VICTORIA UNIVERSITY OF WELLINGTON
*Te Whare Wananga o te Upoko o te Ika a Maui*

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: office@mcs.vuw.ac.nz

# XML Database Support for Program Trace Visualisation

Craig Anslow

Supervisors: Robert Biddle and James Noble

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

**Abstract**

It is important for developers to understand how a reusable component works before it can be reused. This can be achieved by visualising the component in execution. To create visualisations, specific information needs to be extracted from the executing component. One method for extraction is to test drive the component and spy on it's execution, where the events can be gathered in program traces and encoded in XML based languages. This report explores storing and retrieving program traces in XDS: an XML Data Storage environment designed to support visualisations of reusable components, for the purposes of understanding. A native XML database has been used as the storage mechanism because it corresponds to the XML data model and provides XML functionality such as XML based query languages.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

Firstly I would like to thank my family, who have been really supportive throughout this very busy year. Two people who have been really positive and supportive for me have been my project supervisors Dr. Robert Biddle and Dr. James Noble. I would like to thank them for their intiative in setting up the **ELVIS** (Engineering Live Visualisations of Interactive Software) Group. I would personally like to thank the following people, who helped me with my project:

- Stuart Marshall for the work with RCD, XTE, and the paper we published.
- Mike McGavin for discussing PAL issues. Also proof reading and checking the grammar of this report.
- Kirk Jackson for working with Stuart and I to publish our paper and for raising important issues relating to this project.
- Matt Duignan for discussing different approaches for this project.
- Donald Gordon for helping me fix little bugs I had in my project, and discussing technical issues.

Also to Dan, Rilla, Pippen, Daniel, Stephen, Brenda and Angela thanks for listening to my ideas throughout the year.

One other group that I would like to thank which has helped in my project is the XML Database Group who consist of Frank, Ahmed, Amit, Charlie, Hans and led by the extraordinary lecturer, Dr. Pavle Mogin. We have had numerous presentations and all of them have been thought provoking and very insightful.

For financial funding I would like to thank the School of Mathematical and Computing Sciences for awarding me with a MCS Scholarship for 2002. Thanks must also go to Robert and James for paying my registration fees at the Australian Symposium on Information Visualisation, and the Australasian Computer Science Conferences.

Finally I would like to say a big thank you to three of my closest friends at Victoria University, Alex, Nic, and Frank. Without them, I would not have survived, or have had such an enjoyable journey.

# Chapter 1

# Introduction

*Reusing the products of the software development process is an important way to reduce software costs and to make programmers and designers more efficient. Object-oriented programming permits the reuse of design and components of programs.* Ralph Johnson and Vincent Russo[47]

Research in the software engineering discipline is constantly trying to improve its practices and productivity. One approach that other disciplines have used is to reuse products that have already been created. This is because it is easier and cheaper to recycle existing ideas rather than develop a new product from scratch each time. From a software engineering perspective, it would be better to reuse designs and components of existing software programs, as Johnson and Russo [47] state, rather than developing faster ways for writing new software. The benefit of reusing designs and software components is that it can help developers save effort in the analysis, design, implementation, testing and maintenance phases of the software development process. Saving effort may mean a reduction in time and cost for the software project. *This report addresses the role the eXtensible Markup Language (XML) can play in a repository architecture for understanding reusable code by visualisation.*

An aspect for developers reusing designs and software components is understanding how the designs and components work and how they can be reused. However, this aspect is indeed difficult in practice. Helping developers understand designs and software components means that they will potentially be able to use them in their software.

One concept for understanding how designs and components work is to use software visualisation techniques, which can show how the designs or components actually work or behave in a context. Physically viewing how a design or software component operates can help developers understand the inner details of the design or component. For this process, visualisation tools exist, but they are hard to develop and maintain, and are often not frequently used.

To visualise a design or a software component, certain information has to be selected. Extracting the correct information is a difficult procedure. There are many factors which can affect this procedure, such as the language the software component is written in, or that the design is extremely complex. One method for deriving this information is to "spy" on applications executing. This can be done in various ways such as using debuggers or modified execution environments. The spying method generates static and dynamic information about a component such as class descriptions and the methods that have been invoked on an object. The generated information can be collected in a program trace.

We are developing a Visualisation Architecture for REuse [58], VARE ("vahray"), which allows web-based visualisations of remotely executing software. VARE allows developers to "test drive" code to create program traces, transform the traces into appropriate visualisations like class and sequence diagrams, and then view the visualisations over the web.

One approach to encapsulating the information of a program trace is to encode it using eXtensible Markup Language (XML). We have initially used an existing VARE XML based language called Process Abstraction Language (PAL)[59] and have improved it since. We have also recently developed a new program trace language called Resuable Component Descriptions (RCD) and eXtensible Trace Executions (XTE) which is more efficient and has more functionality than PAL. The existing problem with the XML based program traces is that they can be very large in size such as 1-100MB for even small traces. The current limitation with VARE is that it has no formal approach to manage these large generated program traces.

In this report, **XDS, An XML Data Storage environment**, is presented for storing XML based program traces of reusable components, and for retrieving the program traces so that they can be transformed into appropriate visualisations that can help developers understand how they work, and whether or not they can be reused in a new software program.

One issue that is identified for the XDS environment is what type of database should be used to store the program traces and should they be retrieved. Two approaches that are investigated are using a relational database, and using a native XML database. The advantages of each of these approaches is discussed later in more detail.

This report is structured as follows:

Chapter 2, **Background**, introduces the important aspects to code reuse and software visualisation. It also presents two different software visualisation research systems that deal specifically with visualising program traces.

Chapter 3, **An Architecture for Visualising Reusable Components**, describes why we want to visualise reusable components. The Program Mapping Visualisation (PMV) model is then introduced, followed by VARE which is based on the PMV model. The different types of information for visualisation are then examined with the format requirements of a program trace.

Chapter 4, **The eXtensible Markup Language and VARE**, explains different XML technologies that can be used in VARE. It first looks at what XML is and what the benefits in using XML are. The different types of XML documents are then defined, followed by a comparison of Document Type Definitions (DTD) and XML Schema. The different ways to store XML are then examined and is followed by a comparison of the two defacto XML query language standards, XPath and XQuery. A summary of XML parsers, Simple API for XML (SAX) and Document Object Model (DOM), is then presented.

Next we describe Scalable Vector Graphics (SVG) which is one approach that we have used to create visualisations in VARE. Since VARE operates over the web it is important to look at the different types of XML services such as the Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery and Integration. Native XML databases are then defined, followed by the important features that most support, and then a summary is presented of three mature native XML databases.

Chapter 5, **XML Program Trace Languages**, introduces three XML based program trace languages which include Process Abstraction Language (PAL), Reusable Components Description (RCD) and eXtensible Trace Executions (XTE). It also looks at the benefits of each approach by evaluating it against the requirements of a program trace.

Chapter 6, **XDS: An XML Data Storage Environment**, describes a relational database approach for storing XML based program traces, and why it is not the preferred option. The better option, using a native XML database, is then presented. The architecture and the user interface design are explained, and the XDS working environment is shown in action. It then illustrates how XDS has been integrated into the VARE architecture.

Finally, chapter 7, **Conclusion**, concludes this report and outlines the contributions made. It also includes a section on future work, presenting a number of possible features that could be incorporated into the XDS environment.

# Chapter 2

# Background

This chapter introduces the fundamental concepts that are associated with this report: code reuse and software visualisation. We then survey a selection of tools related to the storage of program traces and visualisations. Finally, we summarise these tools and contrast them with our approach.

## 2.1 Code Reuse

> *One of the reasons that object-oriented programming is becoming more popular is that software reuse is becoming important. Developing new systems is expensive, and maintaining them is more expensive . . . Clearly, one way to reuse a program is to enhance it, so maintainenance is a special case of software reuse. Both require programmers to understand and modify software written by others. Both are difficult* Ralph Johnson and Brian Foote[46].

The main reasons for wanting to reuse code are to save on time, effort and costs in both development and maintenance of quality software. This will mean the developer will not have to implement a new solution to an old problem. Instead they can recycle existing code to solve their problem. Research into code reuse has been happening since about the middle of the 20th century [60] and includes many areas such as examining how to reuse code [47, 46, 62], what makes code reusable [63], and metrics to measure code reuse [18, 30, 27].

There are many ways code reuse can be applied. For example copying and pasting code into a new program, inheritance of classes, instantiation of common methods within programs, implementing design patterns into a program [31], using a framework, using existing code from a software library, and using an application programming interface (API). When reusing code it may need to be modified or extended in some way so that it will meet the requirements of the new context. The assumption is that even modifying or extending code will result in the reduction of time, cost and effort compared with designing the code from scratch.

A key benefit from reusing code is that when modifications, bug fixes or updates occur, the developer can save time by incorporating them into their software. Problems then don't have to be solved for every instance. This can happen on a global scale and examples include online updates of both proprietary and open source software.

The reason why *we* are interested in **understanding** reusable code is so that developers can reuse it in their new programs. Currently several techniques exist to understand how software works. They include documentation, experience and visualisation. Documentation is sometimes provided with software, but is often difficult to use, read and understand. Documentation can come in various forms including online form or in written form. Experimenting with the reusable code means the developer will gain practical experience and learn how a component works. Even after practicing with the code in their new program, the developer may still not have a mental picture of how the code actually works. Visualising a component either as a static image or a dynamic image can show a developer how the component has been designed and how it works when executed.

## 2.2 Software Visualisation

Many software visualisation methods exist for a variety of reasons. Reasons include visualising software for teaching students how algorithms work and how language constructs work together, helping programmers understand their code better, using debuggers to correct bugs in software, profiling large applications to determine efficiency, correctness and help during maintenance, and in research from algorithm animation to pretty-printing source code. However, we are interested in visualising reusable code for the purposes of **understanding**.

Software visualisations can be separated into two sets: static and dynamic visualisations. *Static visualisations* can be created from investigating the source or binary files, which can contain class descriptions along with their methods and variables, inheritance hierarchies between classes, and dependency hierarchies amongst classes.

*Dynamic visualisations* can be composed by examining or spying on the software during execution and gathering the events in a program trace. The types of information that can be gathered include object creation and deletion, method calls and returns, field accesses and modifications, exceptions, and multi-threading issues.

There are some general visualisation tools that can present static information as Unified Model Language (UML) class diagrams [54], while dynamic information can be presented in UML sequence diagrams [48].

## 2.3 Related Work

This section looks at two representative software visualisation systems BLOOM[85] and Jinsight[40], and how their developers have approached the use of database technologies. Other systems include the Desert Environment[83], Visor++[69], and Tarantula[24].

### 2.3.1 BLOOM and Bee/Hive

BLOOM[85] is a system for doing software understanding through visualisation and was developed by Steve Reiss at Brown University. It has been designed to collect both static and dynamic information, and then to analyse this data.

BLOOM's program traces are currently stored in two files. One file contains the trace data in a compressed binary format. This file consists of a series of records indicating the following types of events entry and exiting a method, the amount of memory allocated to an object, the amount of time an object takes to execute, the amount of time an object waits for a resource, when an object is created and deleted, and when memory is freed. The other file contains records describing the classes, methods, and objects accessed in the trace. The second file can be stored in a mappable form consisting of hash tables for easy and quick access. It can be mapped into memory and doesn't have to be read or processed, however these files can get rather large, up to 200MB. XML files are used to store the analyses of the trace data.

BLOOM has a visual query language for specifying what information should be visualised. The query language can read either the trace data in XML, or the symbolic hash table trace data file. The latter option, memory mapped, is preferred as it is faster. The architecture of the back end to BLOOM, which is Bee/Hive[84] can be seen in figure 2.1.

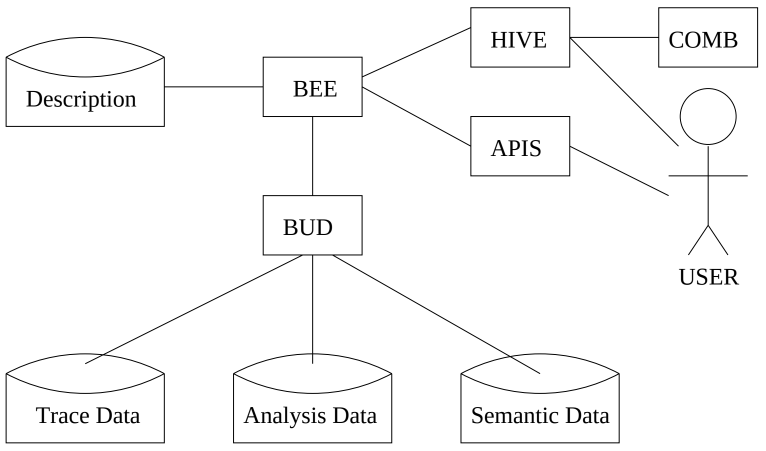


Figure 2.1: Bee/Hive, the back end visualisation architecture of BLOOM.

HIVE is a generic framework that supports 2D and 3D visualisations, using a proprietary Java format. It has two interfaces. One interface, used by BEE, is for letting a client program create and use visualisations. The other interface is designed to support a variety of different plug-in visualisations. The plug-in visualisations are encoded in the COMB package.

BUD is used as a data manager or a common interface for BEE, to access multiple data sources for creating visualisations. Queries are created using Object Query Language (OQL) and Structured Query Language (SQL) to access data. In the trace data, source names and related information is accessed by using IDs. Analysis results are stored as XML files. BUD also allows different data sources to be related and creates virtual objects by these relationships.

APIS is the user interface, and has the following four control panels:

1. **Visualisation control panel:** lets the user set properties of the visualisation and certain data fields can be selected, to be associated with a visualisation.

2. **Restriction panel:** lets the user decide what data objects should be displayed.

3. **Interaction control panel:** supports several visualisations at once.

4. **Grouping control panel:** lets the user dynamically select groups of data objects such as all the methods from a class.

BEE controls the visualisation process by reading in a description file generated by the front end which describes both the data that is needed for the visualisation and the type of visualisation to be used to display the data. BEE uses APIS to ask the user what program and trace files they want to view, and sets up BUD to access and combine the required data sources. The HIVE engine is used to set up the visualisation window and a visualisation method is added in from the COMB package. An APIS interface is then displayed to the user from the data sources and the selected visualisation.

### 2.3.2 Jinsight

Jinsight[40] is a tool for visualising and analysing the execution of Java programs. It is useful for performance analysis, memory leak diagnosis, debugging, or any task in which a user needs to better understand what a Java program is really doing.

Jinsight brings together a range of techniques that lets a user explore many aspects of a program such as:

- **Visualisations:** Let a user understand object usage and garbage collection, and the sequence of activity in each thread, from an object-oriented aspect.
- **Patterns:** Pattern visualisations extract the essential structure in repetitive calling sequences and complex data structures, letting a user analyse large amounts of information in a concise form.
- **Information Exploration:** A user can specify filtering criteria to focus on, or drill down from one view to another to explore details. The user can also create units that precisely match interesting features and then use them as an additional dimension in many of the views.
- **Measurement:** A user can measure the execution activity or memory, summarised at any level of detail along call paths, and along two dimensions simultaneously.
- **Memory Leak Diagnosis:** Special features are provided to help a user diagnose memory leaks.

The main aim of Jinsight is to help a user better understand, tune, and debug a program. It also provides instrumentation for making trace data. Jinsight captures the trace data in a proprietary format, and saves it to a file or sends it over a socket to a visualiser for live analysis. The visualiser works with an in-memory model that it constructs from the trace data using Java collections. It does not use any formal relational or object-oriented database. The visualiser, however has mechanisms that give it some database-like functionality. For example, the ability to formally access attributes of each Java object representing a trace element, and a query mechanism backing the "execution slices". For rendering visualisations, they again have a proprietary format to create the views, using the Java AWT and Swing packages [93].

Figure 2.2 shows different views of Jinsight working. It includes the following views:

- **Call Tree View:** Summarises, in an outline form, the call paths from or to a given set of method invocations.
- **Execution View:** Shows the program execution overview and details of communication among objects per thread as a function of time.
- **Invocation Table View:** Investigates a particular method more closely, and shows each occurrence of that method.

- **Execution Pattern View:** Browses recurring calling patterns arising from a given set of method invocations.
- **Reference Pattern View:** Shows patterns of references in varying detail to or from a set of objects. It is useful for studying data structures and finding memory leaks.
- **Method Histogram View:** Shows methods grouped by class, and indicates their level of activity.

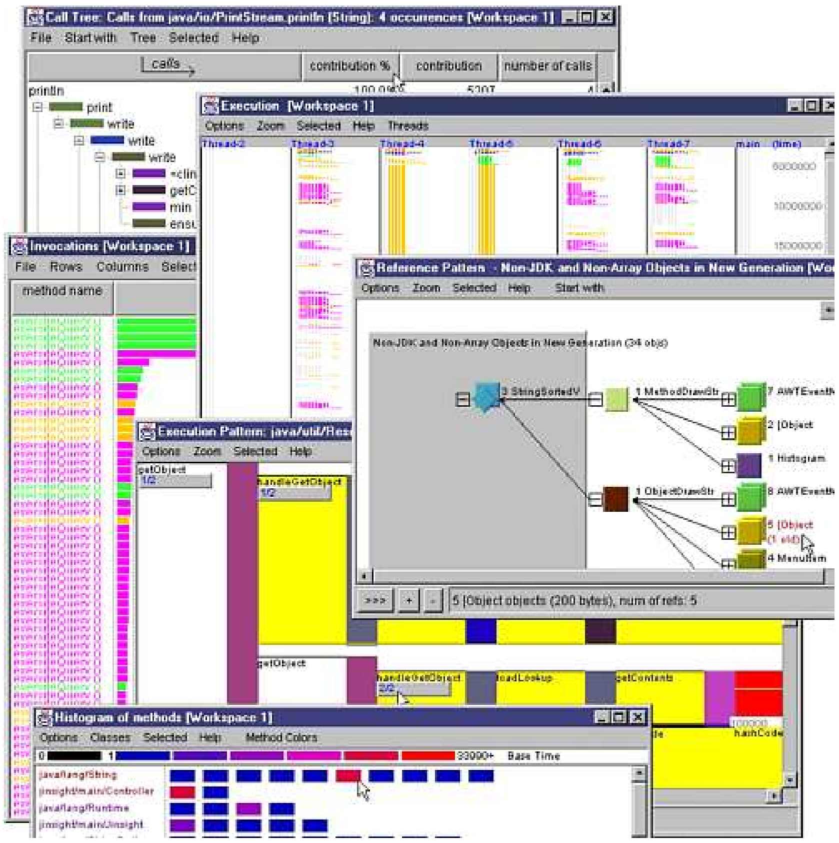

Call Tree: Calls from java/io/PrintStream.println (String): 4 occurrences [Workspace 1]
Execution [Workspace 1]
Invocations [Workspace 1]
Reference Pattern - Non-JDK and Non-Array Objects in New Generation
Execution Pattern: java/util/Res
Histogram of methods [Workspace 1]

Figure 2.2: Different views of the IBM Jinsight project.

## 2.3.3 Discussion

To summarise, BLOOM generates two files for the program traces where they are stored in relational databases. To create visualisations, a visual query language is used and is based on SQL. The visualisations are 2D and 3D and use a propriatrary Java format. Jinsight uses a proprietary format to encode the program traces and stores them in a filesystem. The visualisations also use a proprietary Java format and can't be stored.

As we will discuss later in this report, we encode our program traces using XML based languages, and store them in a native XML database. This can act as a web based code repository. To query the database over the web we use XQuery and the Simple Object Access Protocol (SOAP). Currently we are using the open Scalable Vector Graphics (SVG) standard to create animated visualisations [23].

