Applying Aspect-Oriented Composition to Framework Development – A Case Study

Stefan Hanenberg [1]

University of Duisburg-Essen, Schützenbahn 70, 45117 Essen, Germany

Robert Hirschfeld [2]

DoCoMo Euro-Labs, Landsberger Straße 312, 80687 Munich, Germany

Rainer Unland [3]

University of Duisburg-Essen, Schützenbahn 70, 45117 Essen, Germany

Katsuya Kawamura [4]

DoCoMo Euro-Labs, Landsberger Straße 312, 80687 Munich, Germany

Abstract

Object-oriented frameworks play an essential role in large-scale software projects. Frameworks let us benefit from tested code which provides variation points that can be customized to address project specific needs. The way to adapt frameworks depends on the composition techniques offered by the underlying programming system. Traditionally, frameworks use inheritance as dominant adaptation technique. Unfortunately, since the inheritance mechanism of languages like Smalltalk or Java is quite limited, the way how frameworks can be customized is limited, too. Hence, the reusability of frameworks is restricted to situations anticipated by the framework designers where the variation points meet exactly the developers' needs. Aspect-oriented programming is a new paradigm that provides new composition techniques. This paper shows that some design decisions in framework construction to be made in the context of object-oriented development can be avoided by applying aspect-oriented composition techniques addressing these issues only when required. With that, frameworks based on aspect-orientation allow us to access a larger variety of variation points, making these frameworks highly adaptable and reusable, addressing more unforeseen circumstances.

Key words: Aspect-Oriented Programming, Dynamic Weaving, Frameworks, Unanticipated Software Evolution, Design Patterns.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Introduction

Object-oriented frameworks [24] play an important role in the construction of large-scale applications. Frameworks are collections of related domain-specific, tested, and semi-complete classes that support some architectural aspects to let their instances interact in a well-defined manner. They need to be customized to the particular needs of specific applications. For that purpose, frameworks contain a number of *hooks* [34] or *variation points* which can be used by developers for customization (see [4] for a discussion on explicit and implicit hooks in framework development). How these hooks are realized depends on the design of the framework, but also on the composition techniques provided by the underlying programming language or environment. So, the less powerful the underlying composition techniques are, the fewer possibilities exist to provide appropriate hooks.

White-box frameworks, most commonly used in the object-oriented world, heavily rely on inheritance as the dominant composition mechanism. However, inheritance mechanisms provided by class-based programming languages such as Smalltalk and Java are quite limited. Other languages offer a much larger variety of composition mechanisms that can be used for the framework development such as *mixin classes* [5], *method combinations* in CLOS [25], *F-bounded polymorphism* as introduced in [7], or *class templates* in C++ [36,38]. Restricted composition techniques reduce the reusability of software fragments and limit the possibilities to construct frameworks that are easily customizable to specific needs. Consequently, frameworks based on such techniques tend to be complex in their design, and with that their customization is complex and error-prone.

Aspect-Oriented Programming [31,27] is a new paradigm offering new composition techniques. These techniques can be easily used to introduce custom hooks inside application frameworks in an unanticipated manner to better match the developers needs. Thus, framework construction based on aspect-oriented features can reduce the complexity of both the framework artifact and its customization.

In section 1 we motivate the need for highly adaptable frameworks by introducing the requirements for an object traversal framework. In section 2 we discuss in detail why current object-oriented programming techniques are not sufficient to provide an appropriate implementation of such a framework. Afterwards, we briefly introduce aspect-orientation in general, and AspectS which is an aspect-oriented implementation technique in particular. In section 4 we discuss the effect of using aspect-oriented techniques during the framework development. In the same section we revisit the example and show

[1] Email:shanenbe@cs.uni-essen.de
[2] Email:hirschfeld@docomolab-euro.com
[3] Email:unlandr@cs.uni-essen.de
[4] Email:kawamura@docomolab-euro.com

that by using aspect-oriented features the drawbacks of the object-oriented solution do not exist in the aspect-oriented one. After discussing the related work we conclude our paper.

1 Motivation: Object Graph Traversal

As stated by many authors a large variety of domain-specific problems in object-oriented applications can be solved by the application of graph algorithms [20,32]. Typically, graph algorithms are needed whenever a group of objects provide some common behavior whereby the values affecting this behavior are distributed over an object graph. That means a number of objects need to be traversed, and at each object visited some computation needs to be done. This kind of problem often recurs in the design of object-oriented systems, and the problem (and a corresponding solution) is addressed by the *Visitor* design pattern [11]. However, typical implementations of these traversal strategies are quite similar. Hence, it is not satisfying to have a reusable design element that still needs to be implemented again and again. Instead, it is desired to encapsulate these traversal strategies within a framework.

Our need to encapsulate traversal strategies was mainly motivated by a typical aspect-oriented problem: our objective was to encapsulate the functionality usually provided by *Observers* as described in [11]: A number of objects (the observers) should be informed whenever the state of a particular object changes (the observed object is called subject). Although numerous authors regard recurring observer implementations as typical applications for aspect-oriented programming [8,22], it is argued out in [13] that current aspect-oriented implementation techniques are not sufficient to encapsulate recurring observer implementations. We proposed a mechanism based on *dynamically woven aspects* as supported by systems like *AspectS* [19]. The main idea is to traverse all objects referenced by the subject and add to them the corresponding code that informs the observers about a change within that object. For example, if a student `stefan` as illustrated in Figure 1, needs to be observed, an aspect will be added to all objects reachable from `stefan`.

However, to provide an Observer implementation that can be applied to a number of different object structures, it needs to be highly adaptable. First of all, there must be a number of generic traversal algorithms, because the resulting woven code differs from traversal to traversal, and concrete observers determine the kind of traversal actually needed [13]. In the simplest case we only distinguish between *depth first search* (DFS) and *breadth first search* (BFS) traversals. However, since different observers might be interested in different parts of the object structure, traversals need to be adaptable, too. For example, a user interface that just visualizes a student's name and a student's address does not need to traverse all courses the student attends. The policy of what neighbors should be traversed can be selected independently of whether the underlying traversal is a DFS or a BFS.

