Evolvable Pattern Implementations need Generic Aspects

Günter Kniesel, Tobias Rho
Dept. of Computer Science III, University of Bonn
Römerstr. 164, D-53117 Bonn, Germany
{gk,rho}@cs.uni-bonn.de

Stefan Hanenberg
Dept. of Comp. Science and Inf. Systems, University of Duisburg-Essen
Schützenbahn 70, 45117 Essen, Germany
shanenbe@cs.uni-essen.de

Abstract

Design patterns are a standard means to create large software systems. However, with standard object-oriented techniques, typical implementations of such patterns are not themselves reusable software entities. Evolution of a program into a 'patterned' form (also known as 'refactoring to patterns') and subsequent evolution of a 'patterned' design is largely left to the programmer.

Due to their ability to encapsulate elements that crosscut different modules, aspect languages have the potential to change this situation. For many interesting patterns, a large part of the process of refactoring to patterns can already be implemented modularly in aspects. Still, existing aspect languages can only express a small number of typical patterns implementations in a generally *reusable* way. In many cases, evolution of an application that uses one pattern variant into one that uses another one cannot be achieved at all. In others, it requires duplicating parts of the aspect implementation, thus creating scattered code in the aspects and hindering their further evolution.

In this paper, we argue that aspect languages need to provide genericity in order to support reusable pattern implementations. We sketch the main features of the generic aspect language *LogicAJ*, and show how it supports software evolution. In particular, we demonstrate how *LogicAJ* enables evolution of a non-patterned implementation to a patterned one and easy transition from one patterned implementation to another.

1. Introduction

Design patterns [5] are a standard means of creating large software systems. Catalogues like for example [5, 1, 14] permit developers to benefit from the successful application of certain design elements for a given problem. This increases the quality of software, its comprehensibility and maintainability.

However, with standard object-oriented techniques, typical implementations of design patterns are not themselves reusable software entities. Hence, applying a certain design pattern typically means hand-crafting a number of software elements and embedding them invasively into the application, which is a error-prone process. For example, even the typical Java implementation of a simple design pattern like Singleton is not trivial (cf. [6], pp. 127-133). Furthermore, most patterns are much more complex than Singleton so that their implementations require the developer to perform such invasive changes coherently in a number of classes in the system.

The initial implementation of a pattern, however, is just one of many evolution steps of an application. In general, invasive, non-local changes are required whenever

- an application's implementation is modified to incorporate an initial pattern implementation (refactoring to patterns),
- an existing pattern implementation is to be modified in order to synchronize it with the evolution of the application,
- one pattern implementation variant is to be replaced by another,
- the applicaton of a pattern is to be removed completely from the design and implementation of a system.

Aspect-oriented software development [10] is a promising approach to tackle these evolution scenarios. The application of aspect-oriented languages promises modular implementation of typical design patterns, thus reducing the need for invasive non-local changes.

However, it turns out that current aspect-oriented languages do not fully live up to this promise. For example [7] shows that only a number of typical design pattern implementations from the catalogue in [5] can be implemented in a reusable manner in the aspect-oriented language *AspectJ* [11]. As a remedy, the concept of parametric introductions in the language *Sally* is proposed in [8].

In this paper, we start from the observation that the parametric introductions are a first step towards a generic aspect language but still too restricted to provide a *general* solution for reusable and evolvable pattern implementations. This leads us to the conclusion that genericity must be supported uniformly across an aspect language, not just for member introductions. In support of our thesis we review the fully generic aspect language *LogicAJ*, proposed in [16], and show how it enables reusable and easy to evolve implementations of design pattern variants that cannot be achieved otherwise.

Section 2 illustrates typical pattern-related evolution problems on the concrete example of the decorator pattern. Section 3 describes the main features of the uniformly generic aspect language *LogicAJ*, initially proposed in [16]. Section 4 shows how the two presented variants of the Decorator pattern can be implemented easily using *LogicAJ*'s functionaliy. Section 5 concludes.

2. Addressed Evolution Problems

In this paper we address the problem of non-local, invasive changes that occur in the initial implementation and subsequent evolution of "patterned" designs. In this section we illustrate these problems on two variants of the decorator pattern.

