XML and the Art of Code Maintenance

Martin Klang

XML and the Art of Code Maintenance

Martin Klang

This paper examines the benefits of using XML to represent source code, with a particular focus on the extensibility and processability aspects.

Using as a basis for the discussion o:XML, a general-purpose object-oriented language expressed in XML, it is demonstrated how code can be developed, processed and repurposed using standard XML tools.

Vocabularies for Extreme Programming, Design By Contract, source code documentation and Aspect Oriented Programming are introduced through examples, and their use is examined in some detail. A method for layering information using XML namespaces is presented. Implementations of the extensions using code transformations are also provided.

The paper demonstrates how the often cited separation of content from presentation achieved in XML publishing has a parallel in software development with the separation of logic from implementation. Using XML, software developers are no longer limited by the programming language in what tools and abstraction mechanisms are available. The source code and the development process itself can be customised to individual needs by combining elements from different methods and methodologies.

Table of Contents

List of Figures

List of Examples

Introduction

There seems to be no end to the innovations that we come up with in our attempts to make software development practices more efficient and reliable. The art of writing good code is only one side of it, the art of maintaining that code is quite another. To make it feasible to update, upgrade, and enhance we have to document, test and sometimes prove the code. And code updates are one of the most persistent parts of software project lifecycles. To fix bugs, accomodate changing requirements, add or improve functionality we make hundreds, if not thousands of changes. And then it's time for another set of requirements, and another release!

Good development practices makes code maintenance much less painful. Unit testing and effective use of programming assertions means that we don't go looking for bugs where there are none. Comprehensive documentation helps us understand the code and models that we are working with.

When examining all the different activities of software development, not only code pushing, it becomes clear that traditional source code formats leave a lot to be desired. XML has several distinctive qualities that makes it a highly attractive alternative. This paper focuses on two of them - extensibility, and the ease with which it can be processed.

Using XML we can extend our source code to include exactly the information we want, whether its documentation, test cases or process-related data. With extensions we can incorporate our methodology of choice, or combine elements from different programming methods. The source code becomes not just a program description, but a hub of useful, repurposeable information.

Code transformations make it possible to employ methods and abstraction mechanisms that are not compatible with most programming languages. This does not require specialised compilers or execution environments, simply the fact that the code itself can be processed by standard XML tools is enough.

The focus of this paper is object-oriented development using o:XML, but many of the ideas expressed can easily be extended to other languages and paradigms.

Chapter 1. XML and Software Engineering

This paper examines how using an XML-based programming language can drastically improve several aspects of software development.

Better Quality Code. The quality of the code that we produce is determined not only by our own individual coding talent, but also by the abstraction mechanisms that we use. Theoretically we could write all programs as a single long sequence of commands and gotos, but it is unlikely that we will produce decent programs unless we have some means of abstraction. Procedural and functional programming both provides such means, object oriented design and development gives us several very powerful means of modelling complex systems.

In Chapter 3, *Enabling Technology* we look at another very powerful abstraction mechanism, Aspect Oriented Programming. Used together with XML, AOP can be applied in a simple, declarative way, with an implementation based on code transformations that does not require any special compiler or interpretor. This demonstrates one of the consequences of using XML in software development, that programs are no longer static descriptions of logic but can be transformed and repurposed.

Improved Reliability. We will look at two alternative methods for ensuring that the code we write does what it is supposed to do. The first one is Test Driven Development, often mentioned together with Extreme Programming or Agile Software Development. The idea with TDD is that as a programmer, you first write the test and then the code. Hence the proof that the code works, or doesn't work, exists right from the start. We present a simple, effective method for writing tests into your types and functions, without in any way compromising the code.

The second method presented is Design by Contract. DbC takes the view that the system specification is in fact a contract that has to be fulfilled by both the code and the client/user. For each software element, there should be a clause (or several) in the specification saying what the code can expect as input, and what it must produce as output. These clauses are then written into the code as assertions. If any part of the system does not fulfill the contract, ie fails the specification, it is made obvious by an assertion failure.

Leveraging the extensible properties of XML, either or both methods can be seamlessly integrated in a way that would be impossible with traditional programming languages.

Better Quality Documentation. Often good documentation is created by reusing information that is already available. If you've written a constructor function that takes five parameters, you shouldn't have to repeat that fact in the documentation. A great deal of information is already encoded in the function definition. Using an XML-based programming language, this information can easily be harvested, for example by an XSL stylesheet. Because the information is not tied into a particular syntax, it becomes available to a much wider range of processing tools. Furthermore and thanks to the extensibility of XML, the format for inline documentation can be adapted to the particular needs of documentation, rather than the syntax limitations of the programming language.

The purpose is not to compare development paradigms or methods, simply to demonstrate how different approaches can be easily and naturally integrated using XML. That XML is both extensible and easily processed opens up a world of possibilities when applied to software development.

Introducing o:XML

o:XML is an object-oriented language expressed in XML. Unlike XSLT, which is a functional language created to solve the problem of transformations, o:XML is a general-purpose procedural language. Like many other modern languages it is dynamically typed.

The result of running an o:XML program is either a stand-alone XML document or an XML fragment. Literal XML included in a program is simply copied to the output (as long as it's not in the o:XML namespace). Furthermore o:XML fragments can be included in any XML document - when processed by an o:XML engine the statements will be executed in turn and replaced with their respective output.

The o:XML processing model does not mandate an input document. Programs can open files or input streams

but aren't required to take any input at all. This makes it possible to embed dynamic content in XML documents, and it gives the language a wider range of applicability.

Since every o:XML program is an XML document it lends itself to processing. This also makes it simple and safe to generate o:XML, using for example an XSL stylesheet or another o:XML program.

o:XML has an expression language which is almost identical to XPath. The one additional construct is the dot operator, which is used to denote type functions. Type functions can be invoked on any node, and every object in o:XML is a node. The effect of calling a type function on a nodeset is the same as calling it on every node in the set (composition).

