A Pattern Language for the Design of Aspect Languages and Aspect Composition Frameworks

Uwe Zdun

New Media Lab, Department of Information Systems

Vienna University of Economics, Austria

zdun@acm.org

Abstract

Aspects avoid tangled solutions for cross-cutting design concerns. Unfortunately there are various reasons why it may be hard to use an aspect language or aspect composition framework as a solution, even though developers are faced with cross-cutting design concerns or tangled code structures. For instance, certain limitations of specific aspect composition frameworks might hinder the use of aspects. Or because of particular project requirements, such as constraints for the programming language or limitations of performance and memory, developers are not able to use an existing aspect composition framework. In such cases, developers would benefit from better understanding existing aspect composition frameworks. This would help developers to customize existing techniques or implement (simple) aspect composition frameworks from scratch. For these purposes, we present a pattern language for tracing and manipulating software structures and dependencies, and then explain different, existing aspect composition frameworks as sequences through this pattern language. We also evaluate alternative designs, common design trade-offs, and design decisions for implementing aspect composition frameworks.

1 Introduction

This paper addresses implementation techniques for composing (or weaving) aspects. Different composition frameworks for aspect-oriented programming (AOP) [19] are distinct but comparable. A number of languages, frameworks, and tools have been proposed for AOP. Up to date there are only a few works about the commonalities in these AOP approaches [23].

Some works propose an integrating terminology or model for different AOP approaches. Filman and Friedman [11], for instance, propose a generic definition of AOP. They understand AOP as quantified programmatic assertions over programs written by programmers oblivious to such assertions. Masuhara and Kiczales [23] implement a number of AOP concepts in a simple object-oriented language (namely pointcuts and advice, traversal specification, class hierarchy composition, and open classes). These implementations are used to compare how different AOP concepts realize modular cross-cutting. Other authors generalize from concrete AOP implementations by providing design languages for aspects. For instance, Clarke and

Walker [8] provide an UML-based modeling language for AOP concepts. Some authors propose formal models for specific aspect-oriented concepts. For instance, Lämmel and Stenzel [21] propose a formal model for method call interception.

The works summarized above provide an understanding of AOP beyond a single implementation. All these approaches focus on the *concepts* of AOP. In this paper, we take a different stance and concentrate on the *internal implementation* of aspect composition frameworks[1]. In most cases these implementations are not even visible to the aspect composition framework user.

To explain and compare the aspect composition framework implementations we use a pattern-based approach. The basic idea of our work is that the common and distinctive properties in the design of aspect composition framework implementations can be explained using a pattern language for tracing and manipulating software structures and dependencies [37]. In this paper, we discuss how the patterns are used for implementing successful aspect composition frameworks. The purpose is *not* to propose a new aspect concept, but to explain and evaluate existing technical solutions.

After describing the pattern language briefly, we will concentrate on *different* implementations of aspect composition frameworks. These will be explained as sequences through the pattern language, in particular:

- we discuss generative aspect language implementations using the example of AspectJ [18];
- we discuss Hyper/J [31] as an example of a byte-code manipulation approach using composition rules;
- we discuss JAC [27] and JBoss AOP [5] as two examples of load-time byte-code manipulation for implementing dynamic wrappers and message interception;
- we discuss aspects based on dynamic message redirection using the examples of XOTcl message interceptors [25] and message interceptors in popular middleware (e.g. [3, 16, 35]);
- we discuss aspects based on class graph traversals using the example of DJ [26];
- we discuss some other approaches with interesting characteristics briefly [36, 33, 21, 15].

Finally, we use the pattern language to evaluate alternatives, common design trade-offs, and design decisions for implementing aspect composition frameworks.

1.1 AOP Terms

Let us briefly introduce a few terms that have become quite well accepted in the AOP community (and that we use in the subsequent examples). These terms originate from the AspectJ [18] terminology. They describe the constituents of an *aspect* in a number of AOP approaches (note that there are other kinds of aspects as well):

- *Joinpoints* are specific, well-defined events in the control flow of the executed program.
- An *advice* is a behavior that is triggered by a certain event and that can be inserted into the control flow, when a specific joinpoint is reached. Advices allow one to transparently apply some behavior to a given control flow.
- *Pointcuts* are the glue between joinpoints and advices: a pointcut is a declaration that tells the aspect composition framework which advices have to be applied at which joinpoints.

[1] We use the generic term 'aspect composition framework' in this paper instead of 'aspect language' because some AOP implementations do not include an aspect language. Every aspect language, however, provides some framework for composing the aspects.

- *Introductions* change the structure of an object system. Typical introductions add methods or fields to an existing class or change the interfaces of an existing class. Introductions are not supported by all aspect composition frameworks.

1.2 Motivation and Target Audience

This paper targets at those developers who need to understand how aspect composition frameworks are realized. A conceptual understanding of the internals might be required for effectively working with an aspect composition framework. It might also help to select the best solution for a given task by comparing the different implementations' trade-offs. Finally, in cases where existing aspect composition frameworks and languages do not provide a suitable solution, developers either have to make additions or customizations to existing implementations or develop an in-house aspect composition framework. In the remainder of this section we discuss a number of motivations that might lead to one of these situations.

There are a number of potential practical problems with existing aspect composition framework implementations:

- Even though aspect composition frameworks exist for many programming languages, there are still many language without AOP support (especially in the legacy system context).

- If the computation environment is limited, as in embedded systems, it can be problematic to use the current AOP systems, as they produce some memory and performance overheads for their runtime environment; however, from a conceptual point of view the aspect concept can be used to reduce or eliminate such overheads. For instance, the small components project [33] implements (among other things) a project-specific aspect composition frameworks avoiding the overhead of a runtime environment.

- Sometimes it is simply a business decision that no third-party language extensions should be used in a project.

- Some aspect models are already quite complex languages. For solving simplistic AOP problems the required learning effort might be too large and writing a simple, project-specific aspect composition framework might be less effort.

A main motivation for this work is the observation that current aspect constructs cannot completely untangle all concerns, what might lead to novel aspect concepts and implementations in the future. This issue is discussed already in the AOP community. For instance, Kienzle and Guerraoui [20] report from their experiences that automatically aspectizing non-transaction code with transactions is doomed to failure. Separating transaction interfaces in aspects is also problematic, as it leads to an "artificial" separation of the transaction aspect from the object it applies to. Rashid and Chitchyan [28] propose AOP techniques as an effective means to modularized persistence. Yet they also identify the limitation that trade-offs between the reusability and performance of persistence aspects need to be made. The available AOP tools do not provide the optimal aspect constructs for implementing the persistence aspects. Soares et al. [30] have successfully used AspectJ to untangle the distribution and persistence aspects of a web-based information system. They also identify a number of drawbacks. For instance, (some) aspects might have unintended side-effects that are hard to foresee. There are also problems regarding the reuse of aspects, if the aspect specifications have to lexically refer to elements of the system they are applied in. In the concrete case, the AspectJ pointcuts had to adopt the system's naming conventions and thus were not reusable for other systems.

It is also important to note that the term "aspect" is broader than the AOP concepts currently realized by aspect composition frameworks. These primarily realize extensional (or sometimes called "superimposed") aspects that can be separated in an (object-oriented) language construct (e.g. a class-like structure such as an AspectJ aspect) and are executed for certain events in the call flow. However, this kind of aspects is just one possible interpretation of the term "aspect" in the realm of software engineering. Design disciplines, for example, know other interpretations, and there is no reason to believe that other interpretations are less relevant for the software engineering discipline. For instance, Mørch sees aspect-orientation as a way to interweave the aspects design, programming, and use of software [24]. In the context of reengineering, for example, it is important to be able to separate such aspects or extract them from existing source code. Existing AOP languages might help to architecturally separate some parts of these aspects, but such aspects can hardly be completely untangled.

2 Understanding Software Structure and Dependency Tracing and Manipulation: A Pattern Language

In this section we present a pattern language for tracing and manipulating software structures and dependencies. A *pattern* is a proved solution to a problem in a context, resolving a set of forces. In more detail, the context refers to a recurring set of situations in which the pattern applies. The problem refers to a set of goals and constraints that typically occur in this context, called the forces of the pattern. These forces are a set of factors that influence the particular solution to the pattern's problem.

As an element of language, a pattern is an instruction, which can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant [1]. A *pattern language* is a collection of patterns that solve the prevalent problems in a particular domain and context, and, as a language of patterns, it especially focuses on the pattern relationships in this domain and context.

In an aspect composition framework we require some way to find or trace the joinpoints in a software system. Further, we need to manipulate these spots to apply the aspects. Tracing and manipulating software structures and dependencies is a common problem in many software engineering fields. For instance, many *software maintenance and reengineering* projects need to find the existing structures and relationships in a software system. *Development tools*, such as IDEs, profilers, or architecture visualizations, need to trace structures and dependencies in the source code and/or the call flow. *Programming language implementations and programming language extensions* need to find existing structures and dependencies in the source code when parsing it. *Meta-object protocols (MOP) and meta-level architectures* require (runtime) structure information about their base-level objects in order to control them from the meta-level.

Even though these application fields are quite diverse, many implementations are based on a set of common patterns for structure and dependency tracing and manipulation. Because *aspect-oriented systems* have to interpret and potentially manipulate either the software structures or the call flow, they are often implemented with structure and dependency tracing and manipulation techniques as well. Note that some of the application fields are closely related to AOP, namely programming language extensions and MOPs.

In Figure 1 an overview of the pattern language for structure and dependency tracing and manipulation is presented. In the remainder of this section, we present each pattern briefly (explaining its context, problem, solution, and an example with a figure)[2]. Where possible, we present the pattern already in a typical AOP context or use an example that can be found in some AOP solution. Note that the individual patterns are not limited to the AOP domain; a more generic and detailed description of the patterns can be found in [37] (the two pattern BYTE CODE MANIPULATOR and COMMAND LANGUAGE are not explained

[2] We present patterns from the pattern language in SMALLCAPS font, external patterns are presented in *Italics*.

in [37]).

