Evolvable Pattern Implementations need Generic Aspects

Günter Kniesel[1], Tobias Rho[1]
and Stefan Hanenberg[2]

[1] Dept. of Computer Science III, University of Bonn
Römerstr. 164, D-53117 Bonn, Germany
{gk,rho}@cs.uni-bonn.de
[2] 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. Hence, providing typical implementation of such patterns and connecting them to a piece of software needs to be done by hand which is an error-prone process. Aspect languages have the potential to change this situation, due to their ability to encapsulate elements that crosscut different modules. Still, existing aspect languages can only express a small number of typical patterns implementations in a generally reusable way. In this paper, we point out the limitations of known aspect-oriented languages, define the notion of a generic aspect language (generAL) and argue that generic aspects are a natural way to achieve reusable design pattern implementations. We sketch the main features of one particular generic aspect language, LogicAJ, and show how it enables generic implementations of recurring design pattern implementations. In particular it permits to switch easily from one pattern implementation variant to another, which significantly eases the evolution of software.

1 Introduction

Design patterns [4] are a standard means of creating large software systems. Catalogues like for example [4, 1, 13] 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. The application of a design pattern usually results for a given programming language in a number of typical implementations. However, with standard object-oriented techniques, such implementations are not themselves reusable software entities. Hence, applying a certain design pattern typically means hand-crafting a number of software elements and embedding them into the application, which is a error-prose process. For example, even the typical implementation of a relatively simple design pattern like Singleton in Java is not trivial (cf. [5], pp. 127-133). Furthermore, pattern implementations typically require the developer to perform invasive changes of a number of classes in the system.

Typical implementations of design patterns provide a number of anticipated variation points: for example the Decorator design pattern [4] permits to add easily new functionality to an already decorated object. However, an unanticipated evolution of the underlying application is not that easy, because the application's evolution needs to be synchronized with the evolution of the design pattern implementations. This kind of needed synchronization is known as the problem of *co-evolution of design and implementation* [16]. Furthermore, changing the implementation of a certain design pattern to an alternative implementation is not easy: for example changing from a class-based Adapter implementation [4] to an object-based adapter implementation requires a number of invasive changes in the code. Such changes are also needed if the developer decides to the replace a certain design pattern. For example, the replacement of a Decorator implementation with a class-based adapter requires a number of changes in the code.

Aspect-oriented software development [9] is a promising approach to tackle the previous problems. 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 live up to this promise. For example [6] shows that only a number of typical design pattern implementations from the catalogue in [4] can be implemented in a reusable manner in the aspect-oriented language *AspectJ* [10]. [7] discusses two typical examples of design pattern implementations that cannot be implemented in a reusable way in known aspect languages and proposes *Sally* that supports genericity for aspect-oriented languages.

In this paper we argue for the need of a much stronger degree of genericity to fulfill the promise of evolvable pattern implementations. In support of our thesis we present the generic aspect language LogicAJ [15], and show how its genericity mechanisms enable reusable and easy to evolve implementations of design pattern variants.

2 Evolvable Pattern Implementations – Problem

The problem that we address is the evolution of "patterned" designs and implementations. In this section we introduce two variants of the decorator pattern and discuss the problems that arise if a design starts with the first variant and must later be evolved into the second one. A solution based on the design and implementation of pattern variants as generic aspects in LogicAJ is presented in the rest of the paper.

2.1 Example: Variation of Decorator-Based Designs

The decorator pattern provides a flexible alternative to subclassing. Additional functionality can be added to an object dynamically. **Fig. 1** shows the UML diagram for the basic variant of the decorator[1]. There are four participant roles in the decorator

[1] The variant of the decorator pattern listed in [4] uses inheritance to achieve subtyping between the Decorator and the Component at the expense of undesired inheritance of state. Here, we

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*.