# Chapter 3

# An Architecture for Visualising Reusable Components

In this chapter we present our architecture for visualising reusable components. We first explain our motivation behind wanting to visualise reusable components. Then we present a conceptual model behind our architecture. The architecture is then explained along with some descriptions of existing tools for creating visualisations. Next we look at the types of information that could be present in a visualisation, and finally discuss some requirements for program traces.

## 3.1 Why Visualise Reusable Components?

The aim of visualising reusable components, or existing code that has previously been used in another design, is to allow a developer to determine if and how a given code component can be reused in the developer's new context. Although software visualisation is about understanding software there are some further ideas that are of interest to a code reuser.

Software visualisers and code reusers are interested in what a component does, however a software visualiser may not be interested in reusing that component. The coder reuser is especially interested in how *to use* a component. For this, it may be important to see the order that public methods are invoked in. For example, the specific order of methods that are required to connect to a database could be visualised. Code reusers may not always have access to the underlying source code of a component, however they can still reuse a component as long as they have access to a compiled version for the type of architecture that the code is executing on.

Using the component in a new context may mean that it needs to interact with other components in a different environment than was originally intended. Therefore it would be useful to visualise the external influences of a component. It is also important to understand whether a component matches its specification, so it should be compared against the requirements of the new context and the requirements of the solution.

Code reusers are interested in class or package hierarchies, as well as the methods invoked to access the component's public interface. They are also interested in seeing that the right method gets invoked, and that the correct number of invocations occur. This would help with modification of a class by extending it with inheritance or overloading of methods.

Efficient resource usage may also be an important aspect to visualise, to see that a component is not using too much of the processors time. The following are the main points that a code reuser would be interested in to see if a component is worthwhile for reuse [57]:

- What the component **does**.
- How the component **works**.
- How the component **can** be reused, and whether it needs to be modified.

## 3.2 The PMV Model

The Program, Mapping and Visualisation (PMV) Model is a conceptual model for describing program visualisation systems. It was developed by Stasko [96, 97] and Roman and Cox [92]. It has three separate elements: the program component, the mapping component and the visualisation component. This is demonstrated in figure 3.1.

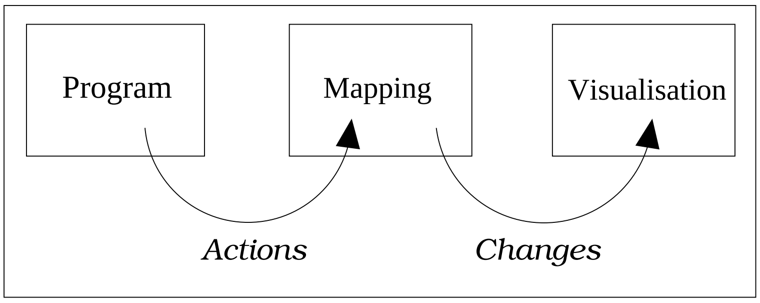


Figure 3.1: The PMV Model.

The role of the *program component* is to collect information from the executing code. This can be done in many different ways. One way is to extract important information that is necessary to make a visualisation from the source code of a program such as class descriptions, methods and fields. Another way is to have the runtime environment interrogate or capture the actions of the program while it is executing. This can be done using debuggers such as GDB (The GNU Debugger)[64, 59] or JDI (The Java Debugger Interface)[55]. The main responsibility of the program component is the detection of runtime events. Once detected these events are passed to the mapping component.

The *mapping component* may receive some information from the program component that will not be required for a visualisation. The mapping component transforms or changes this information into an appropriate format and gives this information to the visualisation component.

The *visualisation component* receives information from the mapping component relating to appropriate events for a visualisation in the runtime execution of code. The information which has been correctly processed into a format for the visualisation component can then be turned into specific visualisations. These visualisations can contain either static or dynamic information, such as class diagrams or sequence diagrams, and can display different levels of abstraction.

## 3.3 VARE: Visualisation Architecture for REuse

*VARE* is an architecture for generating visualisations in programs in a distributed environment and is based on the PMV Model. The design supports multiple programming languages and provides user control for the different parts in the visualisation process. VARE is being designed as part of an overall project by members of our research group, Engineering Live Visualisations of Interactive Software (ELVIS), [23, 57, 59, 58].

### 3.3.1 Architecture

The VARE architecture is summarised below because going into greater detail is beyond the scope of this report, however more information can be found elsewhere [58].

VARE is a client-server architecture as seen in figure 3.2. The server contains repositories and processes. Dashed lines represent test drive traces or visualisation input or output, and solid lines represent control, queries or responses. On the client side, the user manages the activities associated with creating and viewing a visualisation. The component repository interface lets the user select a component from the repository to create a component set. Once this is created, the user can select an engine type from the engine repository to control the test driving of these components. The engine component is synonymous to the program component in the PMV model.

The engine generates a test drive trace as output, which is stored in the test drive trace repository. A test drive trace is then used as input to a transformer, which is synonymous to the mapping component from the PMV model. The transformer then transforms the test drive trace into an appropriate form for a visualisation. The transformer repository interface lets the user select the transformer to use and the test drive trace to use with it.

Finally the finished visualisation is stored in the visualisation repository. The visualisation interface lets a user choose a particular visualisation and control its presentation.

The two important parts of the VARE architecture that are related to this project are the test drive traces repository and the visualisation repository, which are now elaborated on.

#### Test Drive Trace Repository

The data that needs to be stored in the *Test Drive Trace Repository* are test drive traces created by an engine that outputs the PAL trace data, described in section 5.1. A prototype engine called AT (Abstraction Tool) was created by Mike McGavin for his honours project [59]. The test drive traces can be used later as the basis for the creation of a visualisation. The test drive traces contain all the information pertaining to visualisations such as the order of object creation, method invocations, field accesses and field modifications. Meta information such as test drive trace identifiers is also stored [58].

#### Visualisation Repository

The data that needs to be stored in the *Visualisation Repository* are the visualisations created by the transformer, such as the transformer program, Blur, created by Matthew Duignan for his masters thesis [23]. The visualisations contain information such as a description of the components they are associated with, who created them, and notes that help the

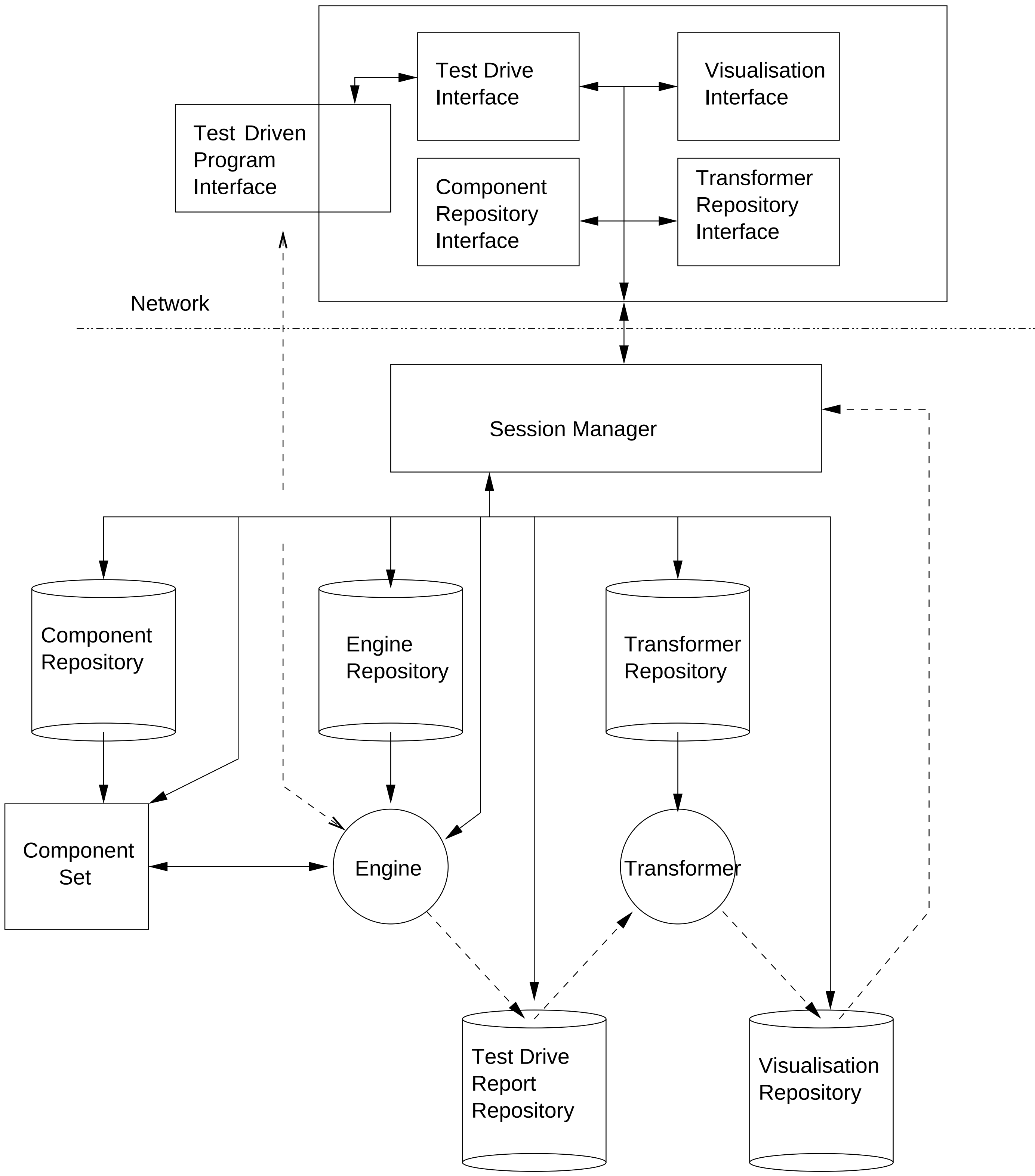


Figure 3.2: The VARE architecture.

understanding of the visualisation. The visualisation data can also be edited once created [58].

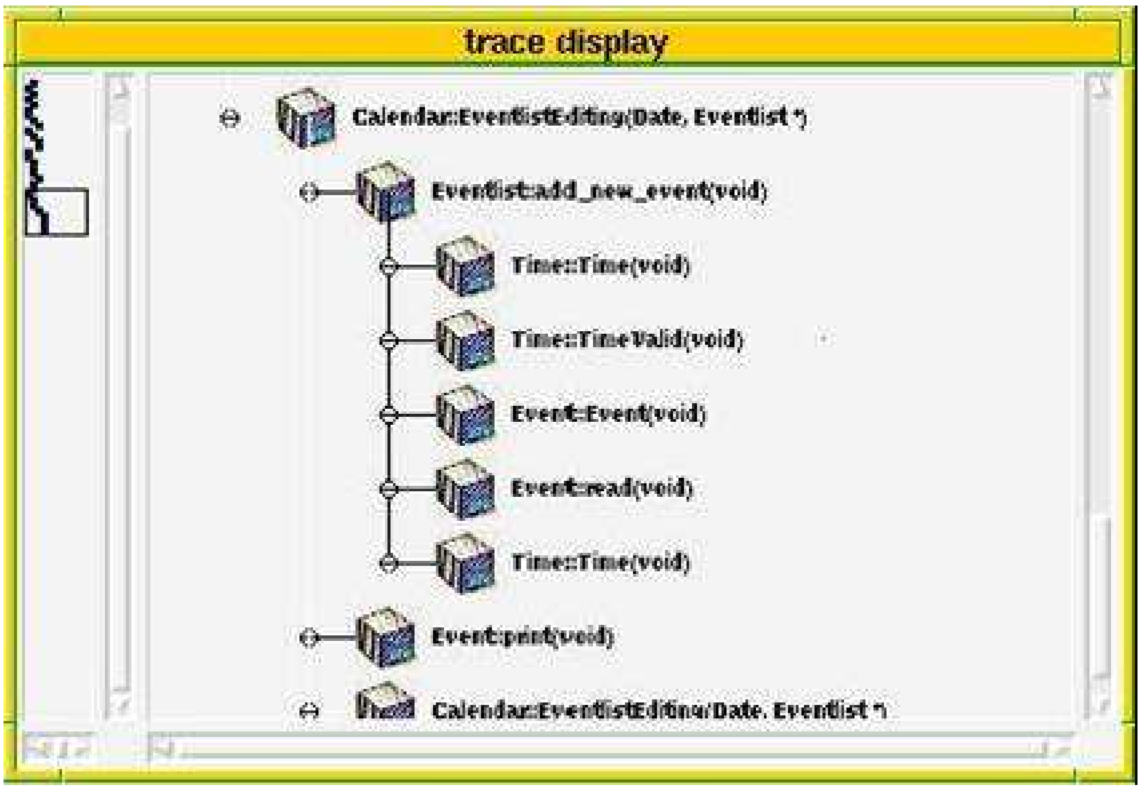
### 3.3.2 Visualisation Tools

There are pre-existing software visualisation tools displaying executions of a program which have been built as preparatory studies to VARE. Several of them will be described in the following sections.

#### Dy-re

Dy-re (Dynamic Re-use) was developed by John Miller Williams for his masters thesis [64]. The project looked at dynamic visualisation of processes, displaying message calls between different objects and allowing for a user to view the program stack to see which methods were calling other methods.

Figure 3.3 is the Dy-re trace display which lets a user see what methods have called other methods with the process as it is executed. The tree in the figure can be expanded or collapsed to view or hide method call information.



```
Calendar:EventlistEditing(Date, Eventlist *)
    Eventlist:add_new_event(void)
        Time::Time(void)
        Time::TimeValid(void)
        Event:Event(void)
        Event:read(void)
        Time::Time(void)
    Event:print(void)
    Calendar:EventlistEditing(Date, Eventlist *)
```

Figure 3.3: Dy-re showing the trace display of method calls.

#### Fire

Fire (Framework Interaction for REuse) was developed by Kirk Jackson for his honours project [43]. The aim of Fire is to visualise frameworks to help people learn how they work.

It can display multiple diagrams of programs in a tiled view. Dynamic visualisations can be generated from the execution of the framework interacting and they can be viewed in real time by a user.

Figure 3.4 is a class instance diagram generated by Fire. When the program is executing arrows are drawn between classes showing which methods are executing. It shows the method, the caller and callee with the most recent method call being highlighted in blue. It is also possible to move backwards and forwards during the execution to see what events happened and when they happened.

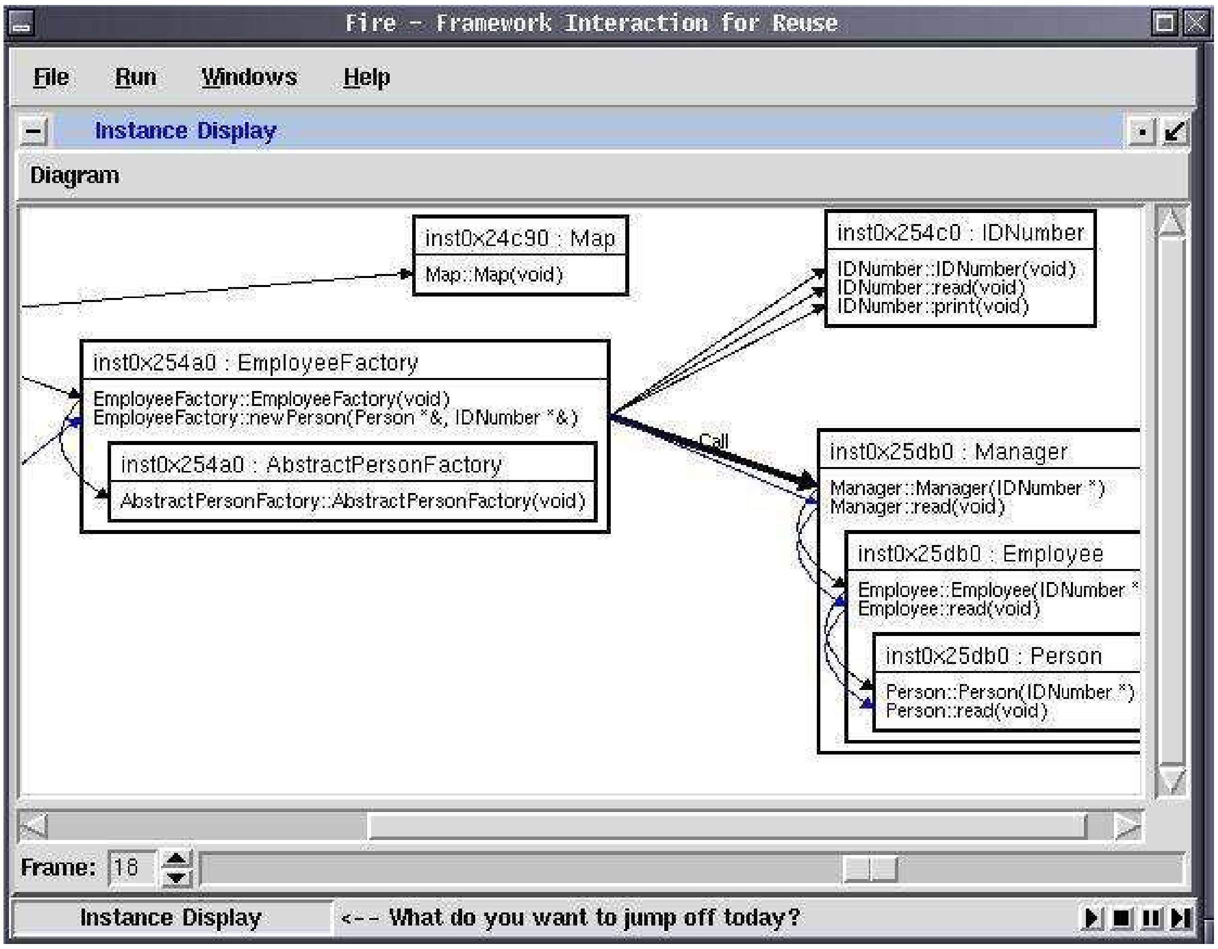

Fire - Framework Interaction for Reuse
File
Run
Windows
Help
Instance Display
Diagram
inst0x24c90 : Map
Map::Map(void)
inst0x254c0 : IDNumber
IDNumber::IDNumber(void)
IDNumber::read(void)
IDNumber::print(void)
inst0x254a0 : EmployeeFactory
EmployeeFactory::EmployeeFactory(void)
EmployeeFactory::newPerson(Person *&, IDNumber *&)
inst0x254a0 : AbstractPersonFactory
AbstractPersonFactory::AbstractPersonFactory(void)
Call
inst0x25db0 : Manager
Manager::Manager(IDNumber *)
Manager::read(void)
inst0x25db0 : Employee
Employee::Employee(IDNumber *
Employee::read(void)
inst0x25db0 : Person
Person::Person(IDNumber *)
Person::read(void)
Frame: 18
Instance Display
<-- What do you want to jump off today?

Figure 3.4: Fire displays a dynamic class instance diagram of a program executing.

### Dyno

Dyno was developed by Stuart Marshall for his masters thesis [55]. Dyno is designed to visualise Java classes by test driving certain components of an application. A user can invoke methods on a class instance and view what happens using a series of different visualisations, such as sequence diagrams as demonstrated in figure 3.5.