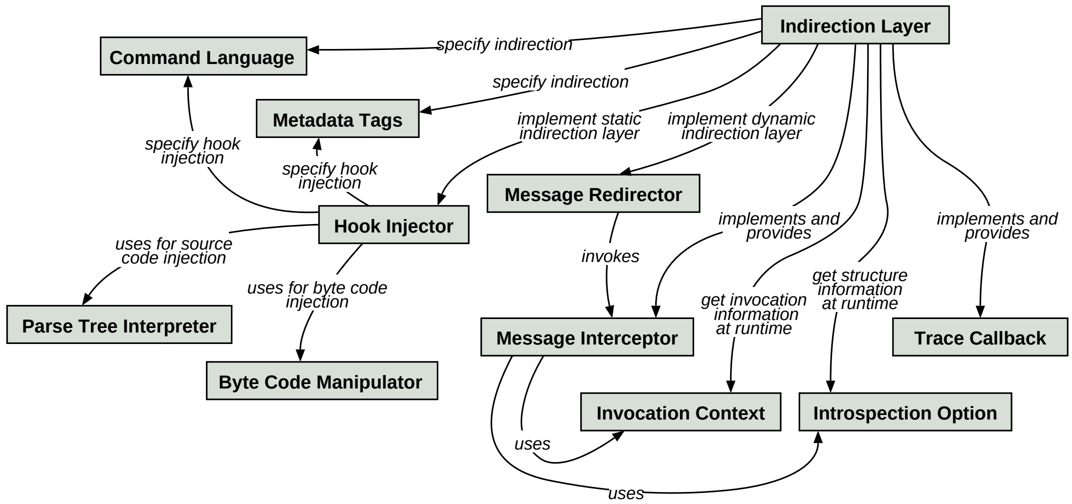

Figure 1. Important relationships of the patterns are represented by labeled arrows

2.1 Parse Tree Interpreter

Consider an aspect composition framework for composing aspects at compile time. It requires information from the source code (and perhaps other formal source documents). Consider further, a parser for the source language(s) is available and can be reused, or it does not seem too much effort to write a (full) parser for the source language(s). Note that, if the aspect composition framework offers an aspect language, this language has to be parsed as well.

A PARSE TREE INTERPRETER parses the sources using the (existing) source language parser and uses the parser's outputs to create a parse tree. The PARSE TREE INTERPRETER offers an API to interpret and possibly modify the parse tree in an application-specific way.

The example in Figure 2 shows a typical PARSE TREE INTERPRETER of an aspect composition framework. It first parses the source files and builds a token tree for each source document. Then the API of the PARSE TREE INTERPRETER is used to find and modify the joinpoints in focus of the aspects. Finally the modified code is emitted (and then handed to the aspect compiler).

2.2 Indirection Layer

In addition to the information available from source documents, most aspect composition frameworks also require runtime call flows (and data flows).

An INDIRECTION LAYER traces all relevant static and dynamic information at runtime. It is a *Layer* between the application logic and the instructions of the (sub-)system that should be traced. The general term "instructions" can refer to a whole programming language, but it can also refer to the public interface of a component or sub-system. The INDIRECTION LAYER wraps all accesses to the relevant sub-system and should not be bypassed. "Hooks" are provided to trace and manipulate the relevant information. Typical hooks provided by an aspect composition framework are the joinpoints.

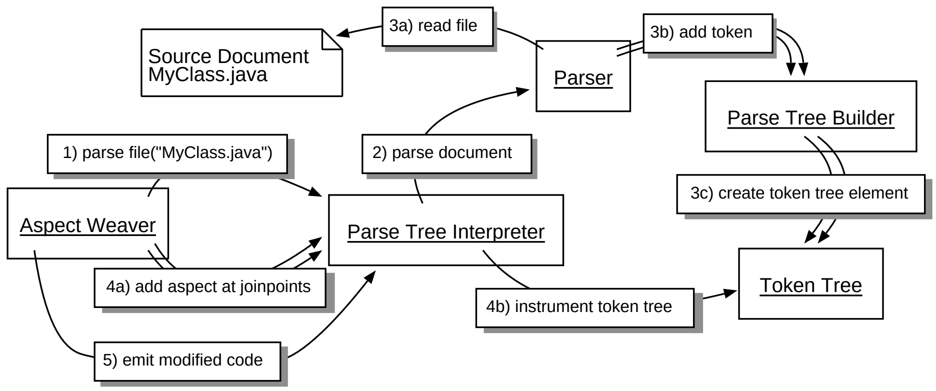

Figure 2. A parse tree interpreter of an aspect weaver

Figure 3 shows an INDIRECTION LAYER consisting of a number of wrappers. These wrappers are used for invoking elements of a sub-system. The wrappers provide hooks to invoke code before or after the actual invocation.

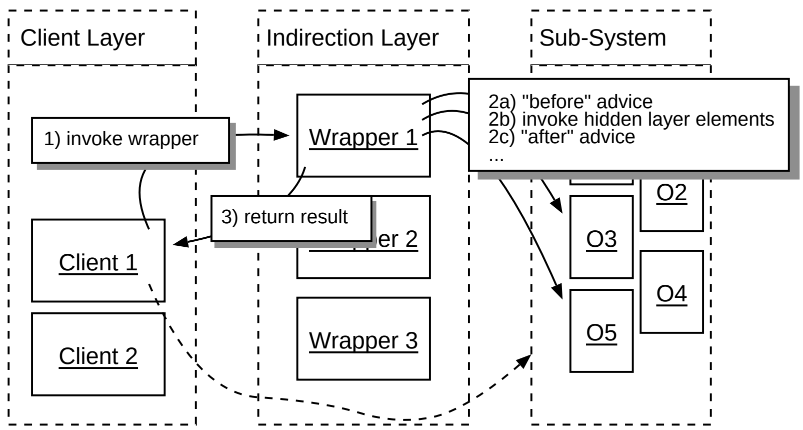

Figure 3. Indirection layer consisting of a number of wrappers

INDIRECTION LAYER is a generalization for patterns wrapping an implementation *Layer* [6] with a symbolic language, such as *Object System Layer* [14], *Microkernel* [6], *Virtual Machine* [13], *Interpreter* [12], and others. In this pattern language, we use the general INDIRECTION LAYER pattern as an abstraction for these individual patterns.

2.3 Trace Callback

Consider one or more specific structures of the runtime system need to be traced. It might not be known until runtime which structures are to be traced.

TRACE CALLBACKS permit a developer to trace a specific runtime structures without (large) performance penalties for untraced structures. A TRACE CALLBACK interface is provided by an INDIRECTION LAYER, and the structures to be traced are only accessed via the INDIRECTION LAYER. With this interface one can dynamically add or remove a TRACE CALLBACK for a specific runtime structure of the INDIRECTION LAYER. When adding or removing a TRACE CALLBACK, the developer specifies the type of the traced runtime structure, the callback event, and a callback operation. The callback operation is a user-defined operation handling the callback event. Whenever the callback event happens for the specified runtime structure, the callback operation is executed by the INDIRECTION LAYER implementation automatically.

Figure 4 shows the example of a TRACE CALLBACK for a method invocation. The INDIRECTION LAYER adds a trace to an object. When the specified method is invoked, the object invokes the TRACE CALLBACK automatically.

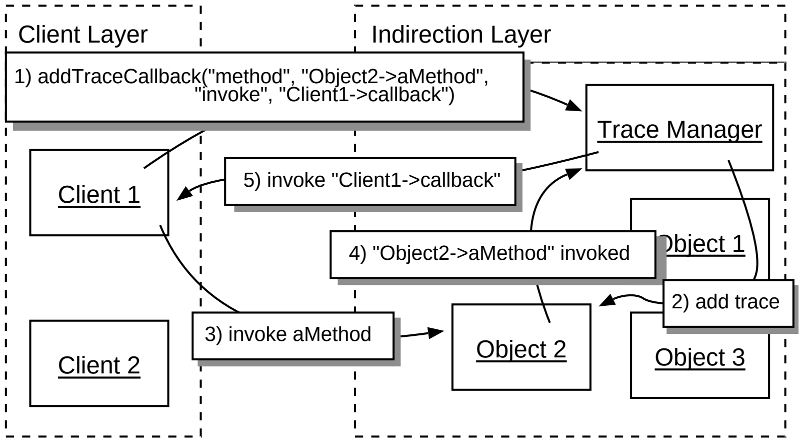

Figure 4. Trace callback for a method invocation

2.4 Message Redirector

An INDIRECTION LAYER is an intermediate layer between the application logic and a subsystem. In object-oriented systems that means it intercepts and adapts all individual messages that are sent from the application logic to the subsystem.

A MESSAGE REDIRECTOR is a *Facade* [12] to the INDIRECTION LAYER. Application layer objects do not access INDIRECTION LAYER objects directly, but send symbolic (e.g. string-based) invocations to the MESSAGE REDIRECTOR. The MESSAGE REDIRECTOR dispatches these invocations to the respective method and object. A MESSAGE REDIRECTOR has some benefits compared to an INDIRECTION LAYER with scattered wrappers. It provides transparent control over the complete call flow: the client does not need to know the wrapper or implementation object. As a central instance, a MESSAGE REDIRECTOR can handle issues cutting across multiple instances and manage common state (such as a call stack).

Figure 5 shows a MESSAGE REDIRECTOR of an INDIRECTION LAYER. All invocation have to pass it, thus it has complete control over the call flow of the INDIRECTION LAYER.

2.5 Byte Code Manipulator

Many programming languages, such as Java or Tcl, are not compiled to native machine code, but use an intermediate representation internally, called the byte code. The byte code is an interpreted language with a small and easy-to-understand set of instructions. These low-level instructions are interpreted by an INDIRECTION LAYER called the virtual machine. Consider, for instance, you want to implement some language extensions for aspects, but you have no access to the source code and/or you want to apply the aspects at load time (or even at runtime). In such cases it is an option to manipulate the byte code (instead of the source code). Even though the byte code is a low-level representation, it contains all relevant information about a program (for instance the Java byte code contains constants, interfaces, classes, methods, class attributes, exceptions, etc.). Typically you do not want to deal with low-level byte-code instructions.

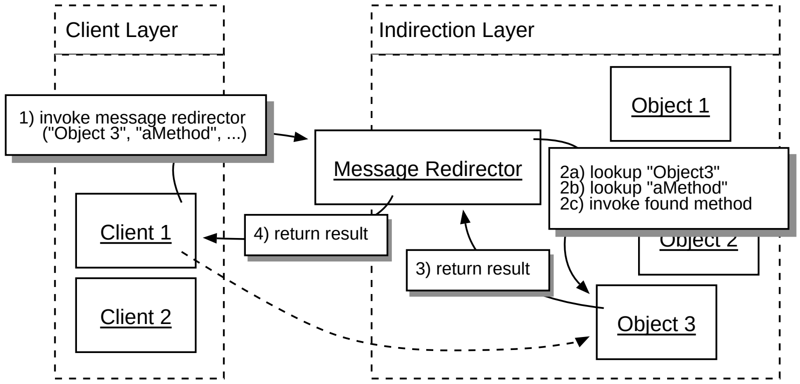

Figure 5. A message redirector is used to implement an indirection layer

As depicted in Figure 6, a BYTE CODE MANIPULATOR first looks up the program structure (in the figure from a class repository). Then the byte code is parsed and a runtime representation of the byte code is produced. For each language element represented in the byte code, the BYTE CODE MANIPULATOR offers an operation for querying them and manipulating them. The BYTE CODE MANIPULATOR can provide source-level abstractions (such as an operation `getMethods` to find all methods of a class and an operation `addMethod` to add a method to a class). Alternatively, it can provide a high-level API to access and manipulate the elements of the byte-code. Source-level abstractions are typically easier to understand than the byte-code details. Typically, there are some byte-code details that cannot directly be reflected in source-level abstractions.

