# Patterns of Tracing Software Structures and Dependencies

Uwe Zdun
*New Media Lab, Department of Information Systems*
*Vienna University of Economics and BA, Austria*
zdun@acm.org

Tracing the structures and dependencies in existing, formal source documents, such as code or design specifications, or in the running software system is required for many different software engineering areas. Typical uses include reengineering tools, programming languages and language extensions, aspect composition frameworks, self documentation, and visualizations. There are many recurring techniques used to gather and manipulate the relevant trace information. This paper presents a pattern language covering common techniques in this area, as well as technology projections and known uses. The target audience of this paper are developers who want to develop new solution or modify an existing solution in one of the named software engineering areas, as well as developers who want to gain a deeper understanding of the internal workings of the software engineering tools and frameworks they use.

## Introduction

**Structure and Dependency Tracing**

Structure and dependency tracing primarily means to extract some knowledge from existing source documents of a software system or from the running system. Typical source documents are the source code, documentations, requirements specifications, and design documents. In this paper, we primarily deal with techniques that extract and manipulate information on static structures, such as pre-defined class graphs or class membership relationships, and dynamic structures, such as call graphs, from a software system.

There are different fields in which techniques for structure and dependency tracing are applied, including:

- *Software maintenance and reengineering* projects are often concerned with finding the structures and relationships in software systems. Typical maintenance and reengineering tasks (that should be supported) are architecture recovery, code analysis, refactoring, visualization, metrics computation, tracing dependencies, history analysis, consistency checking, finding systems' problems/hot spots, analysis of quality attributes, grouping, integration, and wrapping.
- *Development tools*, such as IDEs, performance analyzers, architecture visualizations, etc., also need to trace structures and dependencies in the source code.

- *Programming language implementations and programming language extensions* need to find existing structures and dependencies in the source code when parsing it. They also require knowledge of runtime structures for implementing a language runtime or support of high-level tasks such as reflection, traces, or message interception.

- *Aspect composition frameworks* are used inside of aspect languages and other aspect-oriented solutions [KLM+97]: they find the structure and control flows to which the aspects are applied and introduce the aspects at these points. In [Zdu04] we discuss how the patterns presented in this paper are applied in a number of popular aspect composition frameworks, such as AspectJ [KHH+01], HyperJ [TOHS99], D [Lop97], ComposeJ [W+02], or JAC [PSDF01].

- *Meta-object protocols and meta-level architectures* require (runtime) structure information about their base-level objects in order to control them from the meta-level.

- *Component gluing and configuration* mechanisms provide some means to compose software components in a customized way, either statically at compile time (or load time) or dynamically at runtime. At the composition time, the component composition mechanisms requires a knowledge about the (current) architectural configuration.

Even though these application fields are quite diverse, many implementations are based on a set of common techniques for structure and dependency tracing. These are in focus of our pattern language. In the Known Uses section, following the pattern language, we illustrate these commonalities, when we discuss some concrete implementations from these application fields.

In all the application fields, listed above, tracing is needed, but it is not the main task. That is, in many projects it must be "cheap". In many situations developers, therefore, should search for existing implementations of the patterns that can be used or tailored to the particular problem task, before considering to implement the patterns from scratch.

**Pattern Language Outline**

We present a pattern language consisting of basic, technical patterns that provide successful solutions for implementing such techniques and tools. Important tasks of structure and dependency tracing, concerned in this paper and required in the application fields named above, are:

- extracting relevant structures and dependencies, the trace information, either from source documents or from the running system,

- modifying or adapting the system using the trace information (either statically in the source documents or dynamically in the running system),

- documenting missing architecture knowledge that is yet missing in the source documents, and

- providing an integrated extraction framework for trace information.

In this paper, we first explain what we mean with the term "trace information." Next we present an overview of the pattern language. Then we present the individual patterns. Finally we provide some as known uses examples.

## Trace Information

Tracing structures and dependencies is mainly done by extracting *trace information*. On the basis of extracted trace information, adaptations or modifications of the system can be performed. In this section, we explain the term "trace information" in detail.

**Example: Reengineering to the Web**

Consider developing a web interface for a given, huge legacy system. To write suitable wrappers for the legacy system, at first, it is necessary to aquire an understanding of the overall system architecture. Next the relevant architecture fragments and their APIs have to be found. To find out which architecture fragments are relevant, we have to recover the features they realize. An understanding of the dependencies of these architecture fragments among each other is also required.

There are different sources of information. In the ideal case all required information is well documented. Requirement specifications or design documents may also provide useful information. In many legacy reengineering contexts, however, the code is the only (textual) basis of information. It is quite typical that legacy code that has evolved over a long time is neither well structured nor well documented.

In the context of tracing dependencies and working with views, redundancy of information cannot always be avoided and, if so, consistency has to be ensured. Manually finding dependencies that are neither explicit language relationships nor one-to-one correspondences is hard and error-prone. If information is collected from a legacy system, the consistency of this collected information has to be ensured as well.

**Extracting Trace Information**

As a solution, we can trace the structural dependencies and associated data automatically, either from existing electronic documents (like code, design documents, etc.), from additional documents, or from the running system. Trace information contains information on static structures, such as pre-defined class graphs or class membership relationships, and dynamic structures, such as call graphs.

To produce the trace information it might be necessary that developers markup missing dependencies and metadata (e.g. within one of the documents), but the actual extraction process should require no manual interference.

Documentation in separate documents is often forgotten during maintenance, and self documentation [Mey97] requires less discipline from developers. In other words, it is easier to maintain trace information sources that are part of or close to the program code than maintaining additional documentation artifacts. An example for trace information that is usually better handled in separate documents is history data (that is, information documenting dependencies in time). A typical way to derive history data is to compare the dependency graphs produced from different versions of the same source document. This information should be kept separated from the source document instances.

The patterns in the pattern language presented mainly provide means to document and automatically extract various kinds of trace information. Some of the patterns can also be used to manipulate this information. In this paper we do not deal with information that cannot be extracted directly from the structures of the system, such as logging information, warnings, error messages, memory structures, timing information, etc. Of course, there are applications where such other information need to be combined with trace information – but the other information must be extracted with other techniques than those documented in the patterns presented in this paper.

## A Pattern Language for Extracting Trace Information

In this section, we provide short thumbnails for the patterns, give a pattern language map, and explain the pattern form.

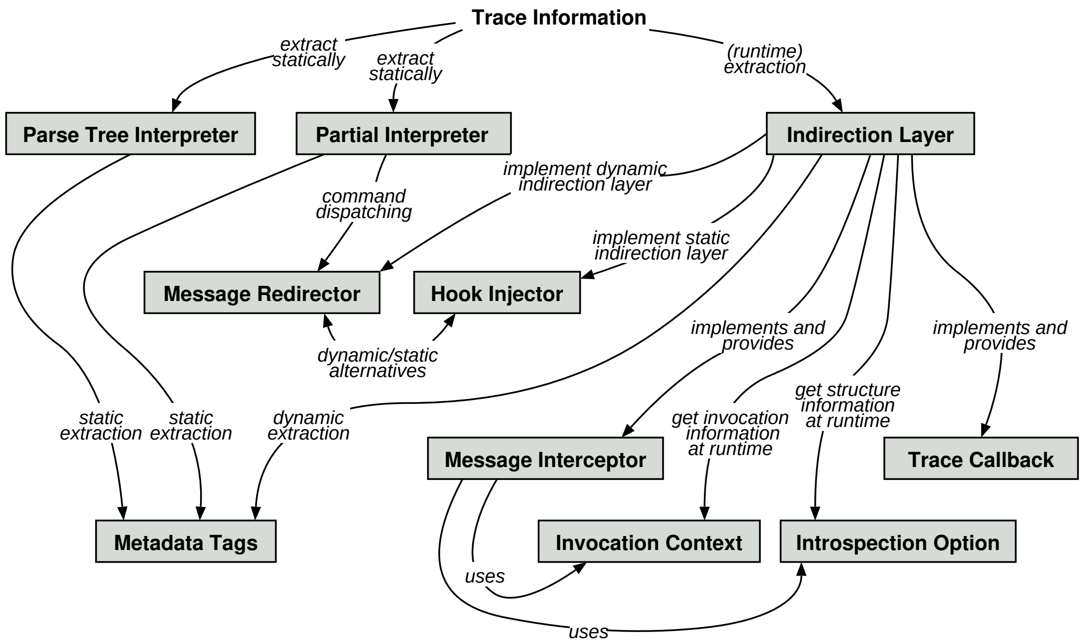
**Pattern Thumbnails**

The individual patterns in the pattern language are:

- P uses a full-fledged parser for the used programming language and extracts or modifies static trace information from the parse tree.
- I hides a sub-system, component, or layer, and provides access to this sub-system, component, or layer using an indirection concept, such as an object system, a virtual machine abstraction, an interpretation mechanisms, or wrapping concept.
- T provides a tracing mechanism for structures of an , such as variable slots, operations, or objects. A can be dynamically added and removed by developers.
- M can be used to implement a dynamic, object-oriented : it redirects symbolic invocations to implementations. During the redirection, trace information can be extracted and operation invocations can be modified or adapted.
- H implement a static by injecting hook invocations into the existing program (either in the code, bytecode, or machine code). These hooks invoke operations for extracting, modifying, or adapting trace information.
- M trace, adapt, or modify operation invocations in a dynamic and transparent fashion. They are typically invoked by a or - ' hooks.
- I expose structural trace information from an at runtime.
- I expose invocation-specific trace information.
- M add missing trace information as embedded metadata elements.
- P provides a limited interpreter for a language to statically extract specific trace information.