The expression language can be extended with user defined functions. There's a range of built-in types representing XML constructs - String, Number, Boolean, Element, Attribute, Document, etc. There are also types for exceptions and exception handlers. User defined types inherit functionality from one or more parent types (multiple inheritance). Type variables are available within the type functions same as any other variable, there is also the special variable `$this` which refers to the node that the function was invoked on. All type variables are private, and so can only be accessed by the declaring type.

The reference implementation of o:XML is ObjectBox, a compiler and interpretor written in Java. o:XML source code can interpreted, or it can be transformed to produce platform-specific code. This gives separation of responsibility, similar to the distinction between content and presentation in XML publishing.

Figure 1.1. Source Code Transformations

Below is a simple example of an o:XML program. See Appendix E, *o:XML Quick Reference* for a summary of o:XML constructs and statements.

Example 1.1. o:XML Type Definition

```
<o:program xmlns:o="http://www.o-xml.org/lang/"
  xmlns:ex="urn:examples">

  <o:type name="ex:Stack">
    <o:parent name="Node"/> <!-- default parent type -->
    <o:variable name="items"/>

    <!-- type function -->
    <o:function name="push">
      <o:param name="item"/>
      <o:do>
        <o:do select="$items.add($item)"/>
      </o:do>
    </o:function>

    <o:function name="pop">
      <o:do>
        <o:return select="$items[last()].remove()"/>
      </o:do>
    </o:function>

    <o:function name="empty">
      <o:do>
        <o:return select="count($items) = 0"/>
      </o:do>
    </o:function>
  </o:type>

  <o:set stack="ex:Stack()"/> <!-- uses default constructor -->

  <o:do select="$stack.push('hello')"/>
```

```
  <o:do select="$stack.push('goodbye')"/>

  <o:while test="not($stack.empty())">
    <o:log msg="pop: {$stack.pop()}"/>
  </o:while>

</o:program>
```

o:XML has many features which will not be covered here. Instead the language is used as the basis for a more general discussion about XML and software development. For more information about the o:XML project, please see the o:XML Web Site [http://www.o-xml.org].

Chapter 2. X is for eXtensible

Abstract

We present three distinct XML vocabularies and demonstrate integration with o:XML source code. The vocabularies relate to interface documentation, unit tests and Design By Contract conditions. By layering the information with XML namespaces, syntax conflicts are avoided and selective processing can be performed using standard XML tools.

Automated unit tests and the implementation of Design By Contract using code transformations is explained in detail. There is also a discussion on the use of extension elements to customize the development process, using Extreme Programming and Design By Contract as examples.

Embedding Documentation

Practically all programming languages provide support for embedding documentation with the source code in the form of comments. Some support a standard syntax for documenting application interfaces, and have tools that automatically generate documentation. In the case of Java, this has resulted in vast improvements in the overall situation of API documentation. One problem is that in order to harvest the documentation, you need to parse and understand the source code.

In o:XML, the core language does not provide a documentation or commenting mechanism. Instead, this is accomplished by adding in a documentation vocabulary, a new syntax superimposed on the code syntax. Clean separation from the actual code is achieved by using separate XML namespaces.

The advantage of using extensions this way is that they can be as simple or as sophisticated as is deemed necessary. Since there is no conflict with the source code, there are no real limitations to how the vocabulary is constructed. Furthermore existing XML vocabularies can be effectively reused. In the case of adding documentation to our source code, it would be quite easy to incorporate an existing format such as DocBook [http://www.docbook.org].

The vocabulary we use for o:XML is based on the specific requirements of documenting code - it covers function and procedure parameters, return values, exceptions and similar information, and it takes full advantage of the code context.

Example 2.1. Embedding Documentation: Node.insertBefore(Node)

```
<o:type name="Node"
  xmlns:o="http://www.o-xml.org/lang/"
  xmlns:doc="http://www.o-xml.org/lang/docs">

  <doc:p>Node is the base type of all other types.</doc:p>

  <o:function name="insertBefore">
    <o:param name="other"/>
    <doc:p>
      Insert another node before this one in the parents set of child nodes.
    </doc:p>
    <doc:param name="other">The node to insert.</doc:param>
    <doc:return>The current node.</doc:return>
    <o:do>
      <!-- function body -->
    </o:do>
  </o:function>
...
</o:type>
```

The above example is taken from the o:XML Node interface definition. All components in the namespace prefixed by *o:* pertain to the core o:XML language and make up the actual type definition.

To produce good quality documentation from this markup, the documenting process must be able to take certain information from the code context - things like type name, function names, parameter information etc. The execution environment however does not need to know anything about documentation - it can happily disregard markup that it doesn't recognise.

This gives us a separation of responsibilities - and the ability to write simple but effective tools to process only certain aspects of the source code. We can easily generate HTML with an XSLT stylesheet using the documentation format in the above example.

Example 2.2. Generated Documentation: Node.insertBefore(Node)

For further examples of what documentation generated from the ObjectBox source tree looks like, see Appendix A, *API Documentation* (ObjectBox is the o:XML reference implementation).

In an attempt to limit the scope of this paper we will not cover the topic of Literal Programming. For a discussion about using XML as the basis for Literate Programming, see for example [Wals01]. It is worth noting that using o:XML, the central problem of embedding code is no longer an issue.

More Extensions - Embedding Unit Tests

Automated tests form an important part of most medium to long term development projects, as regression testing is generally required with every software release. The o:XML compiler and interpretor uses an XML format for embedding unit tests with the type interfaces. This provides verifiable compliance, and the build and deployment procedure automatically produces comprehensive test reports with details on code coverage.

