Towards Pluggable Design Patterns Utilizing Package Templates

Eyvind W. Axelsen and Stein Krogdahl
University of Oslo
Department of Informatics
{eyvinda, steinkr}@ifi.uio.no

Abstract
In this paper, we show how *package templates*, a new mechanism for code modularization, may be utilized to create reusable packages for selected design patterns that can be plugged into an existing architecture with a minimum of "glue code". Exemplified through these implementations we consider some desirable extensions to the previously published version of the mechanism.

1 Introduction

The concept of design patterns [9] is an approach to object oriented design and development that attempts to facilitate the reuse of conceptual solutions for common functionality required by certain *patterns* or classes of common problems. As such, they seem like a perfect candidate for inclusion as reusable components in frameworks, be they completely general such as e.g. the .NET framework [17] or application specific. However, it seems that even though the concepts of many patterns are in relative widespread use, implementing them as reusable components in mainstream languages like C# [6] or Java [11] is hard. This is at least in part due to limitations with regards to e.g. *extensibility of the subtyping relation* [19] and/or lack of support for mechanisms dealing with crosscutting concerns.

A few existing papers explicitly discuss new language concepts in the context of implementing design patterns, notably one by Hannemann and Kiczales [10] utilizing AspectJ [2, 5], and one by Mezini and Ostermann [16] utilizing the Caesar system [1].

The package template (PT) mechanism [12, 13] targets the development of collections of reusable interdependent classes. Such a collection is a template for a package, that may be *instantiated* at compile time, thus forming an ordinary package. At instantiation, the template may be customized according to its usage, and classes from different independent template instantiations (of a single or several distinct templates) may be merged to form one new class.

In this article, we utilize package templates to provide reusable implementations for a few selected design patterns. Exemplified through these pattern implementations, we will consider a few desirable extensions to PT.

This paper was presented at the NIK-2009 conference; see http://www.nik.no/.

2 Overview of the Package Template Mechanism

We here give a brief and general overview of the package template mechanism. The concepts of the mechanism are not in themselves tied to any particular object-oriented language, but the examples will be presented in a Java-like syntax, and the forthcoming examples will be based on Java.

A package template looks like a regular Java file, but we use a syntax where curly braces enclose the contents of templates, e.g. as follows:

```
template T<E> {
  class A { ... }
  class B extends A { ... }
}
```

Templates may have (constrained) type parameters, such as `E` above, but we will not make use of this feature in this paper. Apart from this and a few other constructs (including the extensions we propose), valid contents of a template are also valid as plain Java programs. As such, templates may also be type checked independently of their potential usage(s).

In PT, a template is instantiated at compile time with an `inst` statement, which has some significant differences from Java's `import`. Most notably, an instantiation will create a local copy of the template classes, potentially with specified modifications, within the instantiating package. An example of this is shown below:

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

Here, a unique instance of the contents of the package template `T` will be created and imported into the package `P`.[1] In its simplest form, the `inst` statement just names the template to be instantiated, e.g. `inst T`, without any other clauses. The example above additionally shows how the template classes `A` and `B` are renamed to `C` and `D`, respectively, and that expansions are made to these classes. Expansions are written in `adds`-clauses, and may add variables and methods, and also override virtual or implement abstract methods from the template class.

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

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

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

[1]In the following, we will skip the explicit package declaration in the examples.

Consider now the following usage of these templates:

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

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

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`. Note how the abstract `m2` from `B` is implemented in the `adds` clause (and the expansion does hence not need to be declared abstract), and furthermore that both `m1` and `m2` now have signatures of the form `MergeAB` $\rightarrow$ `MergeAB`.

To sum up, some of the useful properties of PT are: It supports writing reusable templates of interdependent, cooperating classes which may be statically type checked without any information of their usage. Upon instantiation, a template class may be customized, and merged with other template classes. References within a template to a template class will be re-typed according to the instantiation.

3 Design Pattern Implementation and Extensions to PT

In this section, we will look at how three design patterns may be implemented in PT with a few extensions that were deliberately not treated in [13]. Though they are exemplified here through pattern implementations, it is our belief that the extensions are generally usable and viable as integral parts of PT.