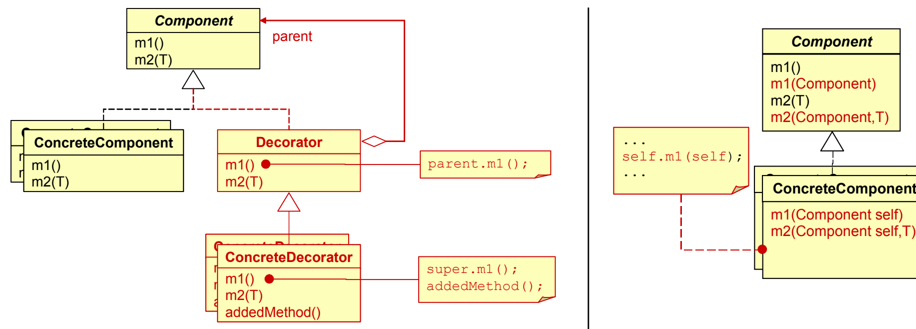


Fig. 1 a) Basic Decorator Pattern according to [4] b) Subsequent evolution of Component. In the following sections the parts in red will be implemented by a generic aspect.

Now assume that, in an application built using the decorator pattern, new requirements imply the additional need to override the behaviour of a component on a per-instance basis. This means, that if some other object has own implementations of the component's methods that implementations should be used. One possible solution is to use *back-reference*s. These can be implemented as an additional parameter (cf. [11, 8]): 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 Fig. 1b.

However, this design decision implies a synchronization of the design pattern implementation with the new design of *Component*. Fig. **2** illustrates how the whole decorator hierarchy needs to be adapted. Comparison of Fig. **1**a and Fig. **2** 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*.

---

use a more general variant, in which we only assume that Decorator is a subtype of Component. In Java, this is achieved by implementation of the Component interface. In other languages it would require inheritance from a purely abstract class.

[2] For a thorough discussion of the issues involved in a simulation of object-based overriding via back references see [12].

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.

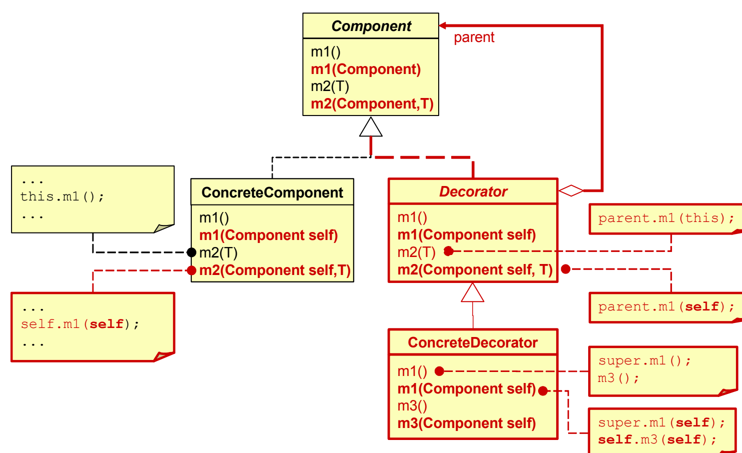


Fig. 2 Decorator Pattern with back reference implemented as an additional method parameter. The parts in red are generated by the aspects described in Section …

3 Towards a Solution – Patterns and Aspects Together

3.1 Patterns describe generic collaborations

Design patterns are an effective way to reuse design knowledge. In a highly standardized way, patterns describe design problems, their context, the forces that influence potential solutions, a generic solution idea and possible variations of its implementation. These descriptions are typically provided in two forms, as concrete examples and as generic abstractions of designs and related object interactions. The generic descriptions of solutions are highly parametric. The names of classes, fields, and methods presented in design pattern "solution" sections are never meant literally but as indicators of *roles* that the respective entities play within the pattern. Correspondingly, object interaction diagrams and related method code refer to such role names.

In order to apply a pattern, a designer must first identify the entities of an application that correspond to the roles mentioned in the pattern. Then he may instantiate the solution, replacing roles by names of concrete entities.

Thus, pattern solutions can be viewed as *generic descriptions of collaborations that are parameterized by the roles of the entities mentioned in the pattern*. Correspondingly, a particular application of a pattern in a given software system is an instantiation of the generic collaboration.

3.2 Aspects describe collaborations

Aspects come in different shapes and sizes. Put differently, there is a wide variety of concepts and systems that call themselves aspect-oriented. Their commonality is that all provide ways to express "crosscutting" structure and behaviour in one single module called an aspect, filter, hyperslice, etc. Structure and behaviour encapsulated in an aspect can range from omnipresent homogeneous behaviour (like logging) to very specific behaviour limited to a particular set of cooperating classes.