2.1. Evolution Scenario 1: Initial Decorator Implementation

The decorator pattern provides a flexible alternative to subclassing. Additional functionality can be added to an object dynamically. Figure 1 shows the UML diagram for the basic variant of the decorator[1]. There are four participant roles in the decorator pattern: *Component* defines an interface for the objects to which additional functionality should be attached. *ConcreteComponent* classes implement such objects. The *Decorator* aggregates a *Component* instance and implements the *Component* interface by forwarding messages to this instance. The *ConcreteDecorator* adds functionality to *Component*.

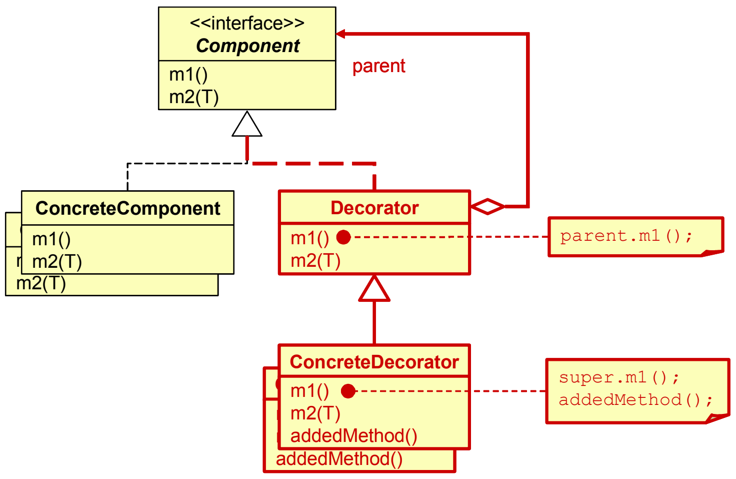


Figure 1Implementation schema of decorator pattern from [5][1]. The parts in red need to be added to an existing code base in order to implement the pattern

As illustrated by the red highlighting in Figure 1 a *Decorator* and many *ConcreteDecorator* classes must be added to an existing implementation. All of them are dependent on the contents of the *Component* interface. If in the existing code base *Component* is not an interface but a class that already contains own state, we additionally

[1] The implementation of the decorator pattern listed in [5] uses inheritance between *Decorator* and *Component* at the expense of undesired inheritance of state. Here, we only assume that *Decorator* implements the *Component* interface.

need to apply first an ExtractIntreface refactoring [4] to create the *Component* interface. This includes changes to type declarations throughout the program [15].

When faced with these implementation tasks, programmers are lucky if they poses tools that automate some of the tasks. For instance, various CASE tools generate code templates for certain patterns and provide a pool of automated refactorings. Still, the basic problem remains: Even after application of a code generating tool, the implementation of the pattern is spread over many classes and tightly interwoven with application specific code, e.g. the addedMethod() in Figure 1. This becomes a problem if the patterned application is to be evolved further.

2.2. Evolution Scenario 2: Evolving the Decorator Implementation

Now assume that, in an application built using the decorator pattern, new requirements imply the need to let decorators additionally *override* the behaviour of a component on a per-instance basis. This means that if the decorator has own implementations of the component's methods that implementations should be used also by the component itself. In a strict sense, we now don't have a Decorator anymore. However, no matter how we name the problem, it remains the same: how to evolve the existing implementation when faced with this new requirement.

For overriding to be effective the component must be aware of the fact that it is working on behalf of another object that might provide own implementations of certain methods. One possible solution is to use *back-references*. These can be implemented as an additional parameter (cf. [12, 9]). Every method inside the component gets a new parameter `self` and every (explicit or implicit) occurrence of `this` is replaced by `self`[2]. The corresponding design of *Component* is illustrated in Figure 2.

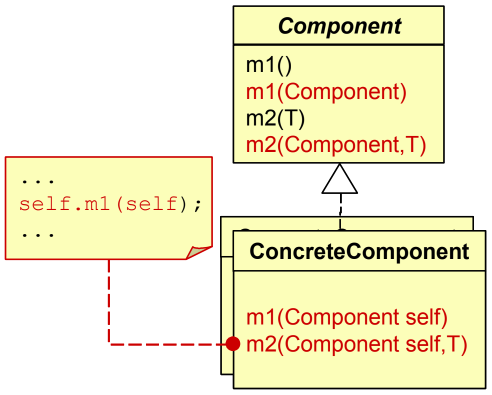


Figure 2 Evolution step: Implementation of instance-level overriding via back reference parameters.