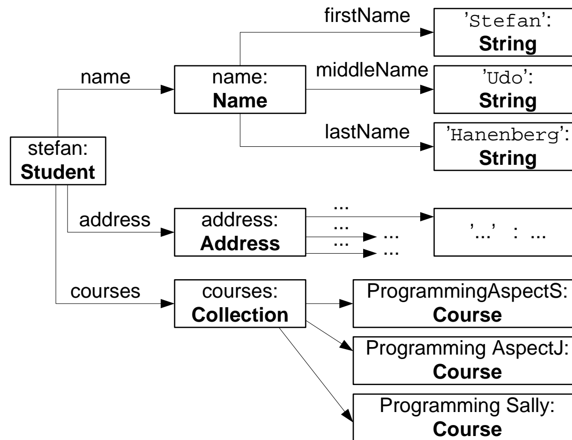

Fig. 1. Object structure of student `stefan` that needs to be observed

Furthermore, a policy for handling cycles in the object graph has to be chosen. Usually, traversing an object twice should be prohibited. On the other hand, there are observers which are mainly interested in paths for accessing individual objects and only traversing the same edge twice should be prohibited. The decision of how to handle cycles is independent of whether the traversal is a DFS or a BFS, and also independent of what neighbors of an object should be visited during traversal. To weave the observer-specific code, it is furthermore necessary to store into a separate object structure how to access neighboring objects [13], possibly the expression to be executed to determine a certain neighbor of a certain object.

And finally, to reduce the developers effort of using the encapsulated observer, we need to provide a default behavior for most of the problems mentioned above. The default behavior we want to provide is a DFS traversal on all accessible neighbors of an object without traversing an object more than once. Developers should be able to refine the default behavior incrementally if needed. For example, it should be possible to use the default behavior and to refine it by describing for certain objects which neighbors to be visited and which not.

We need to consider that in contrast to the usual graph traversal our observer represents a special case: there is no specialized data structure that directly represents a graph. That means there is no graph object that contains a number of node and edge objects used for the traversal. Instead, all information belonging to the graph is distributed over a number of objects which refer to each other, as well as to objects that do not belong to our graph to be traversed. The traversal framework we are about to build should be able to handle this too, i.e. it should be possible to apply the traversal algorithms independently of how the underlying graph is represented.

The result of the discussion above is a set of requirements for our traversal framework:

(i) A number of traversal algorithms like DFS and BFS are to be supported.

(ii) The functionality to be executed at each visited node (in the following we call this the *behavior policy*) is independent of the chosen traversal algorithm.

(iii) The set of neighboring objects of a specific node to be traversed may vary (in the following we call this the *adjacent nodes policy*).

(iv) The way cycles are dealt with should be independent of the selected traversal algorithm and adjacent nodes policy (in the following we call this the *cycle policy*)

(v) The requirements mentioned above are valid independent of the data-structure representing the graph.

Before proposing our solution based on aspect-oriented constructs, we discuss in the following section why object-oriented language mechanisms are not sufficient to encapsulate a framework for object traversal that fits these given requirements.

2 An Object-Oriented Traversal Framework

In the previous section we outlined the problem that almost all elements inside the object graph traversal require some form or degree of variability. Such variability needs to be designed as appropriate hooks within our framework. The requirements stated above necessitate the definition or selection of traversal algorithms, behavior policies, adjacent nodes strategies and cycle strategies in their own modules which can be composed independently of each other.

First of all, we analyze code fragments that need to be encapsulated in their own modules. The above requirements state that these elements represent some behavior that can be selected by application developers to compose a traversal algorithm that fits their needs. Hence, each of these elements should be easily selectable or adaptable since in concrete application these elements usually vary slightly from the default implementations. This means also that developers should be able to replace these elements easily.

Afterwards, we discuss the design alternatives given by object-oriented composition techniques to compose the code fragments. We argue why certain design alternatives have to be chosen to provide hooks that permit to customize the framework according to the given requirements. Then we discuss why the resulting framework is no satisfying solution for the given problem, although the design seems to be reasonable and fits the requirements.

2.1 Code Fragments for Object Traversal

First of all, we consider how different traversal strategies can be implemented (see Figure 2). For our traversals, we define an abstract base class named `Traversal` with abstract methods that need to be implemented by concrete

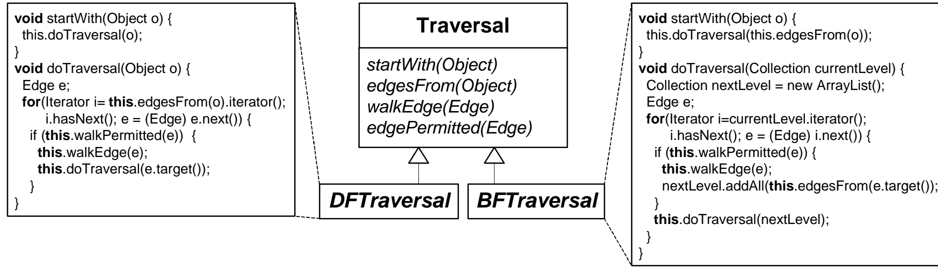

Fig. 2. Using inheritance for different traversals with abstract hook methods

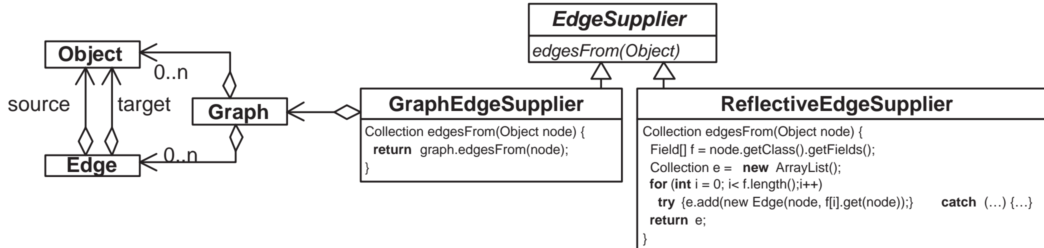

Fig. 3. Frequently used adjacent nodes strategies

subclasses (`DFTraversal` and `BFTraversal`). In class `Traversal` we declare a method `startWith`, which is invoked to start the traversal from a given object.

The class contains a number of hook methods that are invoked by concrete traversal strategies: method `edgesFrom` provides a collection of `Edge` instances which represent the edges starting from a given object, `edgePermitted` is invoked to determine whether a given edge can be selected for the traversal. The method `walkEdge` is invoked to show that a certain edge is selected for traversal. Those methods are declared because it is known that the way how edges are to be chosen and whether an edge is permitted to be traversed depends on the underlying adjacent nodes and cycle policy. Method `walkEdge` is needed to address the requirement that the behavior policy varies (walking an edge just means to traverse the target node from the source node of the edge).

