AOP with design patterns as meta-programming operators

Uwe Assmann
Universität Karlsruhe
Institut für Programmstrukturen und Datenorganisation
Postfach 6980 76128 Karlsruhe Germany
`assmann@ipd.info.uni-karlsruhe.de`

December 2, 1997

Abstract

AOP (aspect-oriented programming) is an important new software construction methodology because it allows to specify parts and behaviors of components in a view-based way, i.e. with aspects. While these aspects can be specified independently from each other, a weaver tool combines them to the final form of the component. This yields software that is better readable (since specifications are modular) and better reusable (since aspects can be re-combined).

We show for an object-oriented setting, how AOP can be performed without weaver tool. The weaving of an aspect into a component (here a class) can be described by an application of a design pattern. Hence design patterns can be used as powerful meta-programming operators that modify components in order to introduce aspects of computations, and to combine several aspects of a component into a single representation. Additionally, the composition can be checked with standard program analysis techniques whether it is harmless, i.e. whether resulting classes still can substitute their unmodified versions. This immediately has the consequence that the development of a specific weaver tool is not necessary.

1 Introduction

1.1 Aspect-oriented programming

The central idea of aspect-oriented programming is to compose a program from several *aspects* (*aspect composition*). These can be specified independently of each other, but must be mixed together by a *weaver tool*. [KLM$^+$97] claims that most of the problems in classical programming result from the fact that these aspects are not programmed independently, but instead in an interwoven way. Consequently, when an aspect has to be updated during software evolution, many places in the program have to be updated consistently. This hampers software evolution.

For instance, in functional decomposition which is a major way to decompose a system, the items of data-flow (i.e. the definition and use of data items) conceptually belong together, but are distributed all over the system. This effect, that aspects are distributed and interwoven in different parts of the final program, is called *cross-cutting.* Hence aspect-oriented programming aims at dividing the programming languages into one or more *component languages* and an *aspect language* (a weaving language). These languages have different goals. The component language is used to describe components in terms of aspects, but it does not specify how aspects are combined in order to arrive at the final program. This is the task of the aspect weaving language; its tool, the aspect weaver, takes the aspect specifications of the components and mixes them together. Hence the cross-cutting in the final code is produced by the weaver tool while the components of the system stay disjunct and independent.

In this paper, we investigate a form of aspect-oriented programming which does not require a specific weaver tool. Instead, the aspect composition is regarded as a transformation process on the class-graph of the program. Since we are interested in a static scenario of an object-oriented setting, components are identified with *classes.* Since a transformation on class-graphs is a graph-rewrite process, the transformation operators are graph-rewrite operators which match and rewrite subgraphs of the class-graph. Since this process transforms and generates code, it can also be called *meta-programming.* We show that the rewrite operators can be programmed in a language with reflection and intercession. In such a scenario, the meta-programming facilities of the language substitute a specific weaver tool.

Furthermore, graph-rewrite operators can be constructed from *design patterns.* It turns out that design patterns grasp specific aspects of systems so that they can be regarded as *aspect composition operators* (*aspect composers*). We also provide a criterion which checks by conventional program analysis that the composition of aspects is *harmless*, i.e. that classes which are extended by aspect composition are still *conform* (i.e. substitutable) to their unmodified versions.

2 From design patterns to aspect composition operators

It is widely acknowledged that design patterns are a useful instrument to describe architectural knowledge in a concise form [GHJV94]. Design patterns describe collaboration of classes and objects. In a pattern, each of the involved classes plays a certain role; normally a class-graph diagram depicts the relations between the components, and a textual description is added to provide the semantics of the classes' relationships.

As example, consider *event-based coupling.* This is a flexible method to link components [SN92] [GHJV94]. In an event-based coupling, events are fired by an *event source* component and delivered to a *mediator* context[1] which redistributes them to *event listeners* components. All event source and listener components have to register with the mediator, in order to enable the mediator