The latter case is the one relevant to patterns. Aspect languages have the potential to express the essence of pattern implementations, due to their ability to encapsulate collaborations. For instance, Hannemann and Kiczales [6] show how 12 of the patterns from [4] can be expressed in AspectJ in a reusable way.

However, 11 other GoF-patterns have no reusable implementation in AspectJ, including often used ones like FactoryMethod, Abstract Factory, Builder, Bridge, Adapter, Decorator, Proxy and State. Moreover, some of the reusable versions are limited in different ways. For instance, Hanenberg and Unland [7] show that even for simple patterns, like Singleton, the reusable AspectJ implementation does not achieve the same effects as a manual implementation of the pattern.

In addition, many solutions are not generally reusable but specific to a particular instantiation of an aspect. Part of the aspect code must be repeated for different instantiations.

Hanenberg and Unland identify the inability to express *context-dependent* introductions as the reason for the problems. As a remedy, they propose "parametric introductions", implemented in the aspect language Sally. They show how parametric introductions improve the implementation of patterns like Singleton and enable reusable, partial implementations of patterns like Decorator, which could not be handled at all with AspectJ.

4 Generic Aspects to the Rescue

In this paper, we start from the observation that the concept of parametric introductions is basically a first step towards a generic aspect language but still too restricted to provide a *general* solution for reusable pattern implementations. This leads us to the conclusion that genericity must be supported uniformly across an aspect language, not just for member introductions.

We define the notion of a *generic aspect language* and sketch the main features of one particular generic aspect language, *LogicAJ*. Then we show how a pattern that

could not be expressed in non-generic aspect languages can be implemented easily. The example of the Decorator pattern serves to illustrate basic functionality missing from previous approaches, when it comes to reusable and easy to evolve implementations. Since the same basic functionality is required for other patterns as well, our arguments can be generalized.

4.1 Generic Aspect Languages

The common cause of the above-mentioned limitations of AspectJ-based pattern implementations is that AspectJ does not provide genericity. Typically, the AspectJ solutions refer to *fixed* names for concrete entities of the base program, where a reusable pattern implementation would require *role* names that can be bound to concrete entities when the pattern (resp. aspect) is instantiated. Therefore, the step to reusable pattern implementations is the step to generic aspect languages.

A *generic aspect language* allows aspects to use logic variables that can range over syntactic entities of the host language[3]. Depending on the host language, these can be syntactic elements of Java, C++, C#, etc. or abstractions of messages or events.

Generic aspect languages can provide different *degrees of genericity*, depending on the range of host language entities that they can match. A *fully generic* aspect language would provide logic variables hat can range over *all* syntactic entities of the host language. In a Java-based fully generic aspect language, for instance, logic variables could match anything from packages and types down to individual statements, modifiers and throws declarations.

Logic variables have two properties that are essential in our context. First, their values cannot be manipulated explicitly. Instead, they are bound to values by the evaluation of particular conditions. Conditions that can bind logic variable values are contained, for instance, in the joinpoint expressions of AspectJ, hyperslice expressions of HyperJ, and filter expressions of Compose*..

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 (e.g in AspectJ), where two occurrences of "*" do not represent the same value.

The following presentation of LogicAJ illustrates the general principles introduced so far.

---

[3] Every aspect language has (at least) one *host language*. This is the language in which the modules to which the aspects refer are written. For instance, the host language of HyperJ, AspectJ, Sally, and LogicAJ is Java. Aspect languages like AspectC++ and AspectC# have C++ and C# as their target languages. The composition filter model and its incarnation in the aspect language Compose* is applicable to many different host languages. Event based AOP is an even more generic model, whose host language is an abstract language of events that can be mapped to any particular object-oriented programming language.

4.2 LogicAJ

LogicAJ is an extension of AspectJ [10] by the above concepts. In LogicAJ, logic variables can range over Java 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. Syntactically, logic variables are denoted by names starting with a question mark "?"[4].

Generic aspects are particularly useful for expressing crosscutting changes that follow a common structure but differ in the names of created or modified entities. A 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.