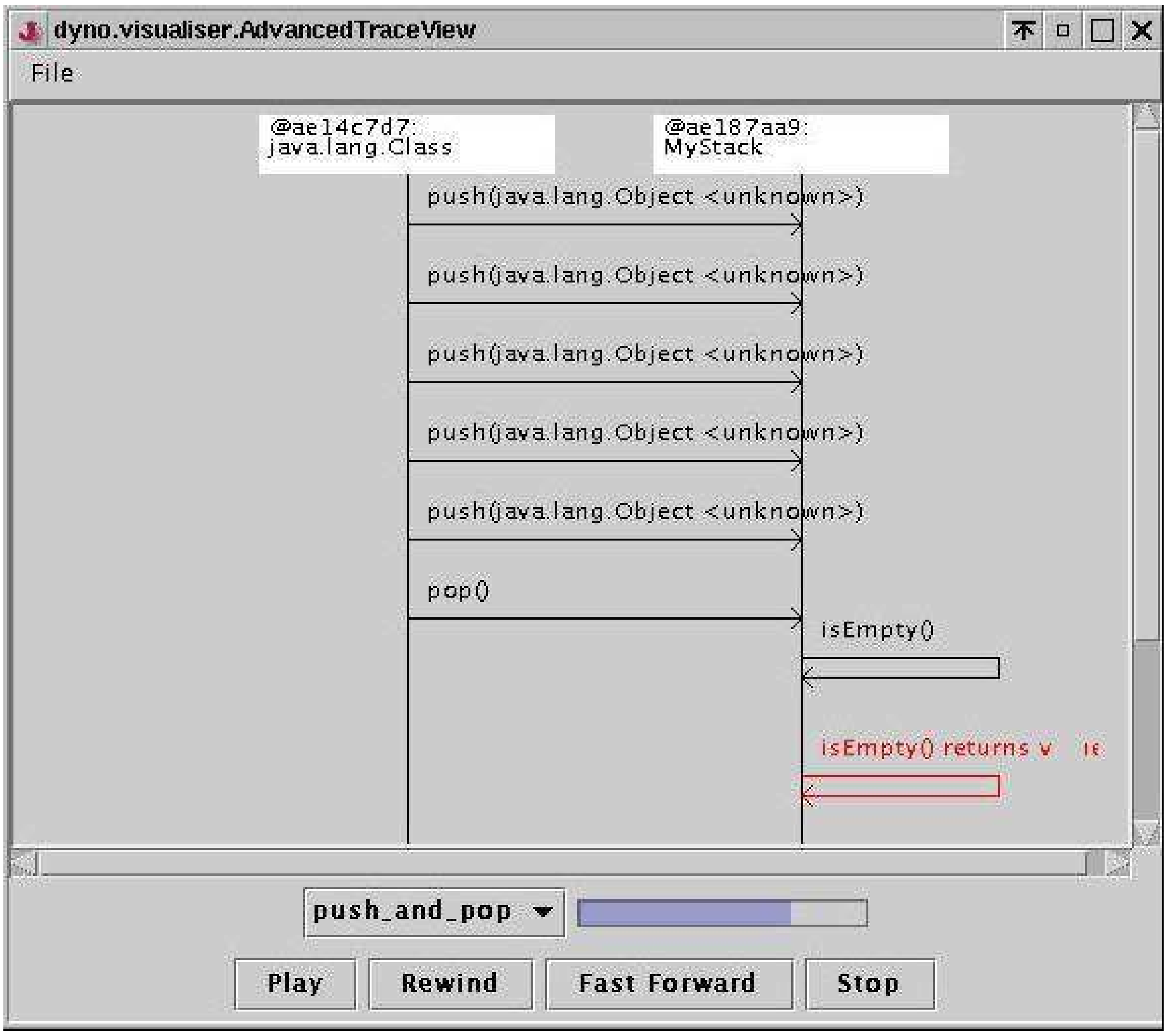

dyno.visualiser.AdvancedTraceView
File
@ae14c7d7:
java.lang.Class
@ae187aa9:
MyStack
push(java.lang.Object <unknown>)
push(java.lang.Object <unknown>)
push(java.lang.Object <unknown>)
push(java.lang.Object <unknown>)
push(java.lang.Object <unknown>)
pop()
isEmpty()
isEmpty() returns v ie
push_and_pop
Play
Rewind
Fast Forward
Stop

Figure 3.5: Dyno displays a sequence diagram of method calls.

## 3.4 Information That Could Be Visualised

Different types of information are required to a visualise a reusable component, but not all the information is needed for any one specific visualisation. On the hand hand all the information should be available if required.

The types of information that can be visualised have been divided into three parts. These parts are the main points that a code reuser would be interested in, deciding whether or not a component is appropriate for reuse. These parts are what a component **does**, how a component **works**, and how a component **can** be reused [57]. For all of these sections it would be good to see both static and dynamic visualisations as this shows the component before execution and how it behaves during execution.

### 3.4.1 What is a Component's Functionality

When investigating what a component does and to see if it can be reused in the new context, it is important to look at the external side-effects and the results that occur as a consequence of interacting with the component's public interface. The component should be treated as a black-box. The following elements are important in understanding what a component does:

- **Documentation:** Authors and users who have designed and used the component can provide documentation, such as README files, which can help describe what a component does. Often they are hard to understand. Documentation could instead be provided by program visualisation, where animated diagrams show pictorially how the code works.
- **Results and Side-Effects:** The result of executing a component will determine what the component does and whether or not a component can be reused. The side-effects may help to show whether a component can work together with other components without compromising their functionality.
- **Data Sent or Requested, External to the Component:** Any information sent or requested by a component to such entities as a network, filesystem or database, needs to be handled in the new context. It is important to understand the requirements and actions of a component with regard to the external environment.
- **User Input and Output Required During Execution:** It is important to know what information is required of a user, what information is given to a user, what the method of interaction is, and what the environment is like. For example graphical user interface (GUI) or command line usage is important.

### 3.4.2 The Behaviour of a Component

Understanding how a component works by looking at the internals of a component, treating it as a white-box, may open up opportunities for modifying its behaviour to what is required by replacing sub-components, extending components or overloading methods. The following internal information could be useful:

- **Documentation:** Having the authors and users describe how an algorithm works would be useful and it could be used in a visualisation.

- **System Permissions:** The system permissions required by a component affect its appropriateness for reuse in a given situation. For example, perhaps only a super user can execute a component.
- **Other Software Applications and Libraries:** If using other software or libraries, visualising these would help to see if it is worthwhile for reuse. It would identify the specifics of what software is required, why and where.
- **Hardware Resource Usage:** The performance of a component with regard to hardware resource usage may mean that it is expensive to reuse in a new context, because it uses a lot of the processor's time.
- **Execution Traces:** Tracing the execution of a component involves catching information such as method calls, method returns, field accesses, field modifications, object creation and object deletion. This can show potential consequences and alternative executions that can be created by overloading or replacing certain parts of a component.
- **Multi-Threading and Synchronisation:** Information gained at runtime from viewing threads and synchronisation can be useful. It can show the sequencing of events and the amount of time spent holding or waiting on a resource or detecting that a deadlock happened. This might show whether or not that it can efficiently work with other components in a new context.
- **Time Length of Execution:** With visualisation, the amount of time a component takes to execute can be measured to see if it meets specific requirements, such as performance. This can be compared with other potentially reusable components that fulfill the same functionality.

### 3.4.3 Integration of a Component for Reuse

Understanding how a component can be reused or modified in a context that it may not have been designed for is important to developers. Such information that could be important in understanding how to reuse a component includes:

- **Uses of a Component Through its Public Interface:** Demonstrate how to link in a component to other code in a new context.
- **Extensions of a Component Through Inheritance and Overriding Methods:** Inheritance could extend certain classes within the reusable component. For example, to reuse what is already been created and add extra functionality that is required. This can also be applied to methods.
- **Details of How to Install Other Software Required by the Component:** Information on how to go about installation of a component and other software that is required for the component to operate would help to reduce the time and effort required in the reuse process. A README file would be helpful in this situation.

## 3.5 Program Trace Format Requirements

The previous section described the type of information that was important for a visualisation. This section is concerned with the format of the information that is generated from the engine and transfered to the transformer in the VARE architecture, so that it can be converted into a useful visualisation. The issues that are of concern are as follows:

- **Storable and Re-playable:** Program traces and visualisations are expensive to generate, so it would be useful to store them in a database so that they could be replayed in the future to create different visualisations.
- **Support Live Streaming to Visualisation Tools:** The format of the execution trace should support the ability for live streaming into a visualisation tool for uses like real time visualisations.
- **Easily Transportable Over the Internet:** This imposes certain restrictions on the format of the execution trace. A format for the execution trace should be able to be transferred over the Internet and preferably in a neutral format such as American Standard Code for Information Interchange (ASCII) text or unicode.
- **Filterable For Extracting Relevant Information:** There could potentially be lots of data that a program could produce in a program trace. The format should support an easy method of filtering so that only relevant information is needed for a visualisation.
- **Able to be Queried:** The execution trace information should be easily queried, so that subsets of information can be used to make a visualisation.
- **Programming Language Independence:** The execution trace format should be programming language independent. Supporting multiple programming languages will allow visualisations to be reused.
- **Platform Independence:** The execution trace information viewed on different machines using different operating systems should be identical. Data transfer should be done in a platform neutral way.
- **Scalable to Large Components and Visualisations:** Execution traces can quickly get large if there are a lot of method calls, objects created or other events. The format chosen will have to scale when it is filtered and queried over large files, on the order of hundreds of megabytes.

There are a number of different requirements that program traces should support, and we have decided on exploring the eXtensible Markup Language (XML) to encode our program traces. This is because XML is an open standard and there are many applications that are based on XML. These include native XML databases and XML query languages. The next chapter goes into more detail why we think that using XML to encode the program traces is an appropriate choice.

# Chapter 4

# The eXtensible Markup Language and VARE

This chapter describes some of the different XML technologies that are used in the VARE architecture and that we consider for use in the XDS: XML Data Storage Environment that we are designing.

## 4.1 What is XML?

eXtensible Markup Language (XML) is a markup language that is used to describe the structure or meaning of a document whereas HTML describes how a document should be displayed. XML is an open and flexible standard created by the World Wide Web Consortium (W3C)[15] in early 1998 for storing, publishing and exchanging any kind of information.

XML allows the user to define new collections of tags that can be used to structure any type of data or document, whereas HTML has a fixed set of tags. XML is an important bridge between a document-oriented view of data implicit in HTML and a schema-oriented view of data that is central to a DBMS. XML has the potential to make database systems more tightly integrated into Web applications.

XML emerged from the Standardised Generalised Markup Language (SGML, established in 1988) and HTML. SGML is a meta-language that allows the definition of data and document interchange languages such as HTML. SGML is very complex and requires sophisticated programs to achieve its full potential, whereas XML was developed to have similar functionality as SGML but remain simple.

The 10 design goals of XML[15] are:

1. *XML shall be straightforwardly usable over the Internet.* The intention is that it should be easily be able to be read much like an HTML document, but only some browsers support this capability.

2. *XML shall support a wide variety of applications.* This includes applications such as web browsers, authoring tools, content analysis and image creation.

3. *XML should be compatible with SGML.* This means that XML should be a subset of the functionality provided by SGML.

4. *It should be easy to write programs that process XML documents.* It should not take too long to write the program.
5. *The number of optional features in XML is to be kept to the absolute minimum, ideally zero.* Optional features may cause compatibility problems between applications.
6. *XML documents should be human-legible and reasonably clear.* The idea here is for a human to look at the XML document and understand what the content means.
7. *The XML design should be prepared quickly.* It should be easy to design the structure of the XML standard.
8. *The design of XML shall be formal and concise.* XML should be easy for machines and humans to understand.
9. *XML documents shall be easy to create.* Using a simple text editor should be sufficient to create an XML document.
10. *Terseness in XML markup is of minimal importance.* There should be no shortcuts because if there were, then it would increase the complexity in understanding the content in the XML document.

### 4.1.1 Basic Syntax

XML is a markup language which is a way for describing a document by placing custom-built tags in the document while the tags in HTML conform to standard tags in the HTML specification. Figure 4.1 is an example of a simple XML document. It contains the XML version tag which represents the version of the XML standard that the document adheres to, in this case it is version 1.0. After that there is a root `<students>` element which has nested (child) `<student>` elements and each `<student>` element has a `<name>`, `<stuid>`, `<phone>` and a `<degree>`. Each element has a closing tag which is the same as the opening tag except it is preceded by a forward slash, e.g `</students>`. All XML elements must be properly nested in that they all have a opening and closing tag. XML is a case sensitive language and all its keywords are made up of lowercase characters, i.e. the `<student>` element is different from `<STUDENT>`. Notice that the `<phone>` elements are different for each `<student>` element. The first one has straight text or parsed character data (`#PCDATA`) while the second one has an attribute (work) which is inside the `<phone>` element, e.g. `<phone work="4637033" />`, and this is known as an empty element. All attributes in XML elements must be enclosed in quotation marks. An XML document can be thought of as representing a tree hierarchy.

### 4.1.2 "eXtensible" in XML

The *extensible* part in XML means that XML is not really a markup language instead it is meta-markup language which means when creating XML documents you create your own markup language. This custom markup language is called an XML based language, such as the languages described in appendices A, B and C.

### 4.1.3 Benefits of XML

XML allows anyone to create their own information and mark it up in the way that meets their requirements and send it over a network or the Internet.

```
<?xml version="1.0" ?>
<students>
 <student>
  <name>Craig Anslow</name>
  <stuid>1234</stuid>
  <phone>4637037</phone>
  <degree>honours</degree>
 </student>
 <student type="grad">
  <name>Stuart Marshall</name>
  <stuid>5678</stuid>
  <phone work="4637033" />
  <degree>phd</degree>
 </student>
</students>
```

Figure 4.1: A Simple XML Document.

The promise of XML is that it is an easy way for different applications to communicate with each other since it is basically ASCII text. As an example, if different email clients stored all information such as bookmarks, address books and email messages in XML, then switching between email clients would be a relatively easy exercise even if it required translation.

Applications should be able to read and parse any XML document and if it contains a schema, see section 4.3, then this must be sent to the application with the XML document. Displaying the schema in a public place gives the ability for other developers to access it, and write their own XML documents based on that schema.

XML is free from any intellectual property rights and has no patents, trademarks, copyrights or trade secrets. Since the creation of the XML specification [15] there continues to be many applications and services that are making use of XML.

## 4.2 XML Documents

XML documents are often classified as being either data centric (section 4.2.1) or document centric (section 4.2.2). Although the border between data and document centric XML documents is sometimes unclear, this classification plays an important role in deciding what kind of database system to use when storing XML related data [11].

### 4.2.1 Data Centric

Data centric documents use XML as a data transport medium and are designed for machine consumption. They have fairly regular structure, meaning there is little or no mixed element character content, and the order of elements is not important. Examples include sales orders, flight schedules, stock quotes and student class data.

### 4.2.2 Document Centric

Document centric documents are usually built for human consumption and have a less regular structure. Nested elements and character data are usually spread out within one element, but the order in which the nested elements and character data appear are significant. Examples include advertisements, product descriptions, e-mails and manuals.

## 4.3 XML Document Structure

XML documents can be represented by different structures and these include Document Type Definitions (DTDs) (section 4.3.1) and XML Schema (section 4.3.3). They are basically dictionaries that XML documents must conform to, if being used. They can also be used as templates of custom markup language to create many XML documents. XML namespaces (section 4.3.2) can also be used in DTDs and XML Schemas to associate elements within one namespace. These concepts are discussed in detail below.

### 4.3.1 Document Type Definitions

A Document Type Definition (DTD) is a text document that defines all parts of an XML document such as all the elements, attributes, lists, entities, their properties or constraints, relationships among elements, what elements are allowed, in what order they can appear and how they can be nested.

An XML document is *valid* if it has an associated DTD and its structure obeys the rules that are contained in the DTD. An XML document is *well formed* even if it does not have a DTD associated with it, but starts with an XML declaration, contains a root element that encompasses all other elements, and it is properly nested.

#### DOCTYPE Declaration

DTDs can be located inside an XML document or it can be located in a separate file. If the DTD is located in a separate file it is called an external DTD otherwise if it is inside the document it is called an internal DTD. DTDs that are located in a separate file are much more flexible as the size of the XML document is smaller and other XML documents can use the same DTD with out having to put it in their XML document.

DTDs are declared using the `<!DOCTYPE root_element [ elements, attributes, entities ] >` syntax and this contains all the elements, attributes and entity definitions. It can be used as an internal DTD in an XML document or it can be put into a separate file and used as an external DTD.

Figure 4.2 shows how to declare an external DTD in an XML document. For the XML document to know that it has an external DTD it has the attribute `standalone="no"` in the xml version attribute. To call the DTD there is a `"DTD_location"` attribute as a Universal Resource Identifier (URI). If the DTD is private then the keyword `SYSTEM` is used, however if it is a public DTD then the keyword `PUBLIC` is used and the name of the DTD `"DTD_name"`, is required.

```
<?xml version="1.0" standalone="no" ?>

<!DOCTYPE root_element SYSTEM "DTD_location">

or

<!DOCTYPE root_element PUBLIC "DTD_name" "DTD_location">
```

Figure 4.2: The DOCTYPE declaration for a DTD.

### Element Definition

An element that appears in the DTD has the following syntax:

```
<!ELEMENT name (ELEMENTS_DATA_NOTHING)>
```

Possible options for `name` and `(ELEMENTS_DATA_NOTHING)` are listed in table 4.1. Also Element definitions can appear in any order in the DTD.

| **DTD Code** | **Description** |
|---|---|
| `<!ELEMENT foo (bar)>` | `bar` appears once in `foo`. |
| `<!ELEMENT foo (bar, bar)>` | `bar` appears twice in `foo`. |
| `<!ELEMENT foo (bar, zip)>` | `bar` and `zip` appear once each in `foo`. |
| `<!ELEMENT foo (bar?)>` | `bar` appears once or not at all in `foo`. |
| `<!ELEMENT foo (bar+)>` | `bar` appears one or more times in `foo`. |
| `<!ELEMENT foo (bar*)>` | `bar` appears zero or more times in `foo`. |
| `<!ELEMENT foo (bar \| zip)>` | `bar` or `zip` appear in `foo`. |
| `<!ELEMENT foo ANY>` | Anything can go inside `foo`. |
| `<!ELEMENT foo EMPTY>` | No elements can go inside `foo`. |

Table 4.1: Summary of the DTD Element Definition Rules.

### Attribute Definition

An attribute is defined as: `<!ATTLIST elementName attributeName type default>`. Table 4.2 shows the possible default states for the attribute definition.

An element can have a list of acceptable type values or enumerated type values as in the case of figure 4.1, which could have multiple type values for the phone number, e.g. `<!ATTLIST student phone (home | work | mobile) #REQUIRED>`.

An `ID` attribute can create a unique element in an XML document. The `ID` value can't be all numbers: it has to have some text in the value. The `stuid` element in figure 4.1 could be represented as an attribute *rather* than an element, by `<!ATTLIST student stuid ID #REQUIRED>` and an instance would be `<student stuid="g_1234">` where "g" stands for graduate and "u" stands for undergraduate. With the `ID` attribute it is possible to use the `IDREF` type which refers to an `ID` elsewhere in the XML document. It is also possible that an attribute can refer to multiple `ID`s by using the `IDREFS` type.

| **DTD Code** | **Description** |
|---|---|
| `<!ATTLIST foo bar CDATA "default">` | If this attribute is not present then the parser adds it with its default value otherwise it is overwritten. |
| `<!ATTLIST foo bar CDATA #FIXED "default">` | If this attribute is not present then the parser adds it with its default value otherwise if the attribute is included that value must equal the default value. |
| `<!ATTLIST foo bar CDATA #REQUIRED>` | The attribute must exist in an element of an XML document. |
| `<!ATTLIST foo bar CDATA #IMPLIED>` | The attribute may or may not exist in the XML document. |

Table 4.2: DTD Attribute Default Values.