[1]Also called *event adaptor*, *event handler*, or *event manager*

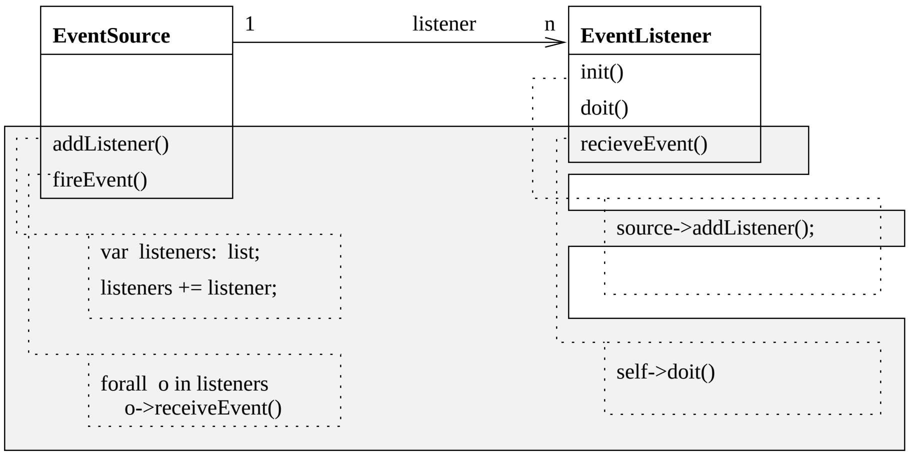

Figure 1: Coupling two components with the design pattern OBSERVER. Coupling-specific parts are shaded. The diagram is in the style of [GHJV94] but it is abstracted from the distinction of abstract and concrete classes.

to distribute events correctly. When new listeners register, or the mediator changes, the behavior of the system changes also. Such a change is transparent to the involved classes, since they do not know to whom an event is delivered and from whom an event originates. This is the reason why event-based coupling is so flexible.

Several design patterns have appeared in the literature which describe event-based coupling [Jav96] [Rie96]. The design pattern OBSERVER [GHJV94] provides the most simple scheme (Figure 1), where the mediator is embedded into event source and the event listener. To this end, the event listener has to provide an interface method `receiveEvent` which is called by the event source in case the event occurred. For initialization, the event listener has to register at the event source, calling the `addListener`-method of the event source. Since both do not know statically to whom they will be coupled, the coupling is flexible.

One disadvantage of this pattern is that if OBSERVER is used in an application to couple two classes, both need to be aware of the coupling: the listener has to register himself, and the event source has to maintain a list of listeners to which the events are broadcasted. Thus in Figure 1 context-related parts of the components can be identified which have to be implemented within the classes to enable the coupling. These are depicted in grey shade, whereas context-independent parts are depicted normally. Suppose a programmer wants to couple components which were not prepared for event-based coupling. Then he/she has to extend their source code with new parts. Thus in a reuse context, OBSERVER can be programmed only with *white-box reuse*. Instead it would be much better, if OBSERVER could introduce the event-based coupling into the classes automatically, in the same way as aspects are composed in AOP automatically.

2.1 Different classes of design patterns describe different aspects

It is seems to be quite difficult to classify the enormous amount of patterns that has appeared up to now. On the other hand, it is obvious that different patterns describe different *aspects* of (architectural) knowledge. [Tic97] provides a preliminary classification that seems to be a first step in the right direction. Tichy proposes that one of the classes are *integration design patterns* which express communication and coupling aspects. Examples for this class are our example OBSERVER or the pattern ADAPTER which wraps a class with a new interface in order to make a coupling possible. Another class consists of design patterns that provide *decoupling design patterns* that decouple components (e.g. hierarchical abstractions with FACADE, introduction of substitution objects with PROXY). A third class consists of those that support distribution (e.g. CLIENT/SERVER or BROKER). Hence this work tries to classify patterns according to their semantic aspects, and hints to the point that design patterns form a *methodology* to describe aspects of systems, but that the aspects which are to be described are totally orthogonal to the principle of design patterns. In other words, it should be possible to classify the aspects for which design patterns are used, and describe aspect-oriented programming with means of design patterns.