For a language using a compiler that creates byte code (such as class files produced by a Java compiler), a BYTE CODE MANIPULATOR can either change the class files on disk, manipulate classes at load time in the class loader, or generate new classes at load time. For a (scripting) language that supports on-the-fly byte code compilation at runtime (such as Tcl), a BYTE CODE MANIPULATOR is used to handle the interaction between the script interpreter and the on-the-fly byte code compiler.

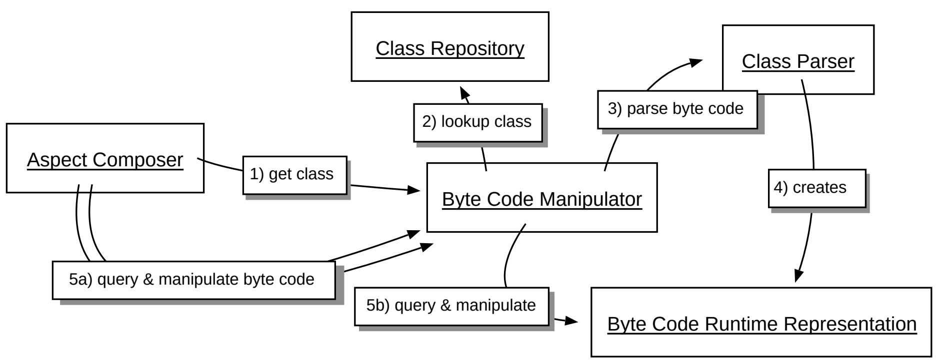

Figure 6. A byte code manipulator used by an aspect composer

2.6 Hook Injector

Structure tracing or behavior modification can be provided using a MESSAGE REDIRECTOR. But this usually requires the (sub-)system or its clients to be changed, what is not always possible or wanted.

A HOOK INJECTOR traces specific, well-defined points of a program by injecting indirection hooks at these points. This can be done using a PARSE TREE INTERPRETER or BYTE CODE MANIPULATOR. Either manipulated source code or byte code is emitted. This manipulated program is then compiled or interpreted, instead of the original program. Semantically the new code is equivalent to the original code, with the exception of the injected hooks for extracting or modifying the relevant invocations.

Figure 7 shows a HOOK INJECTOR that injects hooks by parsing a document, modifying the representation in memory (here a parse tree), and writing the modified source document back. This document is then interpreted or compiled, instead of the original source document.

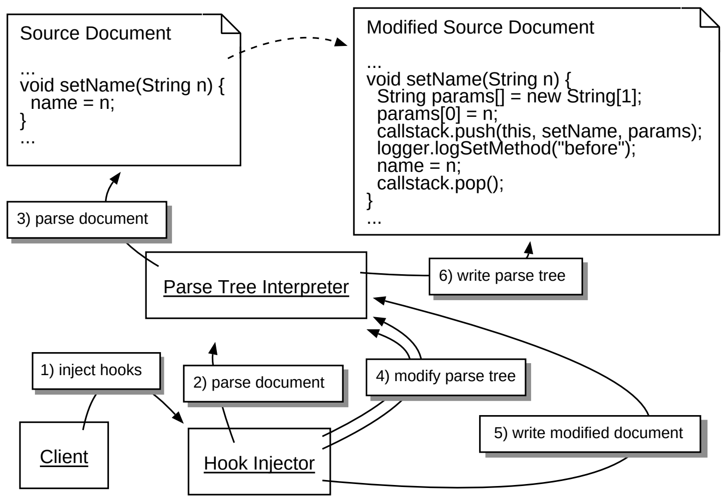

Figure 7. A hook injector implemented with a parse tree interpreter

2.7 Message Interceptor

Controlling the call flow in an INDIRECTION LAYER can be done either with a MESSAGE REDIRECTOR or a HOOK INJECTOR. These patterns provide only low-level support for tracing, modifying, or adapting of message invocations; that is, these tasks have to be hard-coded into the MESSAGE REDIRECTOR or hook implementations.

MESSAGE INTERCEPTORS express dynamic message traces, modifications, or adaptations as first-class entities of the INDIRECTION LAYER. The MESSAGE INTERCEPTORS are invoked for standardized invocation events observable in an INDIRECTION LAYER, such as “before” a method invocation, “after” a method invocation, or “instead-of” invoking a method. This can be done with a callback mechanism built into the INDIRECTION LAYER. The callback mechanism can be triggered either by a MESSAGE REDIRECTOR or the hooks of a HOOK INJECTOR. Optionally a MESSAGE INTERCEPTOR can specify conditions to be evaluated when the invocation event happens, and it is only executed, if the condition is true.

Figure 8 shows two MESSAGE INTERCEPTORS implemented with a MESSAGE REDIRECTOR. These are registered as “before interceptors” and are thus invoked before the actually invoked method. One of these MESSAGE INTERCEPTORS is also registered as an “after interceptor.” Thus it is also invoked after the method invocation returns.

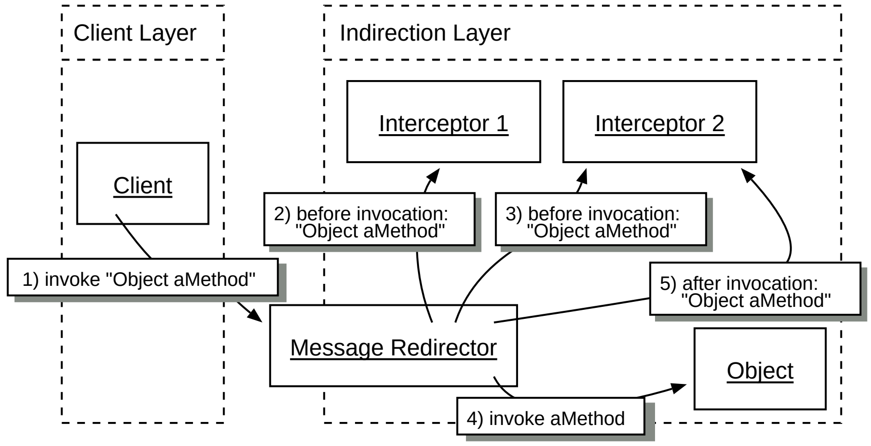

Figure 8. Message interceptors implemented with a message redirector

2.8 Introspection Option

An aspect composition framework requires information about many software structures and dependencies at runtime. These structures and dependencies include dynamic structures (that can change at runtime) as well as static structures (that are defined at compile time and do not change at runtime). But in most programming languages there is no integrated and extensible way to obtain this information at runtime.

INTROSPECTION OPTIONS gather and provide this information for the structures or dependencies of an INDIRECTION LAYER. All messages that are creating or changing structures or dependencies have to pass the INDIRECTION LAYER and can thus be traced. In addition to the pre-defined structures and dependencies, a simple extension API for adding new, domain-specific INTROSPECTION OPTIONS can be offered.

Figure 9 shows an example: the (wrapper) classes in the INDIRECTION LAYER implement a method "info" that provides INTROSPECTION OPTIONS for their structures and dependencies. Clients can access this method via the MESSAGE REDIRECTOR.

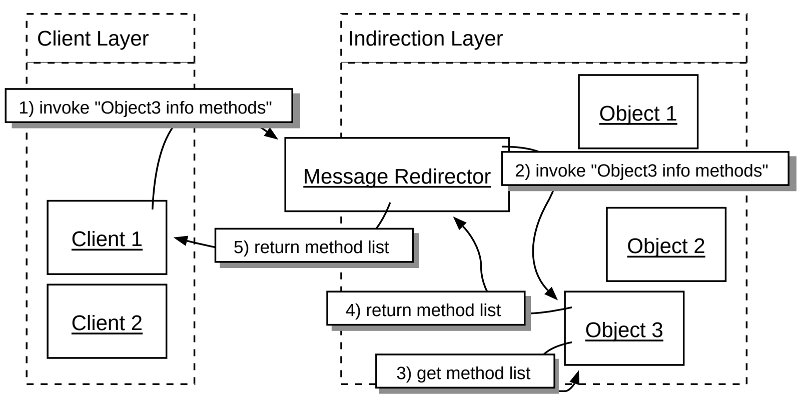

Figure 9. Introspection options for objects in an indirection layer

2.9 Invocation Context

Invocation information is important for object-oriented adaptations that rely on message exchanges. For instance, an aspect requires some knowledge about the invoking and invoked object and method. The invoking method should not have to provide the invocation information as a parameter because the aspect should be applied transparently.

An INVOCATION CONTEXT can be used to obtain the invocation information from inside of an invoked method or a wrapper method. The INVOCATION CONTEXT contains at least information to identify the calling and called method, object, and class. In a distributed context, location information for caller and callee are also required. INDIRECTION LAYERS usually maintain a callstack of runtime invocations, including information like caller, called object, invocation parameters, invoked method scope, and other per-call information. That means, in the context of an INDIRECTION LAYER the INVOCATION CONTEXT essentially is the top-level callstack entry.

Figure 10 shows a MESSAGE REDIRECTOR that puts each invocation onto a callstack. Thus an interceptor is able to obtain the current INVOCATION CONTEXT containing information about the intercepted invocation.

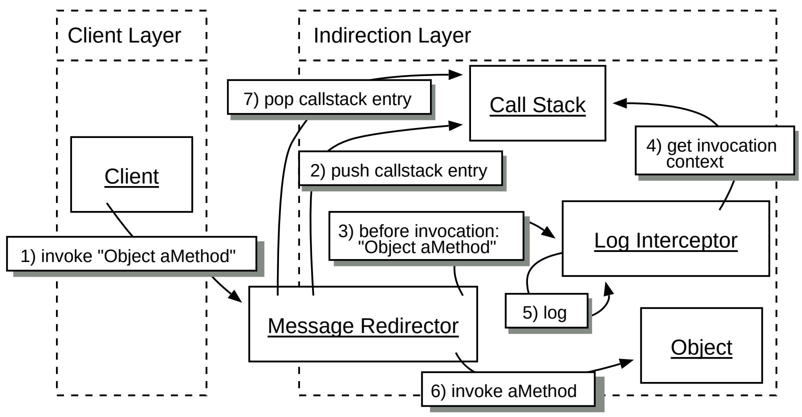

Figure 10. Invocation context obtained from a callstack

2.10 Metadata Tags

Consider some of the relevant information required for an aspect composition framework cannot be given in the source documents. For instance, there might be no proper language resource to specify the aspects.

A solution is to provide a standard notation for embedding METADATA TAGS in the code or in configuration files. These METADATA TAGS contain additional information (such as aspect specifications). Typically, METADATA TAGS are provided as hierarchical key/value lists.