Furthermore, it is known that a traversal should be independent of the underlying data structure used to represent the object graph (requirement 5). Hence, our code uses `Object` as the type for the node to be traversed and introduces type `Edge` for representing edges. An instance of `Edge` refers to two arbitrary objects which represent the connected nodes (the structure of `Edge` is illustrated in Figure 3). Note that in this example we assume that `Object` cannot be invasively extended which is often the case in mainstream programming environments such as Java or C++.

Two typical examples of default adjacent nodes strategies are given in Figure 3. The abstract class `EdgeSupplier` defines a method `edgesFrom` which returns a collection of `Edge` instances. The `GraphEdgeSupplier` refers to a

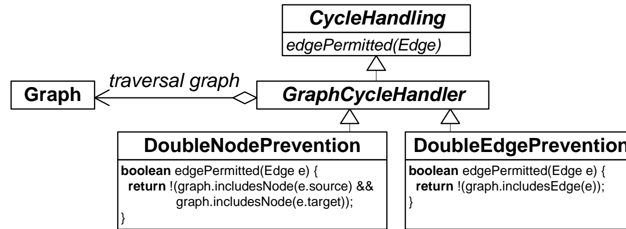

Fig. 4. Often occurring cycle strategies

graph object and returns all edges from the given node (computed by the graph). The `ReflectiveEdgeSupplier` uses the introspective facilities of the underlying language to determine all fields of an object and creates edge instances initialized with the fields values. However, as noted in the previous section, these adjacent nodes strategies need to be refined by the developer. For example, if developers know what adjacent nodes to traverse for a given class or a given object, they should be able to add their customization to any of the default adjacent nodes strategies.

Typical cycle strategies are to prevent the traversal to visit an object or an edge more than once. Both concrete classes in Figure 4 (`DoubleNodePrevention` and `DoubleEdgePrevention`) reference all those elements of our graph which are already traversed. `DoubleNodePrevention` forbids an edge to be visited if both nodes of the edge are contained in the already traversed part of our graph. `DoubleEdgePrevention` forbids the selection of an edge that is already traversed. Another typical example of a cycle policy is to permit the traversal of edges as long as no *Euler cycle* occurs in the traversal graph. Note, as already argued above that a cycle policy not necessary adapts only `edgePermitted` it is also possible that a cycle policy adapts `edgesFrom` (to provide only edges that do not lead to any cycle) and `walkEdge` (to determine the traversal graph that is necessary to determine cycles in the traversal so far).

The behavior policy, that is the code to be executed during the visit of a particular node, usually depends on the concrete application of the graph traversal algorithm. Hence, the behavior policy needs to be specified by the developer. Since the result of the traversal (i.e. the traversal graph) needs to be determined quite often, the framework itself already provides some behavior policies. For example, the framework provides the specification of a default behavior policy inside a class `GraphStorage` that stores edges and nodes to a graph object every time they are traversed because the cycle strategies mentioned above depend on the graph that has been traversed so far. This default behavior policy is used by the cycle strategies described above to determine the traversal graph. Figure 5 illustrates the class `GraphStorage` that implements a method `walkEdge` that adds the edge and its corresponding nodes to the graph in case they are not already contained.

Note, that above we intentionally used the same method names in different

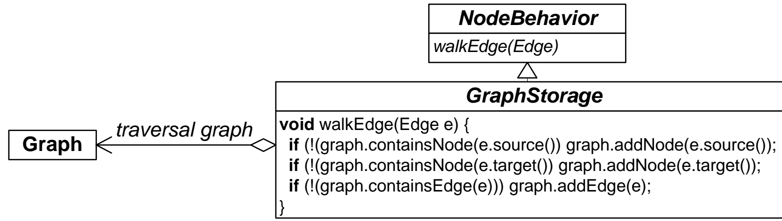

Fig. 5. Storing traversed edges as default behavior policy

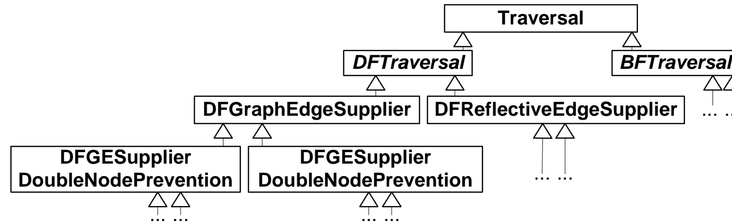

Fig. 6. Chaotic class hierarchy using inheritance as dominant composition technique

classes (for example method `edgePermitted` in Traversal and `CycleHandling`) to emphasize their close relationship. However, until now we have not discussed how to compose these methods defined in multiple policies.

2.2 Unit Composition

The most frequently used composition technique for framework adaptation is inheritance. Methods which are called from within the framework are either left abstract and need to be overridden by the application developer or they provide some default behavior which might be adapted when necessary (according to the *Template Method* design pattern [11]). Hence, the control flow is coordinated by the framework and customized code is invoked from the framework itself. This property of frameworks is usually described as the *Hollywood Principle* [39]. In languages like Java or Smalltalk that provide single inheritance as the only inheritance mechanism, the use of inheritance needs to be quite disciplined, since a class can share only code with at most one direct superclass.

The declaration of class `Traversal` contains the hook methods `edgesFrom`, `edgePermitted`, and `walkEdge` that are invoked by the code defined in `DFTraversal` and `BFTraversal`. However, the connection between those methods and their definitions (in the adjacent nodes policy, cycle policy and behavior policy) needs to be defined. The question is, whether inheritance could be used for that or not.

The availability of just single inheritance and the absence of mixin-based mechanisms to adapt hooks requires the definition of subclasses of `Traversal` for each traversal policy. For example, to compose a adjacent nodes policy with a traversal algorithm, we need to define for each existing traversal algorithm (in our example `DFTraversal` and `BFTraversal`) a corresponding

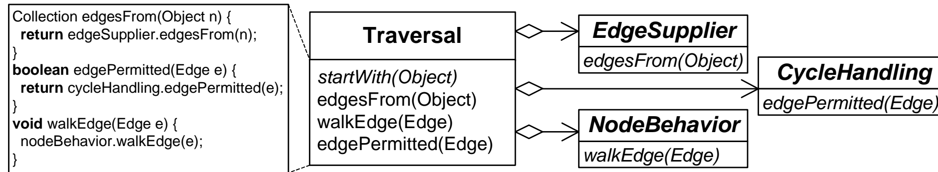

Fig. 7. Problem of (missing) multiple policies by using the *Strategy* design pattern