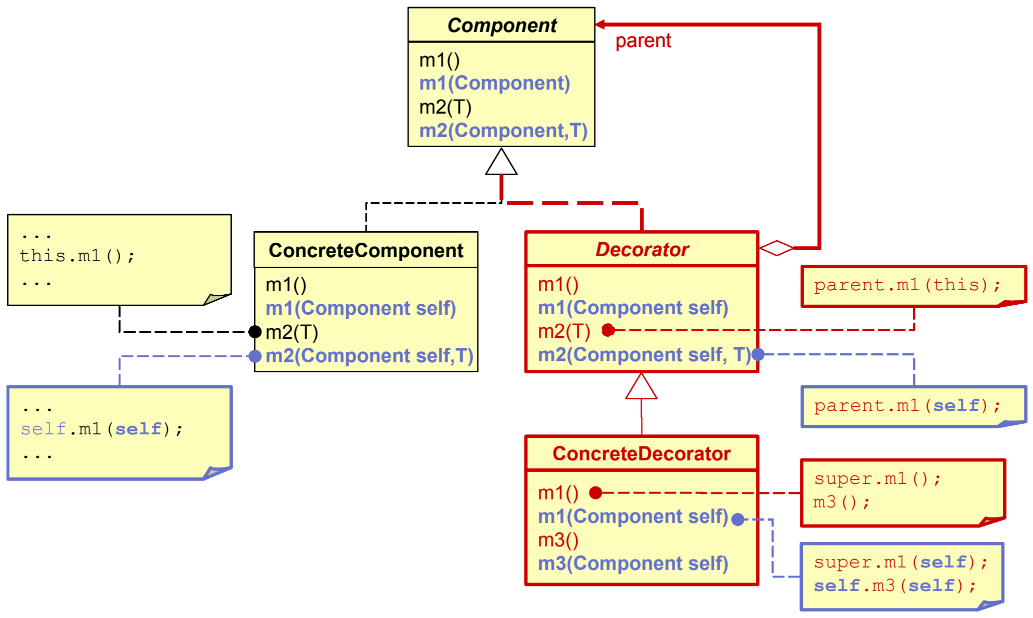
However, this design decision implies a synchronization of the design pattern implementation with the new design of *Component*. Figure 3 illustrates how the whole decorator hierarchy needs to be adapted. Comparison of Figure 1 and Figure 3 shows that the move from the basic decorator to the one with back references involves extensive changes in the design and implementation:

1. Every method in the *Component* interface must be extended by an additional parameter, *self*, for the back reference to the forwarding object. We call such methods *delegatee methods*.
2. Delegatee methods must be included in all subtypes of *Component*. In their body, all messages to *this* must be replaced by messages to *self.*
3. Invocations of original methods must be replaced by invocations of delegatee methods *throughout the program*.
4. For the sake of clients that cannot be adapted, every original method must be preserved.
5. Forwarding methods in *Decorator* now must pass the correct value of the back reference (either *this* – in an original method, or *self* – in a delegatee method).

The result of this simple unanticipated evolution is a number of invasive changes in a number of different modules: large effort is necessary to keep existing design pattern implementations in sync with other design decisions. In practice, it is almost impossible to guarantee such a correct synchronization.

In the next section we introduce LogicAJ, a generic aspect language that can implement both discussed evolution scenarios modularly, in aspects. All the addressed problems are then dealt with by the aspect weaver.

2 For a thorough discussion of the issues involved in a simulation of object-based overriding via back references see [13].



```
...
this.m1();
...
```

```
...
self.m1(self);
...
```

```
parent.m1(this);
```

```
parent.m1(self);
```

```
super.m1();
m3();
```

```
super.m1(self);
self.m3(self);
```

Figure 3Evolution impact: Decorator pattern with back reference implemented as an additional method parameter. Blue highlighting indicates the code affected by the change.

3. LogicAJ

LogicAJ is an extension of *AspectJ* [11]. In *LogicAJ* pointcuts, introductions, and advices are parameterized by logic variables that can range over packages, types, fields, and methods, including method signatures and method bodies. Put differently, logic variables can be used in any place where in *AspectJ* aspects it is legal to use packages, types, fields, and methods.