2.2 From design patterns to aspect composers

Currently, design patterns are specified in a semi-formal form; the class-graph diagramm is the most formal part of the entire description [GHJV94]. Class-graph diagrams in pattern descriptions can be regarded as *graph-based patterns* which have to occur in class-graphs. When a design pattern is used to describe collaboration of classes, there have to be concrete classes in the class-graph which can be matched by the *class variables* in the pattern. Design patterns also require that certain attributes and methods provided by the identified classes. If we do not assume that these items *exist* in the class-graph we can also regard a design pattern as an operator which *introduces* the methods and relations into a class-graph. This means that a design pattern is regarded as a *graph-rewrite operator* which matches classes in a class-graph and introduces additional methods into the classes. This idea has been developed by Zimmer [Zim97] who showed that design patterns can be described as operators that modify class-graphs while matching and manipulating classes and methods. Since design patterns describe aspects of systems, we call such a design pattern operator an *aspect composition operator (aspect composer)*. The original versions of the classes that are matched and rewritten by a graph-rewrite operator, are called *operand-classes*, while the rewritten versions are called *result-classes*.

If we reconsider our example OBSERVER, it can be depicted as graph rewrite rule or graph-rewrite operator (Figure 2). This rewrite rule is shown as a restricted variant of the X-rule [BFG94][2]. Here, on the lower part of a rule those parts are depicted which are matched and left invariant of the rule (the

[2]Embeddings of the rule into the context of the rewritten graph, which are normally drawn in the upper part of the picture, are left out.

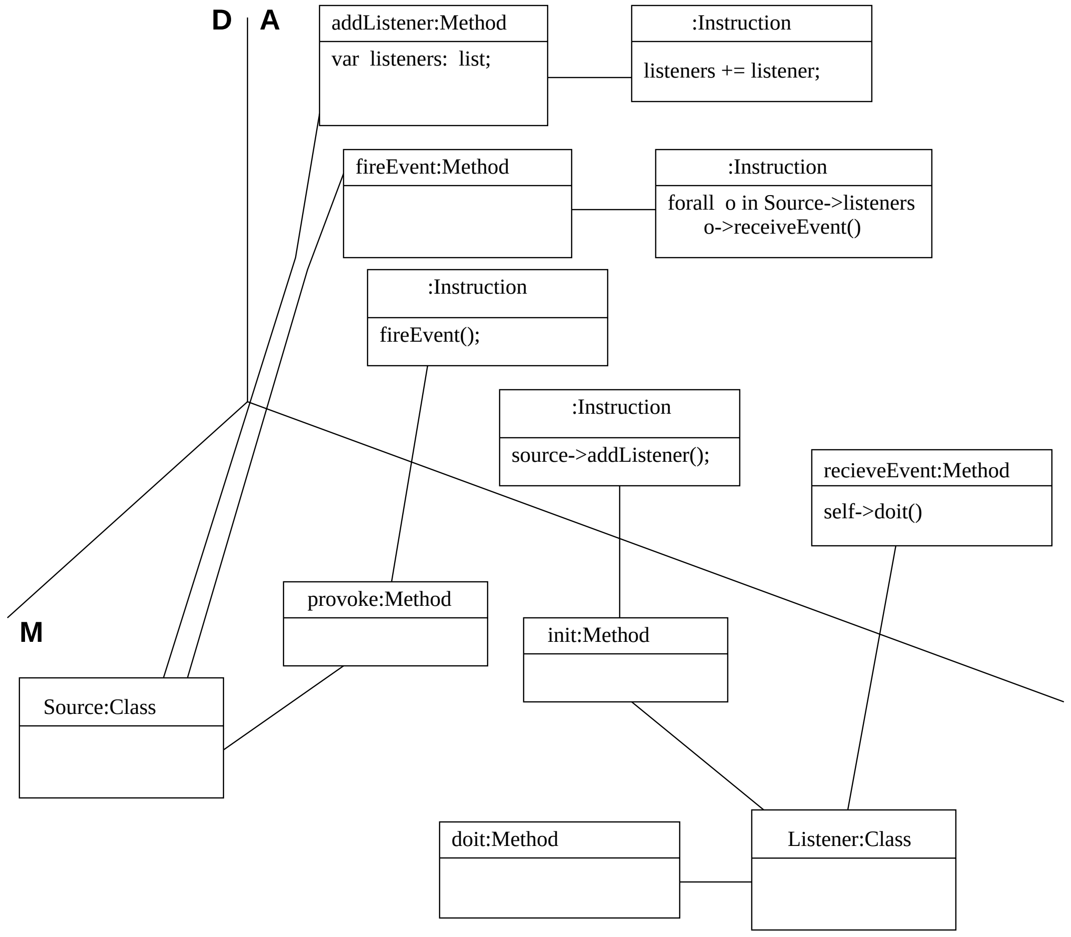

Figure 2: The design pattern OBSERVER, seen as graph-rewrite rule. Now all items are depicted as classes, meta-objects in the class-graph (UML-style). Lower part: invariant items, left side: parts which are deleted, right side: parts which are added.