**Pattern Form**

The form of our patterns is a variant of the Alexandrian form. Each pattern starts with a name. It is followed by the context of the pattern in the language. After that, the problem is described in bold face. Then in plain face, more problem details are given to illustrate the problem and the forces of the pattern, for instance, with one or more problem scenarios. Then, after the word “Therefore” follows the solution, again in bold face. A figure illustrating the pattern and an example of the solution come next. Finally, a discussion of variants, related pattern, and




In the pattern map the most important relationships of the patterns are represented by labeled arrows.

consequences ends the pattern description. S font is used for internal patterns names, *emphasized* font for external patterns.

## Parse Tree Interpreter

**Context** You want to extract trace information from a system, and perhaps modify the system using the trace information.

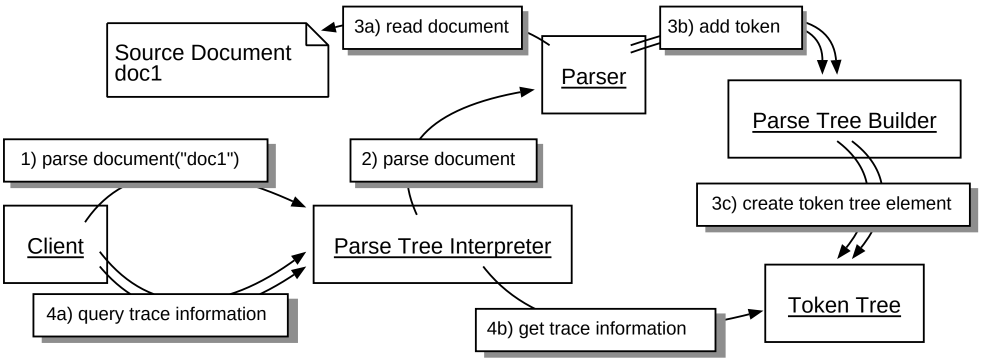
**Problem** **Static trace information, as available from the code or other (formal) source documents, is required. A parser for the source document's language is available and can be reused, or it does not seem too much effort to write a (full) parser for the source documents' languages. But the relevant trace information still has to be extracted from parser's outputs and provided in suitable form to the application logic. How to extract (and possibly modify) these trace information?**

**Problem Details** Consider an architectural view of the class graph of a Java application should be created. Typical applications of a class graph are to calculate metrics or visualize the class graph. In Java, all relevant information for this task is given in the source files. In particular, we have to find all class definitions and class relationships in the source code somehow.

When you want to extract simple trace information from a source document, it might be enough to write a little parser that only finds this kind of trace information. As more requirements for trace information arise, often this little parser is not enough anymore. Writing a full-fledged parser (e.g. for a programming language) is a considerable effort. Often existing source language parsers are usable, but sometimes the output format is not (directly) suitable.

In many cases, a client can directly use a parser's outputs and, for instance, interpret them as an event flow. The advantage of this is that the client has much control over trace information extraction. But when trace information extraction requires more complex, application-specific interpretation, the corresponding extraction code is tangled in the client. That means the client code gets more complex. It also means in many cases that the interpretation code in the client it is not reusable for other clients. If it is necessary to traverse or navigate through the trace information, flow-based output formats can be cumbersome to work with.

**Solution** **Therefore: Parse the source language documents with the (given) source language parser and use its outputs to create a parse tree. Provide a suitable interface for tree traversal. Then use this interface to build an application-specific that offers operations to extract the required trace information. If trace information modification is required, the tree traversal interface additionally provides means to modify the parse tree elements and structure, and the offers operations to modify the trace information application-specifically.**




A typical parses a source document and builds some knowledge representation of the trace information contained in the document. It also provides an application-specific API for querying the trace information.

**Example** Consider again you want to extract the Java class structure. Parsers for Java are available. Consider your parser creates a token stream, one token for each lexical element in the Java code. After a scanner has scanned the code for all tokens, a method `nextToken` of a tokenizer class extracts the next lexical token as a string.

In a token tree *builder* [GHJV94], this method is repetitiously called with a loop and all relevant statements are extracted. For each token containing a statement, the *builder* creates a new node for the parse tree. In case of a statement in which other statements can nest (like a `class` statement), we have to create all nested statements as child nodes. The *builder* can use existing tree and traversal API implementations (like a DOM [W3C00] tree).

The uses the parser and tree builder to create its internal parse tree representation. For clients it provides operations for extracting the Java class structure, such as `getClasses`, `getSuperclass`, or `getInterfaces`. Note that these operations provide application-specific interpretations of the trace information in the parse tree.

**Discussion** To build a the parser's output has to be converted into a tree representation. There are different parser outputs possible, such as token or event streams, parse trees, and others. Some tree-based parser outputs offer a suitable parse tree representation, others have to be converted. Token or event streams also have to be converted into a tree representation. If a conversion of the parsed data is necessary, we have to walk through the whole parsed data in the order of appearance in the source document, and call *factories* [GHJV94] or *builders* [GHJV94] for the targeted tree representation.

The pattern *parser/builder* [Ber96] reads in data from a stream and builds up a respective, generic object structure using a *factory* [GHJV94]. The pattern can be used for the parsing part of a if there is a need to interpret a raw data stream with a generic tree structure.

The parse tree representation should be "suitable," meaning that the tree's node structures should represent the language structure and that the offered tree traversal methods should support the application-specific requirements of the application effectively and efficiently:

- Regarding the tree's node structure, in case of a programming language, typically one node object per statement is created (it is also possible to create one node per lexical token, but this is usually overkill). Nested statements are represented as child nodes. Attributes of statements are represented as node object properties. Each node object type represented one kind of statement in the source language.

- Regarding the offered tree traversal methods the parse tree can either offer a tree traversal API to navigate through the tree and/or a *visitor* [GHJV94] that step-by-step visits all nodes of a (sub-)tree.

The interpretation step following the parsing step is implemented on the - class. This class uses the domain knowledge about the interpreted language and the required trace information to effectively and efficiently extract (and perhaps modify) the trace information. Note that different, application-specific classes can reuse the same parse tree and parser implementations.

If modification of data is supported, usually an operation to write the modifications back into the source language is supported. This can be done with a *visitor* that visits all elements of the tree and converts them into the source language format.

A avoids implementing a new parser for the source language. However, some existing parsers have very complex outputs, bugs, or other practical problems that cause more work than writing a simple would cause. A can only extract information that are (statically) given in the source documents.

## Indirection Layer

**Context** You want to extract trace information from a system or sub-system.

**Problem** **Trace information can consist of information in electronic documents (such as the source code), but also of information derived from dynamic invocation data (and data flows). It is hard to integrate these two kinds of trace information, as the former is statically provided in the sources, whereas the later is obtained from the running system. How to gather and integrate all relevant static and dynamic trace information in a unique way?**

**Problem Details**

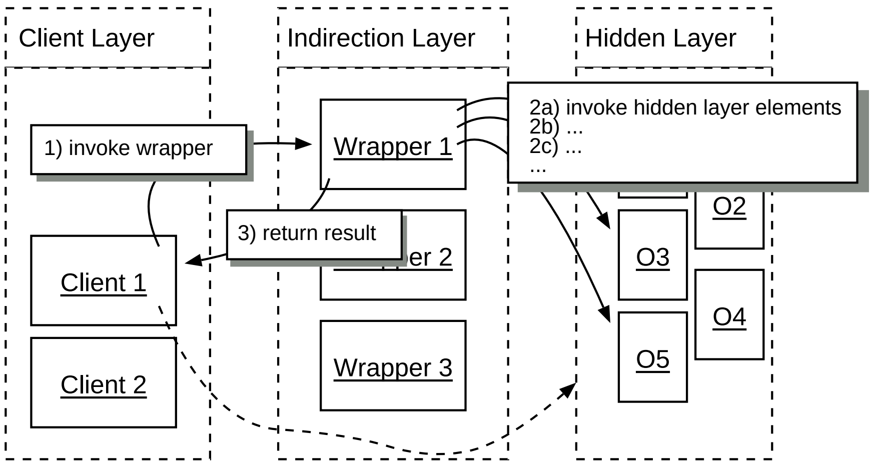
Consider extracting trace information from a large legacy system written in some (dialect of a) programming language. The straightforward solution is to write a for the programming language that statically extracts the information from the code. But this requires a working parser for that language. If no parser is available, it can be a complex, but not very challenging, task to write a suitable parser. This work is also annoying as it has been done already for the compiler or interpreter of the language (which is often not available in source code).

Static information, as contained in the program text, may not be the only required trace information. Consider you also require dynamic invocation data, call graphs, or other runtime trace information. Obviously, scanning the program text is not enough in such cases.

If more than one type of trace information is extracted, these information have to be integrated. To ease development of trace information extraction tools, one consistent API should be provided for all kinds of trace information, instead of a specific interface for each kind of trace information.

**Solution**

**Therefore: Provide an between the application logic and the instruction set of the (sub-)system that should be traced. The general term "instruction set" can refer to a whole programming language, but it can also refer to the public interface of a used component, sub-system, or *layer*. The wraps all accesses to the relevant (sub-)system and should not be bypassed. In this provide custom hooks to extract the relevant trace information.**




A simple consists of a set of wrappers. The client only access the wrappers to interact with the *layer* hidden behind the . The objects handle the complete interaction with the elements of the hidden *layer*.