Logic variables are denoted syntactically by names starting with a question mark "?"[4]. From a semantics point of view they have two essential properties. First, their values cannot be set explicitly but only by the evaluation of conditions in joinpoint expressions. Second, every mention of the same logic variable name within a scope represents the same value. Thus it is possible to refer later to a value matched earlier. In particular, it is possible to create new code based on previous matches. In this respect, logic variables are more expressive than "*" pattern matching in *AspectJ*, where two occurrences of "*" do not represent the same value.

Generic aspects are particularly useful for expressing crosscutting changes that follow a common structure but differ in the names of created or modified entities. An example is the creation of forwarding and delegatee method signatures in the previous section. Another simple example is the use of mock objects. This is a common technique for narrowing down the potential sources of a failure during testing. Its essence is the replacement of some of the tested classes by mock classes, which provide a fixed expected behavior during tests.

```
usrMng = new UserManager(...);
...
dbMng = new DBManager(...);
```

MockAspect

```
usrMng = new UserManagerMock(...);
...
dbMng = new DBManagerMock(...);
```

Below we show a *LogicAJ* implementation of mock objects. The aspect replaces each constructor call with a call to the respective constructor of the associated mock class, if the mock class exists.

```
aspect MockAspect {
    Object around(?mock, ?args, ?class) :
                    // Intercept constructor invocations. Bind ?class to the name of the instantiated class
                    // and ?args to the argument list of the invocation
        call(?class.new(..)) && args(?args) &&
                    // Check if a class with name ?class+"Mock" exists
        concat(?class, "Mock", ?mock) && class(?mock)
    {               // return instance of mock class (includes weave time check for constructor existence)
        return new ?mock(?args);
    }
}
```

[4] LogicAJ shares this syntax with Sally [8] and TyRuBa [2, 3].

The example illustrates the syntax of logic variables, their binding by the evaluation of conditions, and their use in the assembly of generic advice code. The pointcut part of the advice uses three predicates, *call*, *args* and *concat*. The *call* predicate is basically the *call* pointcut of *AspectJ*. The *args* predicate is an extension of the *args* pointcut of *AspectJ* in that the logic variable ?*args* passed as an argument to the pointcut can match an entire argument list. Here it matches all arguments of any constructor invocation. The semantics of the *concat* predicate is that the third argument is the concatenation of the first and the second. It is used here to create names of mock classes by appending the suffix "Mock".

Logic variables also enable generic introductions. With generic introductions, the members to be introduced and the types into which they should be introduced can be determined by the evaluation of predicates. It is also possible to introduce new types. Like parametric introductions in [8], generic introductions generalize the concept of advice in AspectJ. This is reflected in their syntax, which is largely advice syntax except for the keyword "introduction" instead of "advice". We will see various examples of generic introductions in the next section.

4. Evolvable Decorator Pattern Implementation – Solution

In this section we show how the two variants of the decorator pattern introduced in Section 2 are modelled in the generic aspect language LogicAJ in a modular, reusable and evolvable way. As we proceed, we identify different limitations of previous approaches and show in each case how they are overcome in LogicAJ. The examples underpin our conclusion that evolvable pattern implementations need generic aspects.

4.1. Basic Decorator Pattern in LogicAJ

This section models a reusable basic variant of the decorator pattern implementation illustrated in Figure 1. In the following, we assume that an interface playing the *Component* role and classes playing the *Concrete-Component* role exist[6], are implemented in plain Java and are used as base classes for the aspects. The class playing the role of *Decorator* is generated by an abstract aspect AbstractDecorator. Classes playing the *ConcreteDecorator* role are created by concrete aspects derived from AbstractDecorator.

The AbstractDecorator Aspect

This aspect has a double function, as a repository of shared pointcut definitions and as the place where the classes playing the role of decorator are created (Figure 5).