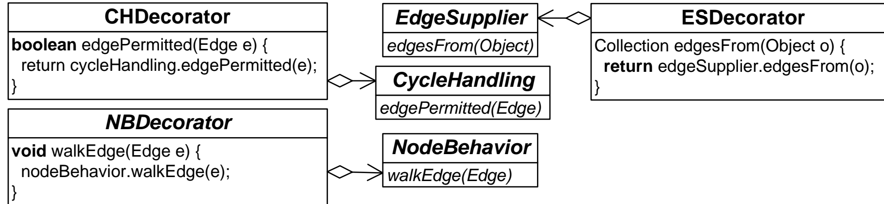

Fig. 8. Using the decorator pattern to adapt policies

subclass including the code for the adjacent nodes policy. Assuming just two traversal strategies, two adjacent nodes strategies and two cycle strategies, fifteen classes need to be defined, whereby most of them contain duplicated code as illustrated in Figure 6. The redundant code base results in a system harder to reason about, to maintain, and to evolve. Furthermore, whenever developers define their own traversal strategies, they need to create a lot of the subclasses mentioned above to meet the outlined design approach. Obviously, this cannot be regarded as scalable approach to the given problem.

To adapt the behavior of individual instances of a particular class, we may consider the application of the *Strategy* or *Decorator* design patterns [11]. Applying the strategy pattern to `Traversal` allows us to configure each instance with a number of strategy objects that provide specific implementations of the `edgesFrom`, `edgePermitted`, and `walkEdge` methods (see Figure 7).

However, besides all known benefits, using strategy objects also has disadvantages. The first disadvantage stems from the cardinality of the policies. We already noticed that, for example, a number of different modules containing behavior policy need to be executed for each traversed node: one for creating the traversal graph (needed by the cycle policy) and one that represents the application specific code provided by developers. Here, instead of having one single strategy object for each kind of policy, we need multiple collections of strategy objects. Hence, the resulting code depends on the way how different strategies are composed. This is not considered in Figure 7. A mechanism is needed that permits multiple policy instances to perform some joint behavior. This is a typical scenario for using a Decorator.

Decorating an individual policy object to add responsibilities dynamically and transparently, we need to implement a decorator class whose instance wrap a policy object with additional behavior. And so, three new classes need to be created in our example (`CHDecorator`, `ESDecorator`, and `NBDecorator` in Figure 8). To add a new policy, developers need to extend the matching

decorator classes and override the corresponding method. Now, an instance of our newly introduced decorator class can be used to wrap, and with that to extend, the behavior of an individual policy instance. The proposed framework applies strategies and decorators to fulfill the requirements given in section 1. We are able to combine a number of different default policies independently of each other and independently of the underlying traversal strategy. Furthermore, the developer is able to customized the traversal strategies and the different policies.

2.3 Conclusion so far

Our framework can be characterized as follows: there are approximately 50 lines of code implementing the basic functionality of our default traversal strategies and default policies. Additional 20 lines of code were spent on implementing base classes or mechanisms to implements the strategy and decorator design pattern. With that, more than 25% of our code does not address the basic functionality but provides some form of infrastructure that allows us to customize the default behavior. This is due to the fact that most of the desired changes such as the customization of the behavior policy or the adjacent nodes policy only affect small entities of the basic functionality, if not only one single method. From the framework developers' point of view the effect is that large effort has to be devoted to design hooks that permit to customize quite small entities.

From the application developers' point of view that means that large effort has to be devoted to customize the framework although only small entities need to be adapted. The effort for adding a new behavior policy to an existing traversal instance is to create a new subclass of NodeBehavior and create an instance of it. Then the developer needs to get a reference to the currently provided behavior policy, decorate this one with the new instance and assign the new policy to the traversal instance. Also, it is the application developer's responsibility to specify the control flow from the customized behavior policies to the one which needs to be adapted. So, although the framework has been designed properly essential parts of the control flow needs to be specified by the application developer. This is in contrast to the well-known Hollywood Principle which states that the main control flow should mainly be managed by the framework and not by the application developer.

The main problem of designing an object-oriented framework is that the hooks need to be designed in a way to be customizable for a large variety of applications. Because of this generalization of the framework's applicability, mechanisms for providing hooks become inherently complex. On the other hand application developers customize a framework to their specific needs. Their needs are usually just a small subset of those cases the framework was designed for. Nevertheless, they have to use the hooks that were provided for a larger variety of problems. Because of this, they have to use (and customize)

hooks that are more complex than suitable for their concrete application. Let us reconsider the basic functionality as explained in section 2.1. Obviously, one problem is that a number of different methods should be able to be combined independently of each other. Unfortunately, traditional object-oriented programming systems do not allow for variations at the level of granularity of individual methods. Because of that, desired change support at the method level needs to be addressed at the much coarser grained object level, utilizing mechanisms like class inheritance and object composition. In the resulting design the mechanisms for supporting appropriate hooks becomes rather complex in comparison to the original problem. As a consequence, customizing (as well as developing) the framework becomes complex and error-prone.

3 Aspect-Oriented Software Development

Aspect-oriented programming (AOP, [31,27]) is a new software technology that deals with code fragments which logically belong to one concern but cannot be modularized because of limited composition mechanisms of the underlying programming language. Such code is said to crosscut other modules, where it gets tangled with code that belongs to other concerns. Such a concern is called *crosscutting concern*. AOP is about modularizing crosscutting concerns by providing special modules called *aspects*. With that, aspect-orientation addresses the issues of *separation of concerns* [9].

As of today there are several approaches that support aspect-oriented concepts, ranging from general-purpose aspect techniques like *AspectJ* [2,26], *AspectS* [3,19], *JMangler* [23,28], or *Sally* [18] to domain-specific aspect languages such as *D* [30]. Many of these techniques allow us to represent crosscutting concerns, down to the methods and instance variables level of granularity. Like objects in object-oriented programming, aspects may appear at all stages of the software development lifecycle. Examples of aspects that can be commonly observed are architectural or design constraints, features, and systemic properties or behaviors (such as error recovery and logging).

The underlying technique for preventing crosscutting code in aspect-oriented programming is the so-called *weaving* mechanism. Two aspects each consisting of its own units are woven together with the original object-oriented base system into a final woven system. The weaver is responsible for combining each aspect with the object-oriented system and to create a final representation of the woven system. For specifying how and where the additional aspects should be woven to the base system, aspect-orientation makes use of the *join point* concept. In [26], join points are introduced as "principled points in the execution of the program". Typical examples for join points are a certain method call from a certain caller object to a certain callee object or the execution of a constructor. Weaving in general can be performed at compile-time or run-time. AspectJ is an example for compile-time weaving. Here, the weaver integrates aspects into the application by transforming the byte code before

runtime. JMangler performs load-time transformation of Java class files. AspectS employs a run-time weaver to transform the base system according to the aspects involved.

