Groovy Package Templates

Supporting Reuse and Runtime Adaption of Class Hierarchies

Eyvind W. Axelsen Stein Krogdahl
University of Oslo, Department of Informatics
Postboks 1080 Blindern, 0316 Oslo, Norway
{eyvinda, steinkr}@ifi.uio.no

Abstract

We show how package templates, a modularization mechanism originally developed for statically typed languages like Java and C#, can be applied to and implemented in a dynamic language like Groovy, by using the language's capabilities for meta-programming. We then consider a set of examples and discuss dynamic PT from the viewpoints of code modularization and reuse, and dynamic adaption of classes at runtime.

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, Dynamic Languages, Templates

1. Introduction

The need for flexible mechanisms for code organization, modularization and reuse is well-known in the domain of software development. Package templates (PT) have been introduced [22] as one approach to this for statically typed languages like C# [8] and Java [14]. Given the promising preliminary results of applying this mechanism to example problems in Java [33, 2], it is an interesting endeavor to examine how this mechanism fits with other languages and paradigms. In this paper, we will examine package templates applied to the dynamic language Groovy [16].

The dynamic nature of the Groovy language opens up several new possibilities for making use of package templates. It also provides an excellent vehicle for studying and experimenting with mechanisms for structuring and reuse of code beyond traditional classes and packages.

In keeping with its dynamic nature, Groovy supports certain modifications of existing classes at runtime. This is a feature that (to varying degrees) is also supported by several other dynamic languages, such as Smalltalk [13], Ruby [30], Python [29], etc. Several different mechanisms may be employed to enable such modifications, such as open classes [5], meta-programming, runtime mixins [4] etc. These mechanisms may be used to achieve things that are difficult or even impossible to accomplish with a static approach. However, with such power comes a set of pitfalls, that in certain scenarios may be significant. We shall explore how dynamic instantiations of package templates may be employed to combine a powerful modularization technique with expressive runtime constructs, and discuss how this may alleviate some of the problems that the other approaches come with.

The Groovy language is in itself an interesting and compelling platform for programming language experimentation from several points of view, and while we evaluated several other languages, its unique set of features contributed heavily to our selecting it as the target language for prototyping our mechanism. To begin with, it compiles to byte code and runs on any standard Java Virtual Machine [23]. This means that the availability of libraries is at least as good as for Java, right out of the box. Also, the syntax, which to a certain extent follows that of Java, should be pleasingly familiar to many developers.

Furthermore, Groovy supports somewhat advanced concepts, such as the Meta-Object Protocol (MOP) [20] and abstract syntax tree (AST) transformations, that are not found in the majority of other languages. We will make heavy use of these features when adding support for package templates to Groovy.

The contributions of this paper are:

- We show how package templates can be beneficial to dynamic languages, and how the nature of such languages can be exploited to make package templates even more flexible. Furthermore, we show how runtime adaption of classes can be done in a comparatively clean and safe manner with PT. We demonstrate the utility of these concepts by a set of examples.
- We describe a prototype/proof-of-concept implementation of package templates for Groovy, supplied in the form of pluggable libraries that require no change to the source code of neither the Groovy libraries nor the compiler. This allows interested parties to modify the semantics of our implementation with relative ease, since it is entirely built from standard Groovy classes.

The remainder of this article is structured as follows: In Section 2, we briefly describe the main concepts of PT, and the distinguishing features of the Groovy language. Section 3 describes the main concepts of dynamic PT, while a set of

DLS'09, October 26, 2009, Orlando, Florida, USA.

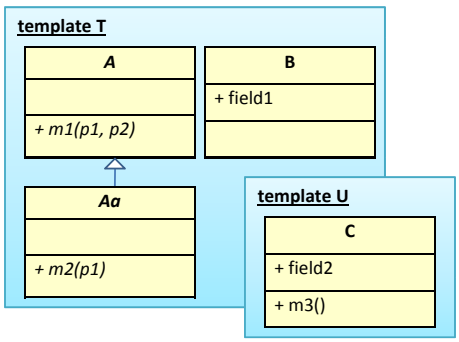

Figure 1. Two example templates, `T` and `U`.

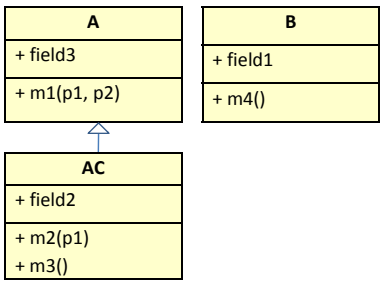

Figure 2. The resulting classes from the instantiation of templates `T` and `U`, with `Aa` and `C` merged. Members have been added to `A` and `B`.

examples is presented in Section 4. The implementation of the mechanism is briefly discussed in Section 5, and Section 6 deals with related work. Section 7 concludes this article, and touches on possible ideas for future work.

2. Background

2.1 Package Templates at a Glance

The package template (PT) mechanism targets the development of reusable collections of interdependent classes. A main idea is that the template mechanism should be flexible enough to allow e.g. framework developers to write quite general templates, and then to provide the application developer with powerful mechanisms for tailoring the templates of the framework to his or her specific needs. An important point in that regard is that such tailoring should be unintrusive with respect to other parts of the application that might be using the same framework.

A package template looks much like a regular package in the target language, with the most notable exception being a `template` construct enclosing classes and interfaces. Figure 1 shows two example templates: template `T` with three classes, `A`, `B`, and `Aa`, where the latter inherits from `A`, and template `U` with only one class, `C`. To model package templates, we will in this paper use a graphical notation where UML classes and UML interfaces are enclosed by a bordered gradiently colored box with an underlined "template [name]" header, representing a template.