**Example**

Consider again that you want to extract the dynamic structure and invocation information of a legacy system. At first, you should search for an existing that can be used: for instance, a public interface layer or a remoting layer are good places to look for. This only makes sense, if it is possible to change this layer's implementation. If no appropriate - can be used, simply implement an consisting of a set of wrapper objects for all operations of the legacy system that are called from within the application logic layer. This task is usually not too much work because in many cases it can be automated and/or it is necessary anyway. For instance, if the goal of your project is reengineering to the web, the web interface requires some public interface for the legacy system. This public interface can also be used for indirections.

Once you have found or built an , trace all methods invoking the legacy system. The traces can be evaluated on-the-fly or logged (and evaluated later on). Required structure information is often maintained within the and can be queried using . For instance, for each open session the will contain a session object. An on the session manager can return the list of all session objects. If this information is needed, but not maintained in the , it can be obtained by tracing the operation invocations for creating and destroying session objects.

**Discussion**

An wraps a relevant (sub-)set of the language's instruction set with objects. This language can be a simple public interface of a component or even a full programming language implementation. The objects may be simple wrappers, just forwarding messages, or may implement more sophisticated behavior. The application logic uses this - ’ objects instead of the original instruction set. Thus all calls can be traced here. The use of an remains hidden for clients, as they use the ordinary interfaces of the application logic.

An is available in many systems, even though it is not provided explicitly for trace information extractions. Consider the following examples:

- A scripting language (implemented in C++) is an , as it wraps the instruction of its own implementation language (here: C++) with new script commands that are solely used by the application objects in the scripts.
- A set of wrappers providing access to a legacy system also implement an , if these wrappers hide a conceptual subsystem completely and the is not bypassed by application logic objects.
- A middleware can be used as a distributed . All invocation to *remote objects* [VKZ02] have to pass a *client proxies* [VKZ02] on client side and an *invoker* [VKZ02] on server side. With access to the middleware structures one can indirect all messages either on client side or server side.
- An object can be seen as a meta-object for the application logic objects. An can implement a meta-object protocol [KdRB91], but it does not have to.
- A virtual machine is a low-level indirection layer that indirects byte-code instructions into machine-code instructions.

An can trace all invocations sent from application logic objects to the subsystem behind the . It can also trace structure information, either by deducing them from the calls (e.g. observing object interactions to find out class relations) or by using reflection or .

Of course, for a given hook invocations for gathering trace information can simply be added by hand into the code of the . There are various solutions for automating trace information extraction from an :

- A invokes a callback whenever a certain runtime structure provided by the (as for instance an operation, object, or variable) is accessed. T mechanisms are part of the implementation.
- M can be used to implement dynamic, object-oriented : all invocations sent to the have to pass the . It can thus be used to extract trace information for the invocations, either hard-coded or using .
- A can be used for automatic insertion of callback hooks into a given program text. It can be used to add a callback to each operation of the code that invokes a method of the hidden subsystem.

I is a generalization for patterns wrapping an implementation layer with a symbolic language, such as *object system layer* [GNZ00b], *microkernel* [BMR+96], *virtual machine* [GMSM00], *interpreter* [GHJV94], and others. Note that these patterns are used in many systems, often for other purposes than tracing structures or dependencies. Thus, if an existing is accessible, it can be (re-)used for extracting trace information.

I can be built in the programming language of the application logic, but also in any other language that can be integrated with the application logic language.

With an the trace information can be extracted from the running system. Usually monitoring can be turned on and off; thus the penalties of observing trace information can be avoided in operational mode. To a certain extent structure information can be deduced as well, e.g. with . However, the indirection causes a performance decrease and structures built for indirecting consume memory.

In many cases an existing can be extended to support extracting trace information (for instance in interpreted languages, the interpreter can be extended). But sometimes there is no or it is not accessible (for instance a closed virtual machine). Then an additional has to be built. Using this additional might imply a change in the programming model of the application logic objects.

## Trace Callback

**Context** You are using an and you want to extract trace information from the running program.

**Problem** **You want to trace one or more specific structures of the runtime system. It is not enough to interpret the source document texts, the program has to be executed and analyzed at runtime. As runtime structures can change dynamically, it might not be known until runtime which structures are to be traced. You want to avoid larger performance penalties due to tracing. How to trace specific runtime structures generically and dynamically?**

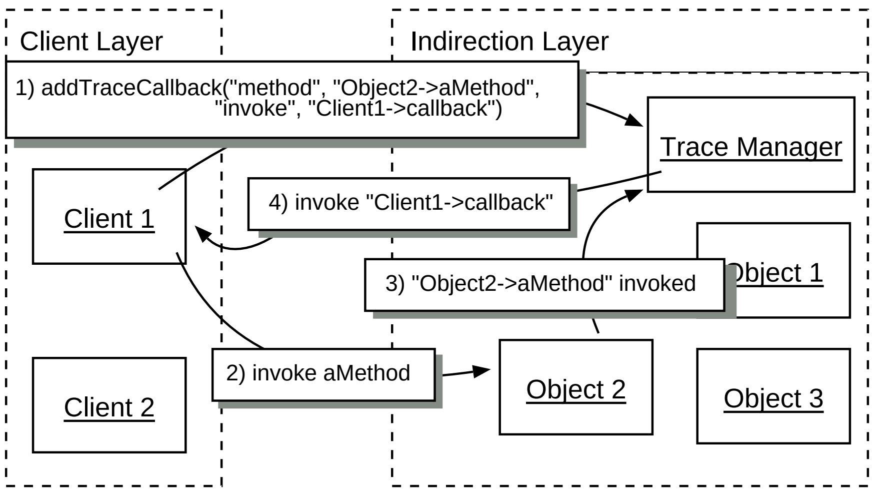
**Problem Details** Consider you are using a component that implements dynamic variable slots for objects in your programming language – that is, variables that can be added and removed at runtime to any object. This component adds a slot table to each object, and handles memory allocation, deallocation, and access of variable slots. You want to implement an automatic persistence mechanism for application objects that use this component. That means you have to trace all variable accesses to persistent variables used within these objects. The persistence mechanism should be transparent to application logic objects and to the dynamic variable slots component. It should be possible to dynamically add persistence tracing for new variable slots, and if a slot is deleted, persistence tracing should stop.

Besides variable accesses, other elements of the may require a similar low-level observation of runtime elements, such as specific methods, objects, commands, or any other structure of the runtime system.

Such low-level tracing can be done by inserting trace invocations in the respective code, wherever the traced element is accessed. But usually this solution produces tangled and complex code as a result.

The client may need to specify – at runtime – which structures are traced, for instance, to minimize the performance impact of tracing. This can hardly be done with a hard-coded solution. An *observer* [GHJV94] may be a solution for this problem. But typical *observers* are not used as low-level traces for specific structures only, and the observed classes have to be prepared for observation, what is not always possible.

**Solution** **Therefore: Provide a interface to an and access the structures to be traced via the only. With this interface one can dynamically add or remove a for a specific runtime structure of the . When adding or removing a , 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 implementation automatically.**




A is typically part of an . Clients can register callback operations for certain events, as for instance a method invocation. The supports a mechanism to automatically invoke the registered , when the respective event occurs.

**Example** Consider again you want to add a persistence mechanism to a given application logic layer transparently. The supports variable slots. T for these slots are required. To the class that manages , we can add the respective operations for adding and removing :

```
void addTraceCallback (String type, String ID, String event, Method traceMethod) {...}
void removeTraceCallback (String type, String ID, String event) {...}
```

For each persistent variable you can add a for the "write" event. Regardless whether the variable is accessed with an operator or with an operation, you can be sure to trace all accesses of variables, as the is rooted in the . The operation is used to write the new variable value into the database.

First, it is required to write a operation handling the persistence tracing (e.g. in a class PersistenceTracer):

```
void persistenceVarTrace (String identifier, String type, String event) {...}
```

Now a specific variable "a" can be traced with this operation:

```
Method meth = null;
try {
  Class cls = Class.forName("PersistenceTracer");
  meth = cls.getMethod("persistenceVarTrace", null);
  varSlotLayer.addTraceCallback("variable", "a", "write", meth);
...
```

This solution assumes that the developer specifies the persistent variables and triggers reloading them upon next application startup. If all variable slots should be persistent in the application logic, you can automate setting up traces and reloading: set up a for the operation that is used for creating variable slots. Whenever a new variable slot is created in the application logic, the is invoked. It first checks the database for an existing persistent value and possibly reloads it. Then it automatically sets up the variable trace for that particular variable so that further changes are also persistent.

**Discussion**

The mechanism is directly invoked from within the implementation of the traced runtime structure in the . That is, callback hooks are placed into the component implementing or wrapping the traced runtime structure. These hooks raise callback events.

For each traced runtime structure the supports lists of registered -, one list for each supported callback event. There are different kinds of callback events:

- for a variable slot typical callback event are: access, write, delete, and read;
- for an object typical callback event are: access and destroy;
- for an operation typical callback event are: invoke, enter, and leave.

T are registered for individual runtime structure using its identifier (such as the variable name for a particular variable). When a callback event of a traced runtime structure is raised, it is checked whether there is a registered for the identifier of the runtime structure that caused the event to be raised. If so, the ’ operation is executed.

All operations are of a specific callback operation type. This operation type's parameters are the identifier(s) of the traced runtime structure, the callback type, and the callback event. Callback operations are provided by the developer when adding the -
.

Besides adding , the interface of the also supports deleting a at runtime. In many cases there is also an for querying the . It returns the list of registered for a particular structure and event.A is a low-level mechanism for gathering dynamic structures dynamically, and it is transparent to the application logic objects. However, it has to be coded into the and adds a slight performance overhead.