We define a test to consist of:

- Test definition - the logic that runs the test
- Input data
- Expected result

To capture this we declare three corresponding elements in our unit test namespace, plus one **test** element to wrap them in. Here follows a simple example of testing a String `length()` function:

Example 2.3. Unit Test: String.length()

```
<o:function name="length">
  <doc:p>Get the size of this string</doc:p>
  <doc:return>The number of characters in this String</doc:return>
  <o:do>
    <!-- function body -->
  </o:do>
  <xp:test>
    <xp:definition>
      <o:return select="string($input).length()"/>
    </xp:definition>
    <xp:input>how long is a piece of string?</xp:input>
    <xp:result>30</xp:result>
  </xp:test>
</o:function>
```

Note

In the o:XML implementation of unit tests, the input data set is available in the test definition through the (implicit) `$input` variable.

Out of the three unit test subelements only the test definition is mandatory, since we can achive much of the actual code verification with assertion statements.

Example 2.4. Unit Tests: Node.copy()

```
<o:function name="copy">
  <doc:p>
    Create a deep copy of this node - all child nodes will be copied as well.
  </doc:p>
  <doc:same>Node.deep(true())</doc:same>
  <doc:return>A deep copy of this node.</doc:return>
  <o:do>
    <!-- function body -->
  </o:do>
  <xp:test>
    <xp:input>
      <test attr1="one" attr2="two">
        <one>one</one>
        <two>two</two>
      </test>
    </xp:input>
    <xp:definition>
      <o:variable name="copy" select="$input/test.copy()"/>
      <!-- assert that we have copied the child nodes -->
      <o:assert test="count($copy/test/*) = 2"/>
      <!-- assert that we have copied the attributes -->
      <o:assert test="$copy/test.attribute('attr1') = 'one'"/>
    </xp:definition>
  </xp:test>
</o:function>
```

Furthermore we want to declare a test definition once and reuse it for many different datasets. Also we would like to declare datasets separately and reuse them in different tests. The format for specifying tests is fairly flexible, the above could have been expressed using a predefined test definition and datasets:

```
<!-- declare the test definition and datasets first -->
<xp:definition name="CopyTest">
  <o:variable name="copy" select="$input/test.copy()"/>
  <o:assert test="count($copy/test/*) = 2"/>
  <o:return select="$copy/test.attribute('attr1')"/>
</xp:definition>

<xp:dataset name="element">
  <test attr1="one" attr2="two">
    <one>one</one>
    <two>two</two>
  </test>
</xp:dataset>

<xp:dataset name="CopyTestResult">one</xp:dataset>

<xp:test name="CopyTest1">
  <xp:definition ref="CopyTest"/>
  <xp:input ref="element"/>
  <xp:result ref="CopyTestResult"/>
</xp:test>
```

Using both documentation and unit test extensions we now have three layers of content, with the source code itself providing the structure and context.

The o:XML test vocabulary has been designed to capture and represent tests as accurately as possible. Since they don't have to bend around the syntax rules of the language, the test declarations provide valuable information and can be reused, processed and transformed. As it turns out it is quite simple to transform the declarations into executable test suites, producing XML reports on both results and code coverage.

Figure 2.1. Test Suite Process Flow

Using XML to carry the information through the test and documentation process means that each step uses the same data format, and can be implemented with standard XML processing tools. Consequently when we want to make alterations to the process, whether to produce a different output format or to completely repurpose the data, these can easily be implemented.

As the diagram shows, the result of transforming our String type will itself be an o:XML program. The `String.length()` unit test would look like this:

Example 2.5. Generated Test: String.length()

```
<o:type name="Test">
  <o:variable name="input"/> <!-- test input vector -->
  <o:variable name="result"/> <!-- expected test result -->

  <o:function name="Test">
    <o:param name="input"/>
    <o:param name="result"/>
    <o:do/>
  </o:function>

  <o:function name="run">
    <o:do>
      <o:set ret="$this.test($input)"/>
      <o:assert test="$ret = $result"/>
    </o:do>
  </o:function>
</o:type>

<o:type name="xp:Test1">
  <o:parent name="Test"/>

  <o:function name="xp:Test1">
    <o:param name="input"/>
    <o:param name="result"/>
    <o:parent name="Test" select="Test($input, $result)"/>
    <o:do/>
  </o:function>

  <o:function name="test">
    <o:param name="input"/>
    <o:do>
      <o:return select="string($input).length()"/>
    </o:do>
  </o:function>
</o:type>
```

The `Test` type is common for all unit tests. The unit tests are controlled by a generated TestSuite that initialises all Tests with the right input and expected result values. The TestSuites also produce a report when running the tests, detailing which ones failed, and why. Coverage data is gathered by the source transformation stylesheet. From the XML test results it is easy to generate for example an HTML formatted report for browser viewing (see ObjectBox report sample below). A more sophisticated example would be a system test report in PDF format, with SVG graphs showing test coverage and compliance, that forms part of the product release docu-

mentation.

Example 2.6. Test Report: String.length()

For a more complete example of generated report output see Appendix B, *ObjectBox Test Report*.

X is for eXtreme - Applying Methodologies

More and more people are taking on board the ideas described by Kent Beck as Extreme Programming, or XP for short (see [Beck99]). A prominent feature of XP is the emphasis on unit testing, or test driven development, and the integral part it plays throughout the development cycle. According to XP, tests should be written before the actual code, to validate that whatever code there is actually works.

For this to work well, writing tests and verifying the results must be as easy as possible and not take up more time than necessary. It also helps if there is a very clear distinction between what is deployment code, and what forms part of the tests.

As part of the requirements process, most software projects (whether XP projects or not) define test cases that determine when the requirements are met. Using XML and the sort of information layering employed in the previous section, it is possible to include both the requirements as well as test cases in a format that makes acceptance test reports part of the build cycle.

It is then easy to monitor the progress of the project - what part of the system is causing problems, what part is near or far from completion. While the generated reports can't be any more accurate than the tests that produces them, it gives a different level of transparency to development work, without introducing any real overhead once the build and test procedure has been created and automated.