Running Example

To exemplify usage of the design patterns we consider below, we will use a simple example that revolves around two classes, `Screen` and `Line`. The idea is that a screen may be used to draw and display lines. This may be (naïvely) implemented by the following package template sketch:

```
template Drawing {
  public class Screen {
    public void update(Line l) { ... }
    ...
  }
  public class Line {
    protected int x1, y1, x2, y2;
    public void setStart(int x, int y) { x1 = x; y1 = y; }
    public void setEnd(int x, int y) { x2 = x; y2 = y; }
    ...
} }
```

The Singleton Pattern

The Singleton pattern is a pattern from [9] that is used to restrict the number of objects of a class to at most one at any given time. This is typically done by making the constructor of the class private, and providing a static `Instance` property, through which the object may be accessed. The object is typically created lazily when requested for the first time.

Despite the inherent conceptual simplicity of this pattern, there are several pitfalls to be aware of when implementing it, that, depending on the language used, might make it tricky to get right (see for instance [20] and [21] for discussions pertaining to the Java memory model, thread safety and more). This points to the need for a safe and reusable implementation. Generic constructs help, but are (in current implementations of Java) unable to express the constraint that the singleton should have a private constructor. With a package template, this can be expressed quite naturally, e.g:[2]

```
template SingletonPattern {
   public class Singleton {
      private Singleton() {}

      // SingletonHolder is loaded on the first execution of getInstance()
      // or the first access to SingletonHolder.INSTANCE, not before.
      private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
      }

      public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
} } }
```

Visibility and access modifiers for template classes is a topic that, save for a few suggestions, is explicitly left for future work in [13]. While we do not have room for a full treatment of this here, we follow and expand on these suggestions, and shall use the following rules:

With respect to access modifiers, a template is considered a separate package. That is, the default Java accessibility ('package-private') means within the template itself. Protected members are visible to subclasses, addition clauses and merged classes. Private members are visible only to the declaring class, with one exception: private constructors are visible to addition clauses, and may be called from constructors declared there. This is vital, since it is required in PT to include a new constructor for merged classes that differ in this respect. However, the addition class may not create new objects using the `new` operator under such conditions, and it may also not increase the visibility of the constructor (e.g. the addition of a public constructor would not be allowed in the example below). Thus, if a template class `TC` with a private constructor is given an addition clause `D` in an instantiation, then objects of type `D` can only be generated by a statement `new TC(...)` from within the template.

Now, to reuse the pattern implementation from above and make a domain class into a singleton, we may simply instantiate the template and map the `Singleton` class to a fitting domain abstraction, e.g. a factory for `Screens` from our running example.

```
inst SingletonPattern with Singleton => ScreenFactory;
inst Drawing;

class ScreenFactory adds {
```

[2]Example implementation of the Singleton class adapted from the Wikipedia implementation available at `http://en.wikipedia.org/wiki/Singleton_pattern`, accessed June 22, 2009.

```
  public Screen createScreen() {
    Screen s = ... < create new or reuse existing screen > ...
    return s;
} }
class Program {
   static void main(String[] args) {
      Screen s = ScreenFactory.getInstance().createScreen();
      s.update(new Line());
      ...
} }
```

The `ScreenFactory` class will now have all the properties of a singleton according to the pattern (including a private constructor) as well as the added `createScreen` method.

This example shows that, through the use of package templates, we may utilize the full capability of the Java (or other targeted) language to specify both constraints and functionality for a generic (in a broad sense of the word) reusable implementation, which goes beyond what is available in the language today. Using techniques similar to the one described above, similar implementations of e.g. the *Flyweight* and *Multiton* patterns from [9] could also be made.

Nested Classes and the Memento Pattern

The Memento pattern is a design pattern from [9] that revolves around three roles; the *originator*, the *caretaker* and the *memento*. The caretaker is, at different points in an application, interested in storing and managing the *state* of the originator. However, an important point of this pattern is that the state of the originator should not be exposed to caretaker. Hence, the memento provides a way to allow the caretaker to store the state of the originator, without knowing its inner details. The pattern could e.g. typically be used to provide an application with undo/redo functionality, and hence the originator provides methods to both retrieve and restore its state by using mementos.