## Message Redirector

**Context** You want to build an object-oriented .

**Problem** **An essentially is an intermediate layer between the application logic and a hidden subsystem. In object-oriented systems that means it intercepts and adapts all individual messages that are sent from the application logic to the hidden subsystem. How to get control over the message flow in an object-oriented system so that we can at least trace (and modify) all messages and their results?**

**Problem Details** Consider you are using a dynamic component loader in a mainstream language, such as Java or C++, and want to allow customization of these components by non-programmers. Pure parameterization is not enough, but you have to expose a few control structures, such as `if` conditions, and provide some functionalities of your component framework. For instance, it should be possible to query the runtime environment (such as which operating system is used) and customize which components are dynamically loaded in which order. Somehow the user customizations have to be specified in a way understandable for users, and then be transformed into the proper invocations.

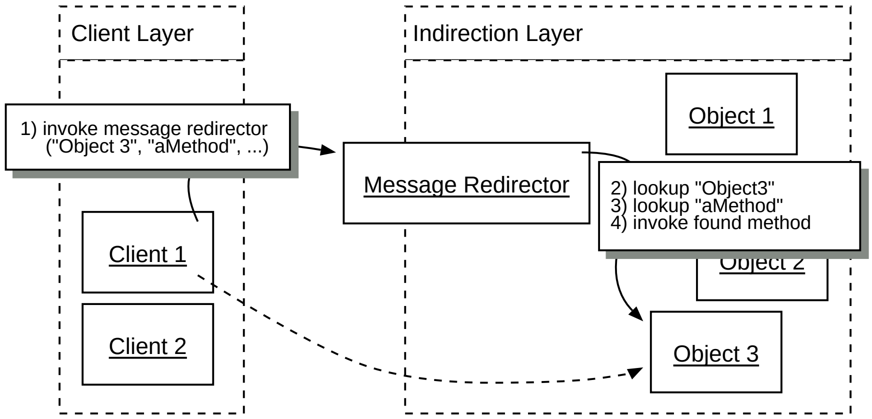
This can be done with an with the users' customizations as application logic. The dynamic component loader is wrapped behind the . The problem is how to transform all messages properly into programming language invocations.

Note that one customization option may have influences on multiple objects. Implementing one orthogonal aspect as code tangled within multiple implementation classes is tedious, and it is error-prone if changes to these scattered code fragments have to be made [Lop97, KLM+97]. H provides a solution in static environments, but as some customization options have to be evaluated at runtime, this compile-time approach for building an does not work here. Tangled aspects can also be added as , but as every possible message sent to the should be customizable, are tedious to apply because a huge number of would be necessary.

Some languages provide means to implement for their instructions, others not. If the language does not implement the natively, we have to gain a control over the message flow. This control over the message flow should be transparent to the developers of the customization script.

The application logic layer may use a different syntax than the ’ implementation language. This means, individual instructions are provided in a script written in the application logic layer’s syntax, and this script should be evaluated by the .

**Solution** **Therefore: Provide a as a *facade* to the . Each object and method in the is registered with a . Application layer objects do not access objects directly, but send symbolic (e.g. string-based) invocations to the . The dispatches these invocations to the respective method and object.**




A is a *facade* to the . All invocation have to pass it, thus it has complete control over the message flow to (and through) the .

**Example** Consider again building a little domain-specific customization language for a dynamic component loader. We provide an that wraps all interface methods of the dynamic component loader. Instead of accessing the operations of the directly, we use an operation `eval` of a . `eval` is applied to interpret the configuration invocations for different components (that are loaded from disk). For instance the following program checks for the operating system type (provided as an ) and loads either an emulator broadcast client component for the PC or it loads the broadcast client when working on a settop box:

```
result = messageRedirector.eval("Runtime info ostype");
if (result == OK) {
  String ostype = messageRedirector.getResult();
  if (ostype.equals("PC") {
    messageRedirector.eval("Component require BroadcastEmulator");
  } else if (ostype.equals("SettopBox") {
    messageRedirector.eval("Component require Broadcast");
  }
}
```

Each invocation contains the object ID of an object, followed by the method name, and a set of arguments. The objects are registered in the . In the example, it looks up the objects for `Runtime` and `Component`, and invokes the given method names with the given arguments.

**Discussion** A implicitly builds a language with its symbolic instruction set. This can be a domain-specific set of a few instructions, or it can be a full-fledged programming language. The objects in the application logic layer use this language.

For each language element there is one or more object(s) responsible for implementing it. The is responsible for parsing the language instructions and dispatching them to the responsible object and method. The client object might be a *command* [GHJV94] that is invokes the .

A often provides hooks for applying . That is, the developers can register as callbacks that are then called before, after, or instead-of the originally invoked operation automatically.

M can be used as a dynamic, object-oriented variant for implementing a custom . Some (interpreted) programming languages already provide an extensible . A custom-built is used if the language does not offer redirections natively.

Note that in many scenarios are already in the system for other purposes and can be used; for example:

- A distributed object framework has to invoke *remote objects* [VKZ02] on server side. The responsible *invoker* [VKZ02] on server side can be implemented as a - .
- Object-oriented scripting languages implement a between script and scripting language implementation.
- Customization or configuration parameters are often written in a little language (for instance using a markup language like XML). The object-oriented interpretation is done with a .
- Domain-specific languages can also be implemented with a .

A bundles all incoming calls from a (set of) client(s). It is often a *facade* [GHJV94] for the indirection layer. Extraction of trace information, message modifications, and message adaptations are transparent for clients. A can provide a language for application developers; then it is usually not visible to application developers. However, if a is used in parallel with ordinary operation invocations, two different styles of operation invocations are used. Building a complete language (including control structures) with the can turn out to be a non-trivial effort. Even though can be implemented quite efficiently, all consume additional computation time. In cases where these issues are a problem, a can be used as a static alternative.

The pattern is also documented in [GNZ01].

## Hook Injector

**Context** You want to implement an .

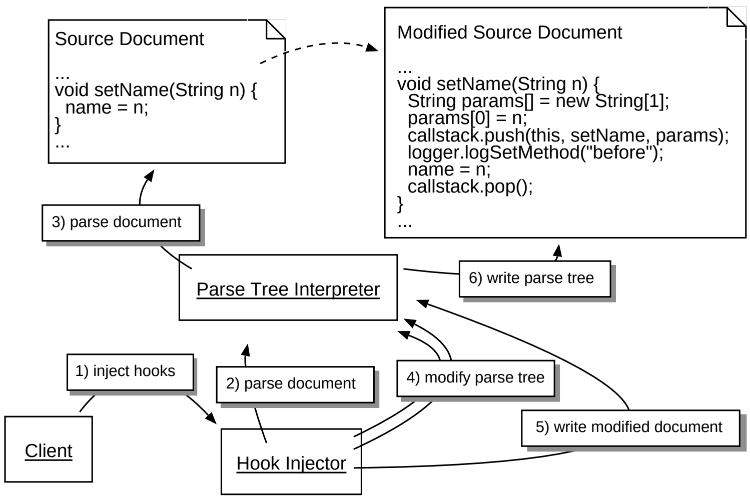
**Problem** **Tracing or behavior modification is required for a (sub-)system. This can be done with an . But you do not want the tracing implementation to require changes of the (sub-)system or its clients, as in the pattern . Also, trace code and application logic code should not be tangled. Efficiency of traces is more important than (runtime) flexibility of traces. How to trace (or modify) specific messages for a (sub-)system transparently?**

**Problem Details** Consider you want to implement an aspect that should log any invocation to a method starting with the string `set` or `get`, say, to calculate metrics based on variable accesses with getter/setter methods. If a is used, one would have to trace all invocations in the system and check whether the method name starts with `get` or `set`. This implies a larger performance impact. However, it is statically known in advance which methods are getter/setter methods; thus the expensive string comparison should not be applied for all invocations at runtime.

Besides performance issues, there is also the conceptual problem that invocations might look different than ordinary invocations in the base language, because the symbolic messages are, for instance, encoded in strings. This might be undesired as well.

A solution to this problem should be straightforward to implement. Especially, the trace code should not be hard-coded into the application classes. Changing such tangled code is tedious and error-prone [Lop97, KLM+97]. In operational mode metrics calculation should be turned off completely.

**Solution** **Therefore: Use a parser for the base language and let a inject the indirection hooks directly into the parse tree (for instance with a ). Either write a custom compiler to directly create machine code or byte code, or, as a simpler alternative, produce a new program in the base language with the injected indirection hooks. Then let this program be compiled or interpreted, instead of the original program. Semantically the new code is equivalent to the original code, with exception of the injected hooks for extracting or modifying the relevant operation invocations.**




A injects hooks by parsing a document, modifying its representation in memory (here: a parse tree), and by writing the modified source document back. This document is then interpreted or compiled, instead of the original source document.

**Example** Consider again you want to log getter and setter methods using an aspect. Once a hook operation for the logging task is implemented, we have to enter the hook into all methods in the system starting with `get` or `set`. The developer has to specify the injection rule, such as:

```
INSERT BEFORE void * set*(*) logger.logSetMethod;
```

This rule lets the inject an invocation to the operation `logSetMethod` of the object `logger` at the beginning of all methods starting with `set` (of any class and with any arguments). For instance, the method `setName`:

```
void setName(String n) {
  name = n;
}
```

is transformed into:

```
Logger logger;
CallStack callstack;
...
void setName(String n) {
  String params[] = new String[1];
  params[0] = n;
  callstack.push(this, setName, params);
  logger.logSetMethod("before");
  name = n;
  callstack.pop();
}
```

The uses a to search all method nodes in the parse tree, and checks whether the method name starts with `set`. Additionally, we have to add the `Logger` class (and `logger` object instantiation) to the existing code base.