Another aspect of XP is collective code ownership. More than ever, here it really helps having a simple yet effective code and design documentation system. The other advantage is that irritating details like indentation, line-breaks etc become irrelevant once you're using an XML editor to write code with. If the team decides on a set of coding standards or conventions, then an XML schema can be created to enforce them, or source code can be transformed into the agreed format when submitted to the code repository.

To summarise, using XML gives the individual, as well as the project team, more room to define the development process, more tools to use and less limitations and trade-offs. By layering information nothing is lost and nothing is compromised. If you can express something well using XML then it can be seamlessly integrated with the source code.

Design by Contract

Extreme Programming is not the only development method to gain popularity recently. An alternative approach is provided by Design by Contract, as exemplified by and integrated in the Eiffel language (see [Meye91]). DbC draws on the notion of viewing the application specification as a contract - if the user or client does this, the program or system component must do that.

Just as in a common-law contract between a supplier and a client, there are two sides to a DbC contract. This is represented by function postconditions and preconditions - preconditions say what the client must do, while postconditions determines what the code must do. (This can also be seen as input/output rules, but only if you take into account the change of state that a function effects.)

Apart from pre- and postconditions, which are defined on individual type functions, DbC also lets the developer define type (or class) invariants. Invariants are a sort of global conditions, they must hold true for every type function both before and after its invocation (the exception is constructor methods, where invariants are only verified when the function exits).

The way that conditions are enforced is generally by the use of programming assertions. Typically an application developed using DbC goes through development and test cycles with all assertions operational, but is then built for production (once specification compliance has been verified) with some or all assertions disabled.

The key point in DbC lies in creating the specification. Firstly it has to cover all parts of the application, so that it can say for each software element what it is supposed to do. Secondly it must be defined in or translated to a form that can be machine-verified. Usually, from the hopefully unambiguous statements of the specification we write expressions that can be used as assertions.

As a proof of concept and for the purpose of this paper, the author has created an XML notation for expressing pre/post conditions and invariants. These components were designed to be used within an o:XML type declaration, using o:Path expressions for conditions, but could probably be easily adapted for other languages.

Example 2.7. Design By Contract: Definitions

```
<o:type ... >
 <dbc:invariant test="Expression"/> *

  <o:function ... >
    <dbc:pre test="Expression"/> *
    <dbc:post test="Expression"/> *
  </o:function>
</o:type>
```

We also want to be able to give certain meta-information about the conditions. This could be a reference to the relevant part of the specification, or simply what error message should be generated if the condition is broken. For simplicity we add an optional *error* attribute to our three DbC elements. Now on to the example - an o:XML FILO (First In Last Out) stack implementation:

Example 2.8. Design By Contract: Stack Type

```
<o:type name="Stack">
  <o:variable name="items"/>
  <o:variable name="capacity"/>

  <dbc:invariant test="count($items) &lt;= $capacity"
    error="stack overflow!"/>

  <o:function name="Stack">
    <o:param name="capacity"/>
    <o:do/>
  </o:function>

  <o:function name="pop">
    <dbc:pre test="count($items) &gt; 0" error="stack underflow!"/>
    <o:do>
      <o:return select="$items[last()].remove()"/>
    </o:do>
  </o:function>
</o:type>
```

The invariant states that the total number of nodes must never exceed our capacity. If it does then that constitues a stack overflow error - we add that information with the *error* attribute.

Another error condition we have to take into account is stack underflow, which would occur if `pop()` was invoked when the stack is already empty. That's stated by the precondition. Though we're still missing a couple of things - for one we don't have a postcondition ensuring that the stack has shrunk after pop has been called.

When writing postconditions, we are generally interested not only in the current object state, but also in the difference before and after the function call. For example, if we have a function that removes resources from a list,

we want to verify that the size of the list has diminished by one. We might also want to assert that the return value of the function call should be equal to the removed item. It would thus help if we had a way of referencing the type variables with their initial values, as well as the functions return value. We can achieve this by assigning shadow variables, with names identical to the original type variables but in a different namespace. For the return value, we assign it to the variable `$dbc:return`.

Example 2.9. Design By Contract: Stack Revised

```
<o:function name="pop">
  <dbc:post test="count($dbc:items) = count($items) + 1"/>
  <dbc:post test="$dbc:return = $dbc:items[last()]"/>
  ...
</o:function>

<o:function name="push">
  <o:param name="item"/>
  <dbc:pre test="$item" error="empty push value"/>
  <dbc:post test="count($items) = count($dbc:items) + 1"/>
  <o:do>
    <o:do select="$items.add($item)"/>
  </o:do>
</o:function>
```

Having adopted a straightforward declarative form, we will now look at how to ensure that the contract is fulfilled. This is of course the implementation aspect, and we can assume that a variety of choices exist. For example we could instruct our interpretor or execution engine so that it understands the DbC declarations and knows how to enforce them. Or we could feed the conditions to a proof system to attempt to prove the correctness of the program. The most straightforward solution however is to transform the code by adding in the required assertions. It gives us full control over which conditions are used in a certain build, and it does not require any knowledge of DbC by the interpretor or compiler (another example of separation of responsibilities).

The o:XML implementation of Design by Contract is an XSL transformation that uses the DbC declarations to process the code. With all assertions enabled, the resulting code will behave exactly like the original as long as no contract clauses are broken. If an invariant or pre/post condition is violated the program will interrupt by throwing an assertion error. To examine the transformed version of the Stack type, see Appendix C, *Design By Contract Transformation*.

Mix n' Match Extensions

In this chapter we have demonstrated how the o:XML source code format can be extended, and how using XML allows programs to be efficiently processed. The programming language no longer defines the limits of development work in the types and variety of constructs it can incorporate, instead it provides an open structure to build upon. Creating a tailored development process can be as simple as picking and choosing from a range of independently developed components, to suit the unique needs of each individual software project.