Figure 11 shows the example of an aspect composer that obtains the aspect composition information from an XML-based configuration file. This metadata needs to be parsed and interpreted first, so that the aspect composer can query this information later on.

2.11 Command Language

METADATA TAGS are good for handling simple, structured configuration options in a declarative manner. It is hard, however, to deal with configuration options that require behavioral specifications and/or programming constructs, such as conditions,

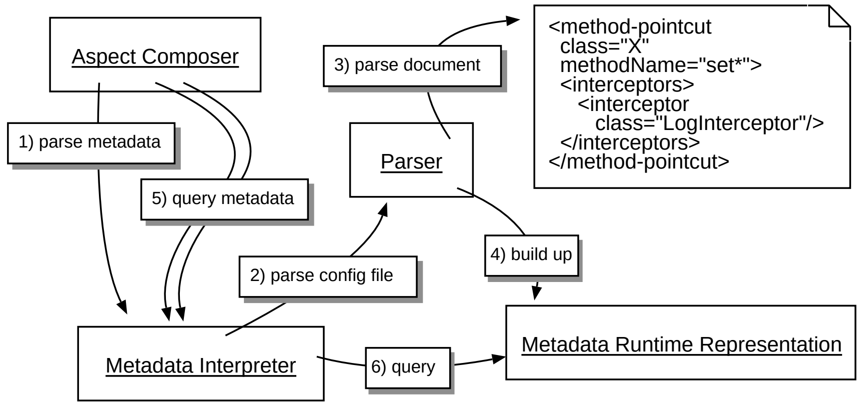

Figure 11. Aspect metadata tags in a configuration file

loops, blocks, substitutions, or expressions.

A solution is to extend the METADATA TAGS syntax to a COMMAND LANGUAGE. That means, each tag is implemented by one *Command* class [12], implemented in an INDIRECTION LAYER. An *Interpreter* [12] for the COMMAND LANGUAGE uses the symbolic names in the COMMAND LANGUAGE representation and maps them to the *Commands*.

Some *Commands* are pre-defined, but the user can provide new *Commands*. Typically, in a general purpose COMMAND LANGUAGE pre-defined *Commands* are conditions like `if`, loop statements like `while`, etc. In a COMMAND LANGUAGE for configuring an aspect composition framework, some *Commands* for annotating aspects are usually pre-defined. The user might be able to define customization *Commands* for specific aspects. For instance, the persistence aspect in Figure 12 requires a configuration of the objects or classes to be made persistent. That means, in this example, the pointcuts are configured using a COMMAND LANGUAGE.

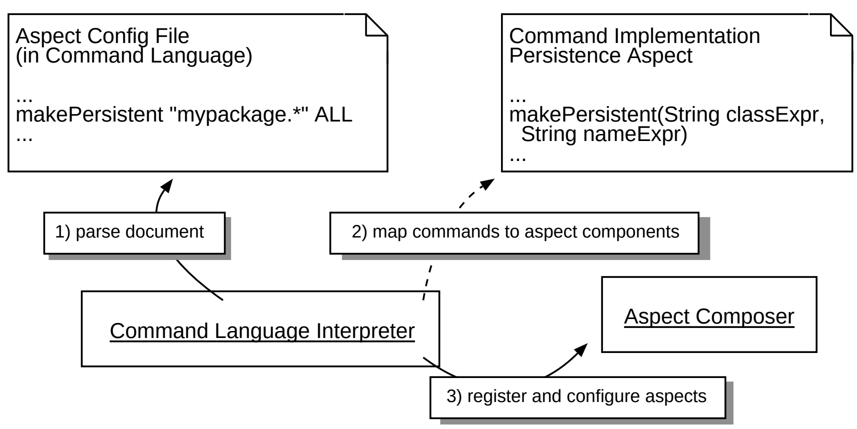

Figure 12. A persistence aspect configured using a command language

2.12 Other Patterns Used with the Pattern Language

A number of other patterns are often used together with the patterns of the pattern language.

A PARSE TREE INTERPRETER requires some means to traverse its parse tree. A typical use of such a traversal functionality is a HOOK INJECTOR. It needs to deal with all elements of the parse tree that are involved in the injection process. There are different patterns that provide solutions for this problem:

- A *Visitor* [12] can be used to visit all nodes of the parse tree and perform some operations implemented by another object.
- An *Iterator* [12] can be used to walk through all nodes in one pass, after the nodes have been linearized (e.g. as an in-order node list).
- A *Composite* [12] can be used to implement operations to be performed on all nodes of the tree.

Note that these patterns can also be used for traversing a runtime representation of the program, such as a class graph obtained with INTROSPECTION OPTIONS.

A PARSE TREE INTERPRETER and a BYTE CODE MANIPULATOR build a runtime representation of the parsed data. Typically there are different node types which have to be created and linked into a graph structure, such as a tree. A *Factory* [12] or a *Builder* [12] can be used to create the node structures.

When more than one MESSAGE INTERCEPTOR applies to a message invocation, the interceptors need to be applied in some order. Typically they are ordered as a *Chain of Responsibility* [12], and an *Iterator* is used to walk through the MESSAGE INTERCEPTOR chain. Some frameworks allow for structuring MESSAGE INTERCEPTORS as *Composites*.

A MESSAGE REDIRECTOR usually is a *Facade* [12] to its INDIRECTION LAYER.

3 Pattern Sequences for Aspect Composition Frameworks

In this section, we explain current aspect composition techniques as sequences through the pattern language. As Alexander points out [2], pattern descriptions alone do not really allow a person to generate a good design, step by step, because they concentrate on the content of the patterns rather than laying the emphasis on morphological unfolding. The creative power lays in the *sequences* in which the patterns are applied. For a given task, the number of possible sequences is huge compared with the number of sequences which work, that is by comparison, tiny. Thus it is important to document the inherent knowledge in the pattern language in form of sequence examples that have proved to work in practice. Discussing such pattern sequences for the technical implementation of aspect composition frameworks is the focus of the remainder of this paper. We also compare these sequences in order to evaluate the common trade-offs of implementing aspect composition frameworks in Section 4.

3.1 AspectJ

Currently, a common way to implement aspects are generative environments, such as AspectJ [18], D [22], or ComposeJ [36] (a tool for adding composition filters [4] to Java). Byte code manipulation as used by Hyper/J [31] or JAC [27] follow a similar sequence with some notable differences (see next sections). For illustration of the generative sequence, we will give some examples from AspectJ.

In AspectJ, the aspects are described in an extension of the Java language, consisting of a set of additional instructions. This aspect language is added to the code written in the base language. Consider a Java class `Point` exists, and we want to assert certain properties using AspectJ.

```
class Point  {
  void setX(int x) { ... }
  ...
}
```

As additional statements, AspectJ introduces the `aspect` statement, as well as pointcuts (`call`, `target`, `args`, etc.) and advices (`before`, `after`, `around`):

```
aspect PointAssertions {
  before(Point p, int x): target(p) && args(x)
      && call(void setX(int)) {
    if (x > 100 || x < 0) {
      System.out.println("Illegal value for x");
      return;
    }
}
```

The aspect contains one advice. The advice itself consists of an advice head, a pointcut definition, and an advice body. The advice head defines the type of advice (before, after, or around) and advice parameters. The pointcut definition defines a set of joinpoints where the advice has to be applied. The above pointcut includes all joinpoints that are a `call` to a `void` method `setX` with one `int` argument. The `target` type of the call must be a `Point`, and the first `int` argument is bound to the identifier `x`. The advice means that the `if` statement in the advice's body has to be executed before any of the specified joinpoints is reached.

Note that some of the elements of a joinpoint (such as `target` and `this`) can only be determined at runtime, as they depend on the control flow of the program. In contrast, the joinpoints for the `call` pointcut can be determined statically by analyzing the program text.

The advice body defines the actions to be taken before the joinpoint is executed. At runtime the advice parameters are passed to the advice body with their current values.

The AspectJ compiler composes the aspect language with the statements in the base language. The required information for this task is contained in the aspect language, the Java class and method structure, and the spots where invocations are sent or received (to handle the call flow). The AspectJ compiler inherits from a Java compiler class because the AspectJ syntax is an extended Java syntax.

A part of the AspectJ compiler is the AspectJ parser. The aspect language parser inherits from a Java parser. The AspectJ compiler implements a PARSE TREE INTERPRETER that parses the program text files and creates a parse tree. The AspectJ parser builds the parse tree for both the AspectJ and Java parts of the system. AspectJ uses an abstract syntax tree (AST) in which each Java or AspectJ statement is represented as a node tree object. For each type of statement there is a class inheriting from a class `ASTObject`. These are instantiated by an abstract syntax tree *Factory* (in the class `AST`).

A HOOK INJECTOR injects hooks at the respective joinpoints (a process called "inlining"). The aspect language code is replaced either by base language primitives or byte-code instructions of the virtual machine.

Parsing, parse tree interpretation, and hook injection are organized in multiple *compiler passes* that create and then transform the information in the abstract syntax tree by iterating over all the compilation units (here, an *Iterator* is used). After the initial parser pass, compiler passes for interpreting the Java structures (type graph, signatures bindings, forward references, etc.) follow. Then joinpoints are collected, static joinpoints are planned, control flow and exceptions are checked, and advices are planned and woven.

All (static) joinpoints are instantiated during these steps and plans for composing them are made. In the weaving pass, these plans are implemented. That means the joinpoints are woven in correct order and according to the connectors "and," "or," and "not."

Then a few helper passes follow. Finally either the passes for source code or byte code generation end the process of parsing and inlining.

These results of hook injection are not visible to the user. The HOOK INJECTOR inserts hooks into the existing base language program. The injected hooks invoke objects implementing an INDIRECTION LAYER that is part of the runtime environment of the woven application. The hooks together with respective implementations are a static form of MESSAGE INTERCEPTORS. In AspectJ the MESSAGE INTERCEPTOR implementation is realized as an advice; the injected hooks call the before, after, or around advices.

At runtime of the woven program, the aspect language runtime environment is implemented as an INDIRECTION LAYER. AspectJ implements a static weaving process. In other words, MESSAGE INTERCEPTORS cannot be composed at runtime. Some tasks, however, are handled dynamically by the INDIRECTION LAYER. For instance, AspectJ provides a partly dynamic joinpoint model and one can specify control flows (`cflow`) as pointcut elements (which are dynamically computed on a control flow stack).

The runtime environment of the INDIRECTION LAYER has to be bound to any AspectJ application. It contains the dynamic parts of the AspectJ joinpoint model and the `cflows`. This joinpoint implementation provides a variant of the INVOCATION CONTEXT pattern for aspects, containing invocation information for dynamic pointcuts, such as the `target` of a joinpoint, the current object (`this`) of a joinpoint, and the arguments of a joinpoint (`args`). Obviously, these information are invocation-specific and cannot be statically determined by the compiler. As we have seen in the above example, these information can be connected to the static part of a joinpoint. Thus the dynamic part of a joinpoint offers INTROSPECTION OPTIONS for static parts: the joinpoint's kind, signature, source location, and string representation.