### Entity Definition

XML also has five predefined entities which XML documents can use:

- `&amp;` which displays ampersand (&)
- `&lt;` which displays a less than sign (<)
- `&gt;` which displays a greater than sign (>)
- `&quot;` which displays a double quote (")
- `&apos;` which displays a single quote (')

Entities can be used to create shortcuts by using the following syntax `<!ENTITY entity_name "entity_content">` and an example of that from figure 4.1 could be `<!ELEMENT address (#PCDATA)>`. The `<student>` element could use the shortcut `<!ENTITY vuw "PO Box 600 Wellington">` and would be represented as `<address>&vuw;</address>`.

Table 4.3 shows the different ways to declare an entity in an XML document or in a DTD. If it is declared in a DTD the entity name is prefixed with a percentage sign (%). To call an external entity or a DTD in another file the keyword `SYSTEM` must be used before the filename to include. The advantage of this is that you can make a DTD from many other DTDs and likewise build up a single DTD using many different entities from other files. This can also help to reduce the complexity in large XML and DTD files.

| **Type** | **Description** | **DTD Code** |
|---|---|---|
| Internal general | Entity created in DTD and used in XML document | `<!ENTITY vuw "PO Box 600 WGN">` |
| External general | Entity created in a different file and used in XML document | `<!ENTITY vuw SYSTEM "vuw.txt">` |
| Internal parameter | Entity created in DTD and used in DTD | `<!ENTITY % vuw "PO Box 600 WGN">` |
| External parameter | Entity created in DTD in a different file and used in DTD | `<!ENTITY % vuw SYSTEM "vuw.dtd">` |

Table 4.3: DTD Entity Types.

### 4.3.2 XML Namespaces

The reason for wanting to use namespaces in a single XML document is to distinguish between elements that have the same name. Namespaces are really just a naming scheme and look like `<prefix:elementName />`. The prefix is the local name of the namespace. A namespace is declared by `<anyElement xmlns:localName="URI">`. The URI in the namespace can be a URL and may or may not point to something, but it must be unique. A default namespace contains no `localName`. All the attribute names of an element become part of the namespace and they do not need to be prefixed by the `localName`.

The scope of a namespace contains all of the elements descendants. It is possible to override a namespace by reassigning a `localName` to a new namespace, i.e. to a new URI. If a namespace has a `localName` it can't be undeclared, i.e. `localName` can't be changed to another `localName` with the *same* URI. Multiple namespaces can also be nested with inside each other.

When using DTDs and namespaces together, each prefixed element has to be declared separately in the DTD. DTDs don't know anything about namespaces and treat `xmlns` as just another attribute, and prefixed elements are just elements whose names have a colon. Namespaces are supported in XML schemas. Finally namespaces do not help in validating XML documents, and this includes DTDs and XML Schemas.

### 4.3.3 XML Schema

An XML Schema is written in XML, and it is a namespace or a collection of elements. It was designed to be better than a DTD by supporting more features. The basic XML schema structure can be seen in figure 4.3.

```
<?xml version="1.0" ?>
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 (other elements contained here)

</xsd:schema>
```

Figure 4.3: Basic XML Schema Element.

Since it is an XML document it has the XML encoding tag at the top. The namespace for the XML Schema is the `xsd` tag and it is also the root element of the XML Schema. The namespace points to the latest W3C XML Schema namespace.

To call an XML Schema in an XML document, the keyword `xsi` is used. It stands for XML Schema Instance and is used as in figure 4.4.

All elements in XML Schema are defined as being *simple type* or *complex type*. Simple type elements have no attributes, contain no elements, and hold text or are empty. Complex types elements contain other nested elements, attributes, and text.

```
<?xml version="1.0" encoding="UTF-8"?>
<rootElementName
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="URI">

 (other elements contained here)

</rootElementName>
```

Figure 4.4: Calling an XML Schema in an XML Document.

### Simple Types

An example of a simple type can be seen in figure 4.5. This shows a student name element which has a string type. Only text can be contained in the `<studentname>` element.

```
<xsd:element name="studentname" type="xsd:string" />
e.g. <studentname>Craig</studentname>
```

Figure 4.5: An example of a simple type.

Other simple types include the following:

- **Number Based:** `xsd:decimal`, `xsd:float`, `xsd:double`, `xsd:positiveInteger`, `xsd:negativeInteger`, `xsd:nonPositiveInteger` and `xsd:nonNegativeNumber`.
- **Date and Time Based:** `xsd:date`, `xsd:time`, `xsd:timeInstant`, `xsd:timeDuration`, `xsd:month`, `xsd:year`, `xsd:century`, `xsd:recurringDate` and `xsd:recurringDay`.
- **Miscellaneous:** `xsd:boolean`, `xsd:language`, `xsd:uri-reference` and `xsd:NMTOKEN`.

Custom simple types can be created. It is possible to limit acceptable values by using the `xsd:restriction base="simpleType"` element, which limits an element to a single type and can be used in conjunction with the `xsd:enumeration` element. Types that match a regular expression pattern use the `xsd:pattern` element. Limiting numerical values can be achieved by using the max and min exclusive and inclusive elements or for accuracy in significant numbers, the `xsd:precision` type can be used. Strings can also be limited to a certain length. Some elements require more than one item per element hence the `xsd:list` element can be used to create lists of items. Simple types can be combined together by using the `xsd:union` element however elements get quite hard to read. An element's content can be constant by using the "fixed" value or it can be given a "default" value much the same way as in a DTD. Finally custom types can be reused in other places so long as it is not declared inside an `xsd:element` and the simple type has a name.

### Complex Types

Complex types contain nested elements, as in figure 4.6. This figure shows a student element which is a complex type, and has a sequence of elements which include a student id and the courses that they are taking, as well as their name and the degree they are studying towards.

To order the nested elements in a sequential order the `xsd:sequence` element is used. For each of the nested elements there is a minimum and maximum number of times that they are allowed to appear as a nested element within a student element. This is achieved by using the `minOccurs` and `maxOccurs` elements. A student name must occur once. A student is allowed to take zero or more courses. Table 4.4 shows the differences in the cardinality of elements between an XML Schema and a DTD.

```
<xsd:element name="student">
 <xsd:complexType mixed="true">
  <xsd:sequence>
   <xsd:element name="stuname" minOccurs="1" maxOccurs="1"
   type="xsd:string" />
   <xsd:element name="courses" minOccurs="0" maxOccurs="unbounded"
   type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="stuid" type="xsd:integer" use="required" />
  <xsd:attribute name="degree" type="xsd:string" use="required" />
 </xsd:complexType>
</xsd:element>
```

Figure 4.6: Complex type element that has nested elements.

| XML Schema | DTD |
|---|---|
| `minOccurs="0"` `maxOccurs="unbounded"` | `element*` |
| `minOccurs="0"` `maxOccurs="1"` | `element?` |
| `minOccurs="1"` `maxOccurs="unbounded"` | `element+` |
| Default `minOccurs="1"` `maxOccurs="1"` | `element` |

Table 4.4: Cardinality differences between an XML Schema and a DTD.

There are other options instead of `xsd:sequence` which are the `xsd:choice` and `xsd:all` elements. The `xsd:choice` element works by selecting one sub element from a list of sub elements nested in the element. The `xsd:all` works behaves exactly the same as `xsd:sequence` but nested elements can appear in any order.

Attributes are added to a complex type in the same way as an element is. They must have a simple type and come after all nested element declarations. Attributes can have a use attribute and in figure 4.6 the student id must be added, and is of integer type. Other options for an attributes use is located in table 4.5.

Attributes can be combined with text using the `xsd:simpleContent` element. The text can have restrictions placed on it by using `xsd:restriction`, or have none at all by using `xsd:extension`. In figure 4.6 the `xsd:complexType` element has a value of `mixed="true"` which means that attributes, text and nested elements can be combined together.

Custom complex types can be created by combining simple types into one complex type by

| **Item** | **Description** |
|---|---|
| `use="required"` | Attribute must be present. |
| `use="optional"` | Attribute may or may not be present. |
| `use="prohibited"` | Attribute is not allowed. |
| `use="fixed"` `value="mandatoryValue` | Attribute must have the value of `mandatoryValue` if it is present otherwise no value is set. |
| `use="default"` `value="defaultValue"` | If the attribute is not present it is added and given the value of `defaultValue`. If it is present then its value overrides `defaultValue`. |

Table 4.5: How attributes can be used in XML Schema.

giving the complex type a name, and nesting the simple type elements inside the `xsd:complexType` element. The `xsd:complexType` element can not appear inside any other XML Schema element. It is possible to define an element or attribute and refer to it later by having a "ref" value inside the element like `<xsd:element ref="elementName">`. Existing types can be used as templates to create new complex types by using the `xsd:complexContent` element inside a complex type element.

Other ways to create complex types include making a group of elements using the `xsd:group` element, which groups a list of elements by `xsd:sequence`, `xsd:choice`, or `xsd:all`. The `xsd:group` element can then be used in a complex type by using the group element or referring to it. This can also be used for attributes which use the `xsd:attributeGroup` element.

To add documentation to the XML Schema the `xsd:documentation` element is nested inside an `xsd:annotation` element. These elements can be placed anywhere in an XML document.

An XML Schema can be created from many different schemas by including the path to the schema location file in `<xsd:include schemaLocation="filename.xsd" />`. The only problem with this approach is that if anything needs to be redefined in the new XML Schema it must be inside the `<xsd:redefine schemaLocation="filename.xsd">` element.

### 4.3.4 Summary

There are a few disadvantages to using DTDs:

- **No Data Types:** DTDs do not support primitive types like boolean or integer while XML Schema does, and allows custom data types.
- **Not in XML:** DTDs are written in SGML while XML Schema is written in XML. Therefore a new syntax must be defined for DTDs.
- **No Inheritance:** DTDs provide no way to support inheritance, while an XML Schema provides a way to extend and restrict an existing type for inheritance.
- **No Namespace Support:** DTDs do not support namespaces at all, and treat the namespace prefixed element as another element with a colon. The `xmlns` is just treated as another attribute. This is because all DTD elements are all global elements and they must all be unique.

The following are the explicit advantages of using an XML Schema as opposed to a DTD:

- **Referential Integrity Constraints:** The ref element can be used to mirror the primary key - foreign key pair relationship.
- **Nested Element Options:** XML Schema has the ability to sort the order and number of nested elements in the mixed mode. Elements can have a minimum and or maximum number of consecutive instances.
- **Namespaces:** Every name is associated with a namespace, such that we can have the same name for different elements without causing conflict. For example, we can define title for a person and title for a book.
- **Extensible:** Since XML Schema is written in XML, it is extensible. This means schemas and data types can be reused.

## 4.4 XML Storage

There are various approaches for storing XML documents such as flat files (section 4.4.1), BLOBs (section 4.4.2), relational databases (section 4.4.3) and native XML databases (section 4.4.4).

### 4.4.1 Flat Files

The quickest and easiest way to store XML is in a flat file located on a file system. Each XML file contains one XML document and can be accessible from the directory it is located in. The downside to storing the XML files on the file system is that they lack the ability to be able to queried and modified efficiently. Simple transaction control can be achieved by using a version control system like CVS.

### 4.4.2 Binary Large Objects

When storing XML documents as binary large objects (BLOBs) the entire XML document is stored as one large object either in a relational database or as a flat file in the file system. BLOBs support transaction control, security, multi-user access, document extensibility, but because BLOBs have no control over the structure of the document there is no access to the node level, no way to do updates, and no option to do any structure dependent querying of any sort. However various text searches are available such as full text, proximity and synonym searches.

When accessing the XML document the whole BLOB is retrieved. When modifying an XML document, it is first deleted, updated and then reinserted into the database. PostgreSQL[37] has support for BLOBs using the `BYTEA` data type which is a descriptor containing the name of the data type with no specific maximum length.

BLOBs are most appropriate when extensibility is needed, and that the level of granularity required by an application is the entire document and not any data within the document.

### 4.4.3 Relational Databases

When storing XML in relational databases there is a need to map the DTD to relational tables. This method is known as *shredding.* This approach gives relational databases XML functionality but cannot support extensibility. It also gives node level access through SQL. Structural information of an XML document, such as the hierarchy and the order of the elements, is lost when mapped into a relational table, therefore features such as round tripping[1], and XML query languages are not supported. Shredding is appropriate for long term storage of durable data. The method in figure 4.7 [10] is a way to implement shredding, but a DTD must be present with the XML document.

1. Each element in the DTD is assigned a table.
2. A table is used for the relationship between an element and it's child elements with the same tag.
3. Each tuple in a table is assigned a unique id and contains a reference to it's parent element.
4. If a parent element has a single child element then this can be an attribute in the parent table.

Figure 4.7: Shredding method of an XML document into a relational database.

### 4.4.4 Native XML Databases

Native XML databases store XML documents in the DOM [2] format. This preserves the documents structure because data elements are stored as parsed hierarchical linked node. It also helps support extensibility. Native XML databases do not support SQL or other relational database tools, however they do support the DOM, XML query languages such as XPath, XQuery, and transformations from one XML document to another by XSLT (eXtensible Style Language Transformations). XML databases can also accept XML documents that do not have a DTD, which helps support node-level access and extensibility. Native XML databases are primarily used to manage active XML documents.

There are many data models used to describe an XML database and an XML document and these include the following:

- **Object Exchange Data Model (OEM):** The OEM[70] is self describing and can store documents, elements, text, data or comments as objects represented by a hierarchical graph. Figure 4.8 is an example of the OEM data model using the simple XML document from figure 4.1. An OEM object contains the following information:
  - Object identifier - used to uniquely identify an object.
  - Label - represents the schema name of an object.
  - Type - represents the type of an object.
  - Value - contains the value of the primitive object or references to other objects.

[1]Round tripping is storing a document and getting the same document back again. It is important for document centric XML applications because they need information which can be hidden by data centric documents, such as comments, entity references and the order the elements appear in the XML document.

[2]DOM is short for Document Object Model. It is an object-oriented language-neutral formal model of the structure of an HTML web page or an XML document. The DOM is an API for dynamically accessing, adding and changing structured content in documents [103].

- **Node-Centric Data Model:** The node-centric data model is based on the W3C DOM. Data represents a tree and works best if there are no bi-directional dependencies; otherwise the edge-oriented approach is better. It has data types for representing the document element (root element), other elements, attributes and character data. A node is the union of element, and character data types.
- **Edge-Oriented Data Model:** The edge-oriented approach is based on a tree structure where edges are the relationships amongst elements [28].

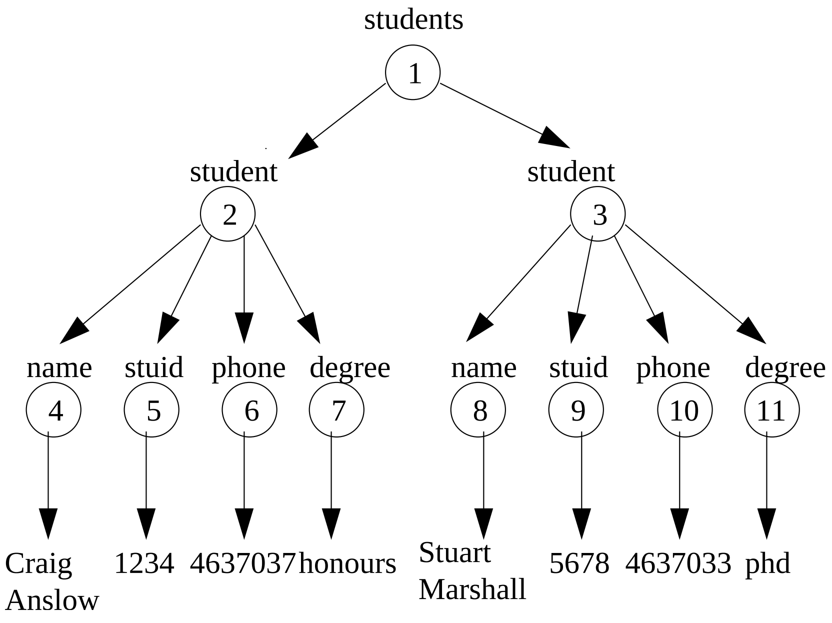
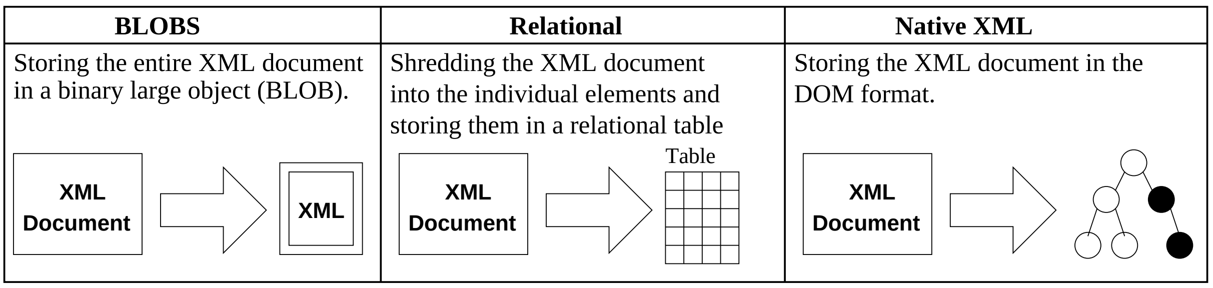


Figure 4.8: Object Exchange Data Model (OEM) representation of figure 4.1.

### 4.4.5 Summary

Figure 4.9 is a summary showing the main points of each of the previous three approaches.

BLOBs are flexible, extensible, have a simple architecture and are optimised for text searches. It is very costly to do updates because the BLOB has to be deleted, updated and then re-added to the database or saved in the filesystem. To access a node requires parsing the whole BLOB linearly because there is no structure information.



| BLOBS | Relational | Native XML |
|---|---|---|
| Storing the entire XML document in a binary large object (BLOB). | Shredding the XML document into the individual elements and storing them in a relational table | Storing the XML document in the DOM format. |

Figure 4.9: Comparing the Storage of XML Documents.

Storing XML data in a relational database requires using the shredding method which needs a schema. The advantage of shredding is that relational database management system (RDBMS) tools are supported and it is optimised for SQL. Once the XML data is stored in the RDBMS it is not possible to extend it. Accessing a node is done through SQL and structure information is lost. Extensions of this kind are likely to be common in VARE, so the relational approach seems problematic.

The advantages of using a native XML database to store XML is that it is flexible, extensible and has direct access to the node level. It is also optimised for the XPath and XQuery languages. The disadvantages of native XML databases is that RDBMS tools are not supported, loading documents takes longer and that XML is a relatively new standard compared with the relational data model.

## 4.5 XML Query Languages

Many XML query languages exist such as XML-QL[22], XQL[91], and Lorel[1]. The main languages that are W3C defacto standards are XPath 2.0 and XQuery 1.0. A comparison of all of these languages and the features that each support can be found elsewhere [32].

There are some main features that XML query languages should contain [17] such as clean and clear semantics, using path expressions in the query, ability to return an XML document, ability to query and return XML tags and attributes, distinguish between different data types like numerical and textual, handle unexpected data that does not conform to a DTD, allow queries when the DTD is not fully known, return unnamed attributes e.g. “return all elements except element A”, return trees instead of sets of XML elements and preserve the order of the XML document.