rule invariant called K). On the left side those parts are shown which are deleted (D); and on the right side those that are added to the graph (A). The left-hand side of the graph-rewrite rule results when D and K are unioned ($L = D \cup K$) and the right-hand side of the rule results from K and R ($R = K \cup A$). Observer matches two classes in the class-graph, an event source and an event listener. The operator does not allocate new classes, but modifies the code of the matched classes extensively: it has to create the method `addListener` and `fireEvent` in the source, and the method `receiveEvent` in the listener. Then it has to add the call to `addListener` to the method `Listener.init` (so that the listener is registered at run-time), and the call to `fireEvent` into method `provoke` (so that the event is signalled and the distribution chain is started). Hence, if we regard Observer as a graph-rewrite operator, it is not required that the matched classes are prepared for event handling; instead the required methods and instructions can be introduced into the classes by the right-hand side of the operator.

2.3 Mixed-in protocols (synchronization aspect)

Another example for AOP is the weaving of synchronization protocols into class hierarchies. It has been identified since several years that synchronization policies should be specified not in class hierarchies, but separately from them (inheritance anomaly [McH94]). If they are specified together, synchronization policies cannot easily be overwritten during inheritance, since they interact tightly with the rest of the inherited code. Hence [McH94] and others [Ber94] propose to handle synchronization policies as separate specification items and to mix them in after the inheritance process. In essence, this approach regards synchronization policies as an aspect of the program which can be exchanged orthogonally.

Also mixing-in of synchronization protocols can easily be described by graph-based rewrite operators on class-graphs. Figure 3 displays a rewrite rule that matches two classes that communicate with a server class. It is assumed that in a specific application the clients work in parallel. In consequence, those methods of the server which modify its state have to be synchronized by a protocol. This can be done by wrapping the methods with entry and exit code of a synchronization protocol. In this case, we mix-in a mutual exclusion protocol with the help of a semaphor. Again, the operator-classes of the composition are matched with the lower part of the X-rule. This time, the left part deletes the relation of the method `work` to its first instruction. We assume that there is only one instruction in `work`. On the right side, a new attribute is added to the server class, a semaphor. Another freshly allocated instruction is inserted as first instruction into the `work` method which initializes the protocol, i.e. locks the semaphor. A new last instruction releases the semaphor again. Hence the client-server pattern can be seen as graph-rewrite operator that mixes-in the aspect of synchronization into an application.