As a benefit, the runtime joinpoint model permits pointcuts and advices to retrieve certain INVOCATION CONTEXT information about the call flow and the current joinpoint at runtime. As a drawback, the runtime environment consumes additional runtime resources.

As we can see in the following code fragment from the AspectJ code, the runtime joinpoint implementation offers the INVOCATION CONTEXT information, as well the connection to the static part (which offers the respective INTROSPECTION OPTIONS):

```
class JoinPointImpl implements JoinPoint {
  ...
  org.aspectj.lang.JoinPoint.StaticPart staticPart;
  ...
  public Object getThis() { return _this; }
  public Object getTarget() { return target; }
  public Object[] getArgs() { return args; }
  public String getKind() {
    return staticPart.getKind();
```

```
  }
  public Signature getSignature() {
    return staticPart.getSignature();
  }
  public SourceLocation getSourceLocation() {
    return staticPart.getSourceLocation();
  }
  public final String toString() {
    return staticPart.toString();
  }
  ...
}
```

INTROSPECTION OPTIONS are used to connect the static and the dynamic part of joinpoints. Some INTROSPECTION OPTIONS about the Java class and method structure are offered by the Java Reflection API.

In AspectJ the original class implementation can be extended with introductions. For instance, in the example above it would make sense to have a method for checking the assertion, but this method requires the self reference of the current object. Thus it should be a method of the `Point` class. An aspect can be used to introduce this method to the existing class:

```
aspect PointAssertions {
  private boolean Point.assertX(int x) {
    return (x <= 100 && x >= 0);
  }
  before(Point p, int x): target(p) && args(x)
      && call(void setX(int)) {
    if (!p.assertX) {
  ...
}
```

In a generative environment such introductions are implemented by injecting hooks into the respective classes with the HOOK INJECTOR.

3.2 Hyper/J

Hyper/J [31] supports multi-dimensional separation and integration of concerns [32] in Java. Object-oriented languages promote decomposition by classes as the single decomposition dimension. Unlike classes, other kinds of concerns cannot be encapsulated easily, and thus, their implementations is scattered across the class hierarchy. Hyper/J provides means to decompose a program according to these other concerns.

The Hyper/J tool provides in first place a sophisticated HOOK INJECTOR. The tool includes a BYTE CODE MANIPULATOR and thus the source code of the Java classes is not required. The hooks are injected according to specifications in three different kinds of files: hyperspace specification, concern mapping files, and hypermodule specifications. These contain primarily COMMAND LANGUAGE instructions describing how to (de-)compose the concerns.

- Hyperspaces identify the *dimensions* and *concerns* of importance and can be seen as a kind of project definition. Dimensions are used to group related concerns, as for instance: `SomeDimension.SomeConcern`.

- Concern mappings describe how various elements of Java classes address different concerns in a hyperspace. The following Java elements can be mapped to dimensions and concerns: package, class, interface, operation, and field.
- A hypermodule specification is a particular composition of the units in some selection of the concerns in the hyperspace. It identifies some dimension and concern names (so-called hyperslices) that are to be composed in the context of the hyperspace.

The composition in hypermodules follows general composition strategies that can be specified by the developer:

- `mergeByName` means that units in the same-named hyperslices are merged together into a new unit.
- `nonCorrespondingMerge` means that units in different hyperslices with the same name are not to be connected.
- `overrideByName` indicates that units in same-named hyperslices are connected by an override relationship: the last hyperslice in the hypermodule specification overrides the others.

In addition to these default composition strategies, exceptions can be declared using more specific composition rules. In these composition rules, units can be declared to equate each other, an order can be specified, units can be renamed, units can be merged or not, specific actions can be defined to override other actions, a given unit can be declared to match a set of units, and methods can be "bracketed." The bracket composition rule is of specific interest because it permits to define one operation as a MESSAGE INTERCEPTOR for another operations, for instance:

```
bracket "*"."foo*"
  from action Application.Concern.Class.bar
  before Feature.Logging.LoggedClass.invokeBefore($ClassName),
  after Feature.Logging.LoggedClass.invokeAfter($OperationName);
```

This declaration means that all methods whose names begin with `foo` in any class in the input hyperslices will be bracketed by the methods `Feature.Logging.LoggedClass.invokeBefore` and `Feature.Logging.LoggedClass.invokeAfter`. The optional `from` clause restricts the calling context from which the bracket methods will be invoked. For example, the `from` clause of the bracket relationship defined above indicates that the before and after methods should only be invoked when `foo` methods are called from within the method `Application.Concern.Class.bar`. Note that this is a static alternative to a callstack and/or an INVOCATION CONTEXT (which would be used in more dynamic environment to limit the caller of a MESSAGE INTERCEPTOR).

3.3 JAC

JAC [27] is a framework for dynamic, distributed aspect components in Java. Here, "dynamic" means that aspects can be deployed and un-deployed at runtime. To prepare the Java classes to be used with aspects, a BYTE CODE MANIPULATOR is applied at load time. As of JAC version 0.10 BCEL [9] is used, in earlier versions Javassist [7] was used. BCEL offer a high-level API to access and manipulate the byte-code details.

The BYTE CODE MANIPULATOR is used by the HOOK INJECTOR of JAC. The inserted hooks have the responsibility to indirect invocations into the JAC INDIRECTION LAYER that implements the JAC AOP features. There are three main features to support dynamic aspects in JAC: aspect components, dynamic wrappers, and domain-specific languages.

Aspect components are classes that subclass the class `AspectComponent`. In JAC a runtime meta-model called RTTI (runtime type information) is defined that provides INTROSPECTION OPTIONS for base program elements. Also, the RTTI can be used for structural changes at a per-class level (similar to AspectJ's introductions). Pointcuts can be defined to add before, after, or around behavior for base methods. In contrast to AspectJ, method's in focus of aspects are specified with strings and looked up reflectively (i.e. using INTROSPECTION OPTIONS). The pointcuts of the aspect components are used to invoke MESSAGE INTERCEPTORS, defined by dynamic wrappers (see below). For each pointcut, hooks are introduced at all respective joinpoints using the HOOK INJECTOR.

Dynamic wrappers can be seen as generic advice. They are implemented as classes extending the class `Wrapper`. Wrappers are ordered in wrapping chains. The method `proceed` can be invoked from inside of the wrapper method. It forwards the invocation to next wrapper in the wrapper chain, and finally to the wrapped method.

The methods of the wrapper have a parameter of the type `Interaction`. This class implements an INVOCATION CONTEXT containing information about the wrapped object, method, arguments, and wrapping chain of the invocation.

A third feature of JAC are *domain-specific languages* that can be defined for configuring aspects. Instead of using simple METADATA TAGS, JAC provides a COMMAND LANGUAGE that can be extended by the user: operations of the aspect component can be provided as *Command* implementations and invoked from the configuration file. This way each aspect can define its own configuration language.

For instance, the predefined authentication aspect component of JAC offers the following configuration method:

```
void addTrustedUser(String username);
```

This method can be invoked from within the COMMAND LANGUAGE script:

```
addTrustedUser "renaud" "renaud"
```

3.4 JBoss AOP

The JBoss Java application server contains a stand-alone aspect composition framework [5]. It implements a similar sequence of the pattern language as JAC, but there are some interesting differences in the design decisions.

Hooks are injected into all "advisable" classes using a HOOK INJECTOR that internally uses the BYTE CODE MANIPULATOR Javassist [7] at load time. Javassist provides a source-level abstraction of the byte-code.

An advice is implemented as a MESSAGE INTERCEPTOR. All MESSAGE INTERCEPTORS must implement the following interface:

```
public interface Interceptor {
   public String getName();
   public InvocationResponse invoke(Invocation invocation) throws Throwable;
}
```

The name returned by `getName` is a symbolic interceptor name. `invoke` is a callback method to be called whenever the advice is to executed. The parameter of the type `Invocation` is an INVOCATION CONTEXT containing the invocation and method parameters. In contrast to many other frameworks, the response is not stored in the `Invocation` object as well, but in the `InvocationResponse` object that is returned.

Forwarding to the next interceptor and finally to the intercepted method is done using a method `invokeNext` of the INVOCATION CONTEXT object:

```
invocation.invokeNext();
```

All pointcut definitions are given using XML-based METADATA TAGS, for instance:

```
<interceptor-pointcut class="mypackage.MyClass">
    <interceptors>
       <interceptor class="TracingInterceptor" />
    </interceptors>
</interceptor-pointcut>
```

The class attribute of interceptor pointcut can take any regular expression as argument. For instance, the expression `mypackage.*` can be used to intercept all messages sent to members of a particular package `mypackage`.

To support some level of composition, one can predefine a set of interceptor chains that can be referenced in any interceptor-pointcut (so-called interceptor stacks). JBoss AOP provides an runtime interface to manipulate the interceptors of an advised class. Interceptors can be appended, pre-pended, or removed from the interceptor chain at runtime.

The INVOCATION CONTEXT provides an INTROSPECTION OPTION to access specific METADATA TAGS at runtime. This way aspects can use METADATA TAGS for aspect configuration.

Introduction-pointcuts can be defined using the METADATA TAGS as well. These introduce one or more interfaces to a class plus a mixin class that implements these interfaces.

3.5 Extended Object Tcl (XOTcl)

Sometimes dynamic composition of aspects is required. JBoss AOP, for instance, supports dynamic configuration of MESSAGE INTERCEPTORS. Note that this does not imply a dynamic aspect composition framework: the aspectized classes are instrumented by a BYTE CODE MANIPULATOR at load time.

A really dynamic aspect composition process can be implemented when the pattern MESSAGE REDIRECTOR is used together with an INDIRECTION LAYER. A MESSAGE REDIRECTOR receives symbolic invocations that are indirected to the actual implementations of all objects in the system. Thus, a MESSAGE INTERCEPTOR can dynamically intercept any message in the call flow, when it is dispatched by the MESSAGE REDIRECTOR.

XOTcl [25] is an object-oriented Tcl variant that uses the pattern MESSAGE REDIRECTOR for its implementation. The symbolic invocations received by the MESSAGE REDIRECTOR are strings extracted from the program code. These invocations are indirected to the Tcl or XOTcl implementation (written in C), or other loaded components.