### 4.5.1 XPath

The aim of XPath[19] is to provide a way to address parts of an XML document. It is possible to locate an exact element in an XML document or to find a pattern based on the XML document elements because it is a concise language for navigating trees. Whenever an XPath expression is evaluated the result is an object and this can be a set of nodes, a number, string or boolean value [52].

XPath is not a complete query language as it can only query one document at a time, has no facilities for joins or grouping, and is weak on references. XPath can be used with other XML technologies such as XPointer which is a technique for building pointers into XML documents and is similar to anchors in HTML. It is also used in XSL for specifying patterns in style sheets.

The data model in XPath treats a document as a tree of nodes for which there are seven: root, element, attribute, text, namespace, comment and processing instruction. An element node may have a unique ID attribute. An XPath expression maps a node called the context node which is the node where the XPath expression is up to in the XML document into a set of nodes.

An XPath expression is written in the form of `axis::node-test`. The axes that exist are `child-descendant`, `parent`, `ancestor`, `following`, `preceding`, `attribute`, `namespace`, `self`, `ancestor-or-self`, `descendant-or-self`, `following-sibling` or `preceding-sibling`. XPath expressions use some of the basic following syntax, as in figure 4.6, for location path

| Expression | Meaning |
|---|---|
| `(nothing)` | nothing or `child::` which means it can be omitted |
| `/` | child or root or `child::node()` |
| `//` | descendant or `/descendant-or-self::node()/` |
| `.` | self or `self::node()` |
| `..` | parent or `parent::node()` |
| `@` | attribute or `attribute::` |
| `*` | any |

Table 4.6: XPath expressions

abbreviations.

The syntax in figure 4.6 can be used to create the following example queries in XPath, as in figure 4.7.

| Query | Description |
|---|---|
| `.//student/course[1]` | find the first course of every student element that is under the context node. |
| `../@name` | find the "name" attribute of the parent of the context node. |
| `student[last()]` | find the last student child of the context node. |
| `//students/student` | find all student elements that are inside students elements. |
| `//*[@ID]` | find all elements that have an ID attribute. |
| `//students[student/stuid > 1000]/name` | find all student names that have a stuid greater than 1000. |

Table 4.7: XPath example queries

An XPath expression in general is a union of one or more paths and each path consists of one or more steps. Usually a step has three parts: the *axis* or direction of movement, a *node test* which is the type and or name of qualifying nodes, and an unlimited number of *predicates* which are placed in square brackets to refine the set of qualifying nodes. The types of functions that can appear in predicates are node-set (last, position, ...), string (concat, substring, ...), boolean (not, boolean(object), ...), and number functions (sum, ceiling, ...).

The advantage of XPath is that it is a compact and powerful syntax for navigation in a tree, although not as powerful as some regular expression languages. It has a W3C specification and is used in some XML technologies such as XPointer and XSL. The downside to using XPath is that it operates on only one document at a time, it has no grouping, aggregation or join conditions that can be used, and has no facility for generating new output structures.

### 4.5.2 XQuery

XML Query Language (XQuery)[8] incorporates XPath's expressions. XQuery is based on the Quilt[90] XML query language which in turn was influenced by the functional approach of Object Query Language (OQL), by the keyword based syntax of SQL, and by previous XML language proposals which include XQL[91], XML-QL[22] and Lorel[1]. One thing to note is that XQuery does not support any update functionality for XML documents, but this is intended for later versions.

The basic form of an XQuery consists of a *FLWR* (pronounced "flower") expression, and is very similar to the **SELECT-FROM-WHERE** construction in SQL. A FLWR expression consists of `FOR`, `LET`, `WHERE` and `RETURN` clauses [8]:

- **FOR**: binds one or more variables, which is a name that begins with a dollar sign (e.g $X, $Y, ...), to a sequence of nodes returned by another expression (usually a path expression) and iterates over the nodes. The variable represents an array of bound nodes.
- **LET**: binds one or more nodes but without iterating. A *single* sequence of nodes is bound to the variable.
- **WHERE**: contains one or more predicates that filter or limit the set of nodes as generated by the `FOR` and `LET` clauses.
- **RETURN**: generates the output of the FLWR expression. The `RETURN` clause usually contains the references to variables and is executed once for each bound node-reference that was returned by the previous clauses.

The input to a `FOR/LET` clause is the `DOCUMENT` function which returns the root node of an XML document. The keyword `DISTINCT` can be applied independently to each expression in a `FOR/LET` clause. Path expressions are used to traverse the XML document and the syntax is similar to the abbreviated XPath syntax. XPath is heavily used in XQuery and any expression following `DOCUMENT` selects one or more child elements below the root element.

In the `WHERE` clause predicates can be used and are based on XPath expressions that contain the variables bound in the `FOR` and `LET` clauses. Predicates may be combined using parentheses and the following types of operators: comparisons (=, <, >, !=, ...), logical (`EXISTS`, `EVERY`, `SOME`), conjunctive/disjunctive and negation (`AND`, `OR`, `NOT`), arithmetic (addition, substraction, multiplication, division, and modulus), compound (`UNION`, `INTERSECT`, `EXCEPT`) and infix and prefix. `IF THEN ELSE` expressions can be used as well. In the `WHERE` clause, joins can be achieved by comparing values that are returned by sub-queries. It is possible to specify the bound variable in the comparison by using one of the data type modifiers (e.g. `text()`, `number()`) added to the end of the bound variable separated by a forward slash.

Aggregate functions are allowed which include `COUNT`, `SUM`, `AVG`, `MAX` and `MIN`. They can be applied in `LET`, `WHERE` and `RETURN` clauses. They may also be combined with `DISTINCT`. Sorting can be done using a `SORTBY` clause. User defined functions can also be created and they may be recursive.

The output in the `RETURN` clause may be a node, set of nodes or a primitive value. The `RETURN` clause is invoked once for each tuple in the variable bindings generated by the `FOR` and `LET` clauses. The `RETURN` clause may contain arithmetic expressions, structured XML text, bound variables with XPath expressions associated, and sub-queries. An element constructor is used in the `RETURN` clause to create new elements because XPath expressions only select existing nodes. It consists of a start and an end tag, enclosing an optional list of expressions that provide the content of the element. `<RESULT>` is the most common element constructor.

Figure 4.10 is an example of a query in XQuery format. The query finds all students who are doing an honours degree from figure 4.1.

```
for $t in document("collectioName/students.xml")//student
where $t/degree = "honours"
return
<honours>
 {$t/name}
 {$t/stuid}
</honours>

Query result:

<honours>
 <name>Craig Anslow</name>
 <stuid>1234</stuid>
</honours>
```

Figure 4.10: Example of a query in XQuery.

## 4.6 XML Parsers

XML parsers can be divided into two categories, tree based and event based [36]. A tree based parser creates an internal tree structure. The tree structure can be navigated to extract the correct information. The DOM parser is a tree based parser that creates objects for each element and character data in the XML document. An event based parser reports parsing events, such as the start and end of an element to an application through call-backs. The tree based parser does not build an internal tree structure. The SAX (Simple API for XML)[61] parser is an event based parser that is more efficient for extracting data from large documents.

Tree based parsers can be used for many different applications, however if the XML document is very large (e.g. over a megabyte) it puts tremendous pressure on system resources. Also most applications build their own typed data structures rather than using a generic tree such as that of the XML document. Therefore it is very time consuming to build a tree of parse nodes and then map it to a new data structure and finally discard the original.

Event based parsers provide a much simpler lower level access to an XML document. The advantages of event based parsers over tree based parsers is that it is possible to parse documents that are much larger than the available system memory, that they can construct data structures using call-back event handlers, and so they are faster and simpler at processing an XML document as they only parse the necessary information rather than the whole XML document. There are many different implementations of XML parsers [104].

## 4.7 Scalable Vector Graphics

Scalable Vector Graphics (SVG)[26] is an XML language for describing two-dimensional vector graphics such as lines, shapes, fills, strokes, Bezier curves, gradients, images, text and effects.

It is possible for graphical objects to be grouped, styled, transformed and composed into previously rendered objects. The features of SVG include nested transformations, clipping paths, alpha masks, filter effects, template objects and extensibility. Another feature of SVG is that it provides events such as `onMouseOver` and `onClick` which can be applied to any SVG graphical object. The animations can be defined and triggered either declaratively or by scripting [102]. The advantage of SVG is that it is written in XML text and not a proprietary binary such as that of its competitor Flash [41].

Figure 4.11 shows a basic SVG diagram which has a blue rectangle, a red triangle and a green circle. Figure 4.12 shows the code that was used to create this SVG diagram. The SVG code shows that it is an XML document with the xml version tag and that it uses the `SVG DOCTYPE` specification. The `svg` element preserves a space for the canvas of the diagram and then the shapes are rendered with their specific properties.

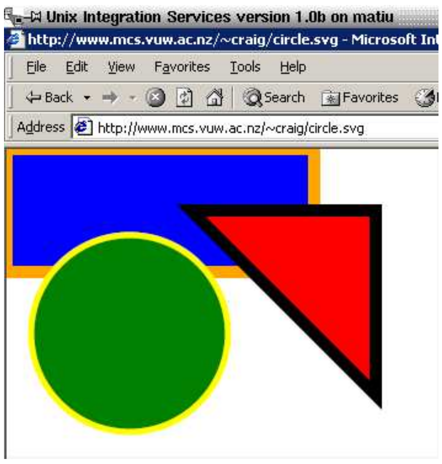

Unix Integration Services version 1.0b on matiu
http://www.mcs.vuw.ac.nz/~craig/circle.svg - Microsoft In
File Edit View Favorites Tools Help
Back Search Favorites
Address http://www.mcs.vuw.ac.nz/~craig/circle.svg

Figure 4.11: A basic SVG diagram of a rectangle, circle and a triangle.

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg xml:space="preserve" width="350" height="250">
<rect style="fill:blue;stroke-width:10;stroke:orange;"
      width="250" height="100" />
<path style="fill:red;stroke-width:10;stroke:black;"
      d="M 150 50 L 300 50 L300 200 z" />
<circle style="fill:green;stroke-width:5;stroke:yellow;"
       cx="100" cy="150" r="80" />
</svg>
```

Figure 4.12: SVG code for the image in figure 4.11

## 4.8 XML Web Services

Web services is a standardised way of integrating web-based applications using XML, Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) open standards. XML is used to tag the data, SOAP is used to transfer the data (section 4.8.1), WSDL is used for describing the services available (section 4.8.2) and UDDI is used for listing what services are available (section 4.8.3).

Web services do not provide the user with a graphical user interface (GUI) because the applications interface while the web service is transparent to the user. Developers can add the web service to a GUI (such as a web page or an executable program) to offer specific functionality to users [33].

Web services allow different applications from different sources to communicate with each other without time consuming custom coding. Since all communication is in XML, web services are not tied to any one operating system or programming language because they are language independent.

### 4.8.1 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) is lightweight, simple, extensible, interoperable and an XML-based protocol for information exchange of structured and typed information or network communication between software services in a decentralised, distributed environment [14].

SOAP is basically a general purpose technology for sending messages between endpoints and it is based in XML and HTTP. It may be used for RPCs or straight forward document transfer. It also supports inter-operation between COM, DCOM, CORBA, Perl, Tcl, Java, C, Python or PHP programs running anywhere on the Internet.

SOAP consists of the following three parts:

1. a *message* that is described by an envelope that defines a framework for describing what is contained in a message and how to process it.
2. a *set of encoding rules* for serialising data (which is beyond the scope of this report).
3. a *convention* for representing RPCs (Remote Procedure Calls) and responses.

#### SOAP Message

In a SOAP message the *first* part is either a request or response and it is a standard HTTP header. Within the header there is an `HTTP POST` operation that points to a Universal Resource Identifier (URI). In the SOAP specification [14] there is nothing about how a component is activated, it is up to the code behind the URI to decide how to activate the component and invoke the specified method. The `Content-Type` field uses a value of `text/xml-SOAP`. SOAP defines this value to identify an HTTP entity body containing an XML-encoded method invocation or response. All SOAP requests and responses must use this Content-Type value. There are also three extended HTTP headers:

- The `MessageType` header (required for all SOAP requests) defines whether the given HTTP body contains a method call or a method response, and must contain the values Call or CallResponse respectively.
- The `MethodName` or `SOAPAction` field (required for all SOAP requests) tells the remote host what the name of the method is to invoke so that the web server or firewall can perform some high level message filtering.
- The `InterfaceName` header (optional) defines the name of the interface to which the MethodName belongs.

The *second* part of a SOAP message is an XML document that contains three parts (see figure 4.13):

- The **Envelope** is the top-level container representing the message. It defines the various XML namespaces that are used by the rest of the SOAP message and usually includes `xmlns:soap` (SOAP envelope namespace), `xmlns:xsi` (XML Schema for instances), `xmlns:xsd` (XML Schema for data types) and `xmlns:soapenc` (SOAP encoding namespace). These represent the SOAP specification [14] namespaces for SOAP and schemas for instances, data types, and encoding.
- The **Header** is an optional generic container for carrying extra information for authentication, transactions, routing and payments. SOAP defines attributes to indicate who should deal with the information and whether understanding it is optional or mandatory. If a header is present it must be the first element of the envelope.
- The **Body** is a container or main payload for mandatory information intended for the final message receiver. SOAP defines one element for the body to report errors. When SOAP is used to perform an RPC, the body contains a single element that contains the method name to be invoked and it's arguments. The namespace of the method name is specified by the web service followed by the type of the target web service. The type of each argument can be optionally supplied using the `xsi:type` attribute. If a header is present the body follows the header, otherwise it is the first element of the envelope.

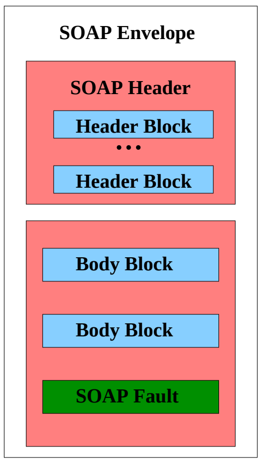


Figure 4.13: The SOAP message structure.

A SOAP request is typically accepted by a servlet, or Common Gateway Interface (CGI) process running on a remote web server. When the servlet gets a request it checks that the request has a `SOAPAction` field and if it it does, it forwards it to the SOAP container. The container uses the `POST` URI to look up the target web service, parses the XML payload and then invokes the method on the component.

The result of the invocation from a SOAP request is translated by the SOAP container into a SOAP response and returned back to the sender with the HTTP reply. The payload of the SOAP response contains the encoded method result. Usually the name of the result is equal to the name of the method followed by *'Response'* and the namespace of the result is the same namespace of the original method.

### Remote Procedure Calls with SOAP

When using RPCs with SOAP, see figure 4.14[3], the client application, i.e browser or traditional application, calls a client side proxy layer by its native RPC protocol (for example COM or CORBA). The proxy uses an XML parser to translate the call into a SOAP packet and it is then transmitted over HTTP across the Internet to the Web server. The Web server handles the URL connection point of the remote service and launches a SOAP translator which may be a CGI program or a Perl script. This translator uses a local XML parser to call the server object by the local RPC protocol and packages the results into a response SOAP packet and finally sends it to the client. Once the response reaches the client it is unpackaged by the proxy and presented to the client application.

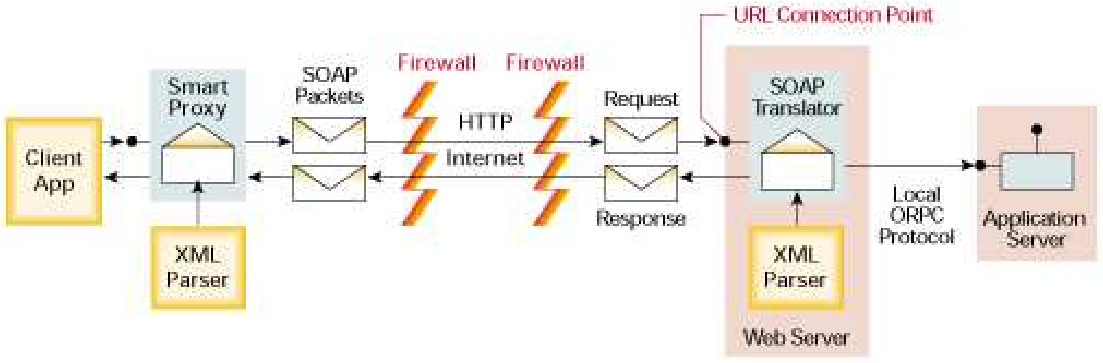


Figure 4.14: How Remote Procedure Calls work with SOAP.

The SOAP specification [14] provides an encoding which is a standard way to serialise objects and other data structures into XML. The objective of the encoding is to provide RPCs transparently to the client. This usually means instantiating an object and invoking its methods or properties on a remote procedure on a server, but letting the client think that it happens locally [100]. The important issue when using RPCs with SOAP is that both the client and the server use the same SOAP encoding otherwise they would not be able to communicate with each other.

[3]SOAP RPC image taken from `http://edu.uuu.com.tw/article/000915c.htm`

### 4.8.2 Web Services Description Language

The Web Services Description Language (WDSL) is an XML-formatted language used to describe a Web service's capabilities as collections of communication endpoints capable of exchanging messages. WSDL is an integral part of UDDI as it is the language that it uses.

### 4.8.3 Universal Description, Discovery and Integration

Universal Description, Discovery and Integration (UDDI) is a Web-based distributed directory that enables people to list themselves on the Internet and discover each other, which is similar to the yellow and white pages directory phone books.

## 4.9 Native XML Databases

Current relational databases do not support XML documents very well as XML and SQL are not compatible with each other in many of the following areas. They each use different data types, XML has the ability to change structure in the middle of a document while relational databases can't do this as they have a table structure. When an XML document is stored in a relational table, information can be lost, such as element ordering and the distinction between attributes and elements.

*Native* XML databases were created to address this problem because they are designed especially to store XML documents. Native XML databases are mostly useful for storing document centric XML documents. They preserve physical document structure, preserve all information that non XML databases drop, allow using XML query languages, speed up retrieving whole documents, and allow storing XML documents without a DTD or an XML Schema. The key difference from other databases, such as relational and object-oriented is that XML databases internal models are based on XML [11].

The following three main points identify what it means to be a native XML database. The definition was developed by the XML:DB Initiative for XML Databases[105], the organisation that is chartered with the development of XML database specific specifications:

1. Defines a (logical) model for an XML document – as opposed to the data in that document – and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX.
2. Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
3. Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

The following items are the most common features of native XML databases [105, 11]:

- **Document Collections:** A document collection contains documents of the same type and its aim is to be able to focus on a certain kind of document when querying the database.

- **Query Languages:**. The defacto standards XQuery and XPath are supported by W3C[15].
- **Updates and Deletes:** When updating and deleting the simplest approach is to replace the whole document, however an advanced option is to use a special query language and allow modification of a document fragment. Neither XQuery or XPath support updating.
- **Transactions, Locking, and Concurrency:** Most support transaction processing with commits and roll-backs, but locking is often at the level of entire documents, rather than at the level of document fragments. Efficient multi-user concurrency can be relatively low.
- **Application Programming Interfaces (APIs):** APIs usually come in the form of an Open Database Connectivity (ODBC) interface with methods for connecting to the database, exploring the meta-data repository, executing queries, retrieving results and communication over HTTP.
- **Round-Tripping:** Is the ability to store a document and get the same document back again. This is important for document centric XML applications because they need exact ordering of a document.

The following sections describe three mature proprietary native XML databases that are available. In each case I trialed and evaluated learning editions. Current open source native XML databases such as Xindice[29] are fairly new and are not as well developed; we expect in the future they will improve.

### 4.9.1 eXtensible Information Server

The eXtensible Information Server (XIS)[25] version 3.1 is a proprietary database developed by the eXcelon Corporation. Documents are not required to have a DTD or conform to a predescribed schema. They can be indexed using both value and structural indexes and are arranged in collections. XIS stores and manipulates XML data with an emphasis on high performance and high concurrency. Data is imported into the database from any OLE DB (object-linking and embedding database) or ODBC (object database connectivity) data source. Validation of an XML document happens when a file is imported, but not with subsequent updates.

XPath and XQuery is supported for querying, but XIS uses it's own XML based language for updating XML documents. For updates XIS locks at the node level and the concurrency scheme lets queries access dynamic data.

Users can link existing documents and create virtual documents that consist of nothing but links. Links are traversed transparently during queries and update operations. This means that virtual documents can be used to perform queries and updates over multiple documents in a single operation. Referential integrity is enforced by triggers for links.

XIS can integrate back-end data such as that stored in relational databases. The back-end data sources can also be updated through XIS. Other XIS features include transactions, distributed caching, partitioning, online backup and restore, tools for mapping back-end data to XML documents, and clustering support. XIS supports the following APIs: Java, COM, and .Net.

### 4.9.2 Tamino XML Server

The Tamino XML Server [20] was the first XML platform to store XML information without converting it to other formats, such as relational tables. There is a built in relational database as well as a native XML database engine and it can create live mappings from the relational database to the native XML database. The advantage of this is that the relational data can remain in the relational database, and it can then be accessed through an XML API.

The storage engine is document-oriented and concurrency locking is done at the document level instead of the node level. Schema oriented XML documents can be added but well-formed XML documents have to be configured correctly. Other features include an extended XPath implementation called X-Query (not W3C XQuery), full-text retrieval, processing of XML documents with server-side XSL and CSS (Cascading Style Sheets) and their own XML based update API.

Tamino supports the following APIs: WebDAV[4] for importing and exporting XML files and the DOM, SAX, COM, Java and XML:DB. There is also limited support for SOAP.

### 4.9.3 Ipedo XML Database

Ipedo XML Database[42] is able to store well formed XML documents and it can provide access to both whole XML documents and fragments of an XML document. The database has an object cache, indexes, and an optimised query and transformation engine.

Figure 4.15 shows how documents are stored in the Ipedo XML Database Server. They are stored by name and are organised into collections, which can be typed or untyped. A typed collection contains either a DTD or an XML Schema, and documents inserted into them are validated against that schema. Untyped collections have no schema which means any document can be inserted into them, and they are not validated. The number of collections that are allowed in this learning edition is five, with 32 documents per collection.

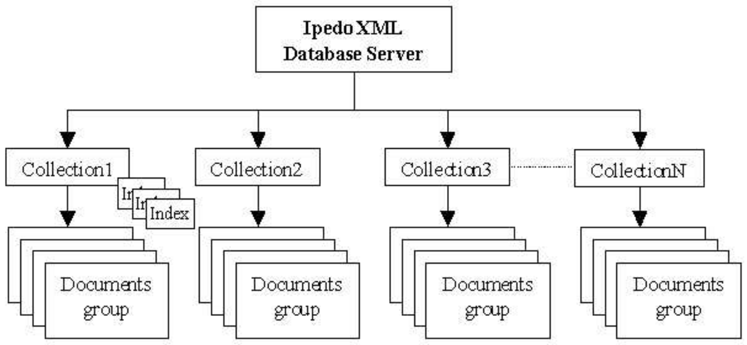


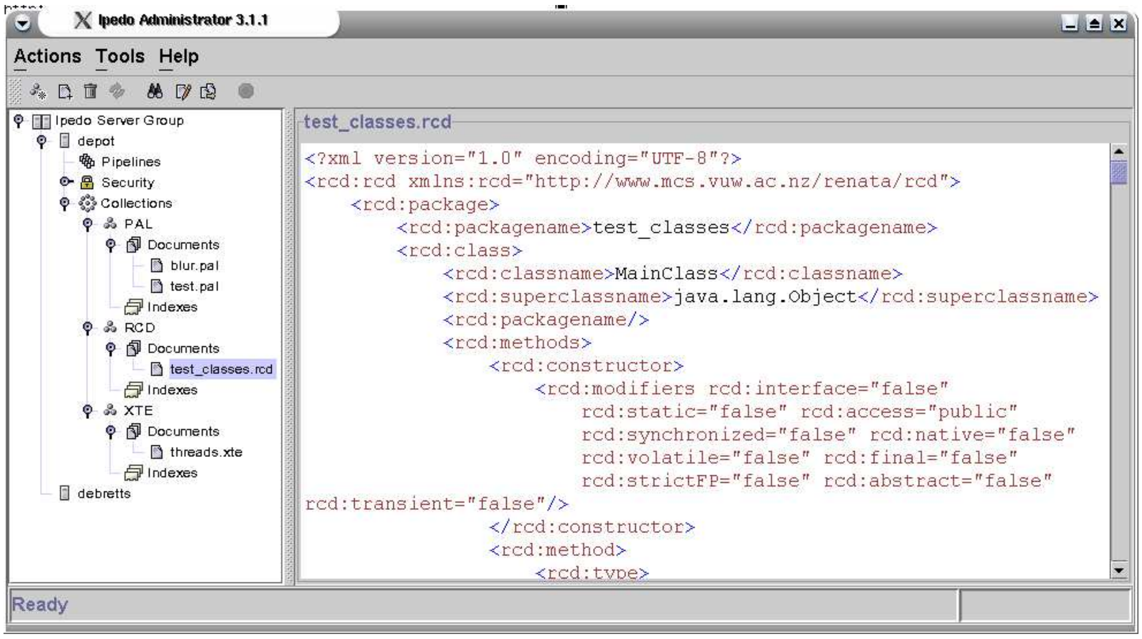
Figure 4.15: XML collections in the Ipedo XML Database.

XPath and XQuery are supported, and multiple documents in a collection can be queried through extensions to these languages. Queries can be combined with transformation operations that execute a query and transform the result set using XSLT. To increase query performance, indexes can be created for specific elements or attributes at the collection level and these include Hashmaps, Treemaps, and B-Trees. An API is provided for updates which

[4]Short for Web-based Distributed Authoring and Versioning, an IETF standard set of platform-independent extensions to HTTP that allows users to collaboratively edit and manage files on remote web servers.

requires specifying the nodes to be modified using an XPath expression, the document or documents to be modified, the operation to perform and the new data value.

Ipedo XML Database supports the following APIs: Java, COM, DOM, WebDAV and SOAP. Other features include document versioning, full text searches, SVG support, distributed database management, and support for very large documents which have a maximum size of 1 GB. An API client or the Ipedo Administrator can access the Ipedo XML Database Server through the Internet. The Ipedo Administrator is shown in figure 4.16. It has three collections with some XML program trace files stored in each collection.