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);
      }
}
```

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. Generic introductions basically generalize the concept of advice in AspectJ. This is

[4] LogicAJ shares this syntax with Sally [7] and TyRuBa [2, 3].

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.

5 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 and reusable way. As we proceed, we identify different classes of 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" and motivate our definition of generic aspect languages.

5.1 Basic Decorator Pattern in LogicAJ

This section models a reusable basic variant of the decorator pattern implementation illustrated in **Fig. 1**. In the following, we assume that an interface playing the *Component* role and classes playing the *ConcreteComponent* role exist[5], 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.

5.1.1 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 (**Fig. 4**).

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.

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

```
abstract aspect AbstractDecorator {

  abstract pointcut component(?component);

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

  introduce(?component, ?decorator) : ... // see Fig. 4

  introduce(?decorator, ?rettype, ?params, ?name) : ... // see Fig. 5
}
```

Fig. 3 `AbstractDecorator` aspect. Participant roles are abstracted as pointcuts. A *Decorator* class specific for a particular *Component* type is generated by two generic introductions.

5.1.2 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 **Fig. 4**. 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;
   }
}
```

Fig. 4 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[6], is essential for our example. We know of no other aspect language that provides this functionality.

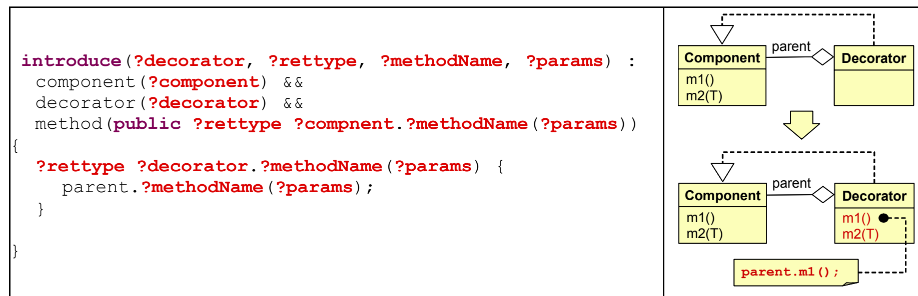
5.1.3 Generic introduction of forwarding methods

The next step is the creation of one forwarding method in the *Decorator* class for every method in the *Component* type[7]. 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

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

[7] 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.

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.



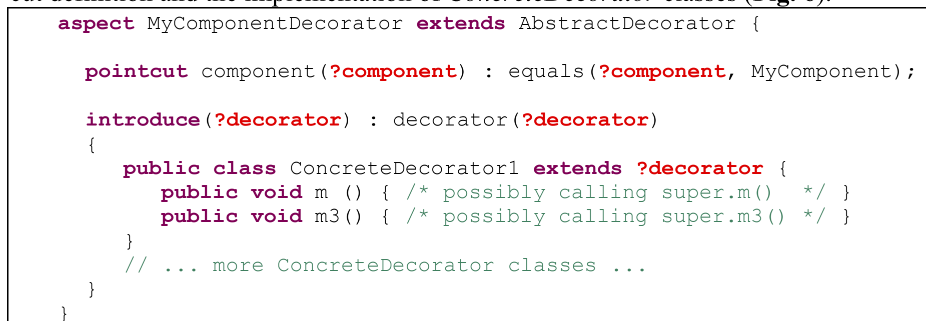
```
introduce(?decorator, ?rettype, ?methodName, ?params) :
  component(?component) &&
  decorator(?decorator) &&
  method(public ?rettype ?compnent.?methodName(?params))
{
  ?rettype ?decorator.?methodName(?params) {
    parent.?methodName(?params);
  }

}
```


Fig. 5 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, [6] concludes that no reusable implementation of decorator is possible with AspectJ. The concept of parametric introductions in [7] 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.

5.1.4 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 (**Fig. 6**).



```
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 ...
  }
}
```

Fig. 6 Instantiation of the decorator aspect for `MyComponent` triggers the creation of the `My-ComponentDecorator` 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 **Fig. 6** preserves separate compilation and encapsulates the knowledge about decorators in the aspect hierarchy.

5.2 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).

5.2.1 Creation of delegatee methods

For every subtype ?*sub* of *Component* and each method in that subtype, the generic introduction in **Fig. 7** adds to ?*sub* a method with exactly the same body but an additional first parameter, self, of *Component* type.