The decorator pattern can be instantiated multiply in an application. Every instantiation is specific for a particular *Component* type. In order to express this dependency, the class playing the role of the *Decorator* in a particular pattern instantiation is generated based on the participant *Component*. Its name is determined by adding the postfix Decorator to the name of the interface that plays the *Component* role. This dependency is abstracted in the decorator pointcut. The abstract pointcut component must be implemented in a concrete aspect, in order to trigger application of the aspect to a particular part of the program.

```
abstract aspect AbstractDecorator {

   abstract pointcut component(?component);

   pointcut decorator(?decorator) :
      component(?component) &&
      concat(?component, Decorator, ?decorator);

   introduce(?component, ?decorator) : ... // see Figure 5

   introduce(?decorator, ?rettype, ?params, ?name) : ... // see Figure 6
}
```

Figure 4 `AbstractDecorator` aspect. Participant roles are abstracted as pointcuts. A *Decorator* class specific for a particular *Component* type is generated by two generic introductions.

[6] The Component interface could be automatically created by an extract interface refactoring on the Concrete Component classes.

Generic Type Introduction for *Decorator* Role

The AbstractDecorator aspect creates a class playing the role of the *Decorator*. This is done via a generic type introduction as shown in Figure 5. Based on the above pointcuts, the name of the *Component* and *Decorator* is determined and bound to the logic variables ?*component* and ?*decorator*. Then the ?*decorator* class is generated and declared to be a subtype of ?*component*. It contains an instance variable parent of type ?*component* to which messages can be forwarded.

```
introduce(?component, ?decorator) :
   component(?component) && decorator(?decorator)
{
   abstract class ?decorator implements ?component {
      protected ?component parent;
   }
}
```

Figure 5 Generic type introduction

Note that the generic type introduction, that is, the ability of creating types in an aspect and defining their name depending on the current context[7], is essential for our example. We know of no other aspect language that provides this functionality.

Generic introduction of forwarding methods

The next step is the creation of one forwarding method in the *Decorator* class for every method in the *Component* type[8]. This is done via a generic introduction. In its condition part, it checks for methods that exist in the Component type. Then it creates methods with exactly the same signature in the *Decorator* class. Every created method forwards its invocation to the object reachable via the parent reference. The method creation process is repeated for all values of logic variables that make the condition part (the pointcut) true. Thus in the end, the *Decorator* class will have one forwarding method for every method from the *Component* interface.

```
introduce(?decorator, ?rettype, ?methodName, ?params) :
   component(?component) &&
   decorator(?decorator) &&
   method(public ?rettype ?component.?methodName(?params))
{
   ?rettype ?decorator.?methodName(?params) {
      parent.?methodName(?params);
   }

}
```

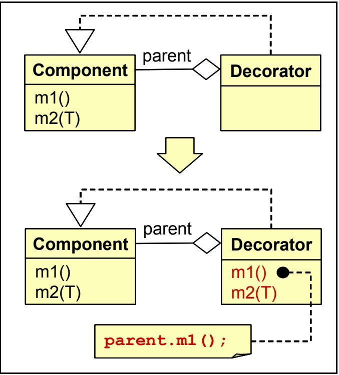


Figure 6 Generic introduction of forwarding methods.

Without generic introductions it is not possible to write *one* introduction that creates *different* methods depending on the context. In AspectJ, for instance, one needs to know the complete signature of the methods to be introduced when writing the aspect. Thus, [7] concludes that no reusable implementation of decorator is possible with AspectJ. The concept of parametric introductions in [8] is very similar to generic introductions. It allows creation of methods with heterogeneous signatures and homogeneous implementations in different contexts. However, the ability to create context-dependent signatures *and* implementations, used above, seems to be unique to LogicAJ.

Instantiation of the pattern

The variant of the decorator pattern implemented in the AbstractDecorator aspect is instanttiated by the creation of a concrete subaspect that supplies the missing pointcut definition and the implementation of *ConcreteDecorator* classes (Figure 7).

[7] In this case, the context is determined by the value of ?*component*.