```
<?xml version="1.0" encoding="UTF-8"?>
<rcd:rcd xmlns:rcd="http://www.mcs.vuw.ac.nz/renata/rcd">
    <rcd:package>
        <rcd:packagename>test_classes</rcd:packagename>
        <rcd:class>
            <rcd:classname>MainClass</rcd:classname>
            <rcd:superclassname>java.lang.Object</rcd:superclassname>
            <rcd:packagename/>
            <rcd:methods>
                <rcd:constructor>
                    <rcd:modifiers rcd:interface="false"
                        rcd:static="false" rcd:access="public"
                        rcd:synchronized="false" rcd:native="false"
                        rcd:volatile="false" rcd:final="false"
                        rcd:strictFP="false" rcd:abstract="false"
rcd:transient="false"/>
                </rcd:constructor>
                <rcd:method>
                    <rcd:type>
```

Figure 4.16: The Ipedo Administrator showing the PAL, RCD and XTE collections.

## 4.10 Summary

In this chapter we reviewed what XML is, the benefits from using XML, and the differences between document centric and data centric XML documents. We then investigated how XML documents can be structured. This can be done either by a DTD or an XML Schema. The different ways to store XML documents were compared next, and this included flat files, BLOBs, relational databases, and native XML databases. XML query languages, XML parsers, SVG, and XML web services were also discussed. We then trialed and evaluated three mature native XML databases.

We concluded that the most appropriate way to store document centric XML documents is in a native XML database and data centric XML documents in a relational database. When querying the XML documents, it depends on the type of the database that the document is stored in. If the document is stored in a relational database then SQL should be used, and if the document is stored in a native XML database then XQuery should be used. XQuery is a superset of the XPath query language. After reviewing XML web services, we found that SOAP is a good mechanism to transport independent information through the Internet. Of the three native XML databases that were evaluated, we decided that Ipedo XML Database met the functionality that we required for VARE. This functionality includes web based code repositories to support program trace visualisation, a Java and SOAP API, XQuery and XPath, storage of large files, SVG support, and useful documentation. In the next chapter we discuss XML based program trace languages for understanding visualisations of reusable components.

# Chapter 5

# XML Program Trace Languages

In this chapter, we present two different XML based program trace languages of reusable components that can be used to generate software visualisations within the VARE architecture. They each have components for representing both static and dynamic information. We then evaluate each of the program traces with the requirements for a program trace that were discussed earlier in this report.

## 5.1 Process Abstraction Language

Process Abstraction Language (PAL) is used to define an XML specification for an object model designed to help visualisation tools get the information they need for useful visualisation. It is defined as a DTD and describes the inner details of executing code [59]. It was originally designed by Mike McGavin as part of his honours project, however enhancements have been made to the DTD by Matthew Duignan, Stuart Marshall and myself. PAL currently stands as version 1.1 and the specification is provided in appendix A.

PAL is designed to describe object-oriented programs. It has elements for describing classes, super-classes, methods, and fields. PAL can also describe the run-time behaviour of programs, including objects, run-time representations of classes, method calls with their arguments and return values, and different threads of control [58].

### 5.1.1 PAL Session Element

A session is represented by the `pal` element and a unique number which can be used in visualisation tools to distinguish them from other visualisation sequences. It also contains a version attribute to indicate the PAL DTD version being represented. A session can also have a unique number which can be used in visualisation tools to distinguish them from other visualisation sequences. Figure 5.1 is an example of a PAL session.

### 5.1.2 Examination of a Program

The following sections look at what types of static information from the PAL DTD can be examined in a program. An example C++ program called Food Chain, in figure 5.2, has been used to show the different parts. Some static elements that can be examined are the `typecollection` and `type` elements.

```
<pal sessionid="1" version="1.1">

''other elements''

</pal>
```

Figure 5.1: A PAL definition of the pal session element.

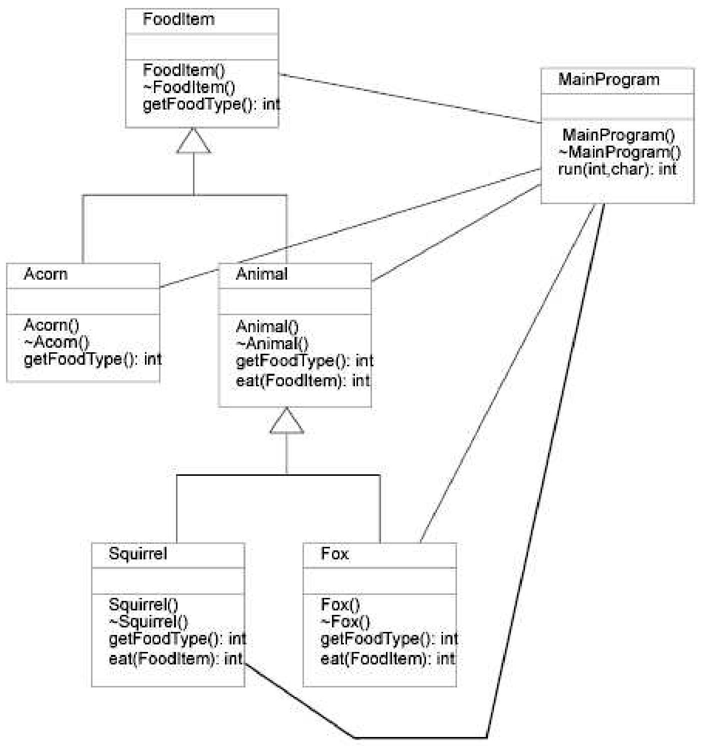


Figure 5.2: UML Class diagram of the FoodChain Class

### Type Collection Element

The `typecollection` contains a list of type specifications about each type in a program and is provided at the start of a session if requested. Each type has a classification category which states what sort of type it is, i.e. generic, alias, or class, or an identifier. The name is provided separately with a name attribute. Types can also be associated with the source file where it was declared, using the `context` element. An example of some of the available type specifications are in figure 5.3.

```
<typecollection>
 <typespec classification="generic" idref="t3" name="__vtbl_ptr_type"/>
 <typespec classification="generic" idref="t4" name="__wchar_t"/>
 <typespec classification="generic" idref="t5" name="bool"/>
 <typespec classification="generic" idref="t6" name="char"/>
 <typespec classification="generic" idref="t7" name="double"/>
 <typespec classification="generic" idref="t8" name="float"/>
 <typespec classification="generic" idref="t9" name="int"/>
 <typespec classification="alias" idref="t10" name="longint"/>
 <typespec classification="class" idref="t1" name="MainProgram">
  <context contextname="sourcefile" contextvalue="tp7.cc"/>
 </typespec>
 <typespec classification="class" idref="t2" name="Animal">
  <context contextname="sourcefile" contextvalue="tp7.cc"/>
 </typespec>
</typecollection>
```

Figure 5.3: A PAL definition of the typecollection element.

### Type Element

Figure 5.4 shows a PAL definition of the `Fox` class and inherits another class `Animal` that has the type identifier of `t2`. This class has a constructor and a destructor, indicated by the appropriate modifier elements. The `getFoodType` element is a virtual function and returns a double which is represented by `typeidref="t7"`. The `eat` element returns a double and takes a pointer as an argument. There is also a constructor for the `Squirrel` class, which takes a const reference to a char. Finally there is an operator overload method of `operator=` which sees if two char objects stored in memory are equal.

## 5.1.3 Execution of a Program

The following sections show the types of dynamic information that can be generated from a program. The Food Chain program has again been used as an example. Some dynamic elements that can be generated are the `execution`, `event`, `newclassinstance`, `deleteclassinstance`, `methodcall` and `methodreturn` elements.

```
<type name="Fox" typeid="t4">
 <context contextname="sourcefile" contextvalue="tp7.cc"/>
 <classdata>
  <superclasses>
   <superclass access="public">
    <typeref typeidref="t2"/>
   </superclass>
  </superclasses>
  <methods>
   <method access="public" methodid="f47" name="operator=">
    <modifier name="shadow"/>
    <typeref typeidref="t4">
     <modifier name="reference"/>
    </typeref>
    <argument argumentid="a46">
     <modifier name="const"/>
     <typeref typeidref="t4">
      <modifier name="reference"/>
     </typeref>
    </argument>
   </method>
   <method access="public" methodid="f49" name="Fox">
    <modifier name="shadow"/>
    <modifier name="constructor"/>
    <argument argumentid="a48">
     <modifier name="const"/>
     <typeref typeidref="t4">
      <modifier name="reference"/>
     </typeref>
    </argument>
   </method>
   <method access="public" methodid="f50" name="Fox">
    <modifier name="constructor"/>
   </method>
   <method access="public" methodid="f51" name="~Fox">
    <modifier name="destructor"/>
    <modifier name="virtual"/>
   </method>
   <method access="public" methodid="f52" name="getFoodType">
    <modifier name="virtual"/>
    <typeref typeidref="t7"/>
   </method>
   <method access="public" methodid="f54" name="eat">
    <modifier name="virtual"/>
    <typeref typeidref="t7"/>
    <argument argumentid="a53">
     <typeref typeidref="t3">
      <modifier name="pointer"/>
     </typeref>
    </argument>
   </method>
  </methods>
 </classdata>
</type>
```

Figure 5.4: A PAL definition of the Fox class.

### Execution Element

The `execution` element tag is used for dynamic information and has nested `event` elements. It may or may not have a unique `execid` as it's attribute. The `execution` element can represent everything that has happened when the program was executed or specific requirements as requested from a visualisation tool. The `execution` element can have interleaved `type` and elements. Figure 5.5 shows an example of the `execution` element.

```
<execution execid="session1">
‘‘event elements’’
</execution>
```

Figure 5.5: A PAL definition of the execution element.

### Event Element

Figure 5.6 shows an event element representing the beginning of the process and the `start` parameter that was passed to the process when it was run from the command line. Other information can be included here depending on what is requested. For example, the process begin event, the name of the process, start time, end time or the number of threads a process has spawned.

```
<event eventid="_200">
 <processbegin>
  <argstring>
   start
  </argstring>
 </processbegin>
</event>
```

Figure 5.6: A PAL definition of the event element.

### New Class Instance Event Element

A `newclassinstance` element is used when a new object is created. It contains a unique `classinstanceid` reference and a `typeidref` which refers to what class the object is an instance of. Figure 5.7 is an example of the `newclassinstance` element of the `Fox` class.

```
<event eventid="_255">
 <newclassinstance classinstanceid="dcl253" typeidref="t4"/>
</event>
```

Figure 5.7: A PAL definition of the newclassinstance element.

### Delete Class Instance Event Element

A `deleteclassinstance` element is used when an object is deleted. It uses the unique `classinstanceid` to refer to the object to be deleted. Figure 5.8 is an example of the `deleteclassinstance` element of the `Fox` class which deletes the `newclassinstance` in figure 5.7.

```
<event eventid="_378">
 <deleteclassinstance classinstanceid="dcl253"/>
</event>
```

Figure 5.8: A PAL definition of the deleteclassinstance element.

### Method Call Event Element

The `methodcall` element is used for when methods are called on an instantiated class. The following are the attributes in a `methodcall` element.

1. `methodcallid` and `methodidref` - are a unique identifier and a reference of the `methodcall`.
2. `classinstanceidref` - refers to the class instance that the method is called on.
3. `threadnum` - refers to the thread that is being used by the process.
4. `callerpositionref` - refers to the precise position in the program where the method was called from.
5. `callerclassinstanceidref, callermethodcallidref, callermethodidref` - refers to identifiers to what methods called a `methodcall` element.

Nested inside a `methodcall` element are arguments represented by the `argvalues` element, containing `argvalue` elements that have an `argumentidref` attribute. The `value` element is then nested in the `argvalue` element and can be either a `rawvalue` (the actual value being passed) or an `abstractvalue` (refers to the PAL identifier of a class instance). In figure 5.9 the `eat` method is called on the `Fox` object from figure 5.7 using `classinstanceidref="dcl253"`. The above enumerated information can also be determined. The `abstractvalue` that is being passed is a `Squirrel` object (`classinstanceidref="dcl232"`) for the `Fox` to eat, hence the Food Chain program.

### Method Return Event Element

The `methodcall` object can be interleaved with other events before there is a `methodreturn` element. The `methodreturn` element has a `methodcallidref` to the method that was called and nested inside is a `returnvalue` which then has a `value` and then finally it can be either a `rawvalue` or an `abstractvalue` if returning objects to the `methodcall`. Figure 5.10 shows that the `Fox` definitely ate the `Squirrel` as the `rawvalue` that is returned is the value “1”, which represents true in this C++ program.