The idea of applying aspects as dynamic MESSAGE INTERCEPTORS on top of a (given) MESSAGE REDIRECTOR architecture is quite simple: if we specify all those calls that are in focus of an aspect as criteria for the MESSAGE INTERCEPTOR, and let the MESSAGE REDIRECTOR execute this MESSAGE INTERCEPTOR every time such messages are called, we can implement any aspect that relies on message exchanges. To receive the necessary information for dealing with the invocations, the MESSAGE INTERCEPTOR should be able to obtain the INVOCATION CONTEXT to find out which method was called on which object (the callee). Often the calling object and method are required as well. INTROSPECTION OPTIONS are typically used to obtain structure information.

For instance, the XOTcl code corresponding to the above AspectJ point class example in Section 3.1 looks as follows:

```
Class Point
...
Class PointAssertions
PointAssertions instproc assertX x {
```

```
  if {$x <= 100 && $x >= 0} {return 0}
  return 1
}
PointAssertions instproc setX x {
  if {[my assertX $x]} {
    puts "Illegal value for x"
  } else {
    next
  }
}
Point instmixin PointAssertions
```

At first, the corresponding code for the class and the aspect (here also implemented as a class) is defined. Then dynamically one of these classes is registered as an instance mixin (a class-based MESSAGE INTERCEPTOR) for all points; thus all calls to the method `setX` are intercepted by the `PointAssertion` mixin's same-named method `setX`.

There are two common ways to ensure the non-invasiveness of aspects (i.e. the obliviousness property in the terminology of Filman and Friedman [11]) when using mixins:

- Mixins can be applied to a superclass or interface, and are automatically applied to all subclasses in the class hierarchy. Thus developers of subclasses can be oblivious to the aspect.
- A mixin can be registered for a set of classes using INTROSPECTION OPTIONS. For instance, one can apply a mixin for all class names starting with `Point*`. This way mixins can be applied in a non-invasive way for any kind of criteria (pointcuts) that can be specified using the dynamic INTROSPECTION OPTIONS of XOTcl.

The instruction `next` is responsible for forwarding the invocation. It thus handles (non-invasive) ordering of the MESSAGE INTERCEPTORS in a *Chain of Responsibility* [12]. At the end of the chain comes the actually invoked method. Thus the placement of the `next` instruction enables us to implement before, after, or around behavior of the MESSAGE INTERCEPTOR.

In contrast to AspectJ, we do not have to "introduce" the method `assertX` on `Point`, as the mixin shares its object identity with the class it extends. However, in other cases we might want to change the class structure. In XOTcl at any time a new method can be defined (because all XOTcl structures are fully dynamic). Such kinds of dynamics require INTROSPECTION OPTIONS to ensure that we do not violate some architectural constraints by re-structuring the architecture. For instance, in the example above we can first check at runtime that there is no method `assertX` defined for `Point` yet, before we introduce it:

```
if {[Point info instprocs assertX] == ""} {
  Point instproc assertX x {
    if {$x <= 100 && $x >= 0} {return 0}
    return 1
  }
}
```

In case of XOTcl (and Tcl) also TRACE CALLBACKS for variable slots are supported. That is, we can dynamically observe specified variables, when they are accessed in the INDIRECTION LAYER. This mechanism is similar to MESSAGE INTERCEPTORS, but it is of a finer granularity, as we can observe single variables. That means, in most cases, a TRACE CALLBACK has a smaller performance impact than a MESSAGE INTERCEPTOR, but it is not well applicable for observing larger structures with multiple callbacks.

For instance, the following code invokes a TRACE CALLBACK method `vartracer`, whenever the variable `x` is read or written. As the trace is introduced in the constructor `init`, the variable `x` of any `Point` instance is traced:

```
Point instproc vartracer {var sub op} {
  puts "[self]->$var accessed"
}
Point instproc init args {
  ...
  my trace variable x rw "[self] vartracer"
  ...
}
```

3.6 Apache Axis Handler Chains

MESSAGE INTERCEPTORS based on a MESSAGE REDIRECTOR can also be found in the many distributed object systems. Here, the symbolic invocations received by the MESSAGE REDIRECTOR are the remote calls that are sent across the network. The *Interceptor* pattern [29] describes this variant of MESSAGE INTERCEPTORS.

Apache Axis [3] implements a MESSAGE INTERCEPTOR framework, as it can be found in many distributed object systems (other examples are Orbix [16] or Tao [35]). Note that these MESSAGE INTERCEPTOR are not primarily designed for AOP, but can be used as an infrastructure for an AOP solution.

In Axis remote messages, sent from a client to a server, are handled both on client side and server side by handler objects, arranged in a *Chain of Responsibility*. Each handler provides an operation `invoke` that implements the handler's task. This operation is invoked whenever a message passes the handler in the handler chain. The *Client Proxy* [34] passes each request message through the client side handlers until the last handler in the chain is reached. This last handler (in Axis called the "sender") is responsible for sending the message across the network using the Axis framework. On server side, the request message is received by the *Server Request Handler* [34] and passed through the server handler chain until the last handler (in Axis called the "provider") is reached. The provider actually invokes the web service object. After the web service object has returned, the provider turns the request into a response. The response is passed in reverse order through the respective handler chains – first on server side and then on client side.

All handlers between *Client Proxy* and sender on client side and all handlers between *Server Request Handler* and provider are MESSAGE INTERCEPTORS. The *Client Proxy* and the *Server Request Handler* act as MESSAGE REDIRECTORS that indirect remote message into the handler chain.

An INVOCATION CONTEXT (in Axis called the `MessageContext`) has to be created first, before the message is sent through the handler chain. This way different handlers can retrieve the data of the message and can possible manipulate it. An INVOCATION CONTEXT is used on client and server side. Each message context object is associated with two message objects, one for the request message and one for the response message.

Note that this infrastructure alone is not an AOP framework. What is missing is a way to specify pointcuts and apply these. This can be done quite easily by hand, because all necessary information is provided to the MESSAGE INTERCEPTORS in the INVOCATION CONTEXT. With this information only one type of joinpoints can be specified: remote invocations.

A simple implementation variant is to make `invoke` a *Template Method*, defined for an abstract class `AspectHandler`. All aspect handlers inherit from this class, and implement the method `applyAspect`. This method is only called, if there is a pointcut for the current aspect and message context defined.

```
public abstract class AspectHandler extends BasicHandler {
  public boolean checkPointcuts(MessageContext msgContext) {
    // check whether pointcuts apply and return true/false
    ...
  }
  public void invoke(MessageContext msgContext) throws AxisFault {
    if (checkPointcuts(msgContext) == true) {
      applyAspect(msgContext);
    }
  }
  abstract public void applyAspect(MessageContext msgContext);
}
```

3.7 DJ and DemeterJ

DJ (and DemeterJ) [26] is a variant of the dynamic aspect composition scheme that does not provide MESSAGE INTERCEPTORS but uses traversal strategies and *Visitors* [12]. A class graph can be traversed as follows:

```
static final ClassGraph cg = new ClassGraph();
...
cg.traverse(this,
  "from Schema via ->TypeDef,attrs,* to Attribute",
  new Visitor() {
    void before(Attribute host) {
      if (host.name.equals("name"))
        def.add(host.value);
    }
  });
...
```

The `cg` object is a MESSAGE REDIRECTOR that first creates a (reusable) traversal graph, and then the object structure is traversed. In DJ, information about the class graph are obtained as INTROSPECTION OPTIONS (in Demeter/J class dictionary files and behavior files are used). At each step in a traversal, the fields and methods of the current object, as well as methods of the *Visitor* object, are inspected and invoked by INTROSPECTION OPTIONS that are obtained via Java's Reflection API.

For the *Visitors* the programmer can define typed before and after methods that are executed before and after an object of a certain type is traversed. Also methods to be executed for start and finish of the traversal can be defined.

3.8 Some other Aspect Composition Frameworks

In this section, we want to discuss some interesting aspects of other solutions that have not been discussed before.

In ComposeJ [36] one can provide a composition filter [4], consisting of a Java part and a composition filter part (written in the composition filter syntax). The composition filter part is written in a COMMAND LANGUAGE that refers to the Java class name. It tells the ComposeJ compiler how to compose the base classes with the composition filter class.

The small components project [33] implements a projects-specific aspect composition framework (among other things), solely using generative techniques. The goal is to avoid overheads of a aspect runtime environment in embedded systems. To reach this goal all compositional information is given as METADATA TAGS. In particular, the class structure is given as an XMI model, and there are template classes, metaclasses, mapping files, and config files that define various component composition aspects (such as structures, instantiation, threading parameters, instance-container mapping, and security settings).

There are various approaches that combine some of the implementation approaches. For instance, there are approaches that integrate the benefits of the dynamic aspect composition into generative environments. The solution in [21] is a generative aspect model; however, it allows for activating and deactivating aspects at runtime. This is done via a central registry for aspects. This registry serves as a central MESSAGE REDIRECTOR for which every class is registered that contains a `superimpose` statement. A HOOK INJECTOR injects hooks into each method of these classes. The hooks call the registry in case of a method `dispatch`, `enter`, or `exit` event. If an corresponding MESSAGE INTERCEPTOR is registered as an advice, it is called by the registry before the original call.

Sometimes it makes sense to apply different aspect interpretation models together. For instance, AJDC (AspectJ Design Checker) [15] is an extension of AspectJ that uses TyRuBa [10] as a logic meta-programming engine for finding errors and problems in AspectJ code. AJDC generates facts and rules out of the parse tree to be compiled, and it provides some rules for retrieving trace information like subclass relationships. Then TyRuBa is used to interpret this output. There are three additional statements understood by AJDC to define errors and problems in AspectJ code, and within them the TyRuBa syntax is embedded as a pointcut language.

4 Evaluation of the Pattern Sequences

As explained before, applying the individual patterns does not necessarily lead to a successful solution. Thus it is important to understand the sequences or combinations of patterns that work in practice. We have explained many individual sequences or combinations of patterns in the previous section. In this section we want to categorize and revisit the general sequences behind these successful solutions. At first, we compare the main features exposed in the individual solutions. Next, we describe the generic pattern sequences and discuss their trade-offs.

4.1 Comparing Features of the Pattern Sequences

In this section, we summarize the main design decisions for an aspect composition framework, based on the most important features of the aspect composition frameworks that we have discussed in the previous section. As Figure 13 illustrates in a feature diagram [17], there are five main design decisions to be considered:

- Which functionalities should the aspects provide?
- In which way should the joinpoints be extracted?
- How should aspects be composed with the system (and with each other)?
- Which context or environment information are available for the application of aspects?
- In which aspect language or aspect configuration syntax should the aspects be specified?

Note that these are technical considerations for the implementation of an aspect composition framework. However, of course, the design decisions for these alternatives are heavily influenced by the concepts to be realized by the aspect composition framework.