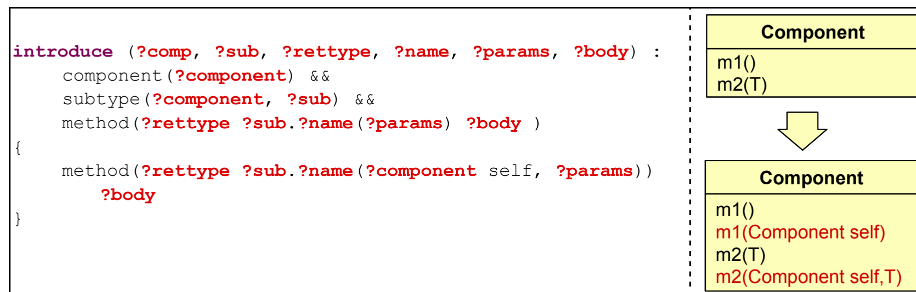


Fig. 7 Introduction of delegatee methods in all subtypes of *Component*

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.

5.2.2 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 **Fig. 8**. 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.

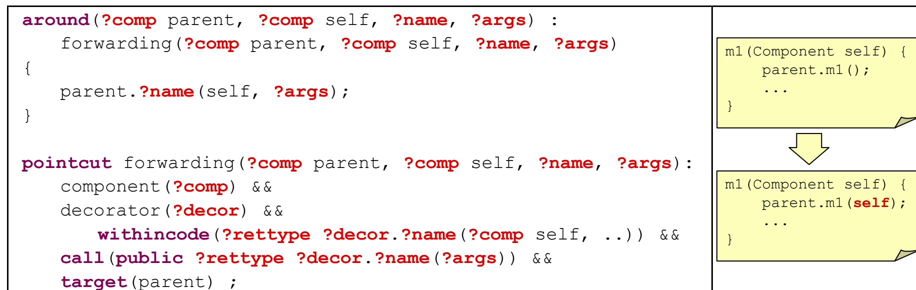
The advice uses two auxiliary pointcut definitions, withinDelegateeMethodInType and callOfComponentMethodOnThis. The first one defines that we are in a delegatee method if the method's first parameter has type *Component* and there is another method in the same type with the signature resulting from deleting the delegatee method's first parameter [9]. 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.

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

Fig. 8 Implementation of “self delegation” by replacing messages to “this”.

5.2.3 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 **Fig. 9** 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.

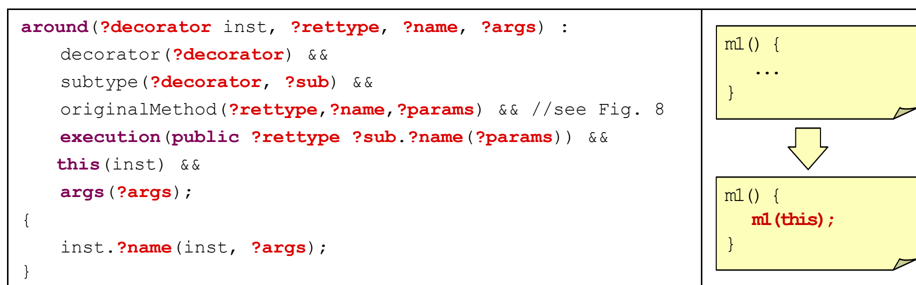


```
around(?comp parent, ?comp self, ?name, ?args) :
    forwarding(?comp parent, ?comp self, ?name, ?args)
{
    parent.?name(self, ?args);
}

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) ;
```


Fig. 9 Forwarding with passing of back reference

5.2.4 Use of delegatee methods

Fig. 10 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 Fig. 8
    execution(public ?rettype ?sub.?name(?params)) &&
    this(inst) &&
    args(?args);
{
    inst.?name(inst, ?args);
}
```


Fig. 10 Redirection of normal method invocations to delegatee methods

The examples shown in **Fig. 8** to **Fig. 10** illustrate various cases in which availability of generic advice was essential in order to express the intended semantics.

5.3 Results

The example studied in this section shows that the adaptation of a decorator-based implementation to new requirements can be performed completely at the level of aspects. 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 one. No single line of code has to be changed manually in the base classes to achieve this result.

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 Fig. 7).

We conclude that non-tangled, reusable implementations of patterns and other highly parametric concepts require genericity at almost *every* level of a language. Whether full genericity (not yet supported in LogicAJ) is desirable, and more generally, what is the “right” degree of genericity, are two of the open questions that we would like to discuss with the workshop participants.

6 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, the 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

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

[16] 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.