```
<event eventid="_306">
 <methodcall callermethodcallidref="dmc212" callerpositionref="dp302"
  classinstanceidref="dcl253" methodcallid="dmc304" methodidref="f54"
  threadnum="1" callerclassinstanceidref="dcl203">
  <argvalues>
   <argvalue argumentidref="a53">
    <value>
     <abstractvalue classification="classref" classinstanceidref="dcl232"/>
    </value>
   </argvalue>
  </argvalues>
 </methodcall>
</event>
```

Figure 5.9: A PAL definition of the methodcall element.

```
<event eventid="_323">
 <methodreturn methodcallidref="dmc304">
  <returnvalue>
   <value>
    <rawvalue>1</rawvalue>
   </value>
  </returnvalue>
 </methodreturn>
</event>
```

Figure 5.10: A PAL definition of the methodreturn element.

## 5.2 Reusable Component Descriptions

Reusable Component Descriptions (RCD) stores static information of a reusable component, where a component is defined as consisting of one or more packages, and each package having one or more classes. While eXtensible Trace Executions (XTE) (section 5.3) stores the dynamic information of RCD components. RCD and XTE combined together provide the same concept as that of PAL.

The RCD DTD was originally created by Stuart Marshall in late January 2003. Since then Stuart Marshall and I have been using it in our software visualisation tools. We have made some modifications to it because we found that there were errors in the DTD and the files that were being generated. The current version of RCD, version 2.0, is provided in appendix B.

The following sections describe the elements in the RCD DTD and uses an example Java program called `test_classes`. The RCD information is stored in a file called `test_classes.rcd`. Figure 5.11 shows a UML diagram of the `test_classes` program.

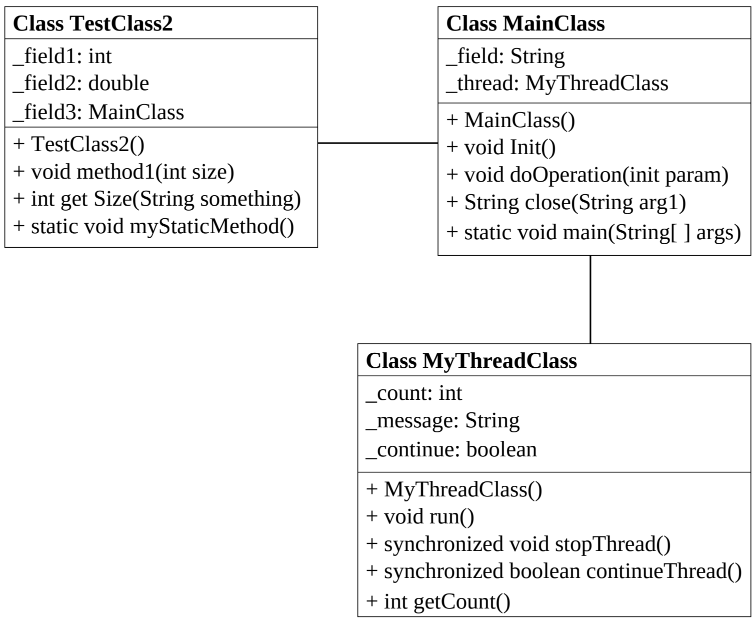


Figure 5.11: UML diagram of the test_classes program

### 5.2.1 RCD and Package Element

An RCD trace starts with the `rcd` element as in figure 5.12. RCD is associated with the renata/rcd namespace and all sub elements are within this namespace. Each element in a RCD program trace is prefixed with the local name of the namespace which is `rcd`. Nested inside the `rcd` element is an optional `version` and `creator` element. There can be many `package` elements that are associated with the `rcd` element. The package contains the `packagename` element and its classes.

```
<rcd:rcd xmlns:rcd="http://www.mcs.vuw.ac.nz/renata/rcd">
 <rcd:version>rcd 2.0</rcd:version>
 <rcd:creator>Stuart Marshall</rcd:creator>
 <rcd:package>
   <rcd:packagename>test_classes</rcd:packagename>
   <rcd:class>

   ‘‘class child elements’’

   </rcd:class>
 </rcd:package>
</rcd:rcd>
```

Figure 5.12: RCD root and package element.

### 5.2.2 RCD Class Elements

The class element represents a class from the source program as in figure 5.13. The class element contains the classname, the superclassname if any and the packagename that it belongs to. In this case the name of the class is MainClass, it inherits from the java.lang.Object class and belongs to the test_classes package. It also has methods and fields. It is possible that this class could have many interface or dependency elements.

```
<rcd:class>
  <rcd:classname>MainClass</rcd:classname>
  <rcd:superclassname>java.lang.Object</rcd:superclassname>
  <rcd:packagename>test_classes</rcd:packagename>
  <rcd:methods>

   ‘‘methods child elements’’

  </rcd:methods>
  <rcd:fields>

   ‘‘fields child elements’’

  <rcd:fields>
</rcd:class>
```

Figure 5.13: RCD class elements.

### 5.2.3 RCD Methods and Fields Element

The methods element, shown in figure 5.14, contains the constructor and methods elements of a class. There may be one initialiser and many constructor and method elements.

The constructor contains modifiers which have a series of compulsory boolean type attributes ranging from whether or not the constructor is abstract to an object being volatile. There is also an access attribute which states what level of access the method is, and the values include `package`, `private`, `public` and `protected`. A `constructor` can have zero or many arguments. An `argument` element is represented by an `argumentname` (PCDATA) and the `type`. For example the `type` value could be a String, int or a float, and it is represented by a `typename` which is also PCDATA.

The `method` element contains the following elements: `type`, `modifiers`, and `methodname`. It may also have many `arguments`. Figure 5.14 shows three methods. These include an `init()` method that does not return anything and is expressed as a void `type` element, a `doOperation(init param)` method that does not return anything but takes an object that is an integer `type`, and the `main(String[] args)` method which takes the standard Java String arguments. There is one method, `close(String arg1)`, that has been removed from this figure for simplicity reasons, but is contained in figure 5.11.

The `fields` element contains many `field` elements. As in figure 5.15 there are two fields for the `MainClass` class. The `fieldname` of each of these fields are `_field`, which is a String type, and `_thread`, which is a `test_classes.MyThreadClass` type. The `field` element also contains `modifiers`.

## 5.3 eXtensible Trace Executions

The purpose of the eXtensible Trace Executions (XTE) is that it stores execution trace information derived dynamically from test driven reusable components. It is the execution information that is associated with RCD, section 5.2. The XTE DTD was initially created by Stuart Marshall in late January 2003. Since then Stuart Marshall and I have been using it in our software visualisation tools and have made some modifications to it, because we found that there were errors in the DTD and the files that were being generated. The current version of XTE, version 2.0, is provided in appendix C.

The following sections describe the elements in the XTE DTD and use the Java program from figure 5.11. The trace information is stored in a file called `threads.xte`. The XTE file shows objects that have been created, threads of control and various types of execution events.

### 5.3.1 XTE Element

An XTE trace begins with the `xte` element as in figure 5.16. XTE is associated with the XML namespace renata/xte and all descendant elements are within this namespace. Each element in an XTE program trace is prefixed with the local name of the namespace which is `xte`. Nested inside the `xte` element is a mandatory `creator` and `date` element. This tells a developer who created the file and when it was created. There can be many `object` and `thread` elements, but there is only ever one `execution` element.

Figure 5.16 shows that the user "alias" was the creator of the XTE trace and that it was created on Wednesday 26th February. It also shows that there are two threads of control, and `main thread` and `new thread`.

### 5.3.2 XTE Object Elements

The XTE `object` element contains a `complextype` and zero or many `field` elements. An ID attribute is also required. Figure 5.17 shows three `object` elements where object1 has

```
<rcd:methods>
  <rcd:constructor>
    <rcd:modifiers rcd:abstract="false" rcd:access="public"
      rcd:final="false" rcd:interface="false" rcd:native="false"
      rcd:static="false" rcd:strictFP="false"
      rcd:synchronized="false" rcd:transient="false" rcd:volatile="false"/>
  </rcd:constructor>
  <rcd:method>
    <rcd:type>
      <rcd:typename>void</rcd:typename>
    </rcd:type>
    <rcd:modifiers rcd:abstract="false" rcd:access="public"
      rcd:final="false" rcd:interface="false" rcd:native="false"
      rcd:static="false" rcd:strictFP="false"
      rcd:synchronized="false" rcd:transient="false" rcd:volatile="false"/>
    <rcd:methodname>init</rcd:methodname>
  </rcd:method>
  <rcd:method>
    <rcd:type>
      <rcd:typename>void</rcd:typename>
    </rcd:type>
    <rcd:modifiers rcd:abstract="false" rcd:access="public"
      rcd:final="false" rcd:interface="false" rcd:native="false"
      rcd:static="false" rcd:strictFP="false"
      rcd:synchronized="false" rcd:transient="false" rcd:volatile="false"/>
    <rcd:methodname>doOperation</rcd:methodname>
    <rcd:argument>
      <rcd:argumentname>this</rcd:argumentname>
      <rcd:type>
        <rcd:typename>int</rcd:typename>
      </rcd:type>
    </rcd:argument>
  </rcd:method>
  <rcd:method>
    <rcd:type>
      <rcd:typename>void</rcd:typename>
    </rcd:type>
    <rcd:modifiers rcd:abstract="false" rcd:access="public"
      rcd:final="false" rcd:interface="false" rcd:native="false"
      rcd:static="true" rcd:strictFP="false"
      rcd:synchronized="false" rcd:transient="false" rcd:volatile="false"/>
    <rcd:methodname>main</rcd:methodname>
    <rcd:argument>
      <rcd:argumentname>args</rcd:argumentname>
      <rcd:type>
        <rcd:typename>java.lang.String[]</rcd:typename>
      </rcd:type>
    </rcd:argument>
  </rcd:method>
</rcd:methods>
```

Figure 5.14: RCD methods element.

```
<rcd:fields>
  <rcd:field>
    <rcd:type>
      <rcd:typename>java.lang.String</rcd:typename>
    </rcd:type>
    <rcd:fieldname>_field</rcd:fieldname>
    <rcd:modifiers rcd:abstract="false" rcd:access="private"
      rcd:final="false" rcd:interface="false" rcd:native="false"
      rcd:static="false" rcd:strictFP="false"
      rcd:synchronized="false" rcd:transient="false" rcd:volatile="false"/>
  </rcd:field>
  <rcd:field>
    <rcd:type>
      <rcd:typename>test_classes.MyThreadClass</rcd:typename>
    </rcd:type>
    <rcd:fieldname>_thread</rcd:fieldname>
    <rcd:modifiers abstract="false" access="private"
      final="false" interface="false" native="false"
      static="false" strictFP="false"
      synchronized="false" transient="false" volatile="false"/>
  </rcd:field>
</rcd:fields>
```

Figure 5.15: RCD fields element.

```
<xte:xte xmlns:xte="http://www.mcs.vuw.ac.nz/renata/xte">
  <xte:creator>alias</xte:creator>
  <xte:date>Wed Feb 26 09:18:54 NZDT 2003</xte:date>
  <xte:object>

  ‘‘object child elements’’

  </xte:object>
  <xte:thread xte:threadid="thread1">main thread</xte:thread>
  <xte:thread xte:threadid="thread2">new thread</xte:thread>
  <xte:execution>

  ‘‘execution child elements’’

  </xte:execution>
</xte:xte>
```

Figure 5.16: XTE root element.

object2 and object3 as its fields. These objects are the fields from figure 5.15.

```
<xte:object xte:objectid="object1">
  <xte:complextype>
    <xte:typename>test_classes.MainClass</xte:typename>
  </xte:complextype>
  <xte:field>
    <xte:fieldname>_thread</xte:fieldname>
    <xte:objectvalue xte:objectid="object2"/>
  </xte:field>
  <xte:field>
    <xte:fieldname>_field</xte:fieldname>
    <xte:objectvalue xte:objectid="object3"/>
  </xte:field>
</xte:object>
<xte:object xte:objectid="object2">
  <xte:complextype>
    <xte:typename>test_classes.MyThreadClass</xte:typename>
  </xte:complextype>
</xte:object>
<xte:object xte:objectid="object3">
  <xte:complextype>
    <xte:typename>java.lang.String</xte:typename>
  </xte:complextype>
</xte:object>
```

Figure 5.17: XTE object element.

### 5.3.3 XTE Entities

Figure 5.18 shows two external parameter entities `TYPE` and `VALUE` that were used in the XTE DTD, located in appendix C, to help with understanding and clarification. The value of `TYPE` can be either `xte:complextype` or `xte:primitivetype`. The value of `VALUE` can be either `xte:objectvalue` or `xte:primitivevalue`.

```
<!ENTITY % TYPE "(xte:complextype | xte:primitivetype)">
<!ENTITY % VALUE "(xte:objectvalue | xte:primitivevalue)">
```

Figure 5.18: XTE Entity Types.

### 5.3.4 XTE Execution Elements

The XTE `execution` element contains the following sub- elements: `objectcreation`, `methodcall`, `methodreturn`, `fieldaccess`, `fieldmodification`, `exceptionthrow`, `exceptioncatch`, and `securitypermission`. There has been no output generated for `exceptionthrow`, `exceptioncatch`, and `securitypermission` elements in any of our testing thus far. All of these sub-elements contain an `eventid`, and a `threadid` attribute which refers to the unique event id and the thread that the event belongs to. Some also contain an `objectid` to refer to the object that the sub-element is associated with.

Figure 5.19 shows the `objectcreation` element. It shows the type of object that was created and in this case it is object1 from figure 5.17.

```
<xte:objectcreation xte:eventid="event1" xte:objectid="object1"
   xte:threadid="thread1">
  <xte:complextype>
    <xte:typename>test_classes.MainClass</xte:typename>
  </xte:complextype>
</xte:objectcreation>
```

Figure 5.19: XTE object creation element.

Figure 5.20 shows a `methodcall` element which calls the `init` method from figure 5.14.

```
<xte:methodcall xte:eventid="event4" xte:receiverid="object1"
    xte:senderid="object1" xte:threadid="thread1">
  <xte:methodname>init</xte:methodname>
  <xte:typename>test_classes.MainClass</xte:typename>
</xte:methodcall>
```

Figure 5.20: XTE method call element.

Figure 5.21 shows a `methodreturn` element. In some cases with XTE nothing is actually returned but the `methodreturn` element still acts as a signal that the method that was originally called has finished.

```
<xte:methodreturn xte:eventid="event10" xte:threadid="thread1">
  <xte:primitivevalue>
    <xte:value/>
  </xte:primitivevalue>
  <xte:methodname>test_classes.MyThreadClass</xte:methodname>
  <xte:typename>test_classes.MyThreadClass</xte:typename>
</xte:methodreturn>
```

Figure 5.21: XTE method return element.

Figure 5.22 shows a `fieldmodification` element where `_field` from object1 is being modified from object2 to object3 from figure 5.17.

```
<xte:fieldmodification xte:eventid="event5"
   xte:objectid="object1" xte:threadid="thread1">
  <xte:fieldname>_field</xte:fieldname>
  <xte:oldvalue>
    <xte:objectvalue xte:objectid="object2"/>
  </xte:oldvalue>
  <xte:newvalue>
    <xte:objectvalue xte:objectid="object3"/>
  </xte:newvalue>
</xte:fieldmodification>
```

Figure 5.22: XTE field modification element.

Figure 5.23 shows a `fieldaccess` element. The type of object that is being accessed is object4 and its `<xte:fieldname>` is `_thread`. The object4 is not listed in figure 5.17 but is contained in the threads.xte file.