Using modern XML editors with namespace and schema support it is practically impossible to write code that is not well-formed. Syntax color-coding and folding trees means that as a developer you see exactly the part of the code that you're currently interested in. Let's have a look once more at the Stack example, this time combining all our extensions in a single type definition.

Example 2.10. Mixing Extensions: Stack Type Revisited

Chapter 3. Enabling Technology

Abstract

This chapter presents an XML vocabulary for Aspect Oriented Programming called Aspect ML. We give a brief overview of the main concepts of AOP, and how they can be encapsulated using XML. We also introduce the Reflection Tree as a model for representing reflective definitions, and explain how it can be used to select and group logical cross-cutting sections of a program. Lastly there is a discussion on the implementation of Aspect Oriented Programming using XML transformations.

Beyond the Object Structure

Aspect Oriented Programming seeks to address the inherent difficulty in applying Object Oriented abstraction mechanisms to implement functionality that doesn't follow the object hierarchy. AOP techniques can also be applied to many problems in code maintenance, such as refactoring or adapting the behaviour of groups of types and functions.

The central principle of AOP is separation of concerns, and the observation that often our concerns do not follow the structure of our programs. Instead they cut across the structure, which makes them much more difficult to implement using traditional methods. Cross-cutting concerns can be implementational in nature, like object instantiation strategies or resource management, or they can be application-specific such as managing screen updates or program output, or providing consistent behaviour across types and functions.

Generally the effect of implementing cross-cutting concerns using standard Object Oriented methods is one of code scattering, which is what happens when the same concern is implemented over and over in many different parts of the system. Sometimes types with scattered code can be refactored, but if the concerns are really cross-cutting this can be anything from difficult to impossible.

In AOP, we use `aspects` to encapsulate concerns, and apply them to logical cross-cutting sections of the application.

As an example, consider the problem of managing connections in a networked application. Say that the code has been developed and is working, but suddenly the requirements change: at no time must there be more than *N* active network connections. Refactoring time. Unfortunately the code is full of explicit and implicit network code, and the libraries used cannot themselves be refactored. To rewrite might involve removing abstraction layers, making some components network aware that otherwise need not be. The changes required are likely to be non-trivial - what happens if there are no connections available, what if some code needs priority access, how to avoid deadlocks, etc. Without very careful modelling, common implementation code will be scattered throughout. After all, there might be dozens of different library classes in use that may or may not initiate network connections.

The concern is clearly cross-cutting and can be captured by an `aspect`. Writing the implementation should be straightforward enough, the trick is to locate the points in the program that we need it to join up with. Aspect ML provides a declarative method for doing just that.

Aspect Markup Language

In Aspect Oriented Programming there are two main constructs used in an `aspect`, `advice` and `introductions`.

An `introduction`, or inter-type definition as they are sometimes called, allows us to modify types without editing the source code directly. It is an extremely effective way of adding in behaviour, or characteristics, to a whole set of types at once. An example of using introductions could be to make a set of value types implement a Printable, or Serialisable, interface.

The first thing we need is a means of expressing the information of an aspect. In the case of introductions this is fairly straigthforward - we declare the target type or types, and the code that we're introducing.

Example 3.1. Aspect ML: Introduction Definition

```
<ao:introduction name="QName"? >

  <!-- specify the target type(s) -->
  <ao:type name="RegularExpr"? parent="RegularExpr"?
           ancestor="RegularExpr"? namespace="RegularExpr"? /> +

  <ao:declaration>
    <!-- added verbatim to matching types -->
    <!-- may include parent, variable and function declarations -->
  </ao:declaration>

</ao:introduction>
```

The target types are selected by any combination of type name, parent type, and namespace (package) name. The *ancestor* attribute denotes a parent type that is not immediate, that is a parent of a parent (of a parent...). We can also specify any of these in nested elements, to pick for example types that inherit from a (or any) type in a certain namespace.

We take advantage of regular expressions to select groups or families of types, such as all types in a certain branch of the package hierarchy.

Another key abstraction mechanism in AOP is `advice`, which can be seen as code fragments inserted at logical points in the program execution. The insertion points are called join points, and are grouped together in what is known as point cuts. In the context of separating concerns, point cuts provide the means with which we single out those parts of the code that relate to a specific concern. For example we might define a point cut for all function calls that print out system status information. The functions themselves might be spread across many different types and modules, but applying the abstraction mechanisms of AOP we can treat them uniformly.

To define advice, we need a more sophisticated method of declaring join points than the simple type selection we used for introductions. The applicability of AOP is to a great extent determined by the flexibility and ease with which we can target specific groups of functionality and logical parts of the system.

By far the most commonly used join point is function calls. Firstly we want to be able to target functions declared by a certain type or family of types (such as a package). It is also important that we can be selective about function parameters.

Example 3.2. Aspect ML: Advice Definition

```
<ao:advice name="QName"? >

  <ao:cut> +
    <ao:type name="RegularExpr"? parent="RegularExpr"? /> *
    <ao:function name="RegularExpr"? > *
      <ao:param type="RegularExpr"? /> *
      <ao:params type="RegularExpr"?
        min="Number"? max="Number|unbounded"? /> ?
    </ao:function>
  </ao:cut>

  <ao:definition when="before|after|returns|throws|instead">
    <!-- code fragment -->
  </ao:definition>

</ao:advice>
```

The *when* attribute of the advice definition refers to one of the possible interception points. A function call can be intercepted right before it is executed, when it returns or when it throws an exception. The `instead` option is the same as `before`, with the difference that the actual function will only be executed when called from within the advice. There's also the option of using `after`, which means that the advice will run after the function call, regardless of whether it was terminated normally or by throwing an exception.

