# Pattern-Based Design and Implementation of an XML and RDF Parser and Interpreter: A Case Study

Gustaf Neumann[1] and Uwe Zdun[2]

[1] Department of Information Systems, Vienna University of Economics, Austria
gustaf.neumann@wu-wien.ac.at
[2] Specification of Software Systems, Institute for Computer Science, University of Essen, Germany
zdun@acm.org

**Abstract** Software patterns have been widely promoted as a means of conveying practical design knowledge in a reusable fashion. Several approaches for providing better implementation variants of certain patterns have been presented. These approaches promise great advantages for flexibility, traceability, and reusability of pattern implementations. However, there are only a few larger practical case studies of these concepts available. In this paper we will present a case study of a component framework for flexible processing of markup languages in the object-oriented scripting language XOTcl. The language offers high-level means and architectural support for component integration ("component glueing"), introspection, language dynamics, and message interception techniques. These language constructs enable developers to extend the language with pattern implementations, and so to provide language support for certain pattern fragments. As a case study domain we discuss an extensible and flexible framework for XML/RDF parsing and interpretation that was developed and evolved over a period of three years and is now in use in numerous applications.

## 1 Introduction

Object-oriented design with patterns and programming language support for design patterns is proposed in several works. One direction proposes code generation for patterns (see for instance [6,4]). Other approaches target on language constructs implementing pattern parts at runtime, as in [23,22,9,19]. But virtually no larger practical case studies of systems, built with these ideas, are available that are actually used in non-trivial applications.

In this paper we present the design and implementation of a parsing and interpretation framework for the Extensible Markup Language (XML) [5] and the Resource Description Framework (RDF) [20]. The case is well suited to demonstrate pattern implementation and variation in an object-oriented scripting languages because it is of a reasonable size, yet comprehensible, and has diverse requirements. These include high flexibility in interpretation of markup and meta-data, on the one hand, and efficient text parsing facilities, on the other. Parsing of XML text does not need to be highly customizable, but it has to be efficient and reliable. Several fast parsers for XML exist which can be reused as off-the-shelf components. The interpretation of the data in the web context is highly application-dependent. It is very likely to frequently change and must provide considerable customizability, depending on the application needs. XML and RDF have been developed to represent data from a diversity of domains. This flexibility of the data representation demands a high flexibility in the application frameworks as well.

An XML and RDF interpretation framework has to provide flexible means of interpretation, suitable for a variety of different application contexts. Typical applications will extract information from the XML and RDF representation, modify it, and create (or recreate) XML and RDF markup from their internal representation. In this case study, the required flexibility is achieved through usage of "high-level," object-oriented scripting in XOTcl [26] and patterns in the "hot spots" of the framework architecture. We will discuss the new pattern implementation variants on top of XOTcl's language constructs briefly to illustrate how partially language supported patterns can be reused and flexibly adapted to the application context. However, we will not focus on language support for patterns here, but on the component-based use of such constructs.

In our case study different programming languages (here: C, Tcl, and XOTcl) are combined to fulfill the application requirements. We generally argue that different expression resources, such as software paradigms, programming language, markup, and pattern languages, have to co-exist for different types of applications, and also within larger application systems. The application developer has to choose proper expression resources for implementing a particular task. For example, for the text parsing task, system languages, such as C or C++, are very useful because of their efficiency. Scripting languages are more appropriate for the semantic part of parsing (parse tree generation) and for the task of interpretation, because they provide powerful string manipulation facilities and flexible language resources for rapid adaptability and customization. However, the two parts have to be integrated properly.

Note that the applicability of these concepts is not limited to their use in the XOTcl language. In [34,14,12,13] we present an architectural pattern language enabling the introduction of the core concepts of XOTcl into languages such as C, C++ or Java. In [14,15] an industrial case study of this pattern language is presented. However, XOTcl directly supports these concepts as language elements; therefore, in XOTcl they do not have to be implemented and maintained by application developers.

In this paper we briefly summarize the application domain of XML and RDF text parsing and interpretation as a motivation (readers familiar with XML and RDF may skip this section). Then we introduce the idea of using patterns in the hot spots of an application, together with dynamic and introspective language constructs that were used for the pattern implementation. These language resources are reused later on. In the remainder of the paper, several crucial excerpts from the actual design of the XML/RDF framework are presented to demonstrate these ideas in a non-trivial case study. Finally, we discuss the results.

## 2 Case Synopsis: XML and RDF Parsing

In this section, we give a brief overview of the domain of our case study: parsing and interpreting of XML and RDF. First we briefly introduce XML and RDF. Secondly, we discuss requirements of parsing and interpreting XML and RDF. In our experience similar requirements for interpretational flexibility are recurring in many business applications that are based on XML (or other generic content formats).

### 2.1 XML and RDF

The Extensible Markup Language (XML) [5] is a simple, extensible language for structuring documents and data. It is primarily designed for Internet usage. XML provides content-oriented ("semantic") markup of data. XML is a standardized language for the notation of markup languages. It provides a meta-grammar for the structure of the application documents, given in a Schema or DTD. An important property of XML is its suitability as a language- and platform-independent intermediate data representation in a distributed environment.

The Resource Description Framework (RDF) [20] is a formal model for describing meta-data. Its primary application domain is the description of web resources. An RDF model [20] represents a set of RDF statements. RDF statements are used to assign named properties to web resources and to define associations between resources. In the RDF terminology a resource is every web element with a Uniform Resource Identifier (URI); that is, an element addressable over the web, such as an HTML or XML document or a part of it. An RDF data model is a directed graph with two kinds of nodes: resources (represented by ovals) and property values (represented by rectangles). The nodes are connected by directed arcs. These are labeled with the property names.

An RDF statement is a triple consisting of a specific resource (*subject*), a property name (*predicate*) and a property value (*object*). The property name is always an atomic name, while the property value can be a resource as well. An RDF model can be represented in XML markup (see [20]) to provide a standardized data exchange for RDF.