All injection is done automatically by a code generator. Thus new getter and setter methods can be written, and it is ensured that they are logged after the hook injector has run again. In operational mode the system is compiled without running the hook injector; thus operation logging is automatically removed from the system.

**Discussion** Providing hooks means to add invocations into given code, for instance, at the beginning and end of each traced operation. These hooks invoke some instance that is responsible for interpreting them at runtime (when the operation containing the hook is invoked). This instance can be of any custom type, but usually it is a generic hook interpretation mechanisms, such as a or an *observer* [GHJV94].

This hook interpretation mechanisms needs to find out which method of which object was called with which parameters. The hook can provide this information as an .

A only provides static message indirection. Note that in dynamic languages limited dynamics are possible: we can dynamically inject new invocations for instance into a method body and use this modified method then. However, frequent changes are hard to implement this way, as we may have to delete old indirection hooks, before we can add new ones. Thus we would have to parse the method body again for each change, and recognize the indirection hooks in this code.

For both, static and dynamic variants, it is in general simpler to apply always on "clean" code without injected hooks, as finding existing hooks in the code (from previous hook injections) is tedious. This has the disadvantage that the must be applied after each change of the code or the injected hooks.

A can use a or to find the relevant places for hook injection in the program text. It adds a high-level API for specifying the insertion spots, for instance, using rules in regular expression style.

The pattern can be very efficient in certain cases where a dynamic is relatively inefficient, for instance, if an indirection condition can be statically pre-evaluated. The pattern also avoids the potential problem of to require another style of calling methods. However, often an additional parser and/or compiler is required. Dynamics are usually not provided (or very limited). It depends on the application case whether it is more work to implement a or a . Developers must be careful that manipulations performed by a do not interfere with verifications performed by a compiler or virtual machine.

## Message Interceptor

**Context** You are using an and want to trace, modify, or adapt some program behavior.

**Problem** **Controlling the message flow in an can be done either with or . 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 or hook implementations. This solution does not provide dynamic composition, extensibility, refinement, or reuse of message traces. How to express dynamic message traces, modifications, or adaptations as first-class entities of the -**

## ?

**Problem Details**

Consider you want to write a dynamic trace tool for messages. A user can specify in a GUI which operations should be traced and from then on these should be logged in a logging text area. At runtime, operations to be logged can be added and removed with immediate effect. This should work for any operation passing the .

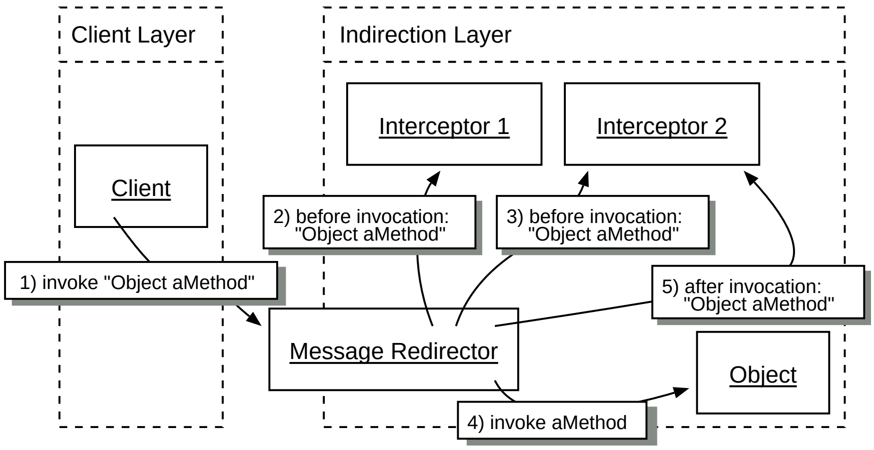
Using a would not work here because it cannot dynamically add and remove hooks. When injecting , to a certain extent the trace operations can be reused. However, you have to bloat the code of all operations in your with invocations. Both solutions are only for handling specific messages, whereas the problem scenario potentially requires all messages in the system.

Thus an unnecessary high number of injected hooks or is used. Logically only one behavior for all messages is required. The developer of the trace tool should be able to handle the concern "tracing messages" as one reusable (and refineable) entity.

Solving the problem with a would mean that you simply have to check in the dispatch operation of the for the respective method names. But here we face the problem that we require some mechanism to change the behavior at runtime so that it traces the operations dynamically, as specified by the user in the GUI.

**Solution**

**Therefore: Let be invoked for standardized invocation events observable in an , 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 . The callback mechanism can be triggered either by a or the hooks of a . Optionally a can specify conditions to be evaluated when the invocation event happens, and it is only executed, if the condition is true. Let the be dynamically composed with the invoked classes/objects and operations.**




Two are registered as "before interceptors" and are thus invoked before the actually invoked method. One of these is also registered as an "after interceptor." Thus it is also invoked after the method invocation returns.

**Example** Consider again implementing a dynamic message trace tool. If a is supported in the system and it allows for registering , we can add the interceptor object for each operation name selected in the GUI by the user:

```
Interceptor traceInterceptor = getInterceptorObject("traceInterceptor");
traceInterceptor.removeFromAllClasses();
String[] classMethodNames = getClassMethodsFromGUI();
for (int i = 0; i < classMethodNames.length; i+=2) {
  InterceptorList il =
    messageRedirector.getInterceptorList(classMethodNames[i], classMethodNames[i+1]);
  il.setInterceptor(traceInterceptor);
}
```

In the example the `traceInterceptor` is first removed from all interceptor lists and then added to all specified classes' methods. Thus these methods are intercepted by the `traceInterceptor` whenever they are called. The `traceInterceptor` is an ordinary object which implements the before, after, and/or instead-of behavior in respective methods.

**Discussion** A needs to be invoked by the method invocation mechanism of the - . This can be done by one of the message indirection patterns, - or :

- A checks before and after it invokes a method for registered . If there are some, it invokes them appropriately.
- A can simply add invocations to into the code. But if the are invoked directly by the hooks, the message interception cannot be composed dynamically. Thus a better solution is to let each hook call a central registry instance and let this registry check for . Note that this registry can be implemented with a that is solely used for handling .

There are different typical types of :

- A "before" forwards the invocation to the original receiver, after it has handled it.
- An "after" is executed after the original receiver and can handle the result before it is passed back to the invoker.
- An "instead-of" handles the message instead of the original receiver.

Somewhere the information has to be stored which methods are intercepted (for which event). If there is access to the relevant entities, this information can be stored within the objects (or classes) of the operations to be intercepted. It can also be stored by a central registry (e.g. implemented in the ).

Consider a system with $n$ interceptors. If always all messages would be intercepted, each invocation would result in $n + 1$ invocations. This is a considerable performance overhead for interception. It is avoided by executing conditionally. That is, a is only invoked, if specific conditions are true. A typical condition is that the message interceptor is only applied for operations with a specific name of a specific class. In some systems, the developer can specify a condition expression to be evaluated before the is invoked.

Also, to avoid recursive interceptions (causing a performance decrease and the problem of endless loops), invocations to interceptors should not be intercepted (by the same interceptor).

The itself can be implemented using many different programming language abstractions, such as objects, classes, or methods. M can also be provided as new constructs that are not supported natively by the used programming language. The later alternative has the disadvantage that either the language has to be extended or a pre-compiler has to be written that translates the new constructs into the native language code.

In a we often need information to discriminate calls. I can be used to find out the relevant per-invocation information. I are useful to obtain the system's structures and dependencies.

A provides controlling behavior for messages. It is transparent for caller and callee. M can be attached and detached dynamically. However, applying a costs performance. For implementing a access to an indirection mechanism is required.

Some variants of this pattern are already documented, including distributed *interceptors* [SSRB00], *before/after interceptors* [GZ02], and *invocation interceptors* [VKZ02].

## Introspection Option

**Context** You want to extract architectural structures as trace information from a software system.

**Problem** **Many architectural structures and dependencies of a software system are needed while it runs (either by an external tool, by the software system itself, or both). 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. How to gather and provide such information?**

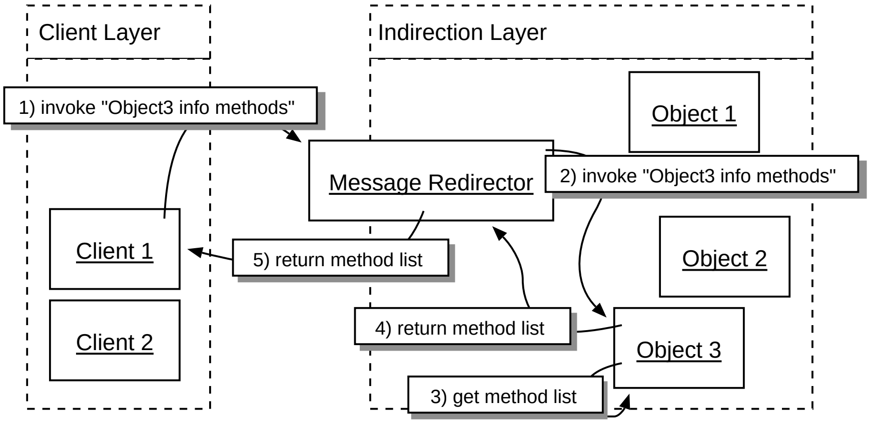
**Problem Details** Consider obtaining structures and dependencies of a software system written in Java. Structures and dependencies of a software system are all those information pieces that do not change for each invocation.