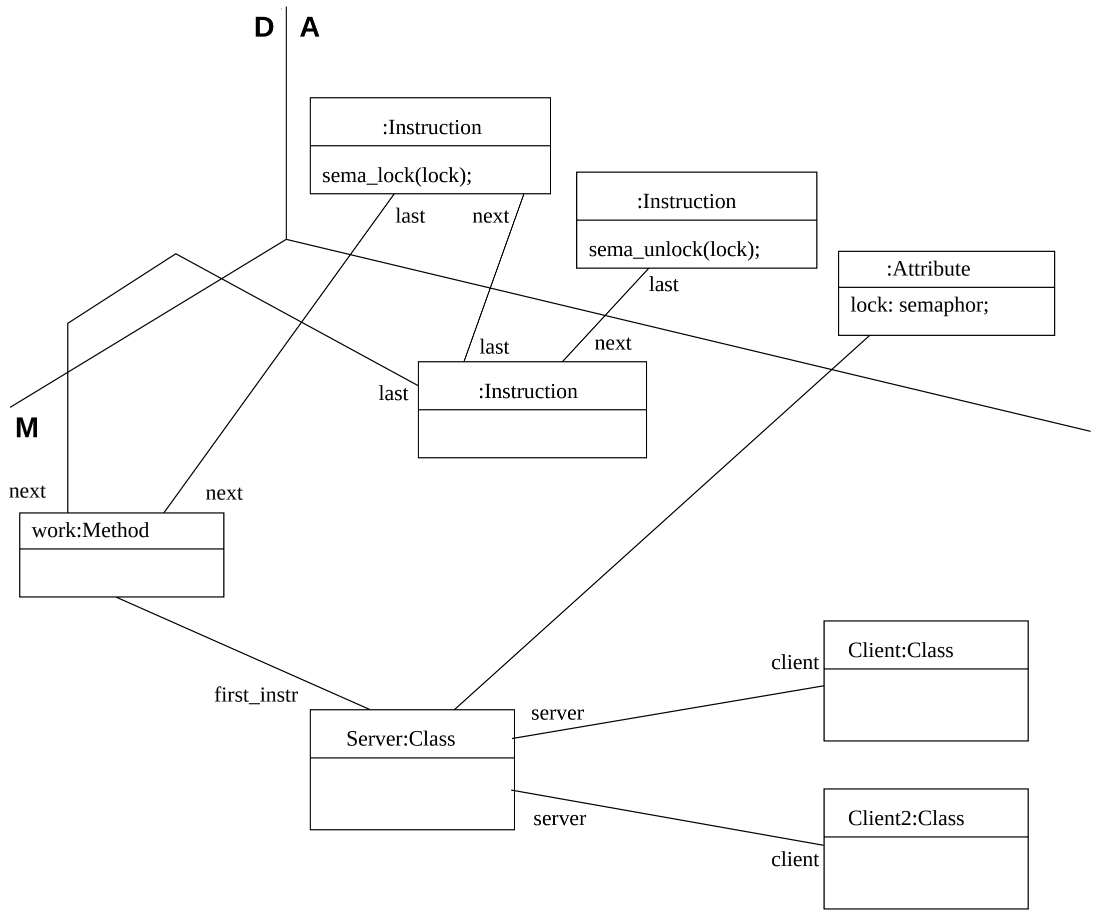

Figure 3: Introducing a mutual-exclusion synchronization protocol into the client-server cooperation, seen as graph-rewrite rule/operator.

2.4 Graph-rewrite operators mix-in aspects into classes

We have shown by example that design patterns can be regarded as graph-rewrite operators that transform class-graphs. More examples can be found in [Zim97]. Since different design patterns grasp different aspects of an application, we may say that graph-rewrite operators may be used to introduce *aspects* into classes. Naturally, since this mixin of aspects transforms code, it can be regarded as *meta-programming* (programming that generates programs).

This insight has immediate consequences for aspect-oriented programming. It should be possible to describe aspects of a system such that the basic components are application-specific classes, while other aspects can be mixed into these classes with graph-rewrite and meta-programming operators. The next section will outline how such aspect composers can be implemented easily.

3 How to implement aspect composition with meta-programming operators

As soon as design patterns and aspect-programming are regarded as graph rewrite operations on class-graphs, the resulting aspect compositions directly generate code, since they perform meta-programming. Each application of an operator modifies classes, and since graph-rewrite rules have a clear semantics (match/add/delete items), it is easy to implement them. All what has to be done is to write a method which implements all parts of the rewrite rule (matching, allocation and linking, deletion). Since such a procedure has to work on class-graphs (i.e. code), it uses reflection (querying meta-objects) and intercession (manipulating them) [KP97]. All that is needed to implement such a procedure is a programming language with a meta-programming interface. For the following example, it is assumed that the reflection interface of Java is extended by intercession. Figure 4 shows the implementation of the Observer operator.

In such an extended reflection interface, the items of the meta-model, i.e. all meta-objects and -relations, are represented with ordinary classes. The following example uses the meta-object classes `Class`, `Method`, and (implicitly) `Instruction` (Figure 4). We assume some basic reflective methods, such as `findMethod` which finds a method with a name, and `findClass` which finds a class with its name. Also several intercessory methods are required: the operator `new` also allocates meta-objects, `addMethod` adds a method to a class, `prefix` prefixes the instruction list of a method with some instructions, and `MakeCodeFromText` constructs instruction lists from Java text.

The entire aspect composer is written as an ordinary Java static method. The graph-rewrite tasks are implemented by matching and manipulating class-graph objects. The left-hand side L is implemented with a list of tests on the operand-class and their relations. The right-hand side R turns into meta-statements which allocate and link meta-objects.