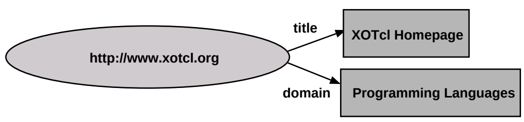


**Figure 1.** Simple Example RDF Graph: One resource with two properties.

Figure 1 presents a simple RDF model consisting of two statements: A certain HTML document (the subject, denoted by the URL) has the properties named *title* and *domain* with associated values. The linearization of Figure 1 in XML is:

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:wp="http://www.xotcl.org/schema/web-page/">
  <rdf:Description about="http://www.xotcl.org/">
    <wp:title> XOTcl Homepage </wp:title>
    <wp:domain> Programming Languages </wp:domain>
  </rdf:Description>
</rdf:RDF>
```

The first line of this example states that the content of the document is XML markup. The `rdf:RDF` tag indicates that RDF markup follows. XML uses namespace prefixing to avoid name clashes between different schemata and to support reusable schemata. The example uses two XML namespaces, denoted by the prefixes `rdf` and `wp`. To obtain the full predicate names, the prefixes can be replaced by the corresponding, unambiguous schema-URI, given by `xmlns` namespace declarations. A `Description` bundles statements about a resource that is named by the `about` attribute.

To provide further structuring, RDF provides containers. They refer to a collection of resources or literals. RDF defines three different kinds of containers, as follows:

– *Bag:* Unordered List, to be used when processing order does not matter. Duplicates are permitted.
– *Sequence:* Ordered List, to be used when processing order does matter. Duplicates are permitted.
– *Alternative:* List representing a set of alternatives for a single value of a property.

Every container must contain a statement declaring its type (`rdf:Bag`, `rdf:Seq` or `rdf:Alt`) and a number of members. These are automatically named `rdf:_1`, `rdf:_2`, ... by the parser, in their order of appearance in the XML text.

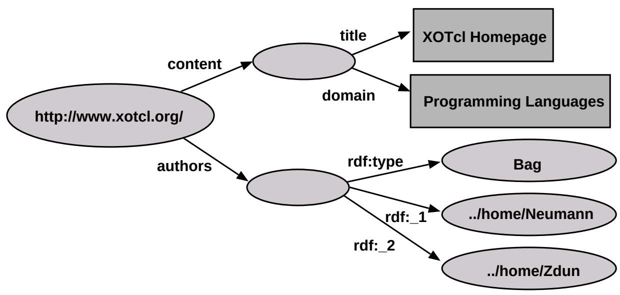


**Figure 2.** Bag Example: Resource with two anonymous resources. One is a bag with two members.

In Figure 2 the earlier example is enhanced with a bag containing the authors of the web page as resources. To serialize the bag we have to add a new property `authors` to the inner description of the preceding example:

```
...
<wp:authors>
  <rdf:Bag>
    <rdf:li resource="../home/Neumann"/>
    <rdf:li resource="../home/Zdun"/>
  </rdf:Bag>
</wp:authors>
...
```

### 2.2 Parsing and Interpreting XML and RDF

In this section we will summarize a few requirements from the domain of markup processing to demonstrate that an XML/RDF interpretation framework must provide high extensibility and flexibility.

Parsed XML markup can be used in different data representations. For example, XML text can be seen as an event flow with events of three types: *start*, *pcdata*, and *end*. Examples for event-based APIs are the SAX API [21] or the Expat API [8]. Sometimes such event flows are parsed into a node tree, like a DOM [32] tree. The tree can be used for further processing.

Different data representations for parsed XML markup have specific benefits and drawbacks, and so it makes sense to choose the most appropriate representation for a given task. The event stream representation has the advantage of potentially high efficiency and low memory consumption. Moreover, it is well suited for incremental parsing. This is especially important for documents that are too large to fit into memory. In a node tree representation, information can be accessed apart from the event stream, random access to the data is possible, and introspection of the tree structure allows for navigation through the tree. If we know where the targeted information is located in the document, a tree representation can be very valuable for reducing search times.

Besides the token event stream and the parse tree, the RDF specification [20] defines two special representations for RDF: a so-called triple representation and the RDF model graph. A triple database with triples of the form ⟨*subject, predicate, object*⟩ is well suited for several reasoning tasks over the whole model. Moreover, in contrast to a parse tree from the XML text, triples and RDF model graphs are “canonical” representations. That is, different XML linearizations of the same RDF statements produce different parse trees, but triples and model graphs are the same.

## 3 Pattern-Based Design and Implementation in Scripting Languages

In this section we will firstly motivate pattern-based design in scripting languages as a way to find flexible and maintainable architectures, as they are required by the XML and RDF parsing domain. Secondly, we will briefly introduce the object-oriented scripting language XOTcl that we have used to implement the XML/RDF parser and interpreter.

### 3.1 Flexible and Reusable Pattern Implementations

Design patterns provide abstractions from reusable designs. They can typically be found in the "hot spots" or centers of software architectures. A pattern describes a recurring *solution* to a *problem* in a *context* balancing a set of *forces*. Patterns cover the problem that expertise is hard to convey. They work by describing frequently used solutions that have proven their inherent capability to be fitted to the environment of numerous applications successfully.

Very often several patterns can be used on the same design problem, and several pattern parts are mutually dependent. A general problem is that of composition based on patterns. Pattern parts resemble architectural roles rather than classes; that is, architectural fragments in patterns enrich the design repository of the developer with elements, mostly of different granularity than classes. Such elements can be reused as architectural roles, which are orthogonal extensions to class composition. In other words, one class in a framework often plays various roles in several different design patterns at once. Alexander's original ideas on patterns [1] suggest using them in a pattern language which defines the correlations between the patterns used; that is, no pattern is used in isolation, but as an element of a language.

To date, there is little or no support in traditional, mainstream programming languages for implementing and using patterns. This implies several problems for the usage of patterns. Recurring pattern implementations cannot be reused, but have to be programmed again for each usage. The architectural fragments in the pattern are conceptual entities of the design repository, but they are split into several entities of the implementation language, like several classes or objects. Missing introspection facilities mean that patterns are neither traceable at runtime nor in the code. Many pattern implementations in mainstream languages have considerable implementation overhead, e.g., for unnecessary forwarding of messages and other recurring tasks (these problems of pattern usage are discussed more deeply in [23,24]).

### 3.2 The Object-Oriented Scripting Language 'Extended Object Tcl' (XOTcl)

In this section, we give a brief overview of the XOTcl language and its use for pattern implementation. XOTcl [26] (pronounced *exotickle*) is a value-added replacement of OTcl [33]. Both XOTcl and OTcl are object-oriented flavors of the scripting language Tcl (Tool Command Language [27]). Tcl offers a dynamic type system with automatic conversion, is extensible with components and is equipped with read/write introspection. In the remainder of this section we briefly describe XOTcl's language concepts.

In XOTcl, all inter-object and inter-class relationships are fully dynamic and can be changed at runtime. Each dynamic language functionality can be introspected. Classes are also objects; therefore, all methods applicable for objects can be applied to class-objects as well. Since a class is a special (managing) kind of object it is itself managed by a special class called a "meta-class." New user-defined meta-classes can be derived to restrict or enhance the abilities of certain classes.

The XOTcl extensions focus on complexity, flexibility, and adaptability in object-oriented systems. Moreover, XOTcl gives architectural support for component glueing and the implementation of larger architectural fragments. In particular, the following language constructs are supported:

- *Dynamic Object Aggregation* supports the 'part-of' relationship [25]. Children are automatically destroyed when the parent is destroyed. With introspection options on child and parent, we can automatically retrieve the current parent and the current list of children. The relation is completely dynamic and can be restructured using deep copy and move operations. We will use the language construct in the XML/RDF case study to implement a "canonical" dynamic object tree representation of the parsed XML data.
- *Nested Classes* reduce the interference of independently developed program structures by letting classes aggregate dependent class descriptions.
- *Assertions* let us provide formal and informal conditions for documenting (and checking) program invariants.
- *Per-Class/Per-Object Mixins* are classes that are dynamically attached to or detached from a class or object. They intercept every message to a class or to an object and can handle the message before/after the original receiver. They are ordered in a chain and inherit from super-classes [22].
- *Per-Class/Per-Object Filters* are special instance methods which are dynamically registered or deregistered for a class hierarchy or an object. Every time an instance of this class hierarchy or this object receives a message, the filter is invoked automatically and intercepts this message. Filters are also ordered in chains and inherited [23].
- *Dynamic Component Loading and Wrapping* allows XOTcl, Tcl, C, and C++ components to be loaded and integrated using the same basic mechanisms, as discussed in [11]. In this work we use the mechanism to load third-party XML parsers (implemented in Tcl and C) and reusable pattern implementations from external components dynamically at runtime.

The XOTcl language constructs are designed to help in the implementation and use of patterns. Filters [23] and mixin classes [22,24] are interception techniques for messages. That is, messages sent to an object, a class, or a class hierarchy are intercepted before they reach the original receiver. The interceptor can adapt the message to another receiver, handle it directly or decorate it with arbitrary behavior before/after the original receiver gets it. Filters are used to implement entities and concerns cutting across an entity as a whole, whereas mixins only intercept certain message calls. Both, filters and mixins, primarily implement extensions to an object or a class hierarchy. Mixins may be used to compose several filters that form a semantic unit in a reusable component. This way, concerns cutting across several instances or class hierarchies, as targeted by aspect-oriented programming [18], can be modeled as well.

By intercepting the calls to a pattern structure, we can implement the recurring pattern parts as separate and reusable entities. These parts can be placed in a component and dynamically loaded at runtime. Language dynamics and introspection can be used to adapt and change the pattern implementation to the current context. In this paper we assume that such pattern implementations are available as components. Note that the XOTcl distribution contains all used pattern components. They are dynamically loaded into the framework and can be reused as if they were native XOTcl language constructs.