Aspect-oriented techniques like AspectJ or AspectS provide the *pointcut* construct to select those join points aspects should be woven to: a pointcut specifies a set of related join points to be addressed by an aspect. The code to be executed at a certain pointcut is called *advice*.

Variations and variation points depend on the underlying modularity mechanism provided by the programming platform a system is built on. Most modern software systems were built using object-oriented technologies where the modularity constructs, and with that the units of change, are that of classes and instances. Although this level of granularity is sufficient in some cases, a more fine-grained approach to modularity is desirable to permit the change of even smaller semantic units such as method implementations. Also, while traditional modules such as classes and instances might support the proper structuring of the initial system, subsequent changes to this system could crosscut these module boundaries affecting more than one location.

AspectS [3,19] extends the Squeak/Smalltalk [21,37] environment to allow for the exploration of aspect-oriented software composition in the context of dynamic systems. It supports coordinated meta-level programming, addressing the tangled code phenomenon by offering aspect modules. In its current implementation, AspectS is realized without changing neither Smalltalks syntax nor its virtual machine. AspectS, instead of relying on code transformations (neither source nor byte code), makes use of metaobject composition instead. In contrast to most other approaches to aspect-orient programming that only focus on class-level aspects, AspectS allows for instance-level aspects and with that allowing for modularization of behavior that crosscuts a set of individual instances.

AspectS mainly draws on the results of two projects: the first is AspectJ from Xerox PARC, a general-purpose aspect-oriented language extension to Java, and the second is MethodWrappers [6], a mechanism to add behavior to a compiled Smalltalk method. In contrast to more static approaches that try to bind all decisions at development or compile time (such as AspectJ), AspectS offers advanced runtime change support.

Method wrappers allow for the introduction of code that is executed before, after, or instead of an existing method. As an alternative to modifying Smalltalks standard lookup process, method wrappers change the objects the lookup mechanism returns. A method wrapper replaces an entry in a class method dictionary (a compiled method or another method wrapper), adds behavior to the method invocation, and eventually invokes the wrapped method itself. AspectS makes use of block method wrappers, special wrappers that allow to plug-in block contexts for additional behavior. For each kind of advice there is a matching method wrapper implementation. AspectS coordinates the placement of block method wrappers into the method dictionaries of the

classes of the receivers.

In AspectS, aspects are implemented via regular classes, so their instances act as regular objects also. An aspect is applied to objects in the image by sending an install message to an aspect instance. The effects of an aspect to the system are reverted by simply sending an uninstall message to the same aspect instance that cause the system transformation. An aspect may hold on to a set of receivers, senders, or sender classes. These objects are added or removed by client code, and will be used by woven (composed) code at run-time to determine if receiver-instance-specific, sender-instance-specific or sender-class-specific behavior has to be activated or not. Client annotations allow the introduction of advice-specific state.

In Smalltalk, object interaction is based on the message-sending metaphor. A message sent by a sender is decoupled from the actual method implementation executed by the receiver on behalf of the sender. In AspectS the receiver of a message is considered the only structural information related to a join point. By naming a target class and a target selector, a join point descriptor partially describes a join points static property of location within the systems class hierarchy. Advice qualifiers allow the description of dynamic attributes of a join point in the context of an advice. Join points of a pointcut can be enumerated statically, or, due to the very open and reflective nature of the Smalltalk environment, collected dynamically by querying the system. AspectS does not introduce a dedicated pointcut language but takes advantage of the expressiveness of Smalltalk itself.

Advice objects associate code fragments (parts of the crosscutting concern to be implemented by an aspect) with pointcuts and their respective join points descriptors that describe targets for the weaver to place these fragments into the system. AspectS uses Smalltalk blocks (lexical closures) to represent code fragments. Furthermore, advice objects are to be qualified to state if the woven code will be receiver-class-aware, receiver-instance-aware, sender-class-aware or sender-instance-aware, combined with additional control or call flow semantics if needed.

AspectS allows us to execute crosscutting behavior before and after the execution of a method invocation, to handle signaled exceptions, and around a method invocation. It is possible to introduce new behavior to the target clients as well. These kinds of advice are not minimal since both a handler advice and a before-after advice can be expressed using an around advice. However, providing these concepts makes the intent of some advice objects more obvious and so the resulting code better to understand. Advice qualifiers allow the description of dynamic attributes of a pointcut related to an advice. These attributes state dynamic characteristics of join points enumerated by a pointcut in the context of an advice. Advice qualifier attributes can be grouped roughly into sender/receiver aware activation, and control or call flow activation. The combination of point cut descriptors and advice qualifier attributes can be compared to AspectJs concept of pointcut designators.

Weaving in AspectS happens every time an aspect instance is installed by sending an install message to the respective aspect instance. To reverse the effects of an aspect to the system, the aspect has to be uninstalled by sending an uninstall message to the aspect instance responsible for the system transformation to be reversed. This process is also referred to as unweaving. Weaving and unweaving in AspectS can be characterized as dynamic. According to the attributes stated in an advice qualifier, a Method Wrapper is configured with one or more activation blocks. Each activation block, represented by a Smalltalk block, is provided with the aspect instance associated with the wrapper, and the base level activation context (base sender) that allows access to not only the receiver of the message, but to the whole chain of activation contexts (Smalltalks stack, [12]). Depending on this information, the activation block evaluates to a Boolean value, either true or false.

4 Framework Construction using Aspect-Oriented Techniques

In the sections above we identified the specification of hooks or variation points as a major problem in the framework construction. The main problem is a result of the limited features provided by object-oriented languages to represent these hooks or variation points. Traditionally, object-oriented languages provide only class inheritance and object composition as adaptation techniques. If the underlying inheritance mechanism is quite simple (e.g. single inheritance with static dispatching), the effort for providing hooks that fit the users' needs is enormous as shown in the example. In addition to that, developers have to make some effort to customize these hooks.

The composition mechanisms provided by aspect-oriented programming techniques are much richer than pure object-oriented techniques. Aspect-oriented techniques provide features for representing join points to which additional aspects can be woven to. So from an aspect-oriented point of view, every existing join point in the base system represents a hook that can be customized by the developer. Nevertheless, different aspect-oriented techniques provide different kinds of join points. For example AspectJ provides a large variety of join points than can be described inside the pointcut language. Other techniques mainly use class, field and method definitions as join points.