With aspect composers as methods, users may write programs of aspect compositions (Figure 5). Suppose two classes `Button` and `Writer` are given,

```
class AspectCompositionSystem {
  public static void ClassGraph parser() {..};
  public static void void prettyPrint(ClassGraph c) {..};
  public static void Observer(
    EventSource:Class, SourceMethod:String, EventListener: Class) {
    /* MATCH whether the required methods exist */
    if (!findMethod(EventListener,"init")) return;
    if (!findMethod(EventListener,"doit")) return;
    if (!findMethod(EventSource,SourceMethod)) return;
    /* no application possible */

    /* REWRITING: Create meta-objects */
    Method receiveEvent = new Method("receiveEvent",MakeCodeFromText("listener->doit()"));
    Method addListener  = new Method("addListener",MakeCodeFromText(
                                 "for (o = first(EventSource.adaptors);
                                        o != NULL;
                                        o = next(EventSource.adaptors,i))
                                 o.receiveEvent();"));
    addMethod(EventListener,receiveEvent);
    addMethod(EventSource,addListener);

    /* insert registration of listener */
    prefix(findMethod(EventListener,"init"),MakeCodeFromText("source.addListener()"));
    /* insert event-firing call */
    prefix(findMethod(EventSource,SourceMethod),MakeCodeFromText("source.fireEvent()"));
  }
}
```

Figure 4: An implementation of the aspect composer OBSERVER as a Java-style method

each of them with a `doit` and `init` method. We want to connect the button class with the writer's `doit` method, in order to call it if a button is pressed. In a static scenario, before a aspect composer can be applied to a class-graph, a parser has to translate some components from program text to a class-graph. In a dynamic scenario, classes might have been loaded dynamically. Then the reflection mechanism can access classes and the aspect composer the OBSERVER can be applied to the components. In order to call the `fireEvent`-method when a button is pressed, the method `ButtonPressed` of the button class should be handed over to the aspect composer. Finally, a pretty-printer has to generate Java code which contains the final standard Java classes. These can be translated to bytecode by an ordinary Java compiler. Hence complex applications can be plugged together with several calls to aspect composer methods.

```
public static void AnnotateButtonToWriter() {
  AspectCompositionSystem acs;
  ClassGraph classgraph = acs.parser();
  Button  button   = findClass("Button");
  Writer  writer   = findClass("Writer");

  /* Compose the classes */
  acs.Observer(button,"ButtonPressed",writer);

  acs.prettyPrint(classgraph);
}
```

Figure 5: Composer applications as ordinary method applications

Since the aspect composer extends the `doit`- and `init`-methods of the classes appropriately from outside, the aspect of event coupling is mixed into the classes. Furthermore, since the aspect composer only adds event-handling calls, and these are independent of the old code, the operand-classes are extended *transparently*. This illustrates the power of our approach: components may be programmed independent of their context and, if the added code does not conflict with old code, the components are embedded into the context transparently.

Naturally, this simple example leaves open a lot of questions. When can a aspect composer know where it should introduce code? In the example, there were two cases: `init` and `doit` were programmed into the composer, but `ButtonPressed` was a parameter to it. The latter is much more flexible since it can be used to parametrize composers and application points. Beyond these cases, a semantic summary annotation to class interfaces would be helpful which could be investigated by the composer for application conditions. This could be provided by mechanisms such as state charts or petri-nets. Also pre- and postconditions could provide semantic knowledge that could be exploited for aspect composition [FZZ95]. A simple standard solution is to use a naming convention by which certain semantic denotations are reserved for certain name prefixes ([Jav96]), but it is obvious that future work has to improve the semantical summaries of components.

Of course, a full-fledged AOP system would offer a library of aspect composers that manipulate different aspects of algorithms. Users may use inheritance or even aspect composition to extend them. We can imagine at least the following aspects for which composers may be developed.

1. Which parts of which classes are coupled to others (coupling and data-flow aspect)?

2. How tight are classes coupled by the introduction of aspects (integration aspect)?

3. Which form do the architectural links have (architectural connection)?

4. Which operand-class executes when (control-flow aspect)?

5. Which classes and objects should survive an application (persistance aspect)?

Programming aspect composers spans up a large design space of aspects compositions, leaving all freedom for users to compose their applications from different aspects arbitrarily.