This pattern may be implemented as a reusable 'skeleton' in PT, as shown below. While this template leaves the bulk of the actual implementation up to users of the template, it provides the benefit of clearly laying out the pattern's participants, and providing well defined (abstract) points in the code that should be concretized by the user.

```
template MementoPattern {
  public abstract class Originator {
    public Memento CreateMemento() {
       return new Memento(GetState());
    }

    public abstract void SetMemento(Memento m);
    protected abstract State GetState();
    protected abstract class State { }

    public class Memento {
      private Memento(State state) { this.state = state; }
      private State state;
} } }
```

This implementation does not explicitly involve the caretaker role, since objects having this role would merely be consumers of the public interface provided by the template. A skeleton caretaker implementation that made use of this interface could be provided if desirable.

The `Originator` class contains nested classes `State` and `Memento`, where the former is abstract. They conceptually belong to the originator, so nesting makes good sense from a code organizational point of view.

Template classes that contain nested classes is an issue that is not treated in [13]. However, since this is an integral part of Java, that appears to be in widespread use both for code organization in general and, in particular, for event handling, it seems like a natural extension of the mechanism, that may additionally provide some of the benefits of *virtual classes* [14], due to the possibilities provided by addition clauses. When implementing support for this in Java PT, care must be taken so that e.g. references to `<EnclosingClass>.this` are re-typed correctly. Note that this also applies to anonymous inner classes, even though these are not allowed to have addition clauses or be merged with other classes. Furthermore, when merging an inner class with another class, care must be taken (by the PT framework) to not introduce any inconsistencies in the inheritance hierarchy.

Consider the following example template containing nested classes:

```
template Nested {
   class Outer{ class Inner { ... } ... }
}
```

Templates such as this may be instantiated in the normal manner, with possibilities for addition clauses or merges of both `Outer` and `Inner`, e.g. as follows:

```
inst Nested with Outer => A { Inner => B }
class A adds { ... class B adds { ... } ... }
```

When providing an addition class for an inner class, a corresponding outer addition class must also be provided, as `B` and `A` in the example above shows, respectively.

Returning now to the memento pattern described above, we can utilize our pattern implementation to provide memento functionality for the `Line` class as shown below:

```
inst MementoPattern with Originator => Line { State => LineState } ;
inst Drawing ; // or explicitly: inst Drawing with Line => Line

class Line adds {
  protected State GetState() {
    return new LineState() {{ x1 = this.x1; x2 = this.x2; ... }};
  }
  public void SetMemento(Memento m) {
     this.x1 = m.state.x1; ...
  }

  protected class LineState adds {
    protected int x1, x2, y1, y2;
} }
```

To utilize the new capabilities added to `Line`, the `Screen` may call `CreateMemento` at fitting points (for instance when a change has been made, utilizing the Observer pattern presented in the next section), and support undo through `SetMemento`.

Aspect-Oriented Programming and the Observer Pattern

The observer pattern is a design pattern 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. Such a notification mechanism is a good example of some of the strengths of aspect-oriented programming (AOP) [8].

AOP revolves around the notion of *crosscutting concerns*, i.e. concerns that are not easily captured in any one part of the chosen class hierarchy of an OO application. Such concerns may in (traditional) AOP be defined within a construct known as an *aspect*. The aspect contains code, called *advice*, that is *weaved* in at locations in the base code specified by *pointcuts*.

Usage of the inherent merge capabilities of PT may in itself be seen as a form of code weaving, so in that sense the addition of more AOP-like constructs to strengthen its capabilities in this regard seems like a natural path to follow. Furthermore, mechanisms that bear some resemblance to PT (such as e.g. [16]) have incorporated AOP to great effect.

An important goal for PT is to retain static safety. Thus we seek to avoid some of the issues posed by the *fragile pointcut problem* [22], and shall therefore consider a somewhat restricted version of common AOP concepts. We believe that paired with PT this may still provide constructs that are sufficiently powerful and expressive for a lot of purposes.