[8] For simplicity, we often identify roles with the classes that play the roles, when the meaning is clear from the context. For instance, we simply say *Decorator* class instead of class that plays the *Decorator* role. For the same reason, we use the role names as class names in all diagrams.

```
aspect MyComponentDecorator extends AbstractDecorator {

  pointcut component(?component) : equals(?component, MyComponent);

  introduce(?decorator) : decorator(?decorator)
  {
     public class ConcreteDecorator1 extends ?decorator {
        public void m () { /* possibly calling super.m()  */ }
        public void m3() { /* possibly calling super.m3() */ }
     }
     // ... more ConcreteDecorator classes ...
  }
}
```

Figure 7 Instantiation of the decorator aspect for `MyComponent` triggers the creation of the `MyComponentDecorator` class and its subclasses

Since the implementation of the *ConcreteDecorator* classes consists almost entirely of plain Java code, one might wonder why we define them in an aspect. The reason is that in a base class that does not declare the extends relationship to the ?*decorator*, invocations of super would not compile. Bases classes that declare the extends relation, however, would be tightly dependent of the naming convention for decorators. The solution in Figure 7 preserves separate compilation and encapsulates the knowledge about decorators in the aspect hierarchy.

4.2. Decorators With Back References in LogicAJ

In section 2 we have shown that the manual evolution of an application that uses the above implementation variant of the decorator pattern can be extremely costly and error-prone. Now we show that the same evolution step can be achieved in LogicAJ incrementally, by the definition of further subaspects of AbstractDecorator. The main work is performed in the AbstractDecoratorBR aspect. It includes:

1. Creation of delegatee methods in *Component* and all its subtypes (via a generic method introduction).
2. Replacement of messages to *this* by messages to *self* in all delegatee methods (via a generic around advice).
3. Replacement of invocations of original methods by invocations of delegatee methods throughout the program (via a generic around advice).
4. Passing of the correct value of the back reference in *Decorator*: either *this* – in an original method, or *self* – in a delegatee method (via a generic around advice).

Creation of delegatee methods

For every subtype ?*sub* of *Component* and each method in that subtype, the generic introduction in Figure 8 adds to ?*sub* a method with exactly the same body but an additional first parameter, self, of *Component* type.

```
introduce (?comp, ?sub, ?rettype, ?name, ?params, ?body) :
    component(?component) &&
    subtype(?component, ?sub) &&
    method(?rettype ?sub.?name(?params) ?body )
{
    method(?rettype ?sub.?name(?component self, ?params))
       ?body
}
```

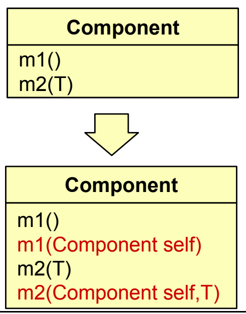


Figure 8 Introduction of delegatee methods in all subtypes of *Component*

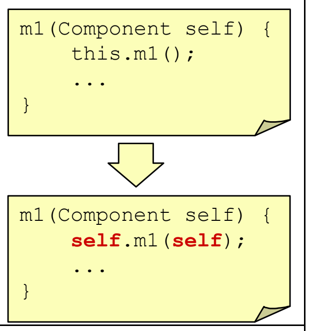
Note that in the above example it is essential that logic variables can also range over unnamed entities, in this case over method bodies. We need the ability to pass a value for the ?*body* variable from the method pointcut to the advice in order to copy the existing method body into the delegatee method.

Replacement of messages to `this`

Consistent use of the additional self parameter in the copied method body is enforced by the generic advice shown in Figure 9. For all subtypes ?*sub* of *Component* it determines within delegatee methods all calls of original methods from *Component* that are invoked on this. These calls are replaced by calls on self, thus enabling execution of the respective method in the self object.

The advice uses two auxiliary pointcut definitions, withinDelegateeMethodInType and callOfComponentMethodOnThis. The first one defines that we are in a delegatee method if the method's first parameter has type *Component* and there is another method in the same type with the signature resulting from deleting the delegatee method's first parameter [10]. The second one selects all invocations of original methods from *Component* and filters those that are invoked on this, using the new pointcut receiverIsThis(). It checks whether the message receiver at the current joint point is the enclosing instance. The pointcut binds the name and arguments of the filtered method calls to the logic variables ?*name* and ?*args*. These are used in the advice to generate the new code.

```
around(?component self, ?name, ?args) :
     component(?component) &&
     subtype(?component, ?sub) &&
     withinDelegateeMethodInType(?sub, self) &&
     callOfComponentMethodOnThis(?sub, ?name, ?args)
  {
     self.?name(?args);
  }
```



```
m1(Component self) {
    this.m1();
    ...
}
```

```
m1(Component self) {
    self.m1(self);
    ...
}
```

```
 pointcut withinDelegateeMethodInType(?type, ?type self) :
     delegateeMethod(?type, ?rettype, ?name, [?comp|?params]) );
     withincode(?rettype ?type.?name(?comp self, ..)) ;

 pointcut callOfComponentMethodOnThis(?inType, ?name, ?args) :
     originalMethod(?rettype, ?name, ?params) &&
     call(public ?rettype ?inType.?name(?params)) &&
     receiverIsThis() &&
     args(?args) ;

 pointcut delegateeMethod( ?sub, ?rettype, ?name, [?comp|?params]) :
     component(?comp) &&
     subtype(?component, ?sub) &&
     method(?rettype ?sub.?name(?params)) &&
     method(?rettype ?sub.?name(?comp self, ?params) );

 pointcut originalMethod(?rettype, ?name, ?params) :
     component(?comp) &&
     method(?rettype ?comp.?name(?params)) &&
     method(?rettype ?comp.?name(?comp self, ?params) );
```