The distinction between dynamic and static structures depends on the programming language used. For instance, in Java the instance list of a class is a dynamic architectural structure, whereas information, such as class structure, operation names, and signatures, are static architectural structures. In other languages all this information represents dynamic architectural structures.

Static structure information could be added to a given program as special metadata fields. The programmer would have to add the same information twice to the code, once as a programming language construct and once for documenting the trace information. But to maintain the same information in parallel by hand is tedious and a waste of memory.

When trace information should be collect, in hand-built solutions, we often have to live with naming conventions (and their violation), such as using special getter/setter methods. Where possible, this should be avoided.

**Solution** **Therefore: Offer for each interesting architectural element of the . This can be done because all messages that are creating or changing structures or dependencies have to pass the . Provide a simple extension API for adding new, domain-specific . I are implemented within the , not the application logic layer. It should be possible for application logic objects to provide new .**




The `methods` is queried by a client. First, the `info` method has to be dispatched for `Object3`. Each object implements the interface; thus, it can be queried for its method list.

**Example** Consider again obtaining structure and dependency information in Java. At runtime Java uses its virtual machine as an indirection layer for its bytecode. The Java reflection API implements some for the class structures. These are extracted from the state of the virtual machine. For instance, the following code obtains the class of object `o` and iterates over the methods of this class:

```
Class c = o.getClass();
Method[] theMethods = c.getMethods();
for (int i = 0; i < theMethods.length; i++) {
  String methodString = theMethods[i].getName();
  ...
}
```

A limitation of this reflection API is that it cannot be extended with new structures. To resolve this problem we can use a specific operation for bundling all . This operation is defined on a superclass for all objects. It uses a string-based interface for providing the :

```
String[] info(String optionName) {
  if (optionName.equals("methods")) {
    return infoMethods();
  } else if (optionName.equals("variables")) {
```

```
    return infoVars();
  ...
```

Sub-classes (e.g. in the application logic) can overload this operation and provide new - . If the sub-class operation is not able to resolve an , it invokes the info operation of its superclass.

**Discussion** I are conceptually coupled with the structures they represent. For instance, one can ask the system which classes it consists of, a class which methods it contains, and a method which parameters it has. Therefore, should be added as operations to the classes of the representing a particular structure.

If it is not possible to change a structure that should offer , a (together with or ) can be used to intercept invocations to and handle them on its own. The is then used to simulate the introspective behavior.

I are always offering introspective information at runtime. Still they can be given for static structures and dynamic structures. In case of dynamic structures, it has to be ensured that the reflects the current state of the architecture. This can be done by calculating the on-the-fly. Or the is cached in the . Then we have to invalidate the respective in the cache when the dynamic architecture element changes.

I are bound to structures and thus should not be used for per-invocation information. These can be extracted with an which are created per method invocation.

The *reflection* pattern [Bus96] implements a variant of that uses a special meta-object type for the introspective information.

I provide one unified API for querying structural information. This information is extracted from the , and thus, it has to be maintained only once. However, providing such information statically (e.g. as metadata), instead of dynamic lookup, is usually faster. This problem can be avoided by caching after the first lookup.

## Invocation Context

**Context** You want to extract invocation information, such as which method has called which other method, as trace information from a software system.

**Problem** **Invocation information are useful for building call graphs and evaluating metrics automatically at runtime. They are also important for object-oriented adaptations that rely on message exchanges: the invoked method requires the information which entity has invoked it. The invoking method should not have to provide the invocation information as a parameter, because this solution is cumbersome and may result in interface incompatibilities. How to obtain the invocation information from inside of an invoked method or a wrapper method without changing its interface?**

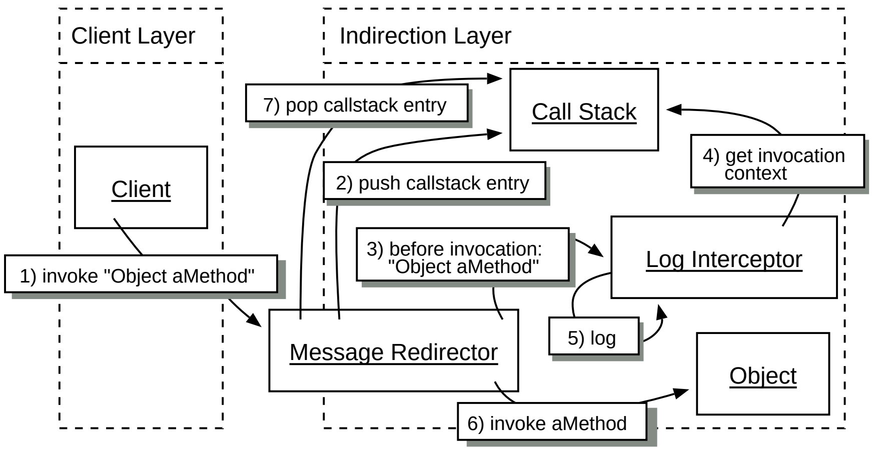
**Problem Details** Consider you have added hooks for logging getter and setter operations with a .
The hook operations need to find out the original operation's name (into which the hook is placed) and the invoked object's ID.

Typical information for an operation invocation, relevant as trace information, are the called and calling object's ID, the called and calling method name, and the called and calling class name.

Such invocation information should not be provided as . I are bound to the structures of the system. In contrast, invocation information is typical per-invocation information, meaning that it changes from invocation to invocation. Asking a structure, such as an object, directly for the current invocation is not a good design, as multiple invocations can be executed by the object at the same time.

In most programming languages invocation information are not accessible. A simple solution is to let the invoking operation pass the invocation information as an additional parameter. However, this solution does not work (well), if the implementation of the invoking operation should not be changed and/or interface incompatibilities of trace methods should be avoided (e.g. because they are used from third-party code).

**Solution** **Therefore: Provide access to the the current , an object (or other structure) that 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. I 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 the essentially is the top-level callstack entry.**




A is invoked before the actual operation. It requires the - for logging. The pushes a callstack entry for each invocation. From the callstack we can obtain the current . The callstack entry is popped when the invocation is done.

**Example** Consider again obtaining the for hook operations that are logging getter and setter methods. The hook operations require the name of the invoked method, as well as the object ID. Both can be provided by the :

```
void logSetMethod (String hookType) {
```

```
InvocationContext ic = getCurrentInvocationContext();
Object obj = ic.getObject();
Method method = ic.getMethod();
String[] params = ic.getParameters();
...
```

The operation `getCurrentInvocationContext` gets the top-level entry from the ’ callstack. The `InvocationContext` object contains all relevant invocation information.

Note that in this example the hook operation `logSetMethod` has no callstack entry, as it was not dispatched, but invoked directly. If traces, hooks, or are added to the callstack, we have to overstep their entries to get the originally invoked .

**Discussion** Providing access to usually means to provide a convenient API for accessing the ’ callstack information. Note that in a distributed context, the callstack is running in a remote process. Thus the is usually passed across the network.

Unfortunately, many existing , such as closed virtual machines, do not allow for open access to the callstack information. In such cases, a or can be used for maintaining an additional callstack. This has the liability that all invocations must be additionally sent through this indirection mechanism. Intermediate indirections, such as wrappers, *proxies* [GHJV94], or a middleware’s *client proxy* [VKZ02] and *invoker* [VKZ02], can also be used to maintain an additional callstack. These solutions have the liability that of invocations that bypass these indirections are lost.

The can be given in form of a set of methods on the entity providing the , one method for each information provided. Or an object can be built that contains the whole information. The later alternative has the advantage that the object can be passed easily to other methods, but this variant consumes more memory for the additional objects. Another variant is to put objects directly onto the callstack and reference these objects. This variant requires access to the callstack implementation.

In a wrapper or that intercepts an invocation before it reaches the callee, information about the caller and the originally called scope are of interest (and should be provided as ).

I are especially useful together with indirection techniques. However, if the callstack information cannot be automatically extracted from existing , we have to restructure the system to provide an additional indirection. This costs memory and performance (but if it is used for pure analysis purposes it can be turned off in operational mode).

The remote variant of this pattern is also documented in [VKZ02, VSW02].

## Metadata Tags

**Context** You want to add some information to the trace information that can automatically be extracted from the existing documents, such as the source code.

**Problem** **In a software development context, sometimes some of the relevant information necessary to produce trace information cannot be expressed with the given implementation or design languages' language resources. In other contexts (like reengineering) additional information might be required to produce some trace information. If such information is maintained in external documents, documentation is likely to be forgotten. But often there's no proper language resource or the system cannot be changed. How to add some trace information to an existing system?**

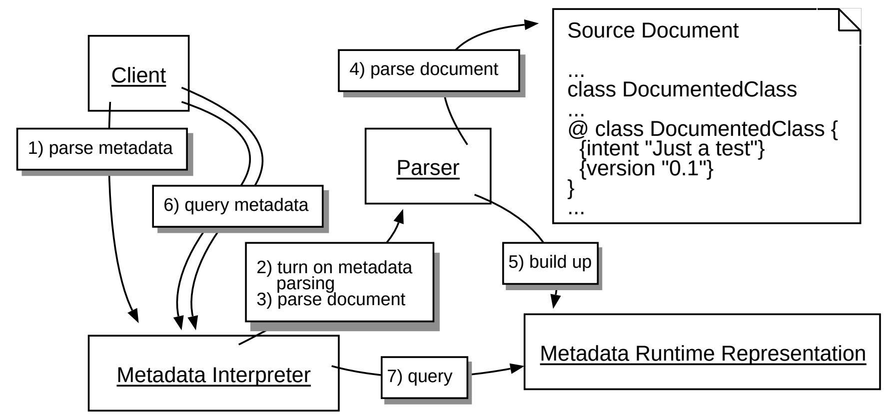
**Problem Details** Consider documenting the intent of a class, that is, a textual description of the requirements it fulfills. In many formal programming (and design) languages there is no proper language resource for this task. But without it we cannot automatically trace which classes realizes which informal requirements. The class documentation should be extensible. For instance, you also might want to document the authors of a class and the version number.