```
<xte:fieldaccess xte:eventid="event12" xte:objectid="object1"
   xte:threadid="thread1">
  <xte:fieldname>_thread</xte:fieldname>
  <xte:objectvalue xte:objectid="object4"/>
</xte:fieldaccess>
```

Figure 5.23: XTE field access element.

## 5.4 Evaluation of Program Traces

In this section we provide an evaluation of the program traces (PAL, RCD and XTE), with the requirements for a program trace format, that were described in section 3.5.

- **Storable and Re-playable:** all of the program traces are storable since they are written in an XML format, with associated DTDs. They can also be replayed, by using appropriate query languages to access the right information.
- **Support Live Streaming to Visualisation Tools:** no live visualisations have been created from the program traces, however we expect the XML based program traces to be able to be parsed efficiently by using an XML parser to meet the requirements of live streaming, based on Matthew Duignan's experiences with PAL [23].
- **Easily Transportable Over the Internet:** since all of the program traces are written in XML, which is ASCII text, there should be no transportation problems. A transport mechanism such as SOAP could be used to meet this requirement.
- **Filterable For Extracting Relevant Information:** using an XML parser such as DOM or SAX would be expected and then the relevant information could be extracted.
- **Able to be Queried:** since all of the program traces are XML based, XQuery and XPath can be used to query them for important information for visualisations.

- **Programming Language Independence:** all of the program traces have been designed to support many object oriented programming languages. PAL has been used with C++ and Java programs, while RCD and XTE have been tested only with Java programs, but we intend to test them with other object-oriented languages.
- **Platform Independence:** the program traces have only been tested on the Intel/NetBSD platforms, but we expect that it would be suitable for other types of closely related UNIX platforms such as Sun/Solaris, and we are beginning to explore other platforms.
- **Scalable to Large Components and Visualisations:** we have done some testing in this area and concluded that in general for medium to large sized files, over 1MB-10MB, that the PAL program traces were much larger. When the files were smaller than 1MB there was not much difference in size between the program traces.

The problems with PAL is that it relies heavily on the `ID` and `IDREF` DTD attribute. When parsing the PAL file, it creates very deep tree structures. PAL contains static and dynamic information for each program trace in one file. Whenever there is an object that is used from an API library, it is contained in the PAL file by the `type` element. If there are many library objects used then this can make the PAL file very large, however it can be turned off. RCD and XTE are better because they separate the static and dynamic information into two separate files. The advantage of this, is that many XTE files can be generated from the same RCD file. PAL would have to regenerate the static information each time. RCD and XTE have the ability not to reference objects from other standard libraries. For large program traces the RCD and XTE files are smaller in size. The next chapter investigates storing these program traces in XML databases.

# Chapter 6

# XDS: An XML Data Storage Environment

In this chapter we present XDS, an XML Data Storage environment for storing and retrieving XML based program traces. We will first examine a relational approach for storing the program traces. Second we will show a native approach, and finally we will describe how XDS can be integrated into VARE.

## 6.1 Relational XDS

An experiment was developed translating the PAL, RCD, and XTE program traces into relational schemas. This is so that they could be used for a relational database. We computed manually the shredding algorithm from figure 4.7 to create the relational schemas. We conducted our experiment for storing and retrieving the program traces with PostgreSQL[37]. It supports many features that we could benefit from such as: inheritance, data types, functions, constraints, triggers, rules, transactional integrity, nested queries, and foreign keys. This proved to be unsuccessful, because the relational schemas became very complex, and hard to understand. The problem came from representing multiple relationships between the program trace elements.

In the PAL DTD the `typeref` element has many relationships with the following elements: `aliasdata`, `superclass`, `method`, `variable`, and `argument`. This caused a deep tree structure as the the `typeref` referred to a `modifier` element. The `modifier` element created many extra tables because it needed to also refer to many other elements that include: `method`, `variable` and `argument`. For all of the tables, a parent id was required, and this added more information. There were over 30 relational tables for the PAL schema. There were extreme costs when querying this model because it would have to create many joins via the parent id attribute in order to get the requested data.

The RCD and XTE relational schemas were more efficient than the PAL schema. The RCD schema had the same problem with the `modifier` element as PAL. The XTE schema has many small elements, that included `creator`, `date`, `details`, `permission`, `typename`, `methodname`, `fieldname`, `parametername` and `value`. Tables were created for all of these elements and they consisted of just a parent id and a unique name.

If a relational database were to be used for storing XML program traces, it would be best to design a more efficient relational schema from scratch, rather than using a mapping technique.

## 6.2 Native XDS

Having found that the relational approach was unsuccessful, we then used a native XML approach. We stored the program traces in a native XML database. The native XML database that we used was Ipedo, because it supported the functionality that we required. This included, a Java and SOAP API, and useful documentation. The following sections describe the architecture, the user interface, and demonstrate XDS in action.

### 6.2.1 Architecture and Technology Overview

Figure 6.1 shows the architecture of the XDS environment. A user can access the native XML database through their web browser on their workstation. The web browser communicates with the Apache Tomcat web-server, executing at:
`http://st-james.mcs.vuw.ac.nz:8888/db/index.jsp`, to request the Java Servlet Pages (JSP). The user can then select one of the user options as described in section 6.2.2. Upon selection the JSP page executes a method from the SOAP client. The SOAP client is implemented in Java. The Apache Tomcat server then communicates with the Ipedo Native XML Database, by SOAP to execute the remote method. The Ipedo Native XML Database, executing at `http://depot.mcs.vuw.ac.nz:8000/db`, has a built in SOAP server. It also has a WSDL file, describing the types of web services that a SOAP client can perform. Once the information is stored or retrieved in the native XML database, a response is sent back to the client, using SOAP. The response of the remote method invocation, is displayed in the users browser, and can be either successful or unsuccessful. In the case of a successful query, the query result will be displayed. In the case of an unsuccessful query, a friendly error message is displayed.

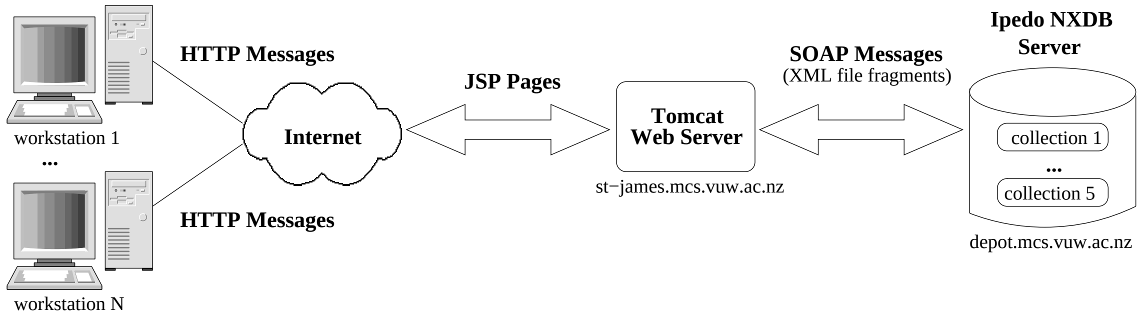


Figure 6.1: Architecture of the XML Data Storage (XDS) environment.

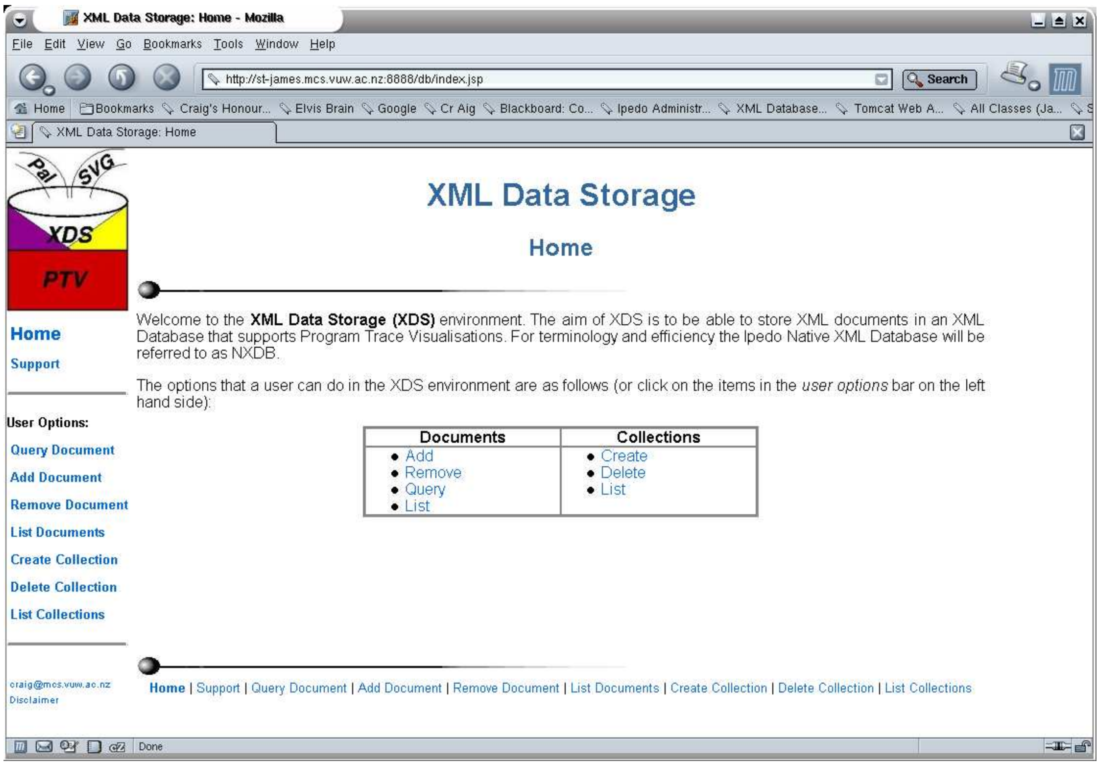
### 6.2.2 User Interface Design and User Options

Figure 6.2 shows the the XDS homepage. It has been designed using JSP pages for user interaction, and for communicating with the Apache Tomcat web-server. At the top left of each page in XDS, the logo is displayed. The meaning of the logo is that PAL and SVG files can be stored and retrieved. At the top of the page, and in the centre, “XDS” (or XML Data Storage for the homepage) is displayed. Below that for each page, is the name of each page. On the left hand side of each page, is the menu for the user options. The user interface has the following user options:

1. Home, a quick link to get to the main page.

2. Support, information about the project.
3. Query an XML document in a collection.
4. Add an XML document to a collection.
5. Remove an XML document from a collection.
6. List all the XML documents in a collection.
7. Create a collection in XDS.
8. Delete a collection from XDS.
9. List all the collections in XDS.

The middle of the page, in figure 6.2, between the ruler objects, all the content for each page is displayed. The homepage has a short introduction and a table of user options. These mirror the options located in the left hand side menu bar. The word "NXDB" is used to refer to the Native XML Database on all of the XDS pages. There is also another menu located at the bottom of the page which has the same options as the left hand side menu bar. It is located at the bottom of the page in case the current page is long and requires scrolling.


XML Data Storage: Home - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://st-james.mcs.vuw.ac.nz:8888/db/index.jsp
Search
Home Bookmarks Craig's Honour... Elvis Brain Google Cr Aig Blackboard: Co... Ipedo Administr... XML Database... Tomcat Web A... All Classes (Ja...
XML Data Storage: Home
Pal SVG XDS PTV
Home
Support
User Options:
Query Document
Add Document
Remove Document
List Documents
Create Collection
Delete Collection
List Collections
craig@mcs.vuw.ac.nz
Disclaimer
Home | Support | Query Document | Add Document | Remove Document | List Documents | Create Collection | Delete Collection | List Collections
Done

XML Data Storage

Home

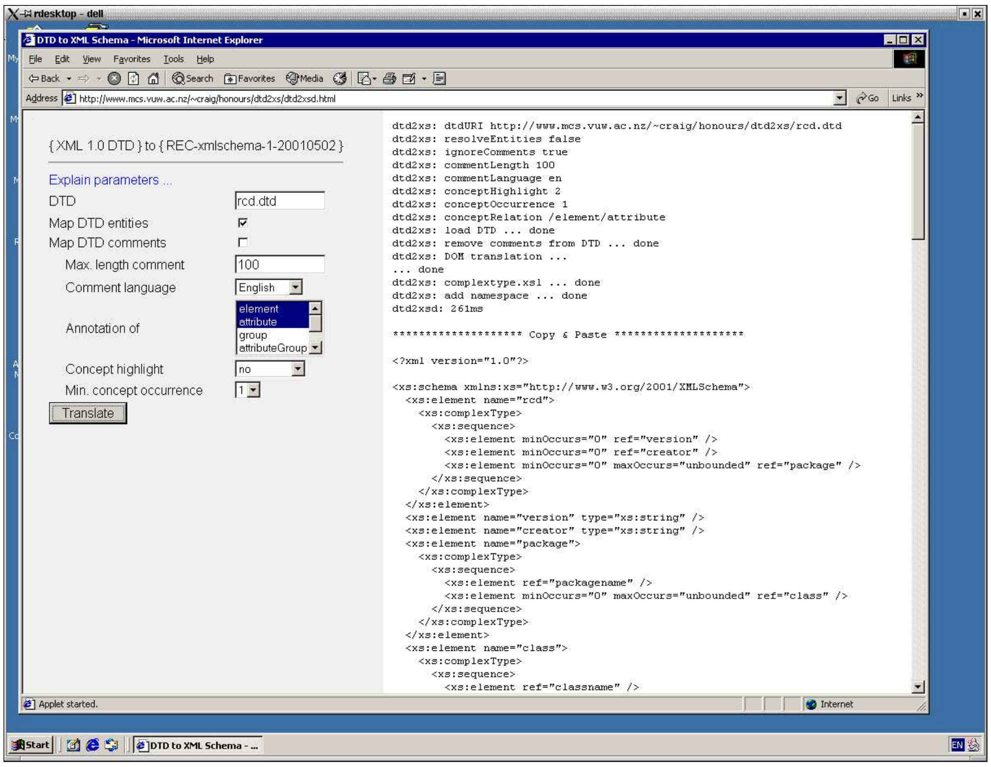
Welcome to the **XML Data Storage (XDS)** environment. The aim of XDS is to be able to store XML documents in an XML Database that supports Program Trace Visualisations. For terminology and efficiency the Ipedo Native XML Database will be referred to as NXDB.

The options that a user can do in the XDS environment are as follows (or click on the items in the *user options* bar on the left hand side):

| Documents | Collections |
|---|---|
| • Add<br>• Remove<br>• Query<br>• List | • Create<br>• Delete<br>• List |

Figure 6.2: XDS User Interface Design.

One other design feature that is incorporated into XDS, is the ability to convert a DTD into an XML Schema. This is done using the dtd2xs[53] Java applet. We did this because we know that XML Schema is a better approach for representing the structure of XML documents. Figure 6.3 shows the interface to the applet. To transform a DTD into an XML Schema, a user has to upload the DTD they want converted and select translate. There are also options



{ XML 1.0 DTD } to { REC-xmlschema-1-20010502 }

Explain parameters ...

| | |
|---|---|
| DTD | rcd.dtd |
| Map DTD entities | ✓ |
| Map DTD comments | |
| Max. length comment | 100 |
| Comment language | English |
| Annotation of | element, attribute |
| Concept highlight | no |
| Min. concept occurrence | 1 |

```
dtd2xs: dtdURI http://www.mcs.vuw.ac.nz/~craig/honours/dtd2xs/rcd.dtd
dtd2xs: resolveEntities false
dtd2xs: ignoreComments true
dtd2xs: commentLength 100
dtd2xs: commentLanguage en
dtd2xs: conceptHighlight 2
dtd2xs: conceptOccurrence 1
dtd2xs: conceptRelation /element/attribute
dtd2xs: load DTD ... done
dtd2xs: remove comments from DTD ... done
dtd2xs: DOM translation ...
... done
dtd2xs: complextype.xsl ... done
dtd2xs: add namespace ... done
dtd2xsd: 261ms

******************** Copy & Paste ********************

<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="rcd">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" ref="version" />
        <xs:element minOccurs="0" ref="creator" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="package" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="version" type="xs:string" />
  <xs:element name="creator" type="xs:string" />
  <xs:element name="package">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="packagename" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="class" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="class">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="classname" />
```

rdesktop - dell
DTD to XML Schema - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.mcs.vuw.ac.nz/~craig/honours/dtd2xs/dtd2xsd.html
Translate
Applet started.
Internet
Start
DTD to XML Schema - ...

Figure 6.3: Conversion of a DTD to an XML Schema using the dtd2xs Java applet.

to transform the meta-data from the DTD. We transformed each of the program trace DTDs into XML Schemas. Currently we have not generated any program trace files that can be validated against the XML Schemas.

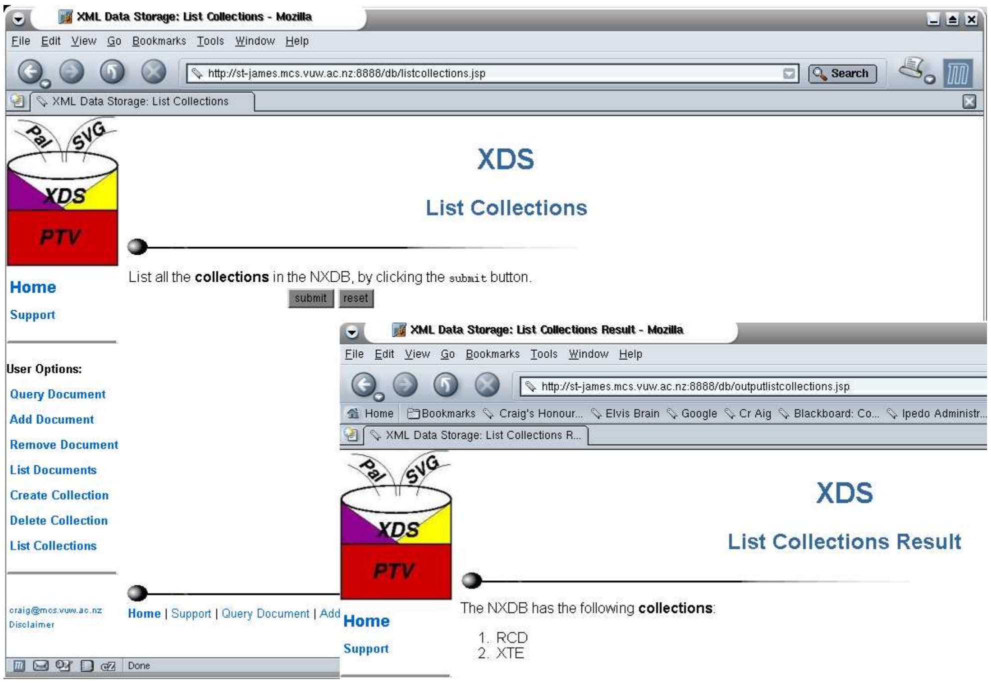
### 6.2.3 XDS in Action

This section shows XDS in action. It demonstrates the following five steps:

1. List the available collections in XDS.
2. Create a collection for PAL program trace files.
3. Add a PAL document to the PAL collection.
4. List the documents in the PAL collection.
5. Query the PAL document from the PAL collection.

#### 1. List Collections

Figure 6.4 lists all the collections in XDS. Once the user has clicked the submit button, the user request is executed. The output of the user request is shown in the bottom right hand corner of figure 6.4. It lists the current collections in XDS. These include RCD and XTE.


XML Data Storage: List Collections - Mozilla
http://st-james.mcs.vuw.ac.nz:8888/db/listcollections.jsp
XDS
List Collections
Home
Support
User Options:
Query Document
Add Document
Remove Document
List Documents
Create Collection
Delete Collection
List Collections
submit
reset
XML Data Storage: List Collections Result - Mozilla
http://st-james.mcs.vuw.ac.nz:8888/db/outputlistcollections.jsp
List Collections Result

List all the **collections** in the NXDB, by clicking the `submit` button.

The NXDB has the following **collections**:

1. RCD
2. XTE

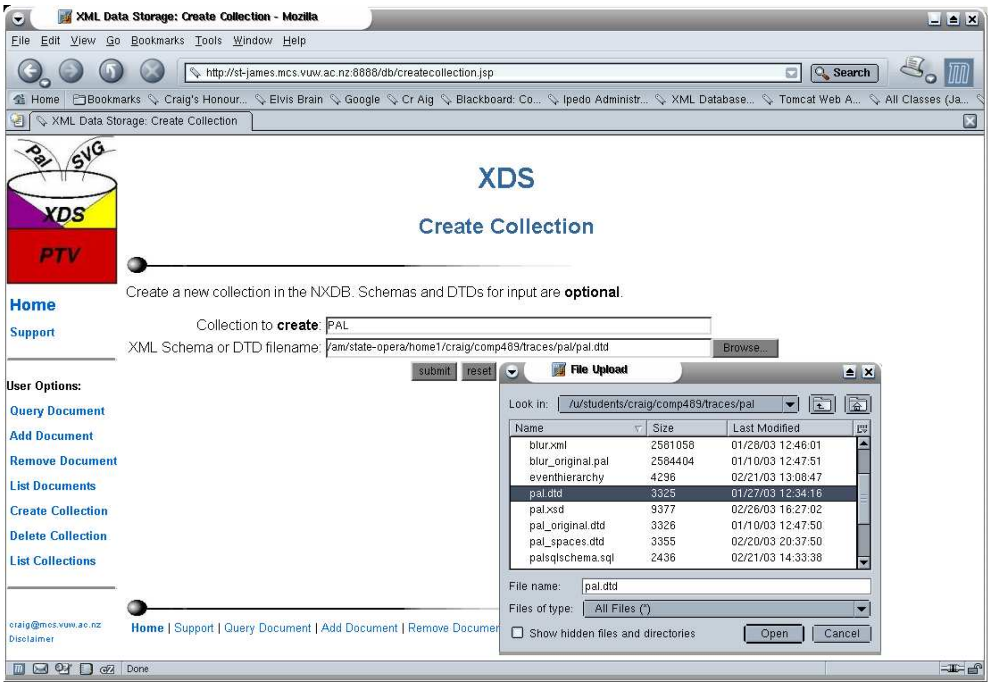
Figure 6.4: List all the collections in XDS.

### 2. Create PAL Collection

Figure 6.5 shows creating a collection in XDS. The PAL collection is created with the PAL DTD. The PAL DTD file is uploaded by the browse option. Once the user clicks the submit button, the uploaded file can then be encoded using SOAP and sent to the database. Upon completion of the method a response message is returned to the user browser. It shows the name of the collection, and the name of the DTD. If there is no DTD associated with the collection then the response will just return the name of the collection. If the method was unsuccessful at creating the collection, an error message will be displayed. Note, it is possible to create a collection without a DTD, or an XML Schema. To do this, the user does not upload a DTD, or an XML Schema.

### 3. Add a Document to the PAL Collection

Figure 6.6 shows adding a document to the PAL collection. The user first types the name of the collection, then browses the file to upload. Once the user clicks submits, the document is encoded in SOAP, and sent across the network to the database. When a document is added to the Ipedo XML Database, it is validated against its DTD, or XML Schema, if it has one. If it does not have a DTD or an XML Schema, the document is added, so long as the user typed in the an existing collection name. Once the document is added, a response is returned to the users browser. The message states whether or not the document was added. If the document was added, the document id number is displayed.


XML Data Storage: Create Collection - Mozilla
File Edit View Go Bookmarks Tools Window Help
Search
Home Bookmarks Craig's Honour... Elvis Brain Google Cr Aig Blackboard: Co... Ipedo Administr... XML Database... Tomcat Web A... All Classes (Ja...
XML Data Storage: Create Collection
Pal SVG XDS PTV
Home
Support
User Options:
Query Document
Add Document