Aspect-Oriented techniques allow us to add multiple aspects to a single join point. Basic mechanisms are available to coordinate the control flow between base program and aspects. For example, it is possible to execute a number of aspect-related code before a certain join point is reached using pieces of advice. Those pieces of advice do not need to know from each other. Instead, it is the responsibility of the weaver to compose the final system that guarantees the execution of all those pieces of advice. If the order of the execution of the pieces of advice is critical, aspect-oriented environments allow to affect their execution sequence.

For framework development, the interesting characteristic of aspect-oriented systems is that a base system inherently provides a number of hooks (represented by the join points) which can be adapted by aspects (and hence by the application developer customizing the framework). As illustrated in our example, we regard the presence of explicit hooks represented by a number of complex design elements as a disadvantage of object-oriented frameworks. The benefit of aspect-oriented programming in the context of framework development is that those hooks are implicitly contained and do not need to be expressed by additional design elements.

Nevertheless, this should not imply that frameworks or extensible libraries based on aspect-oriented techniques no longer provide any explicit hooks. The framework developer still has to be sure that hooks exist that can be used by the applied aspect-oriented technique for weaving. Since the effort for specifying join points in aspect-oriented techniques can be quite high [15], the framework developer must provide selected pointcut specifications (or even a pointcut library) to be used by the application developers for customization.

Hence, developing frameworks based on aspect-oriented techniques we face the following difficulty: We need to provide an appropriate set of default pointcut and advice definitions that can be used by application developers for customization. In the following we show how to design the traversal framework using aspect-oriented composition techniques and discuss their benefits.

4.1 Revisiting the Example

A large effort in our object-oriented traversal framework was spent to provide appropriate hooks. Since from an aspect-oriented point of view all join points represent hooks that can be customized by application developers, we first of all analyze what join points exist in our code fragments and which ones should be offered as extension points. Then, we discuss how to specify the control flow at each join points. Afterwards, we consider the specification of join points inside the framework.

4.1.1 Determination of Relevant Join Points

Figure 9 illustrates the join points 1 to 14 in DFTraversal and TraversalStrategy based on the join point model provided by AspectJ and AspectS. All message-sends inside DFTraversal are join points to whom additional aspects (actually the advice code) can be woven to. All declared methods represent join points (there are far more kinds of join points, but they are not needed in our example).

For a framework developer it is important to determine which join points should be interesting for being customized by the application developers. On the one hand, implementation details (i.e. the implementation of class DF-Traversal and BFTraversal) should be hidden from application developers. Also, methods such as doTraversal which are specific to the implementation

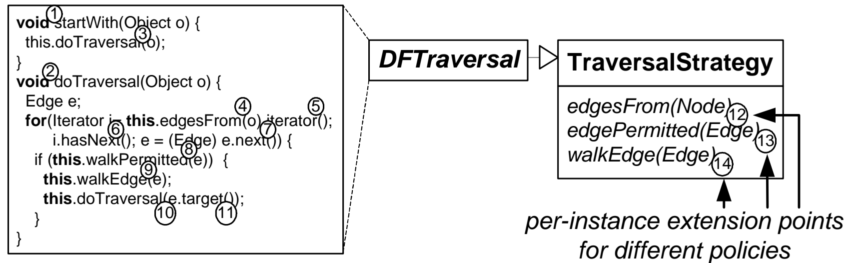

Fig. 9. Join Points in DFTraversal

of DFTraversal (and also BFTraversal) should not be used as extension points either because these methods might not occur in other traversal strategies implemented by the application developer. Since all traversal strategies should have the interface TraversalStrategy, all methods declared in this interface are preferable extension points to be used by application developers.

It must be possible to instantiate a number of different traversal instances and adapt each instance according to the developers needs. Since traversal instances send messages declared in TraversalStrategy to itself it is necessary to adapt TraversalStrategy on a per-instance base: for each traversal instance a different adaptation might occur at join points 12, 13 and 14.

4.1.2 Specifying the Control Flow at Relevant Join Points

All different default policies provide code which might be executed at these join points. A mechanism is needed to combine each policy with the join points in TraversalStrategy. In principle, there are at least two different design approaches. The first approach is to provide join point specifications used by the default policies. Here, the behavior of each policy must be coded by an advice. In the second approach all aspect-oriented constructs are hidden. Here, the policy-specific code is implemented in a regular method and the invocation of this method is hard-coded inside of our framework. The advantage of the first approach is that the framework contains less code and allows the refinement of its join point specification. The disadvantage of this approach is that pieces of advice are hardly reusable [10,16] and that application developers must be highly familiar with the applied aspect-oriented programming and composition technique. An advantage of the second approach is that application developers do not need to be familiar with aspect-oriented constructs since they do not need to deal with aspects and their corresponding pieces of advice directly themselves.

Because of that, we decided to encapsulate the method invocation inside the framework: each superclass of each policy (i.e. class EdgeSupplied, CycleHandler and NodeBehavior) refers to its join point and contains a corresponding piece of advice that simply invokes the method implemented in the policy. Furthermore, we need to decide how these methods should be invoked, that is whether the methods should be invoked before, after, or around the

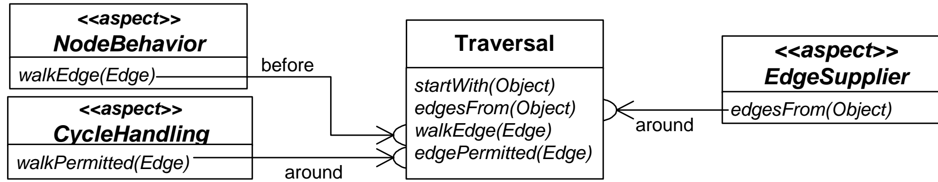

Fig. 10. Invocation of policy instances

corresponding join point is reached. The first two approaches imply that the weaver is responsible for creating an appropriate interaction between different policies. The third approach implies that the application developer who provides a customized policy knows how to specify the interaction between different policies: the developer has to implement how a number of different policies invoke each other. There is no common rule about what approach is more appropriate in general [15]. Instead, it has to be decided case by case.

In section 2 we pointed out that usually a number of different actions need to be performed on every node traversal. In the example cycle policies refer to a traversal graph created by a corresponding default behavior policy (implemented in class GraphStorage, see Figure 5). That means in addition to this policy all customized behavior policies have to be executed. Usually these policies are independent of each other. So, we decided to invoke the methods specified by NodeBehavior instances before the corresponding join point is reached. The situation is different for adjacent nodes policies and cycle policies. Usually, the code belonging to these policies cannot be executed independent of each other. Because of this, we decided to invoke those policies around the corresponding join point. Figure 10 illustrates the design of each policy. The arrow between each method declared in a policy and the traversal strategy designates the join point each policy refers to (in TraversalStrategy) and the advice code invoked at these join points (in each policy). The annotation at each arrow states how the advice code is invoked (before or around).