In our approach, aspects are realized not as separate entities (neither at runtime nor compile-time), but rather as pointcuts and advice defined as members of template classes, where a pointcut can only refer to directly visible methods, fields, etc. Pointcuts and advice can be redefined in addition classes and subclasses in the same way as methods. This entails that the possible changes to the base program through the weaving of a pointcut/advice will be limited to the classes of the template, and the addition classes (and their subclasses) of the corresponding instantiation. Thus, quantification using wild-cards as in e.g. AspectJ is not used in our AOP extension of PT. Thereby, we can control the typing of the program to quite another extent. See syntax and example below.

Pointcuts are declared inside classes according to the following EBNF grammar sketch, where terms in quotes are terminals and the unquoted ones are non-terminals. Terms enclosed in curly brackets may be omitted or repeated, while terms in square brackets may be present one time or not at all. Productions that are equal to their Java equivalents are left out for brevity, and things enclosed in << and >> should be understood as "pseudo-EBNF" in place of parts left out. The abbreviation `pc` is used for *pointcut*:

```
pc_decl ::= { modifier } "pointcut" ( qualified_identifier | "void" | "*" )
 identifier "(" [ <<parameter list>> ] ")" ( "{" pc_expr "}" | ";" ) ;

pc_expr ::= ( call | execution | get | set ) "(" identifier ")"
 | pc_expr "&&" pc_expr | pc_expr "||" pc_expr | "(" pc_expr ")" ;
```

Likewise, an advise has the following syntax:

```
advice_decl ::= { modifier } "advice" identifier
  ( before | after | around ) identifier "{" { <<valid statement>> } "}" ;
```

To illustrate the semantics of the language, we consider the following example:

```
template T {
  class A {
    A m1(A a) { ... }
    pointcut A pc1 (..) { call(m1) }
} }
```

The class A contains a pointcut that matches calls to the method m1. The template may be instantiated as shown below:

```
inst T with A => B;
class B adds {
  pointcut B pc1(..) { call(m1) || call(m2) }
  advice afterM after pc1 { ... }
  B m2() { ... }
}
```

Here, A is re-typed to B, and B adds a method and an advice, and overrides the pointcut pc1. Note that even though pc1 in T refers to the type A, the pointcut will after instantiation refer to B, and continue to match method calls to m1. This is made possible by the fact that pointcuts are not string patterns, but actual bindings, as mentioned above.

Returning now to the Observer pattern, we consider again our example package for drawing lines on a screen. When a line changes its length or position, the observing screen(s) should be updated to reflect the changes. We would like this logic to be abstracted out of the concrete Screen and Line classes, so that it can be reused for other manifestations of this particular problem.

Utilizing package templates, we could implement the pattern with the following code[3]:

```
template ObserverProtocol {
 public abstract class Observer {
   abstract void notify(Subject changee);
 }
 public abstract class Subject {
  List<Observer> observers = new List<Observer>();

  public void addObserver(Observer o) {observers.add(o);}

  abstract protected pointcut * changed(..);
  protected advice ac after changed {
    foreach(Observer o in observers) { o.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 pointcut changed, that will have to be refined in concrete subjects. The pointcut specifies that any parameters and return types are valid for its (as of yet undefined) corresponding join points with the use of wild-cards "*" and "..". We can now do the instantiation as follows:

[3]The removeObserver method is trivial and hence omitted for brevity.

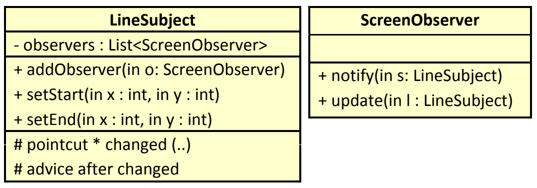
LineSubject
- observers : List<ScreenObserver>
+ addObserver(in o: ScreenObserver)
+ setStart(in x : int, in y : int)
+ setEnd(in x : int, in y : int)
# pointcut * changed (..)
# advice after changed
ScreenObserver
+ notify(in s: LineSubject)
+ update(in l : LineSubject)

Figure 1: *The resulting classes from the merge of the ObserverProtocol and Drawing templates*

```
inst Drawing with Screen => ScreenObserver, Line => LineSubject;
inst ObserverProtocol with Observer => ScreenObserver,
  Subject => LineSubject;
```

`ScreenObserver` and `LineSubject` become a merge of `Observer` and `Screen`, and `Subject` and `Line`, respectively. Furthermore, we make use of addition clauses to concretize the abstract members of the template classes. These clauses are only needed in order to concretize what was left as abstract in the respective templates.