### 3.3 Approaches for Pattern Implementation

In this paper we will present a case study combining scripting languages, high-level language constructs, and design patterns. Much other work has been done on the combination of high-level language constructs and patterns, but unfortunately there are only a few case studies showing the validity of these approaches available. We will discuss some other approaches in this section.

A classic approach to code generation for patterns can be found in [6]. There are several subsequent approaches, some of them with notable similarities to the filter. Bosch, for example, uses static layer definitions [4] to intercept messages to a pattern fragment. Besides several differences of detail, the general difference between our approach for pattern reuse and code generation is dynamics. A pattern is not a reusable entity per se, but a reusable design solution in a context. It therefore has to be fitted to the current implementation and application context. (Static) code generation mechanisms offer no language means for this customization step. Consequently, pattern reuse through code generation seems to be rather rigid. In some sense this contradicts the idea of wholeness often associated with patterns, because in many cases it is hard to produce a flexible and elegant piece of design (and code) from a rigid, pre-fabricated building block.

Some approaches, such as component connectors [9], introduce additional runtime means to represent patterns. These split the pattern, as one conceptual entity, into several runtime entities. In contrast to the interception techniques filter and mixin class, component connectors are not transparent for the client.

A role (as proposed in [19]) is used to express extrinsic properties of a design entity. In this concept a role can be dynamically taken and abandoned. The approach does not provide an abstraction at a broader structural level, as does a per-class filter and per-class mixin. Using roles, a pattern can be implemented as a dynamic composition of its architectural fragments. That means that, by inspecting roles, patterns may be more traceable than in an implementation scattered over classes.

Several patterns express concerns cutting across different design entities. Aspect-oriented programming [18] targets such cross-cutting concerns. There are different approaches for implementing aspects; here, we will compare with the concepts of AspectJ [17]. In AspectJ, so-called pointcut designators let us specify certain events in the message flow. These events are called join points. Before, after, and around advice let us specify behavior to run when a specified join point is reached. The general idea resembles the message interceptors in XOTcl. Thus most concerns in this paper may be implemented using aspects. However, in contrast to the approach presented, AspectJ does not allow for language extension in a component-oriented sense. In other words, aspects cannot be dynamically introduced as language extensions. Instead, an aspect weaver, a kind of generative compiler, is used. Hence AspectJ does not allow for dynamic runtime changes. Aspects are given as non-local extensions to the targeted design units. Since the pattern examples presented in this paper require knowledge about the extended class and runtime callstack information, it is rather unwieldy to implement them with aspects. However, concerns cutting across several entities of the program can be better expressed using aspects, since it may take several mixins with filters to express one complex aspect.

## 4 Component Framework Design

In this section we will present several crucial excerpts from our XML/RDF component framework design. By "crucial" we mean that these framework hot spots are critical for reusability, flexibility, and runtime efficiency of the framework. Before we give the design excerpts in detail, we will present an overview of the baseline architecture of the component framework.

### 4.1 Baseline Architecture

Of several available XML parsers we decided to re-use the C written parser Expat [8] and the Tcl written parser TclXML [3]. Both can be used as Tcl components with the same interface for parsing. In the framework presented here, an object-oriented XML parser wraps these third-party parsers. It forms an abstraction integrating procedural parsers implemented in other languages (here C and Tcl). Those parsers are encapsulated in a distinct layer and are thus exchangeable with other implementations.

The XML information elements form an aggregation tree, in which children have an "is part-of" relationship to their parents. These nodes represent the grammar of XML and RDF in our framework's information architecture. Data and attributes will be treated as properties of nodes.

As a requirement, it should be possible to flexibly extend and change the grammar, so that future changes of the XML/RDF specifications can be easily incorporated. Since XML and RDF are very general data models, it is a requirement of the framework to interpret the representation in different ways and to possibly add new interpretation forms. A factory component for nodes abstracts object creation, thereby enabling extensible node creation. Since there are several node classes having a set of features in common, another goal was to create the node classes automatically.

The basic architecture described applies to both the xoXML and xoRDF components. XML, RDF, and the reused parser part are additionally structured in layers (see Figure 3), as in the Layers architectural pattern [7]. The pattern couples similar responsibilities in a distinct layer. Thus it decomposes complex aspects of the system. The three layers of Figure 3 are the basic components of xoXML/xoRDF.

### 4.2 Nodes of the Parse Tree

RDF provides a relatively simple, formally defined grammar. The central problem of the design is how to build an object-oriented structure from the RDF graph expressed

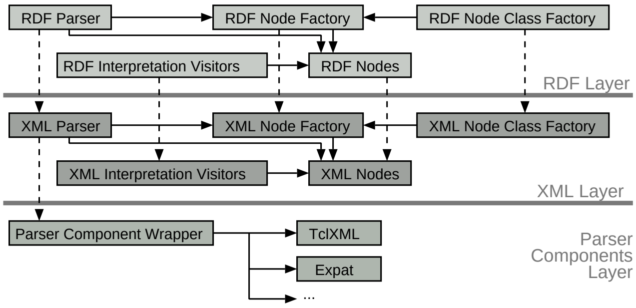


**Figure 3.** The xoXML and xoRDF Layers: The RDF layer specializes the sub-components of the XML layer. Basic text parsing is handled by a procedural parser, e.g., in C or Tcl. It is wrapped by a generic interface. Different layers are depicted in different color.

by the XML linearization. There are several options for mapping the XML elements to a computational representation. Following the line set out in Section 2, we first create a dynamic object aggregation tree; other representations can then be retrieved from the tree.

To provide an XML/RDF interpretation of the node tree, we used the Interpreter pattern [10]. It defines an object-oriented structure for a language grammar, along with an Interpreter. Clients use an abstract interface for node interpretation and, thereby, abstract node implementation details. At runtime node objects form an abstract syntax tree. Terminal nodes, like literals in RDF, terminate the tree structure, while non-terminals are able to aggregate an arbitrary number of nodes. As a consequence of the usage of the pattern, the language represented can be exchanged and extended by object-oriented means, since new node classes are very similar to existing ones.

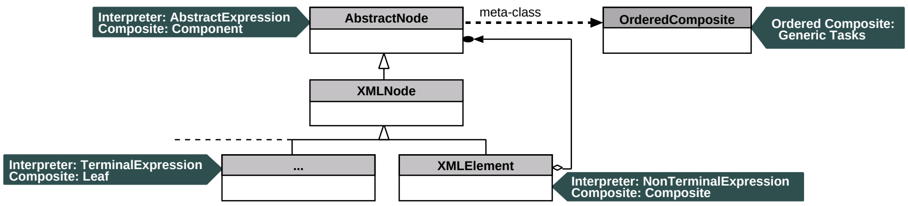


**Figure 4.** Node Tree Structure: The node tree is structured according to the Interpreter and Composite pattern. The Composite implementation is loaded from a separate component.

We will use the common implementation variant of the Interpreter pattern to build the abstract syntax tree: a Composite structure [10]. Composites arrange objects in trees with two kinds of nodes: Leafs terminate the tree, and Composites forward registered Composite operations to all children of the Composite. We have used XOTcl filters to handle the recurring tasks of the Composite pattern, such as forwarding messages through the hierarchy [23]. The pattern implementation is loaded from a pattern component *Ordered Composite* that uses the *dynamic object aggregation* language construct of XOTcl to handle the aggregation structure of the pattern automatically.

As shown in Figure 4, we define a class `AbstractNode` with the meta-class `OrderedComposite`. Each meta-class inherits the ability to define classes from the most general meta-class `Class`. On `AbstractNode` we define a core set of abstract operations, common to all node classes. These let us parse data, as well as starting and ending node tags, and allow them to be printed. Methods for interpretation on concrete classes will be registered as Composite operations. Registration is handled by the registration methods of the `OrderedComposite` class, called `addOperations` and `addAfterOperations`. The pattern implementation will ensure that the Composite operations work recursively on the tree. The XOTcl code for loading of the Composite and definition of the abstract node class is:

```
package require OrderedComposite
...
OrderedComposite AbstractNode
AbstractNode abstract instproc parseStart {name attrList}
AbstractNode abstract instproc parseData {text}
AbstractNode abstract instproc parseEnd {name}
AbstractNode abstract instproc print {}

Class XMLNode -superclass AbstractNode
```

First, the `OrderedComposite` component is loaded. Then an abstract node with abstract methods is defined as a general node interface. Finally, a special node class, the general `XMLNode`, inherits from `AbstractNode`.

### 4.3 XML Namespaces

XML and RDF nodes may contain namespace declarations starting with `xmlns`. Often namespaces are declared within the top node, but it is also possible to declare namespaces within inner nodes. These overlap the namespace declarations of outer nodes, and they are only valid for the current node and its inner nodes. A direct coupling between namespaces and connected nodes would cause the namespace objects to store unnecessary information. A new namespace would be difficult to add in applications with a dynamically changing node tree. A better solution is an indirection to a namespace handler; that is, each node must be connected to one responsible namespace handler. The namespace handler knows its successor in a Chain of Responsibility [10]. If a namespace handler cannot resolve a namespace it delegates the task to its successor.

The Chain of Responsibility pattern is implemented as a reusable component. We add operations to the general `Namespace` class for adding a namespace and for determining a namespace's full name from the prefix, as well as two operations, `searchPrefix` and `searchFullName`, for retrieval of a namespace object from a handler by prefix and full name. The two retrieval operations act on a single handler. To automatically act recursively on the chain, they must be registered as chained operations with `addChainedOperation`:

```
package require ChainOfResponsibility
...
```

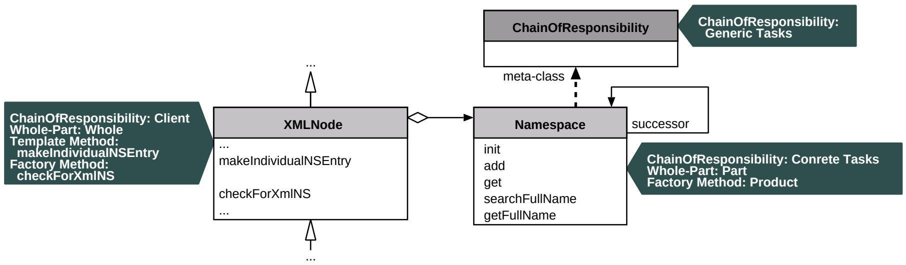


**Figure 5.** XML Namespaces: Namespaces are part-of their node and are structured in a Chain of Responsibility

