta and in J&. It is similar to the one in gbeta in that inner classes are local to objects of the outer class, not to the outer class itself (as in J&). Also, CaesarJ (like gbeta) uses linearization of the superclasses, giving it a mixin capability. However, contrary to gbeta, it cannot easily simulate even a restricted form of type parameters, and the merging mechanism is very similar to the one in J&. You can e.g. not merge fully independent classes, but you will automatically get a merge of further bindings along different paths of a common virtual class. CaesarJ has the same type of restrictions for avoiding run-time tests as J& has.

Mixins and Traits ([Bracha & Cook, 1990], [Curry et al., 1982], [Schärli et al., 2003]). In section 3 we have seen that PT to a very large extent can write templates that simulates mixins and traits. Thus, they are in a sense special cases of what PT can do. The main disadvantage of the PT solution is that a little glue code is needed to combine the parts to form a class (or a larger trait). However, this code is straight-forward to write.

UML2 [UML, 2009] has a mechanism called package merge. It was introduced in UML2 in the first place because it was needed for the definition of the UML2 meta model. This meta model is divided into packages, partly reflecting the different kinds of models (class models, interaction models, state machine models etc.), and partly reflecting compliance levels. Different packages of the meta model contain partial

definitions of some of the meta classes, and when packages are merged, the result is a set of meta classes where the properties from the merged packages are merged.

Package merge of UML2 preserves specialization hierarchies and supports the addition of properties to every class. More than one package can be merged with a single package. There is no mechanism for renaming, in fact the whole idea is that only classes with the same name are merged. When properties of merged classes have the same names, detailed rules covern when the properties are merged and when a merge will not produce a valid merged package.

The package merge was inspired by the notion of package extension in Catalysis [D'Souza & Wills, 1999]. As a package in UML is not a classifier and therefore cannot be extended, and as package extension in Catalysis really was a merge, UML2 went for a pure package merge.

5 CONCLUSION AND FUTURE WORK

We have explored to which extent a combination of packages and templates can cover the need for flexible re-use of collections of related classes, and we have presented this idea by a number of examples. We have demonstrated that package templates have indeed most of the properties that we would like such a collection mechanism to have, and we have compared the idea with other similar approaches.

The main benefit of a template-based mechanism like Package Templates is that it is based upon substitution and thereby supports adaption to specific situations, e.g by renamings. Contrary to templates in C++, Package Templates are based upon semantic substitution: Package Templates can be type checked, and the only requirement to Package Templates is essentially that programs coming from substitutions are type-correct according to the type rules of the underlying language.

Future work on package templates includes detailed rules for name collisions when independent classes are merged. Visibility rules also have to be detailed.

As described above, J& has a mechanism whereby one can write a basic class collection and a number of extensions of this collection. One can then choose a subset of these extensions and make a new class collection that has the properties of the basic one, with the additions of the properties of the chosen subset. We would also like to include a similar mechanism in PT.

Most importantly we will have to try out the mechanism in practice, and for this we need an implementation. The work on this is well under way.

6 ACKNOWLEDGEMENT

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). We would like to thank Eyvind W. Axelsen for many constructive discussions.

REFERENCES

[Aracic et al., 2006] Aracic, I., Gasiunas ,V., Mezini, M., Ostermann, K.: "Overview of CaesarJ." *Transactions on Aspect-Oriented Software Development* LNCS, Vol. 3880. 2006.

[Bracha & Cook, 1990] Bracha, G., Cook, W.: "Mixin-based inheritance". *OOPSLA/ECOOP'90: European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, Ottawa, Canada*, ACM Press, 1990.

[Curry et al., 1982] Curry, G., Baer, L., Lipkie, D., Lee, B.: "Traits: An approach to multiple-inheritance subclassing." *ACM SIGOA Newsletter, 3(1-2)*, 1982.

[D'Souza & Wills, 1999] D'Souza, D. F., Wills, A. C.: "Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach ", Addison-Wesley, 1999.

[Ernst, 1999] Ernst, E.: gbeta -- a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. Department of Computer Science, University of Aarhus, Denmark, 1999.

[Ernst, 2001]. Ernst, E.: Family Polymorphism. *ECOOP'2001- 15th European Conference on Object-Oriented Programming. Budapest, Hungary*. Springer LNCS 2072, 2001.

[Madsen & Møller-Pedersen, 1989] Madsen, O. L., Møller-Pedersen, B. : "Virtual Classes - A Powerful Mechanism in Object-Oriented Programming". *OOPSLA'89 - Object-Oriented Programming, Systems Languages and Applications, New Orleans, Louisiana*, ACM Press, 1989.

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

[Mezini & Ostermann, 2003] Mezini, M., Ostermann, K.: "Conquering Aspects with Caesar". *AOSD'03, Boston, USA*, 2003.

[Nystrom et al., 2004] Nystrom, N., Chong, S., Myers, A. C.: "Scalable Extensibility via Nested Inheritance". *OOPSLA'04- Object-Oriented Programming, Systems Languages and Applications, Vancouver, British Columbia, Canada*, ACM SIGPLAN Notices, 2004.

[Nystrom et al., 2006] Nystrom, N., Qi, X., Myers, A. C.: "J&: nested intersection for scalable software composition". *OOPSLA 2006 - Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, 2006.

[Schärli et al., 2003] Schärli, N., Ducasse, S., Nierstrasz, O., Black, A. P.: "Traits: Composable Units of Behaviour". *ECOOP'2003 - European Conference on Object-Oriented Programming*, LNCS 2743, Springer Verlag, 2003.

[Sørensen &Krogdahl, 2007] Sørensen, F., S. Krogdahl: "Generic packages with expandable classes compared with similar approaches". *Norsk informatikkonferanse (Norwegian Informatics Conference), Oslo*, Tapir Akademisk Forlag, 2007.

[Torgersen, 2004]. Torgersen, M.: "The expression problem revisited". *ECOOP 2004 - 18th European Conference on Object-Oriented Programming, Oslo, Norway*, Springer.

[UML, 2009] Unified Modeling Language 2.1.2, http://www.omg.org/spec/UML/2.1.2/.

About the authors

Stein Krogdahl, Professor at the Department of Informatics, University of Oslo. He can be reached by e-mail: steinkr@ifi.uio.no

Birger Møller-Pedersen, Professor at the Department of Informatics, University of Oslo. He can be reached by e-mail: birger@ifi.uio.no

Fredrik Sørensen, PhD research fellow in the project SWAT at the Department of Informatics, University of Oslo. He can be reached by e-mail: fredrso@student.matnat.uio.no