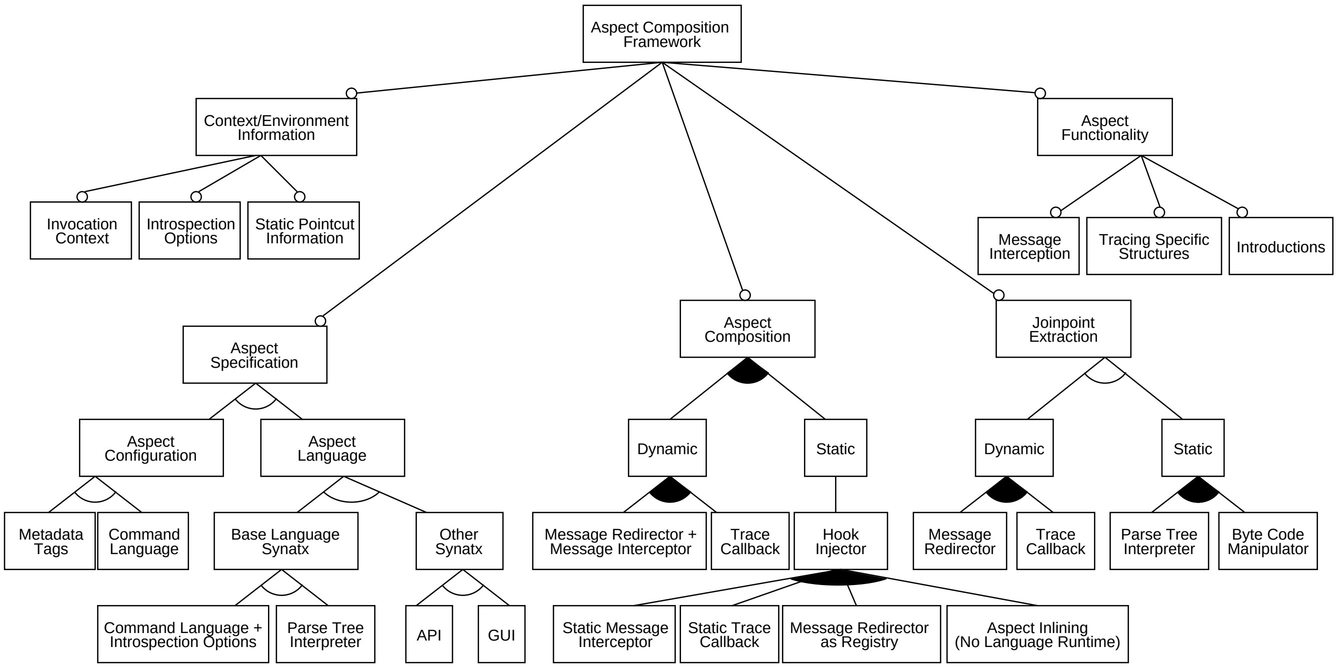

Figure 13. Feature diagram for aspect composition frameworks based on the pattern language

The first, obvious distinction of aspect composition frameworks are the main functionalities provided by aspects:

- Most aspect languages are able to manipulate message invocations. This can be done by inlining aspect code or with MESSAGE INTERCEPTORS.
- Alternatively, specific structures can be in focus of an aspect. This can be done by inlining aspect code or with TRACE CALLBACKS.
- Some aspect languages support introductions as well.

An obvious commonality in different aspect composition frameworks is that we require some joinpoint model. Each aspect sequence provides some way to extract this information. The pattern language offers the following alternatives:

- There are two alternatives for static joinpoint extraction:
  - We can use a PARSE TREE INTERPRETER to extract (and manipulate) the information in the source code.
  - We can use a BYTE CODE MANIPULATOR to find (and manipulate) the joinpoints in a given byte code.
- There are two alternatives for joinpoint extraction at runtime:
  - We can use a MESSAGE REDIRECTOR that can trace all invocations of a (sub-)system.
  - We can use a TRACE CALLBACK that traces all accesses to a specific structure, such as a variable.

Once we have extracted the relevant joinpoints, the aspect can be composed. Again, an aspect can either be composed statically or dynamically:

- The primary pattern for implementing static composition is HOOK INJECTOR. The HOOK INJECTOR can either produce program code in the base language or manipulate the byte code. Again there are multiple alternatives:
  - The HOOK INJECTOR can inline the complete aspect code at any joinpoint. This bloats the resulting byte code or source code, and does not allow for central management of joinpoints at runtime. But this variant can minimize the runtime overhead.
  - The HOOK INJECTOR can inline an invocation of a static MESSAGE INTERCEPTOR dealing with the message.
  - The HOOK INJECTOR can inline an invocation of a static TRACE CALLBACK.
  - The HOOK INJECTOR can inline an invocation of a MESSAGE REDIRECTOR that invokes a MESSAGE INTERCEPTOR. This variant allows for limited dynamics, such as turning aspects on and off.
- There are two variants of dynamic aspect composition:
  - MESSAGE INTERCEPTORS are registered within a MESSAGE REDIRECTOR.
  - TRACE CALLBACKS can be registered for each structure in focus of an aspect.

Note that only the first variant that completely inlines aspects does not require an INDIRECTION LAYER at runtime. The more dynamic variants usually have a larger performance (and memory) impact than the less dynamic ones.

Aspects are applied in the context of a pointcut. In many cases, an advice (or introduction) needs extra information from the context of the invocation or the runtime environment:

- The most simple way to provide such information is to pass the static information provided in the pointcut specification (for instance as parameters) to the advice implementation.
- Only when an INDIRECTION LAYER is supported, we can build dynamic INVOCATION CONTEXT information (which can for instance be used to implement dynamic joinpoints).
- If the language structures are dynamic, we may also require dynamic INTROSPECTION OPTIONS. They can be language supported by a reflection API (as in Java). Often there are additional INTROSPECTION OPTIONS for aspect structures. INTROSPECTION OPTIONS can be used to interweave dynamic parts and static parts of the aspect model, as discussed for the static joinpoint parts and dynamic joinpoint parts in AspectJ.

Another important issue is how aspects and pointcuts can be specified:

- We can provide an aspect language:
  - If we want to use the base language syntax in a compiled language, we typically use a PARSE TREE INTERPRETER together with a HOOK INJECTOR.
  - If we want to use the base language syntax in a dynamic, interpreted language, we can alternatively extend the language with new aspect constructs. To write the system back into a file (after manipulations), we use a program serializer. The serializer introspects the structures using INTROSPECTION OPTIONS and re-builds the program text. For this variant it is necessary that all program structures are introspective and can be serialized.

- The most simple way to implement an aspect composition framework is to provide a simple API in the base language. Using such an aspect extension is more cumbersome than using some dedicated aspect language constructs. This variant is often used for MESSAGE INTERCEPTORS in middleware environments.
- Some approaches provide a GUI-based aspect configuration. This variant can internally use any of the other variants.

• We can also use an aspect configuration language:

- For simple aspect configurations we can annotate a program with METADATA TAGS containing the aspect.
- For more sophisticated aspect specifications we can use a COMMAND LANGUAGE.

4.2 Revisiting the Pattern Sequences and Trade-offs

In this section, we want to revisit the most important combinations of patterns in the solutions discussed in Section 3 and discuss the common trade-offs of different sequences.

The sequence for composing aspects generatively using a PARSE TREE INTERPRETER, as used in AspectJ, is depicted in Figure 14. The PARSE TREE INTERPRETER is used for extracting the information with a parse tree, including the specification of pointcuts and introductions. The HOOK INJECTOR injects hooks and introductions. Optionally, an INDIRECTION LAYER is added that allows for limited runtime dynamics and introspection.

A benefit of this sequence is that aspects can be expressed in a syntax similar to the base language. It is easy to change and extend the aspect language because a full parser is available. Potentially a generative aspect composition framework can offer a performance close to the base language (this largely depends on how much dynamics are supported in the INDIRECTION LAYER). The main drawback of a PARSE TREE INTERPRETER based aspect composition framework is that dynamic aspect composition is not possible. However, a MESSAGE REDIRECTOR can be used to turn aspects on and off, as in the approach of Laemmel et al. [21] (what influences the performance negatively). Another main drawback is that the source code of all aspectized classes has to be be available. If a suitable parser is not available this sequences requires a substantial implementation effort.

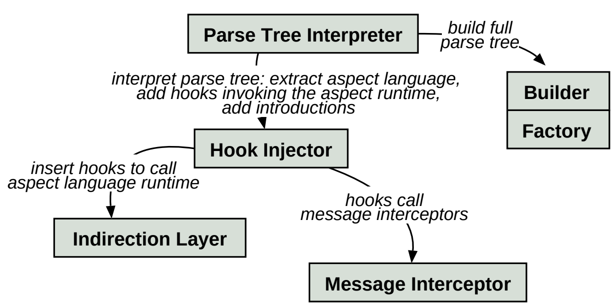

Figure 14. Aspect composition based on a parse tree interpreter

The sequence in Figure 15 is used by the load time composition approaches Hyper/J, JAC, and JBoss. Here, a HOOK INJECTOR use a BYTE CODE MANIPULATOR to read in classes as byte code, add hooks and introductions, and generate new classes. The pointcuts and introductions are specified either as METADATA TAGS or in a COMMAND LANGUAGE. Hooks are injected for introductions, calls to an INDIRECTION LAYER, and MESSAGE INTERCEPTORS. We have seen that the INDIRECTION LAYER can either provide dynamics and introspection for the MESSAGE INTERCEPTORS, as in JAC and JBoss, or the MESSAGE INTERCEPTORS can be static and the context information is hard-coded in the pointcuts, as in Hyper/J.

The general benefits compared to the PARSE TREE INTERPRETER based sequence is that this sequence supports load time aspect composition. That means that the source code does not need to be available, and thus, third party code can be instrumented. Another benefit is that the binding time is postponed. This is important for server environments, such as JAC or JBoss, that support deployment of classes while the server runs. Because this sequence does not assume that the base language code is accessible, it does not make much sense to specify the aspects in a base language syntax. Thus, another way to configure the aspects has to be found. This can be done by using METADATA TAGS or a COMMAND LANGUAGE. Again the performance impact largely depends on how the INDIRECTION LAYER is designed, but there is a residual performance impact for all aspectized classes, even if the aspects are not applied.