Figure 9 Implementation of "self delegation" by replacing messages to "this".

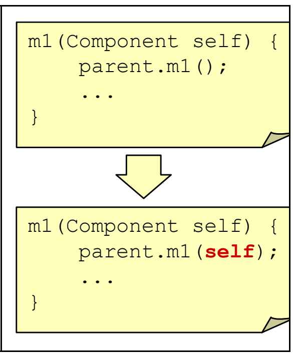
Forwarding with passing of back reference

In order to ensure that the self back reference has the proper value, we must extend the code of the class that plays the *Decorator* role. Each of its forwarding methods must pass on the current value of self to the parent object. In Figure 10 the forwarding pointcut identifies all places where a method invokes itself on the parent object. The advice adds the self parameter to the forwarding invocation.

[10] In the real implementation one would use additional criteria, e.g. a particular naming scheme of delegatee methods, in order to avoid false matches.

```
around(?comp parent, ?comp self, ?name, ?args) :
      forwarding(?comp parent, ?comp self, ?name, ?args)
{
      parent.?name(self, ?args);
}

pointcut forwarding(?comp parent, ?comp self, ?name, ?args):
      component(?comp) &&
      decorator(?decor) &&
      withincode(?rettype ?decor.?name(?comp self, ..)) &&
      call(public ?rettype ?decor.?name(?args)) &&
      target(parent) ;
```



```
m1(Component self) {
    parent.m1();
    ...
}
```

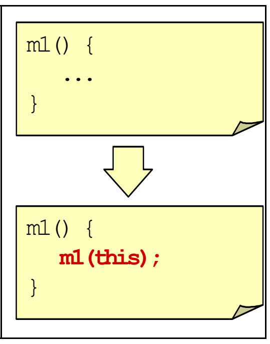
```
m1(Component self) {
    parent.m1(self);
    ...
}
```

Figure 10 Forwarding with passing of back reference

Use of delegatee methods

Figure 11 shows the redirection of the normal method execution to the execution of delegatee methods. The entire body of every original method is replaced by an invocation of the method's delegatee version, with `this` as the value of the first argument. This redirection is applied to every subtype of *Component*.

```
around(?decorator inst, ?rettype, ?name, ?args) :
   decorator(?decorator) &&
   subtype(?decorator, ?sub) &&
   originalMethod( ?rettype, ?name, ?params) && // see Figure 9
   execution(public ?rettype ?sub.?name(?params)) &&
   this(inst) &&
   args(?args);
{
   inst.?name(inst, ?args);
}
```



```
m1() {
    ...
}
```

```
m1() {
    m1(this);
}
```

Figure 11 Redirection of normal method invocations to delegatee methods

The examples shown in Figure 9 to Figure 11 illustrate various cases in which availability of generic advice was essential in order to express the intended semantics.

4.3. Results

The example studied in this section shows that the initial implementation of a decorator-based design and the subsequent and adaptation implementation to new requirements can be performed completely at the level of aspects. In the latter case, the only change that we need in addition to the implementation of the AbstractDecoratorBR aspect demonstrated above, is to turn the concrete subaspects of AbstractDecorator into subaspects of AbstractDecoratorBR. This ensures that after the next weaving, the back reference based implementation of the decorator pattern will be consistently available in the application instead of the basic implementation. No single line of code has to be changed manually in the base classes to achieve this result.