```
ChainOfResponsibility Namespace
Namespace instproc add {prefix ns} {...}
Namespace instproc searchPrefix {prefix} {...}
Namespace instproc searchFullName fname {...}
Namespace instproc getFullName fname {...}
Namespace addChainedOperation searchPrefix {}
Namespace addChainedOperation searchFullName {}
```

A chained operation returns a value indicating whether the operation was successful or not. If not, the next namespace in the chain is tried. This way the chain is searched until the last namespace in the chain is reached.

Namespace declarations are closely coupled to the nodes, and their lifetime is determined by the surrounding node; therefore, the node aggregates its namespaces in a Whole-Part pattern [7]. Moreover, the namespace chain is integrated in the Composite structure of the nodes. Note that this combination of Composite and Chain of Responsibility is a quite typical design for cases in which responsibilities have to be delegated up the tree.

### 4.4 Wrapping the Parser Component

Different XML parsers exist that perform the same task but have no common interface. Since applications should be portable to different components performing the same task, direct invocations of these components should be avoided. Instead an object-oriented interface can encapsulate the functions of the third-party parser components, as in the Wrapper Facade pattern [30]. A Wrapper Facade provides an interface for client objects. It forwards methods to the functions (or procedures) of the procedural (legacy) components. The pattern Component Wrapper [34] integrates different techniques to access foreign language and paradigm components as black-boxes. Here, Wrapper Facade is used as a part of a Component Wrapper.

We define a Wrapper Facade `XMLParser` to incorporate the interface of TclXML or Expat as object-oriented methods (see Figure 6). The method `configure` sets the parser configuration, `cget` queries a configuration option, `parse` invokes the parsing of XML text, and `reset` cleans up the parser. This generic interface may also be used with other parsers, e.g., by using the Adapter pattern [10]. It is also reusable in a component-based fashion [23]. To make the interface more generic we add three methods: `start`, `end`, and `pcdata`. These will be called when the start or end of an XML node or XML data is reached.

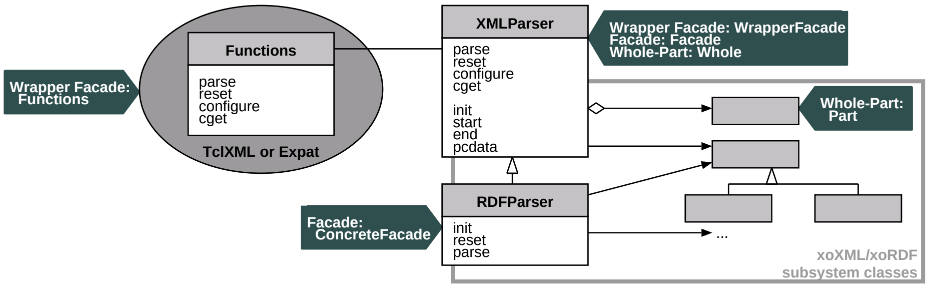


**Figure 6.** Parser Design: The reused parser is wrapped behind an object-oriented interface. It forms a Facade to the sub-system for clients.

As another role of the parser, we define a Facade [10] for the xoXML and xoRDF sub-system. Facades unify the interfaces of a sub-system into one interface. They thus ease access to the sub-system, because client objects have to access a smaller number of interfaces and are more strongly decoupled from the sub-system's implementation details.

As another architectural role, the parser has to serve as a central access point for the namespace chain introduced previously, and for the Node Factory presented in the next section. The parser aggregates both of these roles as parts in a Whole-Part structure.

### 4.5 Node Object Factory

Node creation is a hot spot of the framework; that is, variation requirements can be foreseen in these spots. Let us consider a collective change for all node creations, such as introducing the Flyweight pattern [10] for sharing literal nodes. In XOTcl Flyweights are reusable in a component-based way; for one creation process only three lines of additional code are necessary to load and use the Flyweight component. Integration of the Flyweight into the creation processes of every single literal node class would represent a significant implementation overhead. Node creations may well be scattered throughout the code, and would have to be searched to introduce collective changes. Furthermore, these creations are often located in code of clients of the node classes. A strong coupling between client code and node classes would exist, making all future changes problematic. Since changes in node creations are foreseeable, e.g., in situations where the XML/RDF specifications change, control of all creation processes from a central point is a superior solution.

Factory patterns, like Abstract Factory and Factory Method [10], provide us with a central access point to creations of related or dependent objects (called *products*). To create a node factory, we firstly create the Abstract Factory with a method `getNode`. It returns a newly created node:

```
Class AbstractNodeFactory
AbstractNodeFactory abstract instproc getNode {keyCl objName}
```

Now we concretize the factory with a simple node factory, whose function is to create every requested node.

```
Class NodeFactory -superclass AbstractNodeFactory
NodeFactory instproc getNode {keyCl objName} {
  ...
  return [$keyCl create $objName]
}
```

### 4.6 Node Class Factory

One major benefit of using the Interpreter pattern lies in the similarities between node classes. New node classes can be added rapidly. A trivial way of adding similar classes is to copy the classes' code and change it slightly. Since changes must be propagated throughout the code, changeability may suffer, because for each change all dependent places in the code have to be searched and updated. Replicated parts of the code are an implementation overhead, and so this solution is inelegant and error-prone. Using code generation or putting recurring parts into superclasses minimizes these problems. But the object creational problems sketched in the previous section also occur with class creation.

Central control of the adaptation of newly created classes to the application context and domain let the programmer implement changes on several classes at once. As XOTcl supports meta-classes and dynamics in class-structures, a solution similar to the Abstract Factory pattern can be provided in a meta-class. Each XOTcl class is a runtime object; therefore, there are the same benefits and liabilities as for ordinary object Factories. Our main concerns for applying the pattern for class creation purposes is that the node class configuration tends to change, as the architecture evolves and new requirements for the parsing framework emerge. During the Class Factory's creation process we can introduce interdependent constraints, such as the possible compositions of RDF nodes, in a central place. In XOTcl a meta-class is defined by specifying the most general meta-class `Class` as superclass. Afterwards we create a class for the XML element type dynamically:

```
Class XMLNodeClassFactory -superclass Class
XMLNodeClassFactory create XMLElement -superclass XMLNode
```

For the RDF node tree we specialize the Class Factory, because RDF nodes define the constraints of nodes they may nest in or be attributed to at creation time. We add these initializations to the `create` method of the node class object.