Named point cuts, as well as code definitions, can also be defined independently of any specific advice, and referenced by one or several advice sections:

```
<ao:cut name="QName" ... >
  ...
</ao:cut>

<ao:definition name="QName" ... >
  ...
</ao:definition>

<ao:advice ... >
  <ao:cut ref="QName" />
  <ao:definition ref="QName" />
</ao:advice>
```

Declaring simple advice definitions this way can be very useful, as it allows us to perform tasks before, after or instead of the targetted function calls. However we are limited in what we can do unless we have the ability to pass parameters from the join point to our code fragment. Usually what we want to accomplish is to join up the arguments to the function call that we're intercepting with parameters of the advice definition.

Example 3.3. Aspect ML: Advice Parameters

```
<ao:advice ... >
  <ao:cut>
    <ao:function ... >
      <ao:param name="QName" ... />
      <ao:params name="QName" ... />
    </ao:function>
    ...
  </ao:cut>

  <ao:definition ... >
    <ao:param name="QName" select="this|return|throw|QName"? /> *
    ...
  </ao:definition>
</ao:advice>
```

The *select* attribute must be either a name matching the name given to a parameter in the point cut, or one of the special names `this`, `return` or `throw`. The special names refer to the object the method was invoked on, the return value of the function and the exception that was thrown respectively. In order not to clash, these names should not be used as parameter names in point cut declarations.

What we're ultimately interested in is the aspect itself. We group advice and/or introductions in aspects to organise our code instrumentation. An aspect fills a similar role in AOP as the type, or object class, does in OOP.

Example 3.4. Aspect ML: Aspect Definition

```
<ao:aspect name="QName"? >
```

```
    <ao:cut ... /> *
    <ao:definition ... /> *
    <ao:advice ... /> *
    <ao:introduction ... /> *
  </ao:aspect>
```

We've looked at how to define aspects, advice and instructions using XML, in a way that covers most conventional applications of Aspect Oriented Programming. Now let's move on to include a more interesting use of XML in AOP - to represent type information.

The Reflection Tree

Information about types, functions and other components that make up a programs definitions can be conceptualised as a tree. If we reconstruct the data using XML, you would imagine we get something like this:

Example 3.5. Reflection Tree Type Definition

```
<type name="TypeName">
  <parent name="TypeName"/> *
  <function name="FunctionName" access="public|protected|private"> *
    <param type="TypeName"/> *
  </function>
</type>
```

Of course, this looks an awful lot like o:XML source code. It's worth noting however that it would be quite easy to construct a tree like this using Java's reflection mechanism, or the parse tree of C++ source code. (More interestingly, it would be straightforward to turn a query of the above tree into Java reflection method calls, but more about that later.)

The reflection tree gives us a simple but powerful conceptual model. What's more it's a model that can be queried and indexed using an existing structural query language - XPath. If for example you want to group all functions of `MyType` that takes at least one String parameter, you could do that using the reflection tree model and the following XPath expression:

```
/type[@name = 'MyType']/function[parameter/@type = 'String']
```

The tree would have to be constructed with the specifics of the targetted programming language. For example, in the case of Java we may want to include class fields in the reflection information. The similarities of object-oriented languages makes it possible to define a common subset, probably not too dissimilar to the above proposed definition.

Now we want to include this indexing mechanism in our aspects - we do so by extending the *function* and *type* elements:

Example 3.6. Join Points using Reflection Tree

```
<ao:cut>
  <!-- join all type functions that take parameters of type 'MyType' -->
  <ao:function path="/type/function[parameter/@type = 'MyType']"/>
```

```
</ao:cut>

<ao:cut>
  <!-- join functions of types that inherit directly from 'MyType' -->
  <ao:function path="/type/[parent/@name = 'MyType']/function"/>
</ao:cut>

<ao:introduction>
  <!-- join all types that have factory functions -->
  <ao:type  path="/type[function/@name.matches('create.*')]"/>
</ao:introduction>
```

Note how in the type path we use the o:XML regular expression function `matches()`.

Instrumentation vs Transformation

Now that we've defined a data format for aspects, we want to be able to apply them to our code. As with Design by Contract, there are a few different choices available for the way we implement this. Again we can either create an execution engine or interpretor that understands Aspect Oriented Programming, and intercepts the targetted functions and types as they are being used. This is likely to not only produce an extremely complex interpretor, but also have a serious impact on efficiency and reliability.

If we are lucky to use a programming language that provides the reflection capabilities required, and gives us the ability to make code modifications at runtime (self-modifying code), we could instruct the program to instrument itself. With the new reflection mechanism of o:XML this is a distinct possiblity.

However, since here we are interested in XML processing, we will look instead to transforming the source code itself. One advantage of this approach is that the result of the transformation is easily verifiable. In some cases we might even want to modify the code permanently, and have the transformed code replace the original.

The method we will use gives a further example of an interesting application of XML processing. We create an XSL stylesheet that takes the aspect definitions as input. The output it produces is an o:XML program, one that itself is capable of transforming source code. This is a custom-built, automatically generated code transformer that contains all the information of the aspect declarations.

Figure 3.1. Aspect Oriented Programming: Code Transformation

Chapter 4. Conclusion

XML source code: easy to extend, easy to generate, easy to transform. Transformations can be used to generate documentation, test suites, UML diagrams, platform-specific code. Can also be used for automated code transformations based on XML definitions, such as applying AOP aspects or DbC assertions.

Extensions make it easy to integrate highly specific or proprietary information with the code. This includes acceptance tests, component or unit tests, design metainformation and specialised documentation such as the relevant parts of the requirements or specification.

XML source code can be generated - test suites from test definitions, type interfaces from UML diagrams, code transformers from aspect definitions.

Using namespaces to separate extensions and source code means that there is never any conflict or compromise.

By using well-defined XML vocabularies, the information contained with the source code has increased value and can be repurposed. Test suites can be reused for new implementations. Specifications can be generated from code interfaces that use Design By Contract extensions.

Transformations and processing can harvest specific information to generate automated, up-to-date reports in almost any format.

XML opens up the software development playing field, making available an unlimited choice of programming methods and techniques.

Appendix A. API Documentation

HTML output of documentation generated from the String type source code, part of the ObjectBox source tree.