Package templates must be *instantiated* before use, and access to the contents of the instantiated template will thus be provided from within the instantiating scope. Each instantiation of a given template is completely independent of any previous instantiations. In the following, we shall use the term *package* in a broad sense, meaning a collection of (possibly related) classes and interfaces that is internally consistent, but not necessarily implemented as a Java or Groovy package as such. Using this terminology, each template instantiation will result in a new package being created.

Upon instantiation, the template may be customized according to its usage, and classes from different template instantiations (of the same or different templates) may be merged to form one new class. For instance, we may choose to instantiate the templates `T` and `U` from Figure 1, and to merge the classes `Aa` from `T` and `C` from `U` together, and give the merged class the name `AC`. Furthermore, we might add members to any of the classes, e.g. a new method `m4` to `B` and a new field `field3` to `A`. Template classes and interfaces may also be renamed, and templates may have (constrained) type parameters (though the latter is of little interest in the dynamically typed setting we shall stay within in the rest of this paper). The resulting classes now available in the instantiating scope are shown in Figure 2. Note that the classes in the instantiated template have no relation to their template counterparts, neither through inheritance relationships nor otherwise. Correspondingly, there are also no relations between classes from different instantiations of the same template(s).

2.2 Groovy

Groovy is a dynamic language that runs on the Java Virtual Machine. It can hence integrate nicely with Java, and to a certain extent it provides a familiar Java-esque syntax. The language rests on a purely object oriented foundation, and provides support for several attractive language concepts.

Both static and dynamic typing of variables and method signatures is supported in Groovy. However, there is presently no performance to be gained in Groovy by using static typing as opposed to dynamic (and in many cases it may actually be slower). Furthermore, Groovy cannot detect e.g. invalid calls to statically typed methods at compile time (since a fitting method may be added to the object in question at runtime) [17]. We will therefore, and for other reasons mentioned earlier, not consider the static typing features of Groovy in this paper, and instead focus solely on dynamically typed references, methods and properties defined by using the keyword `def`.

Dynamic typing aside, one thing in Groovy that is resolved entirely at compile-time is the given class from which to create a new object (e.g. as in `new A()`). That is, if a referenced class `A` cannot be statically determined to exist within imported packages and with correct accessibility modifiers, a compile-time error will occur. This will pose problems for our implementation, since the dynamic instantiation of a package template will make new classes available at runtime. We shall look closer at how we resolve this when discussing our implementation in Section 5.

A runtime meta-object protocol (MOP) is supported by the Groovy language. In accordance with the MOP, every object has an associated object instance of its meta-class. This object may be replaced by a version defined by the programmer, and this allows the semantics of method invocation and property getters and setters to be changed on a per-object or per-class basis. Also, new members may be added to classes and objects at runtime using the meta-class.

Further tailoring of the language's semantics may be done using compile-time transformations on the abstract syntax tree. AST transformations may be defined by the developer

to transform or replace parts of the syntax tree during different phases of the compilation process, such as parsing, semantic analysis, code generation etc. Such transformations may be global, or local pertaining to certain targets that typically may be Java-like annotations on source code elements.

Closures are first class citizens of Groovy, and units of code may as such be passed to and from methods.[1] Closures are hence also objects, and have their own meta-class instance that may be replaced by the programmer. They may also have a *delegate*, to which method calls and field accesses are dispatched. The delegate is, as the meta-class, replaceable by the developer. We will make good use of these features when defining our framework for instantiating templates.

3. Dynamic Package Templates

We shall now describe package templates for dynamic languages, exemplified by an implementation for the Groovy language. The two main points that motivate such an implementation are

- to make full use of the freedom that the typeless nature of many dynamic languages provide, in order to achieve greater flexibility when it comes to adaption and composition of templates upon instantiation, and,
- to leverage (and extend) the capabilities of PT in a dynamic setting, and provide a powerful framework for coherent reuse and runtime adaption of entire class hierarchies.

Furthermore, as was mentioned in the introduction, using this implementation as a vehicle for studying the properties of package templates as a software modularization technique without regard for typing issues comes as an added bonus.

3.1 Writing Templates

A package template in Groovy is written as follows:

```
template T {
   // classes and interfaces here
}
```

The contents of the template are ordinary Groovy classes or interfaces. These types, referred to as *template members*, may refer to each other and take part in inheritance hierarchies within the template. A template class is not allowed to have inheritance relationships to, or references to, classes outside of the template, with the exception of classes of templates that are instantiated within the current template. Templates need to be semantically (and, of course, syntactically) valid in their own right, and may as such be checked for validity according to the rules of the target language (in this case Groovy) independently of their potential usage.

None of the classes or interfaces defined in the template are available to the rest of the program before the template itself is instantiated (that is, the class loader does not see any of the members of a template before an instantiation of the template in question is performed).

3.2 Instantiating Templates

In the Java version of PT, template instantiation is performed statically at compile time, providing access to template classes and interfaces to every class and interface within the instantiating package. Instantiation can only be performed at the outermost level of a file, i.e. at the same level as any `import` statements.

In this dynamic version, the Groovy PT framework provides a set of methods[2] that may be invoked at runtime in order to instantiate templates. The motivation for studying dynamic instantiations as well as static, is that it is easy to imagine situations in which it cannot be known until runtime which version of a specific functionality is needed, or if certain modifications should be applied to an entire hierarchy of classes or not. Supporting the stance that this is actually useful for solving real world problems is the fact that many popular dynamic languages, such as e.g. Python, Ruby and of course Groovy, provide support for class modifications at runtime, and popular frameworks such as Rails [31] and Grails [15] make heavy use of this. We shall see examples of how this can be utilized in dynamic PT in the examples in Section 4.