```
Class RDFNodeClassFactory -superclass XMLNodeClassFactory
...
RDFNodeClassFactory proc create args {
  set name [next]       ;# create the class
  ...                   ;# perform initializations
}
```

Subsequently, all types of RDF nodes are defined in a loop. All classes are automatically placed at the right point in the class hierarchy, and all initializations are performed during creation. The loop creates the hierarchy in Figure 7:

```
foreach {name sc content attributeList} {
  RDFTag         RDFNode                RDF         {}
  RDFBag         RDFContainerNodeClass Bag         {ID}
  RDFSeq         RDFContainerNodeClass Seq         {ID}
  RDFAlt         RDFContainerNodeClass Alt         {ID}
  RDFProperty    RDFNode                ""          {bagID ID resource parseType}
  RDFMember      RDFProperty            li          {resource parseType}
  RDFDescription RDFResource            Description {ID bagID about type
    aboutEach aboutEachPrefix}
} {
  RDFNodeClassFactory create $name -superclass $sc \
    -content $content -attributeList $attributeList \
}
```

It is quite common for XML interpretation frameworks to have constraints for node nesting and node attributes, because usually the information architecture reflects the XML structure. The pattern Generic Content Format [31] describes such structural interdependencies in the area of web engineering architectures. Composite and Leaf classes represent each information element type used in the XML documents. As business requirements, standards, or other interpretation requirements change, both, the XML structure and the class hierarchy have to be changed accordingly. These forces can be observed for our RDF framework as well; here, the changes are mainly introduced by the changing RDF standard and the interpretation requirements of our applications.

In our framework, changes of the class hierarchy and its constraints can be rapidly performed in the central place provided by the Class Factory. Introspection options can be used to find out these interdependencies at runtime.

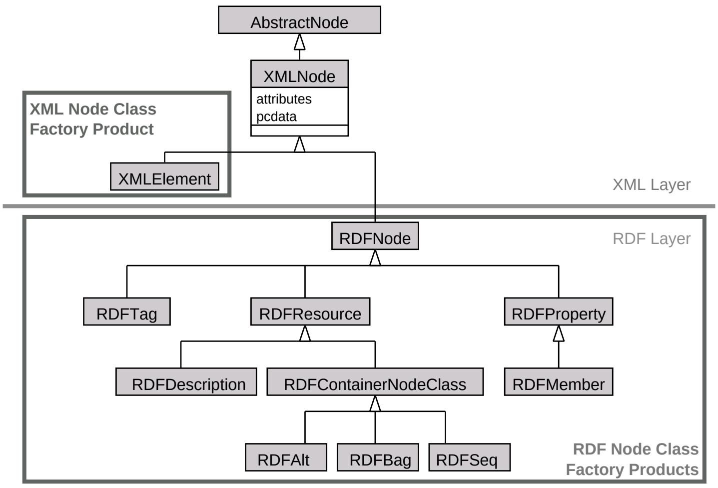


**Figure 7.** Node Class Hierarchy: RDF node classes are specialized, but through their generic superclasses they form a dynamic node tree.

### 4.7 Generic Interpretation with Visitors

In our implementation of the XML/RDF processor we have chosen to provide both an event stream model and a tree structure model. In a first step, the RDF structure is parsed from the event stream into a node object tree. The event stream can be used by applications, if required. Next the tree is interpreted according to the application domain and context. One implementation option would be to insert the interpretation code into the node classes. This is a bad choice when different or multiple interpretations of the node tree are needed. As another drawback, interpretations cannot be changed without making changes to every node class. Parsing cannot be adapted without checking all dependencies against interpretation code.

An interpretation facility is required for abstracting implementation details of the node tree and for supporting several different interpretation forms. The Visitor pattern [10] provides such a facility. Visitors perform operations on all parts of an object structure separated from the object classes. An abstract Visitor declares an interface that can be attached to a given object structure. Concrete Visitors are used for visiting each element of the structure. The elements implement `accept` operations that take a Visitor as an argument and return themselves in the concrete implementations to the concrete Visitor. The concrete Visitor performs the interpretation task on the concrete element.

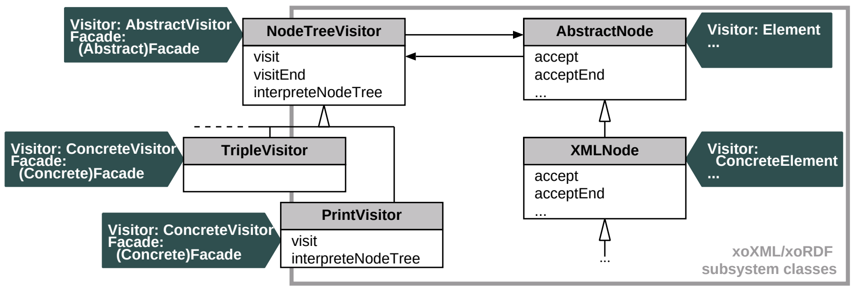


**Figure 8.** Node Tree Visitor: Interpretations tasks are decomposed from the node tree. Customized visitors can be added by sub-classing.

In the case of the XML/RDF parser the nodes are used as elements for the Visitor. So we attach a new architectural role to the abstract node class. It becomes a Visitor's abstract element, while the concrete nodes become concrete elements. We use interpretation to reconstruct the start and end of a node by adding two operations to the abstract node's definition. In conventional implementations both operations would have to propagate `accept` calls to their children. Here, this is handled automatically by the Composite pattern implementation:

```
Abstract Node abstract instproc accept {visitor}
AbstractNode abstract instproc acceptEnd {visitor}
AbstractNode addOperations {accept accept}
AbstractNode addAfterOperations {accept acceptEnd}
```

Now, `accept` is a Composite operation; that is, before the operation `accept` is performed on the node, an `accept` call is performed on each child. `acceptEnd` is performed on the child after the return from the parent operation. Both concrete operations on `XMLNode` merely call the appropriate Visitor with a self-reference as argument:

```
XMLNode instproc accept {visitor} {
  $visitor visit [self]
}
XMLNode instproc acceptEnd {visitor} {
  $visitor visitEnd [self]
}
```

These operations can be specialized to more sophisticated `accept` handling in subclasses. An abstract Visitor is defined with operations for visiting, and an operation starting the interpretation of a node tree. `visitEnd` is not defined as abstract but as an empty method, because it may remain unspecified in concrete Visitors:

```
Class NodeTreeVisitor
NodeTreeVisitor abstract instproc visit objName
NodeTreeVisitor instproc visitEnd objName {;}
NodeTreeVisitor abstract instproc interpretNodeTree n
```

Shown below is the implementation of a simple `PrintVisitor`, using only the node's `print` operation. Each node prints its content to the standard output. Calling the Visitor's `interpretNodeTree` operation on the top node outputs the whole hierarchy.