If more than one representation of some knowledge about a system exists, consistency of these representations has to be ensured. In the daily business, however, consistency often suffers from time pressure, tight budgets, changes in development teams, differing discipline of developers, and many other problems. For a developer it is usually easier to add a documentation directly to the design or program code than to remember to update a separate documentation (see the self documentation principle [Mey97]).

Details are – in contrast to abstractions – volatile. Because details are going to change, they should not be tangled with abstractions, but put where they will create the least amount of friction [HT99].

Separate documentations are usually not accessible at runtime. The running system cannot provide the information about its realization to its clients. For instance, it would be useful, if a documented version number of a class could be used for runtime version checks.

**Solution** **Therefore: Provide a standard notation for embedding in code or formal design documents. Use a language entity that can be distinguished from the other language entities to contain the . In these provide additional architectural knowledge and documentation. Typically, contain hierarchical key/value lists.**




A client invokes a metadata interpreter that parses the relevant document(s). The parser builds up a metadata runtime representation that is used for subsequent queries by the client.

**Example**

Consider again documenting intent (and author and version) for Java classes. A simple solution is to embed a special comment notation in the code. As it is a comment, the Java compiler and virtual machine ignore these tags. But a special interpreter can scan the comments for and interpret them accordingly. This can be done using the patterns or . Note that in some cases should also be exposed at runtime. In such cases, they should not be embedded in comments, but provided with a special special metadata interface. In languages that are extensible with new commands, such as Smalltalk, Tcl, or Lisp, or when using a , we can add a new command which is simply ignored by the standard interpretation, and builds up a metadata context, when activated.

Typically are described as hierarchical key/value lists; for instance with XML:

```
<class name="DocumentedClass">
  <intent> Just a test </intent>
  <version> 0.1 </version>
</class>
```

Such formats can easily be structured and extended. In the programming language context often we do not name each tag, as in the XML example above, but use simply brackets, for starting and ending each tag, e.g.:

```
@ class DocumentedClass {
  {intent "Just a test"}
  {version "0.1"}
}
```

**Discussion**

The standard language interpreter or compiler does ignore the , but special interpretations for extracting trace information can use them. Some entity in an understands these , whereas the standard can also ignore the .

Note that can be embedded in the program that they document, but in some scenarions they can better be stored in a separate file. For example, in the reengineering context, where we usually cannot change the given system, we can only provide in a separate file. Also, when are used for providing configuration options they are stored in a separate file, because not each installation of the system should have the same configuration.

Usually, at least, we want to be able to name the architectural element which is described by a , and give a list of key/value pairs. More sophisticated metadata schemes provide additional structuring and dependencies of metadata. Especially, hierarchies of list elements are often used, as for instance nested brackets or nested markup (as possible with XML). Optionally we can provide a name for each metadata element.

Self documentation of the architecture and relevant (missing) can be embedded in existing documents. However, adding in existing documents means that those documents become more complex and less readable for human beings. M also enable us to put abstractions in code, details in metadata [HT99].

## Partial Interpreter

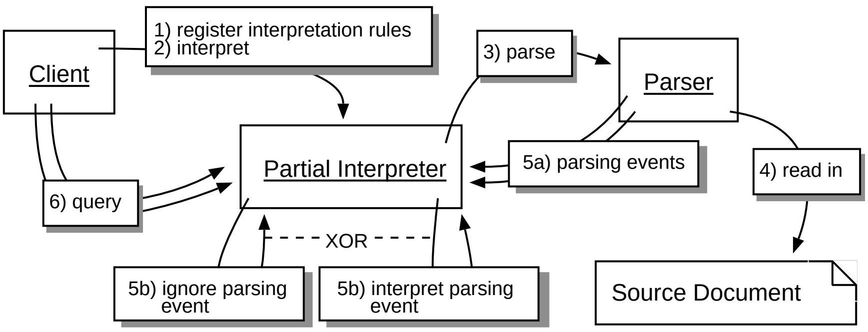
**Context** You want to extract some trace information from the code or other (formal) source documents.

**Problem** **A complete parser for the source document's language does not exist, it is too expensive, it has bugs, or it is complicated to extract the relevant information from the created parser's outputs. The relevant information to be interpreted is only a (small) sub-set of the source document's language, and thus, the complexity of the whole language should be avoided in the subsequent interpretation. How to access the trace information in source documents in a simple and extensible way?**

**Problem Details** In many scenarios only a few language elements are of interest to create a certain architecture view, all others can be ignored. Consider, for instance, we want to find all classes and inheritance relationships in a C++ program. Then we have to get rid of all other information, such as method bodies.

In such cases, writing a complete parser for the source document's language is simply too much work. If we are able to reuse an existing, full-fledged parser for the language, we are only interested in a small subset of its outputs. Using a which builds up a full parse tree from the source document might be overkill in terms of memory and performance.

**Solution** **Therefore: Provide a for the source documents language, an interpreter that only understands the specific language elements required for the programming task. Typically a uses an event-based parsing model, and it interprets the language elements in the order as they appear in the source documents. This can be achieved by using a very simple parser that only raises the respective parsing events, or by selecting specific parsing events from a full-fledged parser's outputs. The language sub-set (i.e. interpretation rules) for selective parsing or interpretation can be defined in a .**




The client registers the rules for parsing and starts the interpretation. The retrieves the parsing events and interprets only those parsing events that were registered before. Clients can use the interpreter's runtime state to query the trace information.

**Example** Consider again writing a for the C++ language that should find all classes in a file. We build a simple parser that is only able to extract C++ statement beginnings and endings. In a simple we register the symbol "`class`" to interpret `class` statements. All other statements are ignored. Then the C++ file is parsed sequentially, and the ignores all other statements.

There are some other statements that cannot be ignored because `class` statements can nest in them, as for instance the C++ `namespace` statement. Thus we also register "`namespace`" in the . The interpretation of `namespace` simply invokes the recursively with the contents of the namespace statement. Thus we find the namespace classes as well. Also the `class` statement's body has to be interpreted recursively in order to find inner classes. For classes we also have to evaluate the statement's contents to find out the superclass relationship in C++.

**Discussion** An important part of a is the underlying parser. There are the following variants:

- *Build a partial parser:* If there is no existing parser, we can provide a very simple parser that only understands the very basic syntax of the language. It is enough, if it is able to find the beginning and the end of a statement and can read the statements one after another. Then we apply this parser to the source document. It only parses the specified language's subset, and ignores all other statements. For known statements, it also parses the statement's arguments to gather the relevant trace information. For nested statements, the parser can be applied recursively.

- *Modify an existing parser:* If there is an existing parser, we can possibly modify it to produce a suitable output. Consider a full can be used and its implementation can be modified, but either the output is too complex or certain elements should not be parsed. For instance, in interpreted languages we can reuse the language's parser, but certain invocations should not be executed, such as starting the event loop (of a server). Then we write a by modifying the in such a way that it ignores all irrelevant or harmful invocations. It then produces a partial output with only the required language subset.

- *Generate Events from Existing Parser:* If there is an existing parser, but we cannot change it, we can let it do it's work and then change the outputs:
  - If the parser is an event-based parser, we can send all parsing events to the and let it ignore the irrelevant events.
  - A 's tree can be visited using a *visitor* [GHJV94]. The *visitor* sends parsing events to the , but only for selected nodes.

A typical implementation of a uses a : in a we provide a set of interpretation rules, one for each interesting statement type. The rules' actions are called whenever the corresponding statement occurs in the program text. All statements that are not registered in the are ignored. The rules' actions are used to build up a runtime representation of the relevant trace information. The interpretation rules registered for the define the subset of the language that the understands. The rules' actions are responsible from extracting the information from the system. In other words, here we use the ' "little" language to build a partial version of the original language.

I can be provided for querying the runtime state of a .

In interpreted languages it might be beneficial to (re-)use the original language interpreter. To build a we do not let it evaluate the source document as is, but a only specific parts of it that comply to the interpretation rules. For instance, in the class example above, we would create all the classes at runtime as empty classes. The nice side-effect of this is that we can then use the interpreter's to find out the classes' names, superclass relationships, etc., without having to write further interpretation code.

The does neither add complexity to the system nor influence performance or memory consumption. For conceptually similar languages the pattern is relatively language-independent, as often a common set of parsing events can be defined. However, the pattern is only applicable if runtime information is not relevant for trace information extraction. Dynamic structures, such as call graphs or object structures, cannot be analyzed.

## Known Uses

In this section we discuss some known uses of the pattern language. We rather concentrate on a few interesting technology projections than discussing all known uses; there are simply too many.

### Generative Aspect Languages: AspectJ

Aspect-oriented programming copes with tangled code fragments which logically belong to one single module (a concern) but cannot be cleanly modularized by the programming language abstractions. Such code is called crosscutting code. Currently, the most common way to implement aspects are generative environments, such as AspectJ [KHH+01], HyperJ [TOHS99], D [Lop97], ComposeJ [W+02], or JAC [PSDF01]. These achieve modularization of crosscutting code by new language constructs for expressing aspects. In this section, we briefly discuss how the pattern language is applied in AspectJ. In [Zdu04] we discuss the application of the pattern language in other aspect composition frameworks in depth.