4 Harmless aspect composition

During aspect composition of class-graphs, immediately the question arises whether extending classes with new aspects disturbs their behavior. Can a class that is modified by an aspect composer still be used in all using contexts it had been used before? When does an mixing-in of an aspect change the behavior of the class significantly?

Apparently, we need to check the *conformance* of a modified class (including a new aspect) to its unmodified version (without the aspect). A class c_1 is called *conform* to another class c_2, if c_1 can substitute c_2 in all use-contexts (substitutability) [FZZ95]. This conformance-checking can be achieved by program analysis. Program analysis methods can estimate conservatively which dependencies between code parts exist (*def-use analysis* [ASU86]). In particular, they can find out, whether new code that is introduced by an aspect composition, is independent on the old code and vice versa. Apparently this case is easy: if both code parts are independent, the semantics of the class to which the code mixed-in is not changed, although it is enriched in its functionality.

In the following, we call a set of instructions of a method a *code-piece*. Two code-pieces are called *independent* to each other, if they do not have data-flow dependencies. A data-flow dependency between a code-piece that defines an item and a code-piece that uses the item, is called a *flow-dependency* (*def-use dependency*). There are also other dependencies, such as use-def dependencies. With these definitions the following theorem can be given:

Theorem 1 *Result-classes of an aspect composition are conform to their operand-classes if*

- *the composition does not delete meta-objects of the operand-classes and*
  - *either all code-pieces of operand-classes are independent to code-pieces that are introduced by the composition*
  - *or between old and new code only flow-dependencies exist.*

In this case, the aspect composition is called harmless.

Proof: Suppose a class c_2 is aspect-composed from a operand-class c_1 which consists of a code-piece m_1. Let U be the set of all use-contexts of c_1. Code deletion is excluded, so c_1 can only be extended with new code or data. If c_1 is extended by a code-piece m_2 which is independent to m_1, m_1 will still behave in the same way. Hence c_2 shows the same behavior as c_1 when it is used in a $u \in U$. Of course, in c_2 both m_1 and m_2 are executed, but since m_2 is independent to m_1, c_2 will show the same behavior in the old context as c_1.

On the other hand, if m_2 only is flow-dependent on m_1, also the visible behavior of c_2 is the same as that of c_1 at every $u \in U$, since m_1 is not flow-dependent on m_2, and the effects of m_2 are not visible at u.

$\blacksquare$

4.1 The algorithm to check harmlessness of aspect compositions

The theorem immediately proposes an interactive algorithm how to check the soundness of aspect compositions (Figure 6). The algorithm is interactive, since graph-rewrite operators cannot be applied automatically like in automatic graph-rewrite systems: because the user intends some semantics with his compositions, he has to apply the operators manually, otherwise the system would choose an arbitrary composition sequence. To check the harmlessness of the composition, all we have to do is to apply standard program analysis techniques incrementally after each composition. For data-flow analysis specification tools exist [Ass96]. In order to avoid that the entire class-graph must be analysed, an incremental data-flow analysis technique can be applied [MR90]. First, such a technique analyses which parts of the class-graph have changed. Based on that, parts of old analysis results are reused. Also techniques from interactive program environments, e.g. for incremental semantical analysis [Sne91], can be used.

```
while composition not completed
     interactively identify application position of aspect composer;
     apply aspect composer;
     check harmlessness with data-flow analysis;
end;
```

Figure 6: An algorithm to check harmlessness of aspect compositions

In essence, meta-programming aspect composition and the incremental data-flow analysis technique economizes a special aspect weaver. This is the major advantage of our method: Weaving is automatically provided by standard techniques, and it is provided for all kind aspects that can be described by graph-rewrite operators on class-graphs.

5 Related work

It is well-known that AOP can be modeled with meta-programming [KLM+97]. However, it has not been realized yet that design patterns can be used to derive meta-programming operators that perform aspect composition.

In his thesis [Zim97], Zimmer develops the idea to use design patterns as transformation operators on the class-graph. Zimmer defines a language in which all actions a design pattern involves can be described systematically (pattern matching on the class-graph, transformations of methods, etc.). This provided one of the starting points for our work, but Zimmer did not recognize that his language uses meta-programming, and that design patterns can be used for aspect composition.