> **Note**
> Many of the String and Node functions have been excluded for brevity.

Figure A.1. String Type API Documentation

Appendix B. ObjectBox Test Report

The ObjectBox uses a simple HTML report format, which is XSLT generated from the output of the test run. The test suite itself is an o:XML program, automatically generated from the unit tests that are embedded in the the source code.

Figure B.1. Node Type Test Report

Appendix C. Design By Contract Transformation

The o:XML implementation of Design by Contract is a straightforward XSLT stylesheet that transforms the input source code by adding in assertions. The result is an modified source file that uses only standard o:XML to enforce the contract declarations.

> **Note**
> At the time of writing, the XSLT stylesheet does not take into account inherited conditions, nor does it produce entirely exception safe code. This is left as an exercise for the interested reader!

Example C.1. Design By Contract: Transformation XSLT

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:o="http://www.o-xml.org/lang/"
  xmlns:dbc="urn:DesignByContract">
  <xsl:param name="shadow" select="true()"/>

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="node()">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="o:type">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>

      <xsl:if test="$shadow">
        <xsl:apply-templates select="o:variable" mode="shadow"/>
      </xsl:if>

      <o:function name="dbc:invariants">
        <o:do>
          <xsl:apply-templates select="dbc:invariant" mode="define"/>
        </o:do>
      </o:function>

      <o:function name="dbc:exit">
        <o:do>
          <o:do select="$this.dbc:invariants()"/>
          <xsl:if test="$shadow">
            <xsl:apply-templates select="o:variable" mode="copy"/>
          </xsl:if>
        </o:do>
      </o:function>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="o:variable" mode="shadow">
    <o:variable name="dbc:{@name}"/>
  </xsl:template>

  <xsl:template match="o:variable" mode="copy">
    <o:variable name="dbc:{@name}" select="${@name}.copy()"/>
```

```
  </xsl:template>

  <xsl:template match="o:function/o:do">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:if test="parent::o:function/parent::o:type
                    [@name != current()/parent::o:function/@name]">
        <!-- type function (that is not a constructor),
             check invariants on entry -->
        <o:do select="$this.dbc:invariants()"/>
      </xsl:if>
      <xsl:apply-templates select="parent::o:function/dbc:pre" mode="define"/>
      <o:variable name="dbc:return">
        <xsl:apply-templates/>
      </o:variable>
      <xsl:apply-templates select="parent::o:function/dbc:post" mode="define"/>
      <o:do select="$this.dbc:exit()"/>
      <o:return select="$dbc:return"/>
    </xsl:copy>
  </xsl:template>

  <!-- post conditions shadow variables:
       $dbc:return = return value
       $dbc:{varname} = value before function entry
       -->
  <xsl:template match="o:return[ancestor::o:function]">
    <xsl:apply-templates select="." mode="value"/>
    <xsl:apply-templates select="ancestor::o:function/dbc:post" mode="define"/>
    <xsl:if test="ancestor::o:type">
      <!-- type function, check invariants and set shadow variables on exit -->
      <o:do select="$this.dbc:exit()"/>
    </xsl:if>
    <o:return select="$dbc:return"/>
  </xsl:template>

  <xsl:template match="o:return[@select]" mode="value">
    <o:variable name="dbc:return" select="{@select}"/>
  </xsl:template>

  <xsl:template match="o:return" mode="value">
    <o:variable name="dbc:return">
      <xsl:copy-of select="node()"/>
    </o:variable>
  </xsl:template>

  <xsl:template match="dbc:invariant|dbc:pre|dbc:post" mode="define">
    <o:assert test="{@test}">
      <xsl:if test="@error">
        <xsl:attribute name="msg">
          <xsl:value-of select="@error"/>
        </xsl:attribute>
      </xsl:if>
    </o:assert>
  </xsl:template>

</xsl:stylesheet>
```

Example C.2. Design By Contract: Stack Transformed

```
  <o:type name="Stack"
    xmlns:o="http://www.o-xml.org/lang/"
    xmlns:dbc="urn:DesignByContract">
```

```
  <o:variable name="items"/>
  <o:variable name="capacity"/>
  <o:variable name="dbc:items"/>
  <o:variable name="dbc:capacity"/>
  <o:function name="Stack">
    <o:param name="capacity"/>
    <o:do>
      <o:variable name="dbc:return"/>
      <o:do select="$this.dbc:exit()"/>
      <o:return select="$dbc:return"/>
    </o:do>
  </o:function>
  <o:function name="pop">
    <o:do>
      <o:do select="$this.dbc:invariants()"/>
      <o:assert test="count($items) &gt; 0" msg="stack underflow!"/>
      <o:variable name="dbc:return">
        <o:variable select="$items[last()].remove()" name="dbc:return"/>
        <o:assert test="count($dbc:items) = count($items) + 1"/>
        <o:assert test="$dbc:return = $dbc:items[last()]"/>
        <o:do select="$this.dbc:exit()"/>
        <o:return select="$dbc:return"/>
      </o:variable>
      <o:assert test="count($dbc:items) = count($items) + 1"/>
      <o:assert test="$dbc:return = $dbc:items[last()]"/>
      <o:do select="$this.dbc:exit()"/>
      <o:return select="$dbc:return"/>
    </o:do>
  </o:function>

  <o:function name="push">
    <o:param name="item"/>
    <o:do>
      <o:do select="$this.dbc:invariants()"/>
      <o:assert test="$item" msg="empty push value"/>
      <o:variable name="dbc:return">
        <o:do select="$items.add($item)"/>
      </o:variable>
      <o:assert test="count($items) = count($dbc:items) + 1"/>
      <o:do select="$this.dbc:exit()"/>
      <o:return select="$dbc:return"/>
    </o:do>
  </o:function>

  <o:function name="dbc:invariants">
    <o:do>
      <o:assert test="count($items) &lt;= $capacity"
        msg="stack overflow!"/>
    </o:do>
  </o:function>