These solutions follow a similar sequence. We will use the example of AspectJ for illustration. The aspects, as well as the points where to apply them (called "joinpoints"), are described in an extended language consisting of a set of additional instruction. This aspect language is added to the code written in the base language. To apply the aspects we require some trace information. First, we need the additional statements written in the aspect language. But we also require the class and method structure, as well as the spots where invocations are sent or received.

A parses the program text and uses an interpretation *command* set to interpret these . During this interpretation a injects hooks at the respective joinpoints. The aspect language code is replaced by base language primitives (or byte-code instructions of the virtual machine).

At runtime of the woven program, the aspect language is implemented as an .
The hooks call their implementations in this . Hooks together with respective implementations are a static form of .

Usually the adaptation defined in the aspect (in AspectJ called "advice") is only applied when certain conditions are true. These conditions are in AspectJ expressed with "pointcuts." In these conditions, information about the current message flow can be used. Limited are offered for instance via the Java Reflection API; these are directly usable in the advice.

In AspectJ the original class implementation can also be extended with so-called introductions. In a generative environment such introductions can also be implemented by injecting hooks into the respective classes using a .

### XOTcl Component Wrapping and Self-Documentation

XOTcl [NZ00] is an object-oriented scripting language that uses the pattern language in various ways, mainly for self documentation, extraction of architecture views, and wrapping. XOTcl is especially designed for component gluing [GNZ00a] (e.g. of legacy component written in other languages). The pattern implementations in XOTcl are also reusable tools to analyze glued component frameworks.

Each object in XOTcl is internally realized as a Tcl *command* that is registered with XOTcl's . The implements an . Thus every call to an XOTcl object passes this dynamic invocation mechanism before it reaches the actual implementation. The implementation of an object can be written in XOTcl itself, but it can also be a *component wrapper* [GZ02] for a component written for instance in C or C++. Trace information can be extracted with various mechanisms:

- There are for each architectural element known to the interpreter.
- The current can be queried.
- There are two . *Mixin classes* intercept specific calls to specific objects or classes. *Filters* are methods intercepting all calls to specific objects or classes.
- Tcl provides a variable and command trace mechanism for implementing - .

For self-documentation XOTcl provides that can be embedded in the program code. These can be attached to any architectural element (class, object, package, relationship, constraint, etc.) and describe the architectural element with structured key/value pairs lists. A *command* "`@`" is used to denote . It is simply ignored in the standard interpretation. There are two special interpretations possible: the can be turned on at runtime, and a can be used that only parses the "`@`" tag plus the described architectural elements (mainly used for automatic documentation of XOTcl code).

### Meta-Object Protocols: CLOS

Meta-object protocols [KdRB91] are that provide many hooks to influence the behavior of objects in the application logic. A meta-object imposes behavior over the objects in its control; that is, meta-objects control the behavior of CLOS itself. For instance, the CLOS meta-object protocols enables several ways to optimize or customize the message dispatch mechanisms and creation and destruction of objects.

A kind of with is also supported. CLOS offers multimethods as a generic and extensible message dispatch abstraction. These multimethods take the complete of a method into account for dispatching, instead of just the `this` pointer as in Smalltalk, C++, or Java. Multimethods are specializations of a generic function. When this function is called, it determines which of its multimethods apply and calls them in the appropriate order. The `:before` methods are called before the primary method is called, and the `:after` methods are called afterwards. An `:around` method runs in place of the actual execution it operates over. All these methods are so-called multimethods.

### Reengineering with Rigi

Rigi [MOTU93] is a tool for program understanding and reengineering that consists of two major parts: the core components and a user interface. Rigi's core components comprise a C parser for trace information extraction as well as selection, organization, representation and measurement components. In early versions, trace information was extracted using a for imperative languages such as C and Cobol. It is also possible to interpret structured documentation in Latex.

The full had its problems in a reengineering project for a very large systems called SQL/DS [WTMS95] (because the abstract syntax trees required several hundred megabytes of storage); therefore, the was replaced by a - in later versions (that produced a database with around two megabytes for SQL/DS). Such amounts of data require a project repository because scanning the information and calculating measurements takes considerable time.

The user interface is based on a Tcl/Tk scripting interface: here, the Tcl interpreter is used as an to support configuration and extension of the reengineering environment, especially custom graph layouts, metrics and analysis, and subsystem decomposition [TWSM94]. Tcl *commands* and the procedural variant of Tcl can be used to provide a dynamic interface for domain-dependent changes that sits between the GUI and the graph editor. Such domain-specific scripts that are triggered as callbacks by the extraction tool are a form of for customization of the analysis and layout processes.

In the SQL/DS project [WTMS95] a very simple heuristic for deriving was used: product-naming conventions, like the prefix `ARI`, were used to modularize the system.

### Moose Reengineering Environment

Moose [DLT00] is a language-independent reverse engineering framework that mainly extracts trace information for tools based on Moose (CodeCrawler, Gaudi, Supremo). As Moose is a pure analysis tool, the patterns that require modification of existing source documents are not used.

Moose is written in VisualWorks Smalltalk, and, for extracting trace information from Smalltalk code, the VisualWorks parser's output can be interpreted with a . This information can be combined with derived via the Smalltalk meta-model (i.e. an is used). Moose provides import interfaces for integrating other source languages (Java, C++, Cobol) via two architecture description formats (containing ), namely CDIF and XMI. Conversion are done using the of the SNiff+ parser tool. Internally, a Smalltalk-based generic format is used.

### Other Known Uses

In this section we discuss a few technologies that use some our patterns in different way than the use in the technology projections above.

In languages like Java some can be derived using the Reflection API; however, some interesting information are not exposed.

Many IDEs add to the source code with . They often use , mostly to achieve syntax highlighting and architectural views. Source Navigator [DOPR03] is an IDE that uses an interesting variant of to achieve a multi-language mapping: the first uses an ordinary to generate a parse tree which is then transformed into an internal architecture description format that is mapped to each supported language. The architecture description format is then partially interpreted for architecture highlighting.

Automatic documentation tools such as JavaDoc or AutoDoc often use special comment - . P extract the embedded metadata. The .NET runtime allows you to add in form of custom attributes to the code.

A profiler (a program that calculates a call graph and the amount of time spent in each operation) needs to inject hooks into a program to gather the profile information. Typically this is done with a during the compilation process. It injects hooks to build an - for all operation invocations. Some runtime structures are added to the program that are filled during program execution (and most often written into a profile file that can be evaluated later on). The inserted hooks can call . A profiler can also be built with . Note that due to the indirection the profiled times are higher the a non-profiled execution; thus a profiler is mostly used for optimizing the call structure and finding performance problems.

## Conclusion

In this paper, we have presented a pattern language of structure and dependency tracing in software systems. The patterns provide a technical infrastructure for tools, languages, language extensions, and frameworks in many different software engineering fields. The patterns are actually used in numerous projects. Note that the individual patterns are often used in other contexts than structure and dependency tracing, extraction, and modification as well. The pattern language contains tightly coupled pattern; however, still many other closely related patterns are to be mined or documented.

## Acknowledgments

Thanks to Markus Voelter for his great comments on the pattern language. Many thanks to Steve Berczuk for his attention as a EuroPLoP 2003 shepherd and his detailed and constructive comments. Finally, thanks to the participants of the EuroPLoP 2003 writers' workshop who provided valuable feedback.

## References

[Ber96] S. P. Berczuk. Organizational multiplexing: Patterns for processing satellite telemetry with distributed teams. In J.M. Vlissides, J.O. Coplien, and N.L. Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[BMR+96] 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.

[Bus96] F. Buschmann. Reflection. In J.M. Vlissides, J.O. Coplien, and N.L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 271–294. Addison-Wesley, 1996.

[DLT00] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the International Symposium on Constructing Software Engineering Tools (COSET 2000)*, Limerik, Ireland, Jun 2000.

[DOPR03] M. DeJong, E. Odenweiller, S. Polk, and I. Roxborough. The source navigator IDE. http://sourcenav.sourceforge.net/, 2003.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GMSM00] J. Garcia-Martin and M. Sutil-Martin. Virtual machines and abstract compilers - towards a compiler pattern language. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.

[GNZ00a] M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct 2000.

[GNZ00b] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.

[GNZ01] M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.

[GZ02] M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(1):1–30, 2002.

[HT99] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[KdRB91] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, October 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[Lop97] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Dec 1997.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[MOTU93] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[NZ00] G. Neumann and U. Zdun. Xotcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[PSDF01] 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*, Kyoto, Japan, Sep 2001.

[SSRB00] 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.

[TOHS99] 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)*, Los Angeles, CA, USA, May 1999.

[TWSM94] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, pages 501–520, Dec 1994.

[VKZ02] M. Voelter, M. Kircher, and U. Zdun. Object-oriented remoting: A pattern language. In *Proceeding of The First Nordic Conference on Pattern Languages of Programs (VikingPLoP 2002)*, Denmark, Sep 2002.

[VSW02] M. Völter, A. Schmid, and E. Wolff. *Server Component Patterns – Component Infrastructures illustrated with EJB*. J. Wiley and Sons Ltd., 2002.

[W+02] H. Wichman et al. ComposeJ Homepage, 2002. http://trese.cs.utwente.nl/prototypes/composeJ/.

[W3C00] W3C. Document object model. http://www.w3.org/DOM/, 2000.

[WTMS95] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, Jan 1995.

[Zdu04] U. Zdun. A pattern language for the design of aspect languages and aspect composition frameworks. Draft; accepted for publication in IEE Proceedings Software, special issue on Unanticipated Software Evolution, 2004.