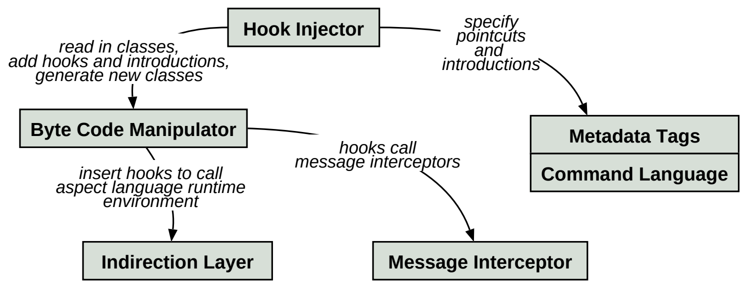

Figure 15. Aspect composition based on a hook injector and a byte code manipulator

Binding at compile time or load time, as in the two sequences explained before, is not always enough. In environments that are dynamic at runtime, a runtime aspect composition is required. Dynamic MESSAGE INTERCEPTORS are provided by many different environments; especially by programming languages such as XOTcl [25] and by middleware environments, such as Orbix [16], Tao [35], or Apache Axis [3] (also described in the *Interceptor* pattern [29]). Dynamic aspect composition can be implemented in these approaches using the following sequence: each invocation in the application logic layer is evaluated using the MESSAGE REDIRECTOR. The MESSAGE REDIRECTOR maps the called symbolic instruction to an implementation object in the INDIRECTION LAYER. Here MESSAGE INTERCEPTORS and TRACE CALLBACKS can be applied. If the MESSAGE REDIRECTOR implements a COMMAND LANGUAGE, the dynamic language features can be used to implement introductions (as in XOTcl).

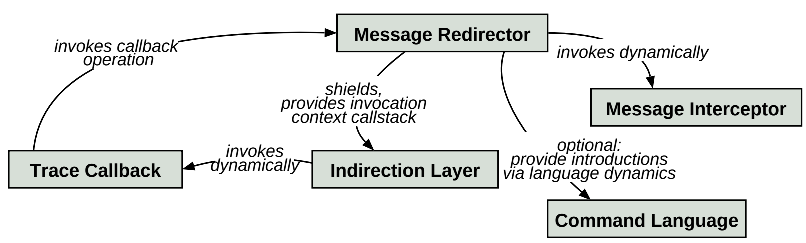

Figure 16. Aspect composition based on a message redirector

The main benefit of using a MESSAGE REDIRECTOR is that both aspect composition and applying the MESSAGE INTER-

CEPTORS is completely dynamic. A MESSAGE REDIRECTOR always has a performance penalty due to the indirection. But in the discussed environments (XOTcl and distributed object frameworks) the MESSAGE REDIRECTOR is used even if AOP is not used at all. A benefit compared to the two HOOK INJECTOR variants that change the aspectized classes' implementation (either the source code or byte code) is that there is no further overhead, when the aspect is not applied.

It requires less work to implement a MESSAGE REDIRECTOR than a BYTE CODE MANIPULATOR or PARSE TREE INTERPRETER. In a MESSAGE REDIRECTOR it is easy to provide INVOCATION CONTEXTS, INTROSPECTION OPTIONS, etc. because all necessary information have to go through the MESSAGE REDIRECTOR. In contrast to injected hooks that are scattered across the code, a MESSAGE REDIRECTOR is a central runtime instance and thus can be used for managing aspects at runtime. A MESSAGE REDIRECTOR can only provide the base language syntax, if it is part of the language implementation; in other words, the language has to be a COMMAND LANGUAGE (such as XOTcl). If it is not used already in a system, the MESSAGE REDIRECTOR requires either the client or the aspectized sub-system to be changed.

By combining the two patterns MESSAGE REDIRECTOR or HOOK INJECTOR, some of the drawbacks of either solution can be avoided.

The pattern INDIRECTION LAYER is central for all AOP sequences that require some runtime indirection or aspect management. If an INDIRECTION LAYER is supported together with MESSAGE INTERCEPTORS the sequences introduced before are typically extended by the sequence depicted in Figure 17. Note that the degree of dynamics of MESSAGE INTERCEPTORS and TRACE CALLBACKS, context information in INVOCATION CONTEXTS, and details of INTROSPECTION OPTIONS varies. The main benefit of adding an INDIRECTION LAYER is that it can be used to provide information such as INVOCATION CONTEXTS and INTROSPECTION OPTIONS at runtime. Also, it can be used to make the MESSAGE INTERCEPTORS dynamically configurable. The drawbacks of this sequence are the performance and memory overheads required for each addition.

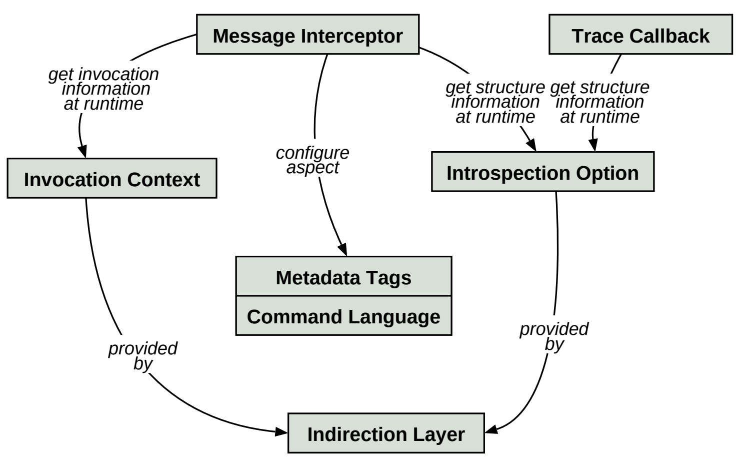

Figure 17. Message interceptor with an indirection layer at runtime

TRACE CALLBACKS are an alternative to MESSAGE INTERCEPTORS. TRACE CALLBACKS are applied locally, within the implementations of the structures to be traced. As a benefit, they have a slighter performance impact than MESSAGE INTERCEPTORS for untraced structures. MESSAGE INTERCEPTORS are applied by some central instance (a MESSAGE REDIRECTOR or a HOOK INJECTOR). Thus it is easier to trace complex structures in a non-invasive way.

Aspects can be specified in the programming language syntax (as in AspectJ or XOTcl). An alternative is to use an aspect configuration syntax. Here we can use METADATA TAGS or a COMMAND LANGUAGE. If no COMMAND LANGUAGE is available, implementing one is much more work than a simple METADATA TAGS syntax (such as XML). However, a COMMAND LANGUAGE as in XOTcl or JAC can be used to provide a domain-specific language for aspect configuration rapidly. Of course, this language has to be understood by developers, and it is more work to understand a COMMAND LANGUAGE than METADATA TAGS.

5 Conclusion

We have described current AOP implementation approaches using a pattern language for structure and dependency tracing and manipulation. In this realm we believe the patterns capture the major implementation variants of existing solutions. The forces and consequences of the patterns lead to the choice of appropriate patterns and pattern variants. Central, domain-specific issues like performance, flexibility, memory usage, program length, program complexity, learning effort, etc. are highly different in different solutions. Pattern language sequences were used to illustrate the existing solutions. The sequences should help developers to better understand existing AOP implementation choices. This understanding should enable developers to use, customize, or implement aspect composition frameworks. Also we can use the patterns and pattern sequences to discuss the main trade-offs of current AOP implementations, so that developers have clear criteria to select for the aspect functionalities and properties as required for a particular task. If there is no sufficient implementation existing, these criteria also help to choose the appropriate patterns for developing an in-house, domain-specific aspect composition framework for a particular project.

Acknowledgments

This paper is a substantially extended version of a paper ("Using Structure and Dependency Tracing Patterns for Aspect Composition") that was presented at the 3rd Workshop on Aspect-Oriented Software Development, Essen, Germany, March, 2003.

Thanks to Stefan Hanenberg for his helpful comments on this paper and to Markus Völter for discussion of the pattern language. Many thanks to Steve Berczuk who provided a lot of valuable feedback as a EuroPLoP 2003 shepherd for the pattern language.

References

[1] C. Alexander. *The Timeless Way of Building*. Oxford Univ. Press, 1979.

[2] C. Alexander and others. Patternlanguage.com. http://www.patternlanguage.com, 2001.

[3] Apache Software Foundation. Apache Axis. http://ws.apache.org/axis/, 2003.

[4] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, Oct 2001.

[5] B. Burke. JBoss aspect oriented programming. http://www.jboss.org/developers/projects/jboss/aop.jsp, 2003.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[7] S. Chiba. Javassist. http://www.csg.is.titech.ac.jp/ chiba/javassist/, 2003.

[8] S. Clarke and R. Walker. Towards a standard design language for AOSD. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 120–129, Enschede, The Netherlands, April 2002.

[9] M. Dahm. The bytecode engineering library (BCEL). http://jakarta.apache.org/bcel/, 2003.

[10] K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[11] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.

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

[13] J. Garcia-Martin and M. Sutil-Martin. Virtual machines and abstract compilers - towards a compiler pattern language. In *Proceedings of EuroPlop 2000*, pages 375–396, Irsee, Germany, July 2000.

[14] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceedings of EuroPlop 2000*, pages 397–410, Irsee, Germany, July 2000.

[15] S. Hanenberg and R. Unland. Specifying aspect-oriented design constraints in AspectJ. In *Workshop on Tools for Aspect-Oriented Software Development at OOPSLA 2002*, pages 641–655, Seattle, USA, Nov 2002.

[16] IONA Technologies Ltd. The orbix architecture, August 1993.

[17] K. C. Kang, S. G. Sholom, J. Hess, W. E. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute (SEI), 1990.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct 2001.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[20] J. Kienzle and R. Guerraoui. AOP: does it make sense? The case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, Malaga, Spain, June 2002.

[21] R. Lämmel and C. Stenzel. Semantics and Implementation of Method-Call Interception. Draft; http://homepages.cwi.nl/∼ralf/sdmci/; accepted for publication in IEE Proceedings Software, 2003.

[22] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Dec 1997.

[23] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 2–28, Darmstadt, Germany, July 2003.

[24] A. I. Mørch. Aspect-oriented tailoring of object-oriented applications. In *Proceedings of the 21st Information System Research Seminar in Scandinavia (IRIS 21)*, pages 641–655, Aalborg University, Denmark, August 1998.

[25] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, pages 163–174, Austin, Texas, USA, February 2000.

[26] D. Orleans and K. Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, pages 73–80, Kyoto, Japan, Sep 2001.

[27] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: a flexible framework for AOP in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, pages 1–24, Kyoto, Japan, Sep 2001.

[28] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129, Boston, Massachusetts, USA, March 2003.

[29] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[30] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02*, pages 174–190, Seattle, WA, USA, November 2002.

[31] P. Tarr. Hyper/J. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm, 2003.

[32] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107–119, Los Angeles, CA, USA, May 1999.

[33] M. Voelter. Small Components Project. http://www.voelter.de/projects/smallComponents.html, 2003.

[34] M. Voelter, M. Kircher, and U. Zdun. Object-oriented remoting: A pattern language. In *Proceedings of The First Nordic Conference on Pattern Languages of Programs (VikingPLoP 2002)*, pages 201–226, Denmark, Sep 2002.

[35] N. Wang, K. Parameswaran, and D. C. Schmidt. Meta-programming mechanisms for object request broker middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, San Antonio, TX, USA, Jan/Feb 2001.

[36] H. Wichman and others. ComposeJ Homepage. http://trese.cs.utwente.nl/prototypes/composeJ/, 2002.

[37] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.