```
Class PrintVisitor -superclass NodeTreeVisitor
PrintVisitor instproc visit objName {
  puts [$objName print]
}
PrintVisitor instproc interpretNodeTree node {
  $node accept [self]
}
```

The xoXML/xoRDF framework contains several Visitors for different interpretation tasks, including one for building up an RDF triple database from the node tree, Visitors for recreation of XML/RDF text from the nodes, and Visitors for pretty-printing of the object tree. These features can be found in the XOTcl distribution. Since they are, from an architectural point of view, similar to the Visitor presented above, we do not discuss them in detail here.

## 4.8 Framework Overview

Figure 9 shows the different components and data representations in the xoXML/xoRDF framework. Arrows indicate the data flows between components, and conversion between data representations. XML/RDF text can be processed by different parsers that are wrapped by the object-oriented parser components. An intermediate object tree representation is created, on which interpretations are performed. Each individual interpretation is implemented as a Visitor. In many applications we provide domain-specific Visitors that extract the required information.

With the same Visitor architecture we can also derive other representation of the information contained in the object tree. In our framework, we have implemented a Visitor for creating RDF triples. Another component recreates XML and RDF text from the object tree.

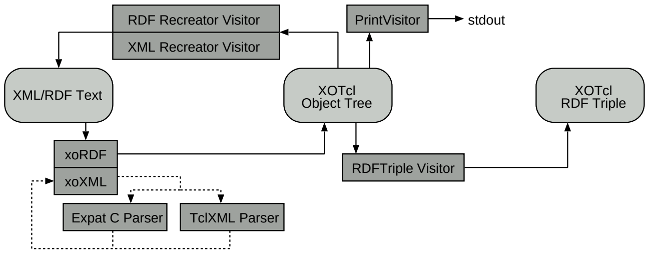


**Figure 9.** Components and Data Representations in the xoXML/xoRDF Framework: From the XML/RDF text either an object aggregation tree or a triple database or both is produced.

# 5 Speed Comparison

In this section we discuss the speed penalties of using high-level XOTcl language constructs, such as patterns and object-oriented scripting, relative to C- and Java-based single-purpose implementations. We provide speed comparisons with the W3C reference implementation SiRPAC [29] 1.14. SiRPAC uses an external parser: we used IBM's XML4J [16] version 3.0.1 in the Xerces-J [2] variant. In addition, we compared with the pure C implementation HTRdf [28]. All speed comparisons were performed on an Intel Pentium III 500 Mhz single processor machine running RedHat Linux 6.1 with 128 MB RAM. The version of XOTcl and the xoXML/xoRDF components used was 0.84.

For each setup we tested each test case 20 times, and used the best result achieved (note that the average results were almost identical; therefore we omit those figures here for space reasons). Finally, we summed the results and calculated the percentage difference between each sum and the figure for HTRdf, as the fastest implementation in the test.

For the comparisons we used example RDF files from the SiRPAC distribution and from the W3C site. For all tests we measured the overall time for instantiating the parser, reading the file and parsing the text. Since xoXML/xoRDF offers different setups, we provide figures for different combinations of parser and created representation. For xoXML/xoRDF the most common variant is to create the object aggregation tree representation with the Expat C Parser. As discussed above, xoXML/xoRDF generates the triples via an object tree representation. Our measurements contain figures for the tree and triple generation. Finally we performed both tests with the Tcl-only parser TclXML as parser back-end. TclXML has a similar interface to Expat and can be plugged into the system by specifying a single configuration parameter.

Even though a comparison of such results is quite hard and subjective, the performance measurements (summarized in Table 1) are very encouraging for the scripting approach with patterns. In all cases the implementation in the scripting language is substantially faster than SiRPAC, despite all high level constructs, patterns and indirections. Even the Tcl-only, but platform-independent, parser and triple generation from the tree still operate at a useful performance level. Of course, the scripting implementation is

| *Filename/Description* | SiRPAC/ Xerces-J (Java) | HTRdf/ Expat (pure C) | xoRDF/ Expat (only tree) | xoRDF/ Expat (tree & triples) | xoRDF/ TclXML (only tree) | xoRDF/ TclXML (tree & triples) |
|---|---|---|---|---|---|---|
| `example1.rdf` Nested Description | 273 ms | 17 ms | 28 ms | 44 ms | 44 ms | 57 ms |
| `example2.rdf` Two Descriptions | 284 ms | 27 ms | 27 ms | 49 ms | 40 ms | 63 ms |
| `example3.rdf` Bag & aboutEach | 274 ms | 23 ms | 28 ms | 43 ms | 40 ms | 56 ms |
| `example4.rdf` Resource Property | 282 ms | 19 ms | 26 ms | 42 ms | 35 ms | 51 ms |
| `pics.rdf` 2 BagIDs & aboutEachs | 366 ms | 116 ms | 67 ms | 165 ms | 81 ms | 180 ms |
| `dc.rdf` Larger Dublin Core Example | 1183 ms | 131 ms | 215 ms | 363 ms | 277 ms | 429 ms |
| Sum: Whole Test Suite | 2389 ms | 316 ms | 363 ms | 662 ms | 473 ms | 779 ms |
| Comparison to HTRdf (in %) | +656 % | 0 % | +15 % | +109 % | +49 % | +149 % |

**Table 1.** Speed Comparison: Different xoXML/xoRDF Setups Compared to the HTRdf/Expat C Implementation and the SiRPAC/Xerces-J Java Implementation

slower than an efficient, pure C implementation. xoRDF is, as expected, 0.14-1.46 times slower than HTRdf in our tests, but HTRdf only builds triples.

The performance comparison indicates that the xoXML and xoRDF implementation can be used in practical applications, when considerable control and customization is needed. Nevertheless, the speed comparisons are still unsatisfactory if the task is merely to create triples. Since we argue for the component reuse aspect, we can provide a wrapper for HTRdf (or another C implementation), enabling HTRdf to directly create triples. In this case, construction of the object tree representation is skipped.

## 6 Conclusion

In this paper we have presented an extensible design of an XML/RDF parser and interpreter framework. The case is non-trivial, yet of a comprehensible size. The design presented has evolved over three years and, during this time period, we had to cope with diverse new requirements, such as higher performance demands, less memory consumption, simplification of interfaces, and changes in the RDF standard (and its interpretation). The use of patterns and high-level constructs in crucial parts of the design has allowed us to easily adapt the design to new, unexpected situations. Despite considerable new requirements, the original design core is still stable.

Flexible glueing of components, combined with the Component Wrapper pattern, has given us the opportunity to reuse existing parser implementations in a manner transparent to the object-oriented implementation. The high level language constructs of

XOTcl, such as filters, mixins, and dynamic object aggregations provide flexible means of adapting and supporting higher-level design constructs, say, certain design patterns, roles, or Whole-Part structures. Several examples have demonstrated how patterns and high-level constructs interact, and how this interaction produces pattern variations. For example, we have discussed the language-supported Composite pattern and the Ordered Composite variant. Variants often can be derived by runtime language means. The idea of the Class Factory came from combining that of the Abstract Factory pattern and the class object and dynamic classes language constructs of XOTcl.

Our approach does not force developers to use XOTcl as the implementation language of the whole system, since components can be integrated with little effort in any C or C++ program. Moreover, the concepts can be introduced into almost any procedural and object-oriented language with the pattern language from [34,14,12,13]. Thus, the ideas can be used independently of the concrete XOTcl implementation, and the case study discussed here can be seen as an example of a usage of this pattern language with XOTcl. To a certain extent, other implementation languages, such as AspectJ, can be used. Our experiments with AspectJ and other Java-based AOP approaches indicate that current implementations have similar liabilities as our approach in respect to speed penalties, but, as generative approaches, they do not allow for runtime changes. XOTcl is completely implemented in C and heavily optimized for runtime performance. Therefore, we believe, future implementations of dynamic message interception techniques in Java or C++ will have similar consequences as using XOTcl together with C or C++ components.

The benefits of our approach are only useful for applications requiring high flexibility and component glueing. There are also some drawbacks in the approach discussed. Scripting languages are slower than system programming languages, such as C or C++, because of dynamic conversions and method lookup. We can minimize these problems by using C components wherever high flexibility is not required. If the tasks performed in the scripting language are not even required by the application, however, the performance penalty becomes significant. For example, if an application only needs a triple representation of an RDF structure, the creation of an intermediate object tree is unnecessary.

It may be hard for application developers, not used to the scripting language, to understand the "new" language. Similarly, the patterns and their relations have to be understood and cannot simply be used "out-of-the-box," but they require usage experience and knowledge of possible implementation variants. In our experience, usage experience and understanding of patterns is especially relevant to inexperienced programmers who want to *extend* the framework (as for instance observed in student projects). Here, our approach helps novices to trace the patterns as design units in the code. However, for modifications and customizations of the design, the intent, the effects on quality attributes, and the structure of the used patterns have to be fully understood. The component-oriented structuring hides the patterns from users writing applications that only *use* the framework; therefore, for merely using the framework its internal design has not to be fully understood but only the component interfaces. If the pattern language from [34,14,12,13] is used instead of XOTcl, the implementations of the wrapping and indirection mechanisms have to be maintained by the development organization.

In general, our case study shows that the choice of the expression resources used for programming, such as programming languages, patterns, paradigms, etc., has to deal with conflicting forces. These can be at least partially resolved by a glueing approach. We have introduced component glueing as a wrapping technique for combining several languages and paradigms properly. Note that some of the glued components were XOTcl components for implementing recurring pattern variants. Thus the component glueing concept has allowed us to extend XOTcl with the concerns implemented in pattern components. These components implement a variant of the pattern that can be further customized to the application context by different dynamic language resources. The patterns have extended the XOTcl language dynamically with new language elements that are as well elements of the design repository. The loaded language element, implementing the pattern variant, uses the standard XOTcl syntax, and it is dynamic, introspectible, and traceable as a runtime entity of the language like all other XOTcl language elements.

XOTcl and the XML/RDF components are freely available from:
`http://www.xotcl.org`.

## References


1. C. Alexander. *The Timeless Way of Building*. Oxford Univ. Press, 1979.
2. Apache XML Project. The Apache XML project. http://xml.apache.org/, 2000.
3. S. Ball. XML support for Tcl. In *Proc. of the Sixth Tcl/Tk Conference*, San Diego, CA, USA, September 1998.
4. J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2), 1998.
5. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, 1998.
6. F. Budinsky, M. Finnie, P. Yu, and J. Vlissides. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.
7. 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.
8. J. Clark. Expat - XML parser toolkit. http://www.jclark.com/xml/expat.html, 1998.
9. S. Ducasse. Message passing abstractions as elementary bricks for design pattern implementation. In *Proceeding of ECOOP Workshop on Language Support for Design Patterns and Frameworks*, Jyväskylä, Finland, 1997.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
11. M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proc. of Second International Symposium on Generative and Component-Based Software Engineering (GCSE'2000)*, Erfurt, Germany, Oct 2000.
12. M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.
13. M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.
14. M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. Accepted for publication in Journal of Software Maintenance: Research and Practice, 2001.

15. M. Goedicke and U. Zdun. Piecemeal migration of a document archive system with an architectural pattern language. In *5th European Conference on Software Maintenance and Reengineering (CSMR'01)*, Lisbon, Portugal, Mar 2001.
16. IBM. XML4J, Version 3.0.1. http://www.alphaworks.ibm.com/, 2000.
17. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, October 2001.
18. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241. Springer-Verlag, 1997.
19. B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996.
20. O. Lassila and R. R. Swick. Resource description framework (RDF): Model and syntax specification. http://www.w3.org/TR/WD-rdf-syntax/, 1999.
21. D. Megginson. SAX 2.0: The simple API for XML. http://www.megginson.com/SAX/index.html, 1999.
22. G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.
23. G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.
24. G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proceedings of NOSA'99, Second Nordic Workshop on Software Architecture*, Ronneby, Sweden, August 1999.
25. G. Neumann and U. Zdun. Towards the usage of dynamic object aggregation as a foundation for composition. In *Proceedings of Symposium of Applied Computing (SAC'00)*, Como, Italy, March 2000.
26. 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.
27. J. K. Ousterhout. TCL: An embeddable command language. In *Proc. of the 1990 Winter USENIX Conference*, January 1990.
28. J. Punin. W3C sample code library libwww RDF parser. http://www.w3.org/Library/src/HTRDF, 1998.
29. J. Saarela. SiRPAC - simple RDF parser & compiler. http://www.w3.org/RDF/Implementations/SiRPAC/, 1998.
30. D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.
31. O. Vogel and U. Zdun. Dynamic content conversion and generation on the web: A pattern language. submitted to EuroPlop 2002, 2002.
32. W3C. Document object model. http://www.w3.org/DOM/, 2000.
33. D. Wetherall and C. J. Lindblad. Extending TCL for dynamic object-oriented programming. In *Proc. of the Tcl/Tk Workshop '95*, Toronto, July 1995.
34. U. Zdun. *Language Support for Dynamic and Evolving Software Architectures*. PhD thesis, University of Essen, Germany, January 2002.