From a software evolution point of view, it is also worthwhile noting that there is no code duplication between the aspects that implement the decorators with back reference and the ones that implement the basic decorator schema. Genericity eliminates the need to repeat any code that would be identical up to the names of referenced entities. This eases evolution of the aspects themselves.

Looking back to our small case study from a language designer's point of view, we note that we needed the joint expressiveness of generic type introductions, generic member introductions, and generic advice. It was essential that logic variables ranged over all named entities of the host language (from types to arguments) plus some unnamed entities (in the case of method bodies, see Figure 8).

We conclude that non-tangled, reusable implementations of patterns and other highly parametric concepts require genericity at almost *every* level of a language. What is the "right" degree of genericity, is a question that we would like to discuss with the workshop participants.

5. Conclusions

Object-oriented design patterns are a standard means to create more dynamic and easier to evolve systems. However, with standard object-oriented techniques, pattern *implementations* are not themselves reusable software entities. Therefore, initial implementation and subsequent evolution of a "patterned" design *beyond* its anticipated variation points can be arbitrarily difficult. Synchronizing the pattern implementation with changes in the design of the application, switching to another implementation variant for a particular pattern, and combination of multiple patterns within one application typically require extensive changes in a design and code base.

In this paper we have shown that *generic aspect languages* are a promising solution. Their characteristic is the use of *logic variables* for program entities (packages, types, fields, methods, parameter lists, argument lists, method bodies, etc.), which enables expressing generic transformations of a program which can be performed subject to generic conditions.

In particular, we described *LogicAJ*, a generic aspect language design for Java that provides logic variables ranging from types and packages down to the level of individual method invocations, method arguments and method bodies. Using the decorator pattern as an example we have demonstrated the expressive power of the resulting language design and in particular, how it fosters reusable and evolvable implementations.

6. References

[1] Buschmann,F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stahl, M: *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, 1996.

[2] De Volder, K.; D'Hondt, T.: *Aspect-Oriented Logic Meta Programming*, Pierre Cointe (Ed.): 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), Saint-Malo, France, July 19-21, 1999, LNCS 1616, Springer-Verlag, 1999, pp. 250-272.

[3] De Volder, K.: *Type-Oriented Logical Meta Programming*, PhD thesis, Vrije Universiteit Brussel, Belgium, 1998.

[4] Fowler, M.: *Refactoring - Improving the Design of Existing Code*, Addison-Wesley, 1999.

[5] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[6] Grand, M: *Patterns in Java*, Vol. 1, John Wiley & Sons, 1998.

[7] Hannemann, J.; Kiczales, G.: *Design Pattern Implementation in Java and AspectJ*, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November 2002.

[8] Hanenberg, S.; Unland, R.: *Parametric Introductions*, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, Boston, MA, March 17 - 21, ACM, 2003. pp. 80-89.

[9] Harrison, W.; Ossher, H.; Tarr, P.: *Using Delegation for Software and Subject Composition*. Research Report RC 20946 (922722), IBM Research Division, T.J. Watson Research Center, Aug 1997.

[10] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwing, J.: *Aspect-Oriented Programming*. Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997, pp. 220-242.

[11] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold William G.: *An Overview of AspectJ*, Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Springer-Verlag, 2001, pp. 327-353.

[12] Kniesel, G.: *Delegation for Java – API or Language Extension?*, Technical Report, IAI-TR-98-4, ISSN 0944-8535, University of Bonn, Germany, May, 1998.

[13] Kniesel, G.: *Dynamic Object-Based Inheritance with Subtyping*, PhD Thesis, Computer Science Department III, University of Bonn, Germany, July, 2000.

[14] Rising, L.: *The Pattern Almanac 2000*, Addison Wesley, 2000.

[15] Tip, F; Kiezun, A.; Bäumer, D.: *Refactoring for generalization using type constraints*. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA. pp. 13-26.

[16] Windeln, T.: *LogicAJ -- eine Erweiterung von AspectJ um logische Meta-Programmierung*, Diploma thesis, CS Dept. III, University of Bonn, Germany, August, 2003.

[17] Wuyts, R.: *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*, Phd Thesis, Vrije Universiteit Brussel, Belgium, 2001.