It is well-known that *architectural environments* such as Unicon [SDK+95] or Aesop [GAO95] try to extract the communication aspect of a system out of the components in order to facilitate reuse and composition [GS93]. When we use design patterns to describe aspect composition, it seems that architectural systems allow only a certain kind of graph-rewrite operator during the composition of the communication aspect. For instance, repository systems coupled components tightly, and use design patterns that mix-in synchronization by wrapping slot access methods. Implicit-invocation systems only allow event-based design-patterns. Pipe-filter systems only allow design patterns that introduce unidirectional flow of work packages on links that are built from pipes. Procedure-call systems only use design pattern operators that introduce method call links. Hence architectural environments could very well turn out as a specific instance of AOP, since in AOP a system is divided into more aspects than the communication and the algorithmic one.

Composition-filters [Ber94] [ABV92] and the *layered object model* [Bos95] represent context-related actions of a class by *filters* or *layers* that encapsulate it. Each message that arrives at a class has to cross this set of filters that modify it. However, composing filters (i.e. wrapping code around methods and objects) is a simple meta-operation that is used by design patterns and aspect composers very frequently.

Code generation from design patterns has been attempted only recently [BFVY96]. Here design patterns are described in the form of [GHJV94], with an additional description in a special language COGENT. This macro-based language is expanded by the `perl` interpreter to C++ code. Since the items of COGENT are classes, this approach uses meta-programming, although it has not been described as such. In our work, code generation results naturally from rewriting the class-graphs, since the composed classes can be compiled.

6 Conclusion

This paper presented an approach how – for an object-oriented setting – aspects can be composed without weaver tool. Design patterns can be used to derive graph-rewrite operators that match and rewrite parts of the class-graph in order to mix-in aspects into classes. Hence for certain classes of applications it is not necessary to develop new aspect languages and weavers.

Additionally, we presented a criterion on orthogonal composition which allows to exactly determine when an aspect composition is harmless, i.e. when a result-class of a composition still can substitute an operand-class. As this criterion is based on standard program analysis it is valid for all application domains and easy to check.

References

[ABV92] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 372–395, Berlin, Heidelberg, New York, Tokyo, June 1992. Springer-Verlag.

[Ass96] Uwe Assmann. How To Uniformly Specify Program Analysis and Transformation. In P. A. Fritzson, editor, *Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, pages 121–135, Heidelberg, 1996. Springer.

[ASU86] Alfred A. Aho, R. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ber94] Lodewijk M. J. Bergmans. *Composing concurrent objects*. PhD thesis, University of Twente, Enschede, 1994.

[BFG94] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical Use of Graph Rewriting. Technical Report Queens University, Kingston, Ontario, November 1994.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[Bos95] Jan Bosch. *The layered object model.* PhD thesis, University Twente, 1995.

[FZZ95] Arne K. Frick, Walter Zimmer, and Wolf Zimmermann. On the design of reliable libraries. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 17 — Technology of Object-Oriented Programming*, pages 13–23. Prentice Hall, August 1995.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[GS93] David Garlan and Mary Shaw. *An Introduction to Software Architecture*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[Jav96] JavaSoft. JavaBeans^{TM}. http://java.sun.com/beans, December 1996. Version 1.00-A.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopez, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP 97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[KP97] Gregor Kiczales and Andreas Paepcke. Open implementations and metaobject protocols. Technical report, Xerox PARC, 1997.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[MR90] T. J. Marlowe and B. G. Ryder. An Efficient Hybrid Algorithm for Incremental Data Flow Analysis. In *ACM Symp. on Principles on Programming Languages*, pages 184–196, 1990.

[Rie96] Dirk Riehle. The event notification pattern – integrating implicit invocation with object-orientation. *Theory and Practice of Object Systems*, 2(1):43–52, 1996.

[SDK+95] Mary Shaw, Robert DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, pages 314–335, April 1995.

[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions of Software Engineering and Methodology*, 1(3):229–269, July 1992.

[Sne91] Gregor Snelting. The calculus of context relations. *Acta Informatica*, 28(5):411–445, 1991.

[Tic97] Walter F. Tichy. Classification of Design Patterns. Lecture slides, Universität Karlsruhe, January 1997. http://wwwipd.ira.uka.de/ tichy/entwurfsmuster.html.

[Zim97] Walter Zimmer. *Frameworks und Entwurfsmuster*. PhD thesis, Universität Karlsruhe, February 1997.