4.1.3 Specifying Relevant Join Points

Once it is clear what relevant join points exist within the framework and what aspects might be woven to these join points, the way of how these join points should be accessed has to be determined. There are at least the following two alternatives: either the relevant join points should be made accessible to the developer, or the join point specifications should be hidden from the developers. To make join points explicit, we need to provide constructs that directly refer to the relevant join points. From a technical point of view it is not necessary to make relevant join points accessible since these join points are already in the base system. Application developers just need to address them from within their aspect specifications.

The benefit of providing join points specifications lies in a reduction of

redundant code: if a number of aspects refer to the same join points it is not necessary to specify these join points redundantly. Instead, all aspects just refer to the join points specification provided by the framework. Redundant join point specifications are critical if the join points in the base program change. Here, all join point specifications that refer to the corresponding join points need to be changed [14]. The disadvantage of specifying join points is that additional code needs to be created.

Usually, aspect-oriented programming techniques provide constructs for making join points explicit. For example, a named pointcut in AspectJ is a construct for externalizing join points that can be shared by different pieces of advice. In the same way join point descriptors in AspectS are usual instances that can be delivered using object-oriented constructs.

Applied to our framework it seems clear that the specification of the relevant join points in TraversalStrategy has some benefits: a number of different policies provide some additional code to be executed at these join points. Furthermore, application developers who provide their own policies need to weave their own code to these join points. In such a case it needs to be determined what modules should provide this join point information. In the traversal framework there are two reasonable alternatives: either the join points specifications are provided by TraversalStrategy or by each policy class. The first approach has some advantages in respect to the reusability of the join point specifications: single instances of customized policies can easily adapt all relevant join points just by referring to the corresponding traversal strategy. Nevertheless, since our intention was to keep the interfaces of each class inside the framework as small as possible and the join points for each method declaration in TraversalStrategy are quite easy to specify we neglected to provide join point specifications within the framework to be used by the application developer. Instead, the join points used by each policy are hidden from the application developer.

4.2 Implementation Issues

Our argumentation in the previous section was based on mechanisms provided by aspect-oriented techniques without referring to a concrete one. Nevertheless, since aspect-oriented techniques differ from each other we discuss implementation issues of the framework here by comparing an implementation in AspectS and AspectJ. Our original motivation for the discussed framework came from the need to encapsulate often occurring object-traversal strategies needed to add the functionality provided by the observer pattern to object during run-time. Therefore, we implemented the traversal framework originally in AspectS. In its current version AspectS does not directly provide aspects that only work on single instances. As a consequence, developers need to manage the reference to the target object on their own. Within the activation blocks developers need determine whether or not the receiver object

corresponds to the target object. To reduce the effort of managing the target object and the activation block, we simple created a class PerTargetAspect that extends the root aspect class AsAspect, added a field target that refers to the object the aspect is woven to and modified the activation block for each advice within instances of PerTargetAspect. All policy classes are extensions of PerTargetAspect and are initialized with the TraversalStrategy object they are woven to.

4.3 Benefit of the Aspect-Oriented Framework

If we regard the design of our framework based on aspect-oriented composition techniques in comparison to the object-oriented solution from section 2 we see that the design elements which were responsible for providing the appropriate hooks have vanished. The reason for this phenomenon is the existence of aspect-oriented composition techniques that provide more powerful mechanisms to compose different software units.

One problem we faced in the object-oriented framework was to provide hooks that permit to add new behavior to single objects. This was solved using an implementation of the strategy pattern. But since aspect-oriented techniques directly permit to specify pieces of advice for single objects there is no necessity to consider object adaptation during the design of the framework.

Another problem we faced was to provide hooks that permit to compose different objects that provide some behavior in cooperation. We solved that by implementing a decorator for each policy. In the aspect-oriented framework we do not need to consider this problem since the aspect-oriented weaver already permits to weave a number of aspects to the same join point.

In comparison to the object-oriented framework adapting the aspect-oriented one is much easier. Application developers just need to create their policy subclass that overrides the corresponding method, instantiate this class and weave the instance to an traversal instance. In contrast to that the object-oriented solution assumed each developer to access the corresponding policy instance, decorate this instance and replace the traversal's object reference to the original policy instance with the new decorator instance. Because of that, adapting the aspect-oriented framework is less complex and error-prone.

5 Related Work

The use of aspect-oriented techniques in framework construction as described in our paper is highly related to collaboration-based design with mixin-classes [36,38]. Nevertheless, these approaches proposed utilize C++ class templates in that only allow the creation of new classes, but not the adaptation of existing classes or even instances. In [35] aspect-oriented techniques were analyzed with respect to their ability to implement collaboration-based design. The authors mainly propose to use introductions, a language or system construct

of AspectJ and AspectS. However, using introductions are similar to the use of class templates, introductions are performed before compile time and with that do not allow for instance specific adaptations.

Roles as described in [29] propose similar composition techniques as used in our example. Roles add dynamically new behavior and fields to objects. Hence, very similar results can be achieved if the traversal strategies are adapted by roles (see [17] for a comprehensive discussion of aspects and roles). In contrast to roles, aspects permit to adapt a number of different objects by selecting different join points from different instances. For framework construction, such a characteristic is needed to affect the behavior of different elements at different locations.

The framework proposed in this paper has a number of parallels to DJ [33] which also encapsulates traversal strategies and permits to add customized code. However, in *DJ* a once created traversal strategy cannot be adapted by the developer anymore.

6 Conclusion

In this paper we discussed the benefit of using aspect-oriented composition techniques for the implementation and adaptation of application frameworks. By means of an object traversal framework, we have shown that the object-oriented solution is not satisfying. Since object-oriented languages usually provide composition mechanisms only for classes and objects but not at the level of individual methods, the resulting object-oriented design is more complex than necessary. A mechanism was needed to compose several methods independently of each other.

We discussed how aspect-oriented techniques can be used for framework development. We illustrated the relationship between join points of base program aspects and hooks within framework to be customized by application developers. We pointed out similarities between join points and hooks that both allow for individual adaptations. In comparison to the object-oriented approach, aspect-oriented framework design gives us the opportunity to address changes at the appropriate level of granularity, extended to that of individual methods, without being confined to coarser grained class and instance levels. Hence, in frameworks that need to contain a large number of fine grained hooks the object-oriented solution is necessarily more complex than the aspect-oriented solution based on fine grained join points. Furthermore, dynamic aspect composition mechanisms allow for framework adaptations at runtime.