  <o:function name="dbc:exit">
    <o:do>
      <o:do select="$this.dbc:invariants()"/>
      <o:variable select="$items.copy()" name="dbc:items"/>
      <o:variable select="$capacity.copy()" name="dbc:capacity"/>
    </o:do>
  </o:function>

</o:type>
```

Appendix D. Aspect Oriented Programming Implementation

Aspect Oriented Programming paradigms, preferrably expressed using an XML vocabulary such as Aspect ML, form an efficent basis for implementing aspects as source code transformations. The transformations are generally straightforward. This is especially true for introductions, where there's no need to intercept function calls or generate new types.

Below is the definition of an o:XML function that transforms o:XML source code by adding in introductions (aka inter-type declarations). It takes two parameters - the first one is the aspects in Aspect ML form, the second is the source code to transform.

For a complete AOP implementation in XML, follow the links on the o:XML web site [http://www.o-xml.org].

Example D.1. Aspect Oriented Programming: Introductions Transformation

```
<o:function name="aop:introduce">
  <o:param name="aspects"/>
  <o:param name="input"/>
  <o:do>
    <!-- handle all inter-type declarations (introductions) -->
    <o:for-each name="dec" select="$aspects//aop:introduction">
      <o:variable name="targets"/>
      <o:for-each name="join" select="$dec/aop:type">
        <o:set match="$input//o:type"/>
        <o:if test="$join/@path">
          <!-- filter out types with matching paths -->
          <o:set match="java:evaluate($join/@path, $match)"/>
        </o:if>
        <o:if test="$join/@name">
          <!-- filter out types with matching names -->
          <o:set match="$match/o:type
                        [match(@name, $join/@name)]"/>
        </o:if>
        <o:if test="$join/@parent">
          <!-- filter out types with matching parents -->
          <o:set match="$match/o:type
                        [match(o:parent/@name, $join/@parent)]"/>
        </o:if>
        <o:do select="$targets.add($match)"/>
      </o:for-each>
      <o:for-each name="type" select="$targets/o:type">
        <o:log msg="adding introduction to type: {$type/@name}"/>
        <!-- for each matching type, append declaration -->
        <o:do select="$type.append($dec/aop:declaration/node().copy())"/>
      </o:for-each>
    </o:for-each>
  </o:do>
</o:function>
```

Appendix E. o:XML Quick Reference

```
<!-- conditionals -->

<o:if test="Expr">
  ...
</o:if>

<o:choose>
  <o:when test="Expr"> +
    ...
  </o:when>
  <o:otherwise> ?
    ...
  </o:otherwise>
</o:choose>

<o:while test="Expr">
  ...
</o:while>


<!-- iterations -->

<o:for-each name="QName"? from="Number"? to="Number" step="Number"? >
  ...
</o:for-each>

<o:for-each name="QName"? in="MixedExpr" delim="MixedExpr"? >
  ...
</o:for-each>

<o:for-each name="QName"? select="Expr">
  ...
</o:for-each>


<!-- exceptions and assertions -->

<o:catch exceptions="QName (, QName)*"? handler="Expr"? >

</o:catch>

<o:throw select="Expr"/>

<o:assert test="Expr"/>


<!-- process flow -->

<o:return select="Expr"/>

<o:return>
  ...
</o:return>


<!-- result output -->

<o:eval select="Expr"/>

<o:element name="MixedExpr" namespace="URI"? />

<o:attribute name="MixedExpr" namespace="URI"? select="Expr"? />

<o:processing-instruction target="MixedExpr" select="Expr"? />
```

```
<o:comment select="Expr"? />

<o:text select="Expr"? />


<!-- log output -->

<o:log msg="MixedExpr"? level="debug|info|warning|error"? />


<!-- expression evaluation -->

<o:do select="Expr"/>


<!-- variables -->

<o:variable name="QName" select="Expr"? />

<o:set QName="Expr"+ />


<!-- procedures -->

<o:procedure name="QName">
  <o:param name="QName" select="Expr"? /> *
  <o:do>
    <!-- procedure body -->
  </o:do>
</o:procedure>

<!-- procedure calls -->

<QName QName="Expr"* />


<!-- functions -->

<o:function name="QName" access="private|protected|public"? >
  <o:param name="QName"/> *
  <o:do>
    <!-- function body -->
  </o:do>
</o:function>


<!-- constructors -->

<o:function name="QName" access="private|protected|public"? >
  <o:parent name="QName" select="Expr"/> *
  <o:param name="QName"/> *
  <o:do>
    ...
  </o:do>
</o:function>


<!-- types -->

<o:type name="QName">
  <o:parent name="QName"/> *
  <o:variable name="QName"/> *

  <o:function name="QName"> *
    ...
  </o:function>
</o:type>
```

References

[Spec02] Martin Klang. *The o:XML Programming Language*. available online [http://www.o-xml.org/spec/] . November 27, 2002.

[Prog03] Martin Klang. *Programming in o:XML*. available online [http://www.o-xml.org/docs/tutorial/programming.pdf] . July 16, 2003.

[Wals01] Norman Walsh. *Literate Programming in XML*. available online [http://nwalsh.com/docs/articles/xml2002/lp/paper.html] . October 15, 2001.

[Fowl99] Martin Fowler. *Refactoring*. ISBN: 0201485672. Addison-Wesley, June 28, 1999.

[Beck99] Kent Beck. *Extreme Programming Explained*. ISBN: 0201616416. Addison-Wesley, October 5, 1999.

[Beck02] Kent Beck. *Test Driven Development*. ISBN: 0321146530. Addison-Wesley, November 8, 2002.

[Meye91] Bertrand Meyer. *Eiffel: The Language*. ISBN: 0132479257. Prentice Hall, October 01, 1991.

[Kicz97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. *Aspect-Oriented Programming*, In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241, June 1997.