A package template may be instantiated at run-time by utilizing the `instantiate` method of the supplied `PT` class, as shown in the following construct:

```
PT.instantiate { template T }
```
[3]

This statement will make *a copy of* all the classes and interfaces defined in the template `T` available to the caller of the static `instantiate` method, preserving potential type hierarchies as defined in the template.

However, instantiations may also be more elaborate. The `instantiate` method takes a Groovy closure as its sole argument, and this closure may in turn contain several calls to the `template` method. The `template` method takes a template name (such as `T` in the example above), and optionally a specification (in the form of a closure) of *adaptations*, such as mappings to new names etc.

If multiple templates are listed in the argument provided to `instantiate`, this will lead to one single instantiation of these templates, forming one single new package. Taking the example from Section 2.1 (as illustrated in Figures 1 and 2), we would employ the following call to merge the classes `Aa` and `C` (we will get back to how to add fields and methods, as in the example, in Section 3.3):

```
def INST_TU = PT.instantiate {
    template T { map Aa, AC }
    template U { map C,  AC }
  }
```

Here, the classes `Aa` and `C` are both renamed (or *mapped*) to `AC` (using the `map` method), and they will thereby be merged (more on that in Section 3.4). A new, identical (from a textual point of view), copy of each of the classes `A` and `B` will be available to the instantiating context, while the template classes `Aa` and `C` will not be available as individual classes, but only as their merge.

[1] While Groovy closures are not actually closures in the formal sense of the word, we shall in this paper use the term *closure* to refer to closures as the syntactic constructs that define subclasses of `groovy.lang.Closure`, and instances thereof.

[2] Or, if you will, a tiny internal DSL

[3] The syntax of Groovy is quite flexible, allowing the developer to omit parenthesis and semicolons where desirable. Furthermore, we massage the resulting expression a little at runtime, to allow the developer to omit commas and quotation marks. The instantiate statement could hence also be written in a more elaborate (and perhaps more immediately familiar) way: `PT.instantiate({ template("T") });`.

In the preceding paragraphs, we have not discussed the issue of the scope of instantiated templates. In a statically typed scenario, where template instantiations are performed at compile-time, the scope is always the instantiating package. However, when we are looking at dynamic instantiations, which may be performed at arbitrary points at runtime, the issue of instantiation scope becomes important.

When dynamically instantiating templates (either from within methods or as part of the static initialization phase of a class), a variable holding the resulting package needs to be explicitly specified, in order to provide the correct scope for the new package. E.g., given templates `U` and `V`, with template classes `A` and `B`, respectively, one could for instance have the following instantiation:[4]

```
void f() {
   def INST_UV
   if(someCondition) {
      INST_UV = PT.instantiate {
         template U { map A, Z } }
   } else {
      INST_UV = PT.instantiate {
         template V { map B, Z } }
   }

   def z = new INST_UV.Z()
}
```

This allows the runtime boolean condition `someCondition` to potentially affect an entire hierarchy of classes, providing specific adaptions to classes which may be used in subsequent stages of the application. The scope of the instantiated class hierarchy is well defined by the scope of `INST_UV` (which follows the normal scoping rules for variables in the language). New objects may be created from the instantiated template classes by referring to them with their qualified name, e.g. as shown above as `INST_UV.Z`.

Given a mechanism such as this, it is now quite possible (yet perhaps somewhat contrived) to write code that performs the same instantiation multiple times, e.g. within a loop such as the code snippet below.

```
for(;;) {
  INST_T = PT.instantiate { template T }
  a = new INST_T.A()
}
```

For each iteration of this loop, a new template instantiation would be performed, and the class `INST_T.A` in a given iteration would be different (though equal in structure) to that of the previous or next iteration. Since there may be multiple instantiations of any given template (or sets of such), each instantiated template class is given a reference to its instantiation, so that other classes from its own instantiation will take precedence over external classes when new objects are to be created.

Arguably, one would often want the classes and interfaces of a template to be available to every class in a given package, without having to perform several instantiations of the same template, in the same way as is the case for the static version of PT. The recommended way of achieving this in dynamic PT is to create a class that instantiates the template when it is loaded by the class loader. This class may then be statically imported for easy access in other files. The example below illustrates this:

```
class TemplateT {
   static def INST_T = PT.instantiate { template T }
}
```

Given the class above, other classes may utilize the instantiation of `T` by importing `TemplateT`:

```
import static TemplateT.*
class C {
   def m () { def a = new INST_T.A() }
}
```

3.3 Additions

When instantiating a template, template classes may be customized to the current usage by so-called addition classes. Such a class specifies method overrides and new members, and can be written as an ordinary class, with an additional `@Addition` annotation. For instance, consider the following template, and a corresponding instantiation of it:

```
template T {
   class C { def f() { /* something */ } }
}

def INST_T = PT.instantiate { template T }
```

The virtual method `f` in `C` may be overridden in an addition class, and new members may be added to the class, e.g. as follows:

```
@Addition class C {
   def f() { */ something else */ }
   def g = ""
}
```

The instantiated class `C` will have the added members and overrides of the addition class `C` (based on their sharing a name). Stated explicitly, the class `C` will after the instantiation have the method `f`, with the implementation given in the addition class, and a field `g` which is initialized to the empty string.

Once more, looking back to the example in Figures 1 and 2, the addition classes for `A` and `B` can be written in a straightforward manner, as follows:

```
@Addition class A { def field3 }
@Addition class B { def m4() { /* impl. here */ } }
```

The `@Addition` annotation ensures (through a local AST transformation defined by the PT framework) that objects may not be created from its belonging class until a template instantiation that targets this class has been made. It also serves as a marker interface for the implementation that needs to record certain pieces of meta-data about this class during the compilation process.

Keeping with tradition in Groovy, a method may also be defined by providing a closure containing its new implementation, and this closure may be used to provide a suitable override directly when instantiating the template, e.g. as follows:

```
def INST_T = PT.instantiate {
    template T {
      map C {
        override f { /* something else */ }
 } } }
```

[4] For clarity, we shall name all variables holding instantiated templates with the prefix `INST_` followed by the name of the template(s) or a derivation thereof. This is however only a convention, and not something that is enforced by our implementation.

If a template class is declared to be abstract, it may be made concrete upon instantiation by utilizing a non-abstract addition class, that provides concrete implementations of all abstract class members. This has turned out to be especially useful for implementing templates that represent some form of abstract protocol, where parts of the interaction can be specified in a totally generic way, while other parts need to be specified by the domain classes implementing the protocol. We shall see an example of this when looking at the Observer design pattern in Section 4.2.

3.4 Merging, Renaming and Exclusion

It seems to be a well established recognition within the OO community that traditional single inheritance alone is too limited to capture scenarios where complex reuse of code is needed. Many real-life situations actually require (or would benefit from) the reuse of code from more than one distinct entity. Multiple inheritance is one way to achieve this, but that comes with a host of well-known problems of both technical and conceptual nature. Mechanisms like traits [32] and mixins [4] aim to solve this problem by creating conceptually simple units of code intended only for code reuse, and not for object creation. PT offers a different approach to this kind of reuse, in which classes from independent template instantiations may be merged together, while preserving type hierarchies and avoiding multiple inheritance (we will get back to the latter).

The merge of two classes `A` and `B`, is defined as taking the discriminated union of the set of all the members (methods, fields and constructors with associated meta-data[5]) from `A` and the set of all the members of `B`. Members with the same name and signature stemming from different classes are hence not considered equal, and will therefore appear twice in the resulting set. This is defined as a conflict. For instance would the following not be allowed, and would lead to a runtime exception if not explicitly handled by the programmer:

```
template T { class C { def f(arg1) { ... } } }
template U { class D { def f(arg2) { ... } } }

def INST_TU = PT.instantiate {
    template T { map C, CD }
    template U { map D, CD }
  }
```

Here, the classes `C` and `D` are merged under the common name `CD`. Since they both define a method `f` with a single argument, there is a conflict. Every such conflict should be resolved manually by the developer. In static PT, one typical way to resolve conflicts, is to rename members (statically) until there are no more conflicts (e.g. in the example above, `C.f` could be renamed to `C.g`). Since the semantic analysis will bind every method call to a particular signature, renaming may safely be done (of both the member itself and all references to it). Given the dynamic nature of both the type system and method call resolution/dispatch in Groovy, this is not possible. With statements like e.g. `def f` followed by a call such as `f.m()`, it would be impossible to safely rename `m`, since the type of `f` can not be determined statically.

One general approach to renaming could be to keep a list (or some other fitting data structure) of all renamed members inside a class, and perform dynamic dispatch through this mechanism when in a *message not understood* situation. This works for renames of members from a single class, but it will not be of any help when there are merge conflicts involved, since it cannot be known if the call/field reference was intended for the renamed method/field or for the originally conflicting one that was left untouched in the merge.

Given these issues, it seems clear that renames cannot be used to resolve merge conflicts. Instead, a possible option would be to exclude one of the conflicting members, so that all calls/references would be routed to the remaining member. Exclusion may be done with e.g. the following instantiation (referring to the example templates `T` and `U` defined above):

```
def INST_TU = PT.instantiate {
    template T { map C, CD { exclude f } }
    template U { map D, CD }
  }
```

If one excludes members that are not the source of a merge conflict, there is always the possibility that these members are being referenced other places in the template. Since this cannot be known in advance (due to the dynamic typing issues discussed above), we have essentially two options; either to disallow exclusions that are not part of conflict resolution, or to allow it and let the developer deal with it. We have opted for the latter, and allow exclusion of members upon template instantiation[6]. One exception to this rule that is probably reasonable to make (though we have not yet implemented it), is to disallow the exclusion of members that are part of an interface that the class in question is declared to implement. To allow exclusion in such a scenario, the member would then first have to be excluded from the template interface, and then from the implementing class(es).

Another way to resolve a merge conflict when the conflicting members are methods, is to provide an override in an addition class. This override will in that case take precedence over all methods with equal signature stemming from template classes, and may access the original implementations through `super` calls, qualified with the name of the respective template class to which it belongs, as in e.g. `super[C].m()`[7].

The merge of two independent classes could implicitly lead to multiple inheritance, which we do not allow. Thus, some restrictions on allowable merges are needed. The first restriction is that we forbid template classes to have superclasses defined outside the template. 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. Superclass relationships between classes inside a template are of course allowed. The second rule is that if, in an instantiation, 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 instantiation. We also have to add the requirement that a merge must not result in a cyclic superclass-structure.

After a merge or a rename has been done, statements inside the instantiated template classes that create new objects (e.g. of the form `new C(...)`) need to processed by the PT

[5] Inner classes are not mentioned because they are presently not supported in Groovy

[6] Allowing the exclusion of members, though potentially unsafe, is also the approach taken by the designers of Traits.

[7] In some situations, particularly when instantiating the same template twice, a more elaborate way of referring to the original template class of a method may be needed.

framework so that they refer to the new name of the merged and/or renamed classes.

4. Examples

In this section, we will look at some examples in order to further motivate and explore the dynamic PT mechanism. We shall look at some properties that apply to PT in general, and some that are only attainable in the dynamic version.

4.1 Lists and Matrices

To begin with, we will here look at how a simple standard linked list implementation (adapted from an example in [22]) can be implemented and utilized with dynamic PT. First, we define a template containing classes for the list itself, and for elements of such lists, as shown in the following code:

```
template Lists {
   class List {
       def first, last
       def add(element) {
          if(first == null)
             first = element
          else
             last.next = element
          last = element
       }
       ... methods for removing etc ...
   }
   class Element { def next }
}
```

The `Element` class is of course not all that useful as is, and the intention is that upon instantiation, this class should have an addition class specifying additional members corresponding to the given problem domain, or be merged with a class from a different template. For instance, we may create a list of students by merging the `Element` class with a `Person` class as follows:

```
template Persons {
   class Person {
      def lastName, firstName
      ...
   }
}

def INST_PList = PT.instantiate {
   template Lists { map Element, Student }
   template Persons { map Person, Student }
  }
```

However, a `Student` class would probably need a few more properties and perhaps some added logic compared to a regular `Person` class. The necessary additions and/or potential overrides can easily be done by creating a corresponding addition class:

```
@Addition class Student {
   def studentNumber
   def attendingClasses = []
   ...
}
```

It is also possible to merge classes from the sample template, when this template is instantiated more than once. Below is an example of this, where we show how we may now use our `Lists` template to create a sparse matrix by instantiating this template twice, and mapping the respective classes accordingly (remember that every template instantiation is independent of any other instantiation of a given template).

```
template Matrix {
  PT.instantiate {
     template Lists, {
        map List, SparseMatrix
        map Element, Column
     }
     template Lists, {
        map List, Column
        map Element, Item
     }
  }

  @Addition class Column { def cNo }
  @Addition class Item { def rNo }

  @Addition class SparseMatrix {
     Item getItem(columnNo, rowNo) {
        def col = first
        while (col!=null && col.cNo!=columnNo){
           col = col.next
           ... and so on ...
        }
     }
  }
}
```

To use the `Matrix` template in a real program, one might now simply instantiate the template and map the `Item` class to a fitting domain abstraction, or create new addition classes to specify additional required state or behavior.

4.2 A Reusable Observer Pattern Implementation

The observer pattern is a design pattern [12] with two roles, the subject and the observer. Each subject maintains a list of observers that are interested in being notified when certain (yet unspecified) changes occur in the subject. An observer may choose to observe one or more subjects at any given time. We looked at this pattern in [2], utilizing an aspect oriented extension to Java PT to create a reusable implementation. Here we will look at how the same can be achieved with dynamic PT in Groovy.

To exemplify the use of the pattern, we consider a package for drawing objects on a screen or printing them out on a printer (strongly resemblant of the examples in [18] and [26]). The `Drawing` package is realized as a package template as shown in Figure 3. The `Screen` and `Printer` classes can be considered potential observers, while the `Line` and `Point` potential subjects, respectively. Hence, `Screens` and `Printers` should be notified when changes occur in `Lines` or `Points`.

The Observer pattern can be programmed as a template consisting of two classes, `Subject` and `Observer`, as shown below:

```
template ObserverProtocol {
   public abstract class Observer {
     abstract def notify(changee);
   }

   public abstract class Subject implements
      GroovyInterceptable {
      def observers = []

      def invokeMethod(name, args) {
         def result = this.class.metaClass.
```

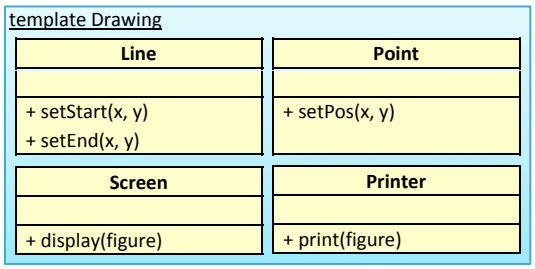

Figure 3. The `Drawing` template with two potential subjects and two potential observers

```
            invokeMethod(this, name, args)
         if(getMutatingMethods().contains(name))
            afterChanged()
         return result
      }

      def addObserver(observer) {
         observers.add(observer) }

      def removeObserver(observer) {
         observers.remove(observer) }

      abstract def getMutatingMethods()

      def afterChanged() {
         foreach(observer in observers) {
            observer.notify(this);
         }
      }
   }
}
```

The `Observer` class has only one method, the abstract `notify`, that will have to be implemented at a later stage to make it meaningful to the concrete observers.

The `Subject` class has methods for adding and removing observers from the current subject instance. Furthermore, it defines an abstract method `getMutatingMethods`. This method should return a list describing the methods that mutate the state of the subject in such a way that observers should be notified, and it must clearly be left abstract at this point. The `Subject` also implements the empty `GroovyInterceptable` marker interface, which is a part of the Groovy MOP implementation. This interface signifies to the runtime that method invocation should be routed through the virtual `invokeMethod`. By overriding this method, we may intercept every method invocation, and use the `name` argument to check if it is a call we are interested in. Our overriding `invokeMethod` calls the method that was originally called by using the meta-class, and then checks to see if the method called is a mutating one. If so, it notifies observers by calling the `afterChanged` method.[8]

To apply the observer pattern to the `Drawing` template, we instantiate both templates as follows:

```
def INST_DO = PT.instantiate {
     template Drawing
     template ObserverProtocol {
         map Subject, Line
         map Subject, Point
         map Observer, Screen
         map Observer, Printer
     }
  }
// addition classes here, see below
```

Here, we are using a feature only possible in the dynamically typed version of PT, and that is that the `Subject` class and the `Observer` class are both mapped to more than one target class (i.e. `Line` and `Point`, and `Screen` and `Printer`, respectively) in the merge done upon instantiation. In a statically typed world, this would have introduced typing issues for references that were previously typed as either `Subject` or `Observer`, and would hence not be allowed.

What is missing form the example at this point, is to specify which methods that are considered to mutate state in `Line` and `Point`, and which action should be taken upon call to the `notify` method for `Screen` and `Printer`. For the latter two classes, we would quite simply write addition classes to call the `display` and `print` methods, respectively, from the `notify` method, providing the changed subject (i.e. a `Line` or a `Point` object) as the argument.

For the `Line` and `Point` classes, we assume that every method with a name starting with '`set`' mutates the state, and we specify this by using addition classes:[9]

```
@Addition class Line {
  def getMutatingMethods() {
     return this.metaClass.getMethods().
        findAll { it.name.startsWith("set") }
}
```

The addition class for `Point` would be equal to that of `Line`, with the exception of the class name itself. Having only two subjects, that is probably not much of an issue, but if there were many subjects that all would share the same definition of mutation, duplicating that same code many times would not be desirable. With dynamic PT, we could instead have put the concretization of `getMutatingMethods` into a template class (that is, in a template that is separate from both the `Observer` and the `Drawing` templates), and upon instantiation mapped the same implementation directly to the subject role.

An example program utilizing the instantiated templates could look as follows:

```
def screen = new INST_DO.Screen()
def line = new INST_DO.Line()
line.addObserver(screen)
line.setX(10)
```

The call `line.setX(10)` would now ultimately result in the `screen`'s `display` method being called with `line` as its argument.

[8] The check for mutating methods is for the purpose of this example a simple one based on name only, but it could of course with relative ease be extended to take the entire signature of the method into consideration, or to allow wild-cards in the vein of e.g. AspectJ [6] etc.

[9] This approach would include the base class methods `setProperty` and `setMetaClass` in the list of mutating methods, which we in a real situation clearly would not want (at least not the latter of the two). This could easily be amended by a little more elaborate approach to the `getMutatingMethods` method, however for the sake of our example and what we want to demonstrate, the simple approach will suffice.

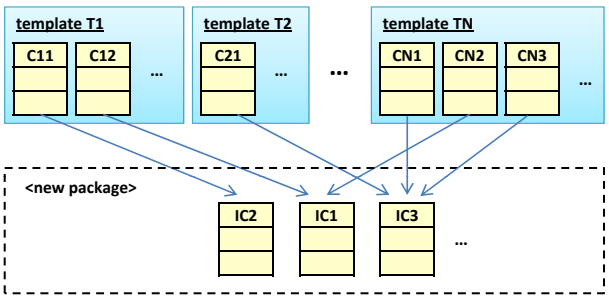

Figure 4. A schematic view of how PT can be used to quite freely mix features from different libraries to form one new package. The arrows represent an example instantiation with mapped template classes.

Clearly, it is also possible to make an implementation of the Observer pattern in plain Groovy. There are, however, certain worthwhile benefits of using package templates. To begin with, PT allows for one *single* and *coherent* abstraction (the template) to capture the concept of the pattern. The protocol of interaction between the two roles may be specified in one place, and may be reused as one conceptual unit. Furthermore, PT allows for pluggable reuse of the entire pattern, even when the targeted domain abstractions already are a part of an inheritance hierarchy, without having to resort to multiple inheritance. This does, however, require that the domain abstractions in question are also implemented as templates or addition classes. Last, if the classes of the `Drawing` template were in use other places in the application, adding the `Subject` and `Observer` functionality could be done unintrusively only where needed, and the rest of the application could, if desirable, be left with the original versions of the classes from the template.

If we generalize this example, we see that PT allows the developer to enjoy a great deal of freedom to mix and match several different libraries (templates), and merge required functionality into new classes to form an independent, new, package. Figure 4 illustrates this. Package specific adjustments may be made through addition classes, or as parameters to the `map` method.

4.3 Adding logging capabilities dynamically

For this example, we shall consider a Groovy application for simulating graph-like structures of various forms that is deployed at a customer's site (and the availability of debugging tools are hence limited). The application is using a graph library in the form of an instantiated package template to model certain aspects of a physical Ethernet network, as well as other structures such as telephone lines etc. A skeleton of the `Graph` template is defined as follows:

```
template Graph {
   class Node {
      def edges = []
      def insertEdge(toNode) {
         def e = new Edge()
         edges.add(e)
         return e
      }
      ...
   }
   class Edge {
      def fromNode, toNode
      def delete() { ... }
      ...
   }
   ...
}
```

The library is used by the rest of the application by utilizing classes with static members holding instantiated template classes, as illustrated in Section 3.2:

```
class EthGraph {
   static def INST_EG = PT.instantiate {
       template Graph {
          map Node, Computer
          map Edge, Connection
        }
    }
}
// possible additions to Computer and Connection here

class TelephoneGraph {
   static def INST_TG = PT.instantiate {
       template Graph {
          map Node, Central
          map Edge, Line
        }
    }
}
// possible additions to Central and Line here
```

Consider now a situation in which a certain part of this application is misbehaving, and we suspect that the graph libraries are involved in the bug. In order to debug this, we would like to enable extended logging at the client site, but only for a very limited part of the application (since extensive logging may be expensive in terms of performance and storage space). To enable logging, we utilize similar techniques as for the observer pattern presented in Section 4.2 to create a simple `Logging` template consisting only of a `Logger` class, that overrides the Groovy `invokeMethod` method to provide functionality resembling the aspect-oriented programming construct of *before advice*:

```
template Logging {
  class Logger implements GroovyInterceptable {
     def invokeMethod(name, args) {
        log(name, args) // perform logging
        return this.class.metaClass.
          invokeMethod(this, name, args) // proceed
     }
     ...
  }
}
```

In order to enable logging, we may now create an instantiation of `Logging` that merges the `Logger` class with the classes whose method invocations we wish to log. To minimize the effect on the efficiency of the application, we may choose to only enable logging for a selected class that uses the library. To achieve this, we may change that class so that every object of this class includes its own reference to the instantiated templates. For instance, given a `Simulation` class that uses the `INST_EG` and `INST_TG` instantiations defined above, we may create local variables that hold the instantiated templates, and define methods that may set these variables to instantiations in which logging is merged in at runtime.

```
class Simulation{
   def INST_EG = EthGraph.INST_EG
   def INST_TG = TelephoneGraph.INST_TG

   def setupNetwork() {
      def c1, c2
      c1 = new INST_EG.Computer()
      def wire = c1.insertEdge(c2)
      ...
   }

   def enableEthLogging() {
     INST_EG = PT.instantiate {
       template Graph {
          map Node, Computer
          map Edge, Connection
       }
       template Logging {
          map Logger, Computer
          map Logger, Connection
         }
      }
   }
   ...
}
```

After the `enableEthLogging` method has been called, every new-statement involving `INST_EG.Computer` and `INST_EG.Connection` will refer to the instantiated package that has the logger class merged with `Computer` and `Connection`. Furthermore, every such subsequent object created by any of these instances will again also have logging enabled, but the rest of the application will remain unaffected. This means that after a log-enabling call, the `Computer` object created in `setupNetwork` will have logging enabled (as its class will be a merge of the `Node` and `Logger` classes). Logging will hence also apply to the `Connection` object that is created by the call to `c1`'s `insertEdge` method in the example above. To turn logging off again, one would simply reset the `INST_EG` variable to its original value (though this would of course not affect any objects that are already created).

This exemplifies how dynamic instantiations can be applied in order to selectively turn features on and off in a application, with a well-defined scope for the changes. One can easily imagine other scenarios where this would be relevant, for instance customizing certain parts of application based on the contents of a license file (resolved at runtime), adapting the GUI to different kinds of devices etc. Dynamic class modifications are supported by many languages, but the pitfalls of using such features is that the modifications may have unanticipated effects on other parts of the application, making the program harder to debug and reason about (since global changes to classes may be done at any point). The main incentive for using PT in such scenarios would therefore be its ability to apply coherent changes to an entire hierarchy in one operation at runtime, and to be able to constrain the scope of such modifications.

5. Implementation

Having decided upon Groovy as the target language for an implementation, the inherent flexibility and power of Groovy allowed us to follow a self-imposed restriction: to only make use of the standard features of Groovy itself, and not modify any of its source code. This enables anyone that has Groovy to just plug in our libraries, and experiment with Groovy package templates. As a result of this, the syntax may not be as nice as it could have been (i.e. in contrast to the Java implementation of PT, which employs special keywords for template instantiations and addition classes), but on the other hand it should be familiar to any developer versed in Groovy. It also makes making modifications and extensions to the syntax (and the semantics!) a lot more manageable, effectively making the PT framework open for modification.

The dynamic PT framework is implemented as a Java archive (jar) that can be included in any Groovy solution to provide access to its features. The implementation is mainly divided into three parts; (1) the tiny DSL that allows for expressing template instantiations within Groovy, (2) the actual implementation of such instantiations, and (3) the AST transformations that take place at compile time to allow `new`-statements that include references to as-of-yet not instantiated template classes, and to make sure that addition classes are not used to create objects without having been targeted by a template instantiation.

The first part relies on a set of closures for which the execution is delayed until they are assigned a specific delegate that overrides the `methodMissing` and `propertyMissing` meta-object protocol methods. Utilizing this in a builder-like fashion (resembling e.g. the Groovy `MarkupBuilder`), the framework dynamically builds a tree representing the requested instantiation at runtime. This allows for a quite clean syntax, without the need for modifying Groovy's grammar.

When the tree representing the instantiation has been built, control is handed over to a transformer class that builds a new tree representing the instantiated template. The framework will then use this tree to look for each template that is to be instantiated in a file named `[template name].groovy.template`. If a fitting file is found, its contents are handed over to a simplified template parser, that separates the template classes and interfaces from template instantiations and `import` statements at the outermost level of the template. For each template class or interface, a specialized class loader that records the AST for each such respective element is applied.

The result of a template instantiation is an instance of the provided `InstantiatedTemplate` class. To this class, properties for each instantiated template class are added (at runtime). For instance, for a template containing classes `A` and `B`, the object instance of `InstantiatedTemplate` will have corresponding properties named `A` and `B`. Each such property contains an instance of the `Class` class, that provides access to the instantiated template classes to the rest of the program at runtime. For each instantiated template class, a property `__INST` is added, and set to a reference to the instantiated package of which the class is part. By doing this, we can ensure that classes from the same instantiation will take precedence over potential different instantiations of the same class when new objects are to be created.

As the final part, calls to constructors (e.g. like `new C(...)` or `new INST_T.C(...)`) need to be processed at compile-time. This is because classes and interfaces are resolved statically by Groovy, while dynamic PT allows for dynamic class generation at runtime. Our approach is hence to detect the object instantiations for which the class cannot be statically resolved, and instead inject a runtime call to the `newInstance` method of the instance of the corresponding `Class` class, by applying an AST transformation to the source tree. Furthermore, classes with the `@Addition` annotation are transformed so that they appear as ordinary classes only after a template instantiation has targeted them.

6. Related Work

A trait [32] is a construct that encloses a stateless[10] collection of provided and required methods. A trait may 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 [27].

Mixins [4] 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. In Groovy, however, mixins are mixed in at runtime using the `mixin` method of the `Class` class.

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 we have shown for the Observer pattern example. 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 larger units of code. 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'. In contrast to the Groovy mixin implementation (which makes global changes to existing classes at runtime), the PT approach is to define a new set of classes (that may be given a local scope) for each runtime instantiation of a template, so that such changes do not inadvertently affect code in unrelated parts of an application. One might therefore argue that dynamic PT offers a safer way to perform runtime class modifications, given that existing objects need not be affected.

Open classes in MultiJava [5] supports adding new methods to existing classes without resorting to subclassing or modifying the original code. (Multiple dispatch is also supported.) Several other languages, like Ruby, JavaScript and Groovy (and to a certain extent C#), also support open classes with varying syntax and capabilities. One of the main obstacles overcome in [5] has to do with static typing in the form of modular type checking and compilation, which is not an issue for us. With open classes, one can achieve some of the same end results as with PT, with some important distinctions. First, PT supports adding both state and behavior to existing classes, whereas with open classes only behavior can be added. Furthermore, PT allows the developer to make several different and unrelated additions and modifications, resulting in separate class hierarchies that may be localized to certain parts of an application. MultiJava does not support runtime addition of methods (probably just because it is tightly bound to Java), but the open classes in Ruby, JavaScript and Groovy support this.

Groovy also supports an open classes-like concept called *categories*, that enables the developer to specify that certain methods should be added to appropriate types inside a scope enclosed by curly brackets. The added functionality will apply to every statement inside this lexical scope, and also propagates down the call stack. However, only static methods are supported, and instance state cannot be added. As such, categories are similar to C# extension method [11].

BETA [25, 24], gbeta [10] and J& [28] (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.

Aspect-oriented programming (AOP) [21] involves concepts related to PT. For instance, intertype declarations in AspectJ [6] may (statically) add new members to existing classes. The Caesar language [1, 26] 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). In [9], Tanter describes a mechanism for controlling the scope of dynamically deployed aspects (including propagation down the call stack and to new objects), however, these aspects may affect behavior only, and not class structure or hierarchy.

Context-oriented programming (COP) [7] 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). This differs from PT, where runtime modifications to template classes will be visible according to the visibility of the variable that holds the instantiated package template (so for instance a `ThreadLocal` variable could yield different instantiations per thread). While doable, to mimic COPs stack-like activation and deactivation of features at runtime in PT, one would have to write quite a lot of 'glue code'.

In a subject-oriented [19] 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 others), that tells the system how to compose separate subjects with overlapping methods and/or state. Like with mixins and traits, there is 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.

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

7. Concluding Remarks and Future Work

In this paper, we have described how package templates can be applied to and implemented in the dynamic language Groovy. We have argued, through a set of examples, that it provides powerful features for code modularization and runtime adaption of entire class hierarchies, and that this provides a set of advantages over conventional approaches to runtime class modifications.

For future work, it seems worthwhile to investigate the reconciliation of some of the dynamic features from this paper with the static version of PT, so that, for instance, templates may be instantiated at runtime in a type safe manner within a statically typed language like Java or C# .

Further utilization of the openness of this implementation, by providing the programmer with direct means to manipulate and add e.g. merge strategies and conflict resolution options, seems like an interesting endeavor that could add useful flexibility to our framework.

Applying the mechanism to a real-world scenario is also a priority in our continued research and development efforts in this area.

Acknowledgments

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). The authors would like to thank the kind individuals on the *groovy-user* mailing list for helpful hints during the implementation process. Furthermore, we would like to thank Fredrik Sørensen and Birger Møller-Pedersen for fruitful discussions, and the anonymous reviewers for valuable feedback.

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, 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.

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

[4] 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.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

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

[7] 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.

[8] Ecma International. *Standard ECMA-334 C# Language Specification*, 4th edition, 2006.

[9] Éric 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.

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

[11] Extension methods (C# Programming Guide). URL: `http://msdn.microsoft.com/en-us/library/bb383977.aspx`.

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

[13] A. Goldberg and D. Robson. *Smalltalk 80 : The Language*. Addison-Wesley Series in Computer Science. Addison-Wesley Professional, January 1989.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] The Grails homepage. URL: `http://grails.org`.

[16] The Groovy homepage. URL: `http://groovy.codehaus.org`.

[17] Groovy: Runtime vs compile time, static vs dynamic. URL: `http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic`.

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

[19] 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.

[20] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[21] 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.

[22] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. To appear in the Journal of Object Technology (available now from `http://home.ifi.uio.no/steinkr/papers/`), 2009.

[23] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] The Python homepage. URL: `http://python.org`.

[30] The Ruby homepage. URL: `http://ruby-lang.org`.

[31] The Ruby on Rails homepage. URL: `http://rubyonrails.org`.

[32] 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.

[33] F. Sørensen and S. Krogdahl. Generic packages with expandable classes compared with similar approaches. In *NIK 2007*. Tapir akademisk forlag, 2007.