```
class ScreenObserver adds {
 void notify(LineSubject l) { update(l); }
}
class LineSubject adds {
 pointcut * changed(..) {call(setStart) || call(setEnd)}
}
```

The two resulting classes are shown graphically in Figure 1 (where a fourth compartment has been added to the UML class representation to hold AOP members). Note that that a re-typing has occurred, such that for instance the `addObserver` method now takes a `ScreenObserver` as its only parameter. Similarly, the `notify` method now takes a `LineSubject` parameter. Due to the re-typing done by the PT mechanism, no casts are required, and all the code is statically known to be type safe.

Admittedly, this example is a simple instance of the more general class of subject/observer problems. However, the approach presented above applies (with minor variations) to the general case as well. For a more detailed exposition, with multiple interacting subjects and observers, the interested reader is referred to [3].

4 Related Work

The original description of the design patterns by the so-called "Gang of Four" (GoF) found in [9] comes with example implementations in C++. For the Singleton pattern, suggestions for reusable implementations are provided (for instance by relying on a registry in which singleton subclasses need to register themselves), but none of these provide the ease of reuse that the PT version provides.

For the Memento pattern, the GoF implementation makes use of C++'s `friend` construct, to allow only the originator access to the internals of the memento's state. Still, it does not provide the plugability that the PT version does.

The Observer implementation makes use of C++'s support for multiple inheritance to provide subject and observer classes for the pattern. Since we are targeting Java in this example, such inheritance hierarchies cannot be used (but we can, in this case, achieve the same end result, and more, with class merging). The GoF version of Observer also makes use of a special "trick" in which the observer knows the identity of the (single) subject for changes in which it is interested. Because of this they can avoid having to do a type cast that would otherwise have been necessary in the general case.

In *Design Pattern Implementation in Java and AspectJ* [10], Hannemann and Kiczales implement the design patters from the GoF in AspectJ, and contrast these implementations to their potential pure Java counterparts. The Singleton pattern is realized with an *around advice*, that wraps the constructor of the class in question to return a single instance every time it is called.

The Memento pattern may take advantage of AspectJ's possibilities for superimposing functionality onto existing classes. However, a general reusable implementation would be hard, if not impossible, to make without sacrificing type safety.

In the Observer implementation, the pattern is handled by one single abstract aspect, the `ObserverProtocol`. This aspect will exist as an entity at runtime, and contains a global map of observers to subjects. This is in contrast to the PT version, in which there is no notion of the aspect as a separate entity at runtime, but rather that the roles of the aspect are imposed on (and localized in) the classes that are to play the respective roles.

The AspectJ aspect defines empty "marker" interfaces, that conceptually declare the existence of the two roles of the pattern. However, since the interfaces are empty, in contrast to the PT version the aspect does not make it explicit which operations/behaviors belong to either participant in the pattern. In order to update the observer(s) when a change has occurred, a runtime cast must be made from the interface `Observer` to the class `Screen`, since the interface is empty and hence has no knowledge of the `Screen`'s `display` method.

In PT, the concretization happens through the `inst` statement (with optional addition classes for e.g. concretizing abstract members). The re-typing may in most cases eliminate the need for casts completely. The fact that pointcuts are not explicitly bound to join points, means that the AspectJ version is more prone to errors stemming from faulty pointcuts, while the PT version sacrifices some flexibility for a greater degree of pointcut safety.

In *Conquering Aspects With Caesar* [16], Mezini and Ostermann use the Observer pattern as a running example to address two main points with regards to the AspectJ implementation [10] discussed above; (I) the need for expressing aspects not as a monolithic entity, but rather as a set of interrelated, interacting modules, and (II) the need for flexible and reusable aspect bindings and implementations.

With respect to (I), Caesar achieves this through so-called *aspect collaboration interfaces* (ACIs). The ACIs are hierarchical, and may contain several (related) sub-interfaces. Each interface may describe a set of provided and/or required operations. The provided operations must be realized by implementers of the interface (e.g. the `addObserver` method must be implemented by an implementation of the `Subject` interface). This brings us over to point (II).

The required operations of an ACI must be implemented by a *binding* (which is separate from the implementation discussed in the previous paragraph), that binds the ACI to classes in the base program. Bindings may be defined independently of ACIs and their implementations, offering greater flexibility than the AspectJ counterpart.

In the PT example, the template is the rough equivalent of both an ACI and its implementation. We could, however, have used interfaces for the different pattern roles to separate actual implementation from public interface, but we are not obliged to.

A CasearJ binding can be compared to the `inst` statement (with optional addition classes) in PT, in the sense that required/abstract members will (or at least might) be concretized, and roles will be mapped to base program classes.

In *Explicit Programming* [4], the authors present an orthogonal approach in which design concepts (including patterns) may be added directly to the target *language* (i.e. Java). This allows, in principle, any pattern to be represented as a first class language construct, and this is exemplified with the Flyweight pattern. These constructs are implemented as transformations based on *textual* templates.

BETA [15, 14], gbeta [7] and J& (pronounced "jet") [18] are systems that in many ways are similar to each other, and in some 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. A distinguishing feature of PT compared to these three, is that it allows the developer to quite freely merge previously unrelated classes, and have the corresponding typed references re-typed accordingly.

5 Concluding Remarks and Future Work

We have extended the package template mechanism with a couple of desirable extensions, and shown how this can be used to implement components that may be plugged into an existing architecture with a minimal amount of "glue" required from the developer. Though exemplified here by three relatively simple design patterns, we believe that the constructs are usable for a broad range of problems, and that they may aid the developer in his/her continued strive for reusability and separation of concerns.

The AOP constructs introduced here should not be viewed as set in stone, as we acknowledge that the assumptions on which our restricted set of constructs are based may need some tweaking. An interesting topic in that respect is to investigate further the interplay between PT and AOP through experimenting with different degrees of AOP complexity with regards to e.g. pointcut scope and expressiveness, and see how different points along this axis pair up with inherent PT mechanisms such as re-typing and merging.

Another possible direction would be to see how far one could get with support for merging nested classes compared to what is attainable with the different approaches to virtual classes.

Furthermore, it would be clearly interesting to try to validate the applicability and usefulness of the constructs we have here, with regards to a broader set of real-world problems.

6 Acknowledgements

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 the anonymous reviewers for their comments, and Fredrik Sørensen and Birger Møller-Pedersen for valuable input and participation in work upon which this paper builds.

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] AspectJ Team. The AspectJ programming guide, 2003.

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

[4] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit programming. In *AOSD '02: Proc. 1st int. conf. on AOSD*, pages 10–18, New York, NY, USA, 2002. ACM.

[5] A. Colyer. AspectJ. In *AOSD*, pages 123–143. Addison-Wesley, 2005.

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

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

[8] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *AOSD*, pages 21–31. Addison-Wesley, 2005.

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

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

[11] Java platform, standard edition 6 api specification.

[12] S. Krogdahl. Generic packages and expandable classes. Technical Report 298, Department of Informatics, University of Oslo, 2001.

[13] 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.

[14] 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.

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

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

[17] Microsoft .NET Framework class library. URL: `http://msdn2.microsoft.com/en-us/library/ms229335.aspx`.

[18] 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.

[19] K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121 – 145, 2008.

[20] B. Pugh. The Java memory model. URL: `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

[21] J. Skeet. Implementing the singleton pattern in c#. URL: `http://www.yoda.arachsys.com/csharp/singleton.html`.

[22] M. Störzer and C. Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *EIWAS*, Berlin, Germany, September 2004.