However, using join points as extension points for the framework development requires a disciplined approach to customization. Current aspect-oriented programming techniques do hardly support access control to internals that should not be used for the customization by application developers. It is recommended that application developers only customize frameworks at the

join points specified by the framework developer.

The construction of frameworks based on aspect-orientation allows us to provide a larger variety of variation points, making these frameworks highly adaptable and reusable, addressing more unforeseen circumstances.

References

[1] Akşit, M., editor, "Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2003)," ACM Press, 2003.

[2] AspectJ homepage, *http://eclipse.org/aspectj/*.

[3] AspectS homepage, *http://www.prakinf.tu-ilmenau.de/ hirsch/projects/squeak/aspects/*.

[4] Aßmann, U., "Invasive Software Composition," Springer-Verlag, 2003.

[5] Bracha, G. and W. Cook, *Mixin-based inheritance*, in: *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications* (1990), pp. 303–311.

[6] Brant, J., B. Foote, R. E. Johnson and D. Roberts, *Wrappers to the rescue*, in: E. Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming*, LNCS **1445** (1998), pp. 396–417.

[7] Canning, P., W. Cook, W. Hill, W. Olthoff and J. C. Mitchell, *F-bounded polymorphism for object-oriented programming*, in: *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture* (1989), pp. 273–280.

[8] Clarke, S. and R. Walker, *Towards a standard design language for AOSD*, in: G. Kiczales, editor, *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2002)* (2002), pp. 113–119.

[9] Dijkstra, E. W., "A discipline of programming," Prentice-Hall, Englewood Cliffs, NJ, United States, 1976.

[10] Ernst, E. and D. H. Lorenz, *Aspects and polymorphism in AspectJ*, in: Akşit [1], pp. 150–157.

[11] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, Boston, MA, United States, 1995.

[12] Goldberg, A. and D. Robson, "Smalltalk-80: The Language and its Implementation," Addison-Wesley, Boston, MA, United States, 1983.

[13] Hanenberg, S., R. Hirschfeld and R. Unland, *Aspect weaving: Using the base language's introspective facilities to determine join points*, in: *Workshop on Advancing the State-of-the-Art in Runtime Inspection (ECOOP 2003)*, 2003.

[14] Hanenberg, S., C. Oberschulte and R. Unland, *Refactoring of aspect-oriented software*, in: *Proceedings of Net.ObjectDays*, 2003, pp. 19–35.

[15] Hanenberg, S., A. Schmidmeier and R. Unland, *Aspectj idioms for aspect-oriented software construction*, in: *European Conference on Pattern Languages of Programs*, 2003.

[16] Hanenberg, S. and R. Unland, *Using and reusing aspects in AspectJ*, in: *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, 2001.

[17] Hanenberg, S. and R. Unland, *Roles and aspects: Similarities, differences, and synergetic potential*, in: Z. Bellahsene, D. Patel and ColetteRolland, editors, *Object-Oriented Information Systems*, LNCS **2425** (2002), pp. 507–521.

[18] Hanenberg, S. and R. Unland, *Parametric introductions*, in: Akşit [1], pp. 80–89.

[19] Hirschfeld, R., *AspectS – Aspect-oriented programming with squeak*, in: M. Akşit, M. Mezini and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS **2591** (2003), pp. 216–232.

[20] Holland, I. M., *Specifying reusable components using contracts*, in: *Proceedings of the European Conference on Object-Oriented Programming*, LNCS **615** (1992), pp. 287–308.

[21] Ingalls, D., T. Kaehler, J. Maloney, S. Wallace and A. Kay, *Back to the future: The story of squeak, a practical smalltalk written in itself*, in: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1997), pp. 318–326.

[22] Jézéquel, J.-M., N. Plouzeau, T. Weis and K. Geihs, *From contracts to aspects in UML designs*, in: *To be included!*, 2002.

[23] JMangler homepage, *http://javalab.cs.uni-bonn.de/research/jmangler/*.

[24] Johnson, R. and B. Foote, *Designing reusable classes*, Journal of Object-Oriented Programming **1** (1988), pp. 25–35.

[25] Keene, S. E., "Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS," Addison-Wesley, Boston, MA, United States, 1989.

[26] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, in: J. L. Knudsen, editor, *Proceedings of the European Conference on Object-Oriented Programming*, LNCS **2072** (2001), pp. 327–353.

[27] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: M. Akşit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, LNCS **1241** (1997), pp. 220–242.

[28] Kniesel, G., P. Costanza and M. Austermann, *JMangler—a framework for load-time transformation of Java class files*, in: *First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)* (2001).

[29] Kristensen, B. B. and K. Østerbye, *Roles: Conceptual abstraction theory and practical language issues*, Theory and Practice of Object Systems **2** (1996), pp. 143–160.

[30] Lopes, C. V., "D: A Language Framework for Distributed Programming," Ph.D. thesis, College of Computer Science, Northeastern University (1997).

[31] Lopes, C. V., *Aspect-oriented programming: An historical perspective (what's in a name?)*, Technical Report UCI-ISR-02-5, University of California, Irvine (2002).

[32] Mezini, M. and K. Ostermann, *Integrating independent components with on-demand remodularization*, in: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2002), pp. 52–67.

[33] Orleans, D. and K. Lieberherr, *DJ: Dynamic adaptive programming in Java*, in: A. Yonezawa and S. Matsuoka, editors, *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS **2192** (2001), pp. 73–80.

[34] Pree, W., "Design Patterns for Object-Oriented Software Development," Addison-Wesley, Boston, MA, United States, 1995.

[35] Pulvermüller, E., A. Speck and A. Rashid, *Implementing collaboration-based designs using aspect-oriented programming*, in: *Proceedings of TOOLS-USA*, 2000, pp. 95–104.

[36] Smaragdakis, Y. and D. Batory, *Implementing reusable object-oriented components*, in: *Proceedings of the International Conference on Software Reuse* (1998), pp. 36–45.

[37] Squeak homepage, *http://www.squeak.org/*.

[38] VanHilst, M. and D. Notkin, *Using role components to implement collaboration-based designs*, in: *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1996), pp. 359–369.

[39] Vlissides, J. M., *Protection, part i: The hollywood principle*, C++ Report (1996).