A Reusable Observer Pattern Implementation Using Package Templates

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

Fredrik Sørensen
University of Oslo
Department of Informatics
Postboks 1080 Blindern, 0316
Oslo, Norway
fredrso@ifi.uio.no

Stein Krogdahl
University of Oslo
Department of Informatics
Postboks 1080 Blindern, 0316
Oslo, Norway
steinkr@ifi.uio.no

ABSTRACT

In this paper, we show how *package templates*, a new mechanism for code modularization, paired with a comparatively (and intentionally) small AOP mechanism may be utilized to create a reusable package for the Observer design pattern that can be plugged into an existing architecture with a minimum of "glue code".

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—*Patterns*

General Terms

Languages, Design

1. INTRODUCTION

The concept of design patterns [6] 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, infrastructure software, etc. 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# or Java is hard. This is at least in part due to limitations with regards to e.g. *extensibility of the subtyping relation* [13] and/or lack of support for mechanisms dealing with crosscutting concerns.

One commonly used design pattern is the Observer pattern. A few implementations utilizing various new language extensions already exist, notably one by Hannemann and Kiczales [8] utilizing AspectJ [2], and one by Mezini and Ostermann [11] utilizing the Caesar system [1].

ACP4IS'09, March 2, 2009, Charlottesville, Virginia, USA.

The package template (PT) mechanism [9, 10, 17] 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 templates may be merged to form one new class.

In this article, we utilize package templates with a comparatively (and intentionally) small and simple aspect oriented extension to provide a reusable package for the Observer pattern, and compare our approach to the other solutions mentioned above.

2. OVERVIEW OF THE PT 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.

A package template looks like a regular Java package, but we will use a syntax where curly braces enclose the contents of both templates and regular packages., e.g.:

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

Templates may have (constrained) type parameters, such as `E` above, but this will not be treated in any detail in this paper. Apart from this and a few other constructs (such as the extensions we propose in Section 2.1), 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 U {
  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 `U`. In its simplest form, the `inst` statement just names the template to be instantiated, e.g. `inst T`. 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 expansion classes (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); }
}
```

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 furthermore how both m1 and m2 now have signatures of the form MergeAB → MergeAB. We shall use a similar construct in Section 3.

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.

2.1 AOP Extensions

We here extend the basic PT mechanism described above with a minimal set of constructs for aspect-oriented programming. Thus, we consider a restricted version of common AOP concepts, that paired with the inherent possibilities in PT for e.g. merging and expanding classes may provide some of the power and flexibility found in "traditional" AOP languages.

"Aspect-oriented programming is quantification and obliviousness" [5] is often seen as an imperative quote for what AOP is or should be. However, in this paper we wish to investigate the possible benefits of a slightly different approach. To begin with, 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. In practice, this means that the possible changes to a base program by a (conceptual) aspect will be limited in scope by a corresponding template instantiation. Furthermore, it means that the pointcuts are local to the defining class, including its subclasses (given the appropriate modifier), and that they may only refer to members within the defining class or any of its superclasses, following the OO principle of encapsulation. Hence, they may also be refined or redefined by subclasses or in addition clauses. The same will apply to advice.

This may seem like a severe restriction compared to e.g. AspectJ, but we believe that paired with the flexible tailoring mechanisms offered by PT (in particular the merging possibilities), this provides a sufficiently powerful and expressive construct for many purposes.

Given that all pointcuts will refer to local members, *quantification* using wild-cards over member or type names should not be as necessary as in e.g. AspectJ. Therefore, we find it worthwhile to explore the disallowing of wild-card usage with regards to member names, and rely instead on explicit join point specification. This will result in a mechanism where the pointcuts do not specify a pattern to be matched against join points, but instead an actual *binding* to join points.

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. 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:

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

To keep this exposition short, we will not discuss the EBNF or the generated language in any further detail, but rather turn to a small example to illustrate how the mechanism works. Consider the following template:

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

The class A contains a pointcut that matches calls to the 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 refines 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, and this would hold even if m1 was renamed in the instantiation. This is made possible by the fact that pointcuts are not string patterns, but actual bindings, as mentioned above.

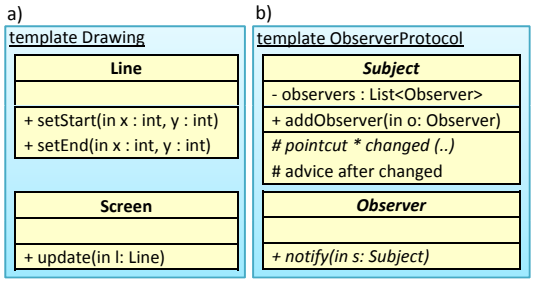

Figure 1: *a) The Drawing template, and b) the roles of the Observer pattern*

3. THE OBSERVER PATTERN EXAMPLE

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.

3.1 Single Subject/Single Observer Classes

To exemplify the use of the pattern, we consider a package for drawing objects on a screen (strongly resemblant of the examples in [8] and [11]). In this section, we look at an example with only two classes, `Screen` and `Line`, as shown in Figure 1a. When a line changes its length or position, the screen 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 Observer pattern as shown in Figure 1b with the following code[1]:

```
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 "`..`". Using the drawing package of Figure 1a, we could do the instantiation as follows:

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

[1]The removeObserver method is trivial and hence omitted for brevity. The classes in the diagrams contain a fourth compartment for AOP related members where this is relevant, and the names of abstract classes and members are written in italics.

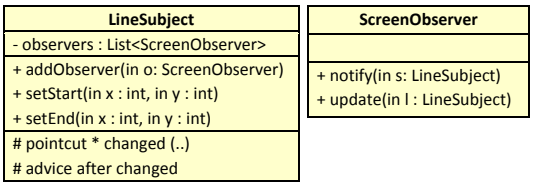

Figure 2: *The resulting classes from the merge of the ObserverProtocol and Drawing templates*

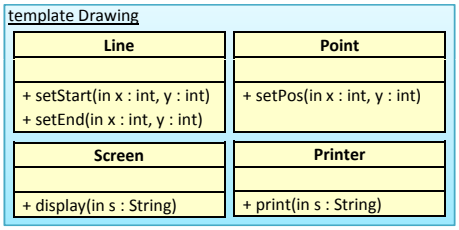

Figure 3: *The drawing template with two potential subjects and two potential observers*

```
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 `adds` 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 2. 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 retyping done by the PT mechanism, no casts are required.

3.2 Multiple Subject and/or Observer Classes

In the previous section we looked at a rather simple scenario in which there was only one class having the subject role, and one having the observer role. In this section, we will look at the more general problem, with multiple classes playing the roles of subjects and observers.

To exemplify, we extend the template `Drawing` such that there are several classes acting, respectively, as subjects and observers, and hence we need to modify our instantiation code a little. The good news, however, is that our implementation of the Observer pattern itself needs no modification. The classes of the new `Drawing` template are shown in Figure 3. Applying our Observer pattern to this package, we would use the following instantiation:

```
inst Drawing with
  Screen => ScreenObserver, Printer => PrinterObserver,
  Line   => LineSubject,    Point   => PointSubject;
inst ObserverProtocol with
     Observer => BaseObserver, Subject  => Figure;
```

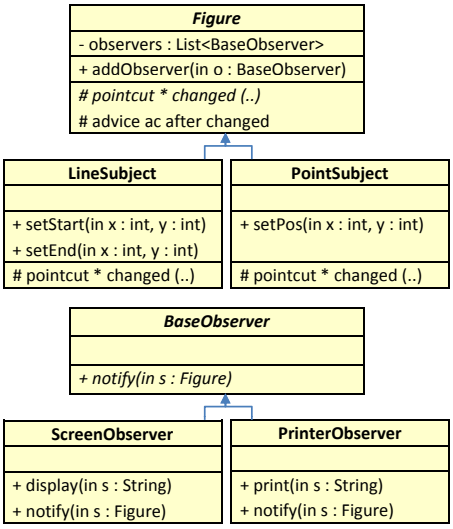

Figure 4: *The resulting classes from the merge of the ObserverProtocol template and the new version of the Drawing template*

The `Drawing` template does not contain any base classes into which the required functionality for subjects and observers can be merged. However, PT allows us to *introduce* superclasses directly, in a type safe manner, as shown in the instantiations above and the addition classes below:

```
abstract class BaseObserver adds {}
class ScreenObserver extends BaseObserver adds {
 void notify(Figure f) { display(f + " has changed"); }
}
class PrinterObserver extends BaseObserver adds {
 void notify(Figure f) { print(f + " has changed"); }
}
abstract class Figure adds {}
class LineSubject extends Figure adds {
 pointcut * changed(..) {call(setStart) || call(setEnd)}
}
class PointSubject extends Figure adds {
 pointcut * changed(..) { call(setPos) }
}
```

The resulting package is shown in Figure 4. Note how the functionality for the Subject and Observer roles are distributed to their respective functional counterparts from the `Drawing` package through the newly introduced common superclasses.

4. RELATED WORK

The original description of the Observer pattern by the so-called "Gang of Four" (GoF) found in [6] comes with an example implementation in C++. This 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.

The GoF version 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 have been necessary in the general case.

In *Design Pattern Implementation in Java and AspectJ* [8], Hannemann and Kiczales implement the design patters from the GoF in AspectJ, and contrast these implementations to their potential pure Java counterparts. The Observer pattern is used as a running example. In their 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, the aspect does not make it explicit which operations/behaviors belong to either participant in the pattern.

The abstract aspect is concretized via inheritance as concrete aspects, that override the abstract pointcut `subjectChange`, and define which concrete classes should be designated as either subject or observer through the AspectJ `declare parents` construct. Different concrete aspects may be defined based on the abstract aspect. 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 retyping may in many 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 (the *fragile pointcut problem* [16]), while the PT version sacrifices some flexibility for a greater degree of relative pointcut safety.

AspectC++ [15] adds full-blown AOP support to the C++ language, including support for templates and non-OO programming. Contrary to our approach, the authors explicitly state as one of their goals to *"not make any compromise regarding obliviousness and quantification"*, i.e. to not put any restrictions on their language with regards to the definition from [5]. This provides for a very powerful paradigm, which pairs up well with the nature of C++ itself.

One of the examples in this article shows a reusable Observer pattern implementation. It makes use of C++'s multiple inheritance capabilities and a distinguishing join point API to add the required members to the classes designated as subject and observer, respectively. The solution is similar to ours in the respect that only pointcuts and the update method for the observer need to be concretized for a particular utilization, yet differ in the sense that while our implementation syntactically distributes these members to their respective classes, the AspectC++ version keeps everything in a central aspect.

In *Conquering Aspects With Caesar* [11], Mezini and Ostermann show an alternative implementation of the pattern, utilizing the CaesarJ system. The work mainly addresses two points with regards to the AspectJ implementation 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 op-

erations. 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 subjects and observers 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 be concretized, and roles will be mapped to base program classes.

Like in Caesar (and in contrast to AspectJ), aspects in PT must be *explicitly deployed*. That is, the mere inclusion of an ACI or a package template that contains pointcut and advice in the compilation process does not alter any part of the base program. For this to happen in PT, an explicit `inst` statement must be present. Similarly, Caesar employs the `deploy` keyword for much the same purpose, with a couple of important distinctions: Caesar supports aspectual polymorphism and dynamic as well as static deployment of aspects, while we (at present) only support static instantiation of templates. However, since every instantiated template class will correspond to an actual class in the base program, and pointcuts and advice reside within ordinary classes, ordinary OO polymorphism may possibly be used in such situations, though we have not yet explored this issue in any detail.

Concerning the PT mechanism itself, several others have defined mechanisms that in some ways are similar (disregarding the AOP extensions presented here). Examples are J& [12], Classboxes [3], Traits [14] and Mixins [4]. For a more thorough treatment of their respective similarities and differences compared to PT, the interested reader is referred to [17].

5. CONCLUSION AND FUTURE WORK

We have suggested how package templates extended with a (compared to its contemporaries) rather minimal AOP mechanism may provide a sufficiently powerful framework for implementing reusable components that map to interdependent entities in a base program. Exemplified here through the Observer design pattern, this is clearly an area that we hope to generalize and expand on.

One 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.

Furthermore, it would be interesting to try to assess how applicable and useful the constructs we have used here really are when applied to a broader set of real-world problems. With that in mind, it would be interesting to relate the findings from [7] to an implementation of the same patterns in PT.

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 valuable comments.

7. 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] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: controlling the scope of change in java. In *OOPSLA '05*, pages 177–189, New York, 2005. ACM.

[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] 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.

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

[7] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05*, pages 3–14, New York, 2005. ACM.

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

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

[10] 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, 2009.

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

[12] 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.

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

[14] 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.

[15] O. Spinczyk and D. Lohmann. The design and implementation of AspectC++. *Know.-Based Syst.*, 20(7):636–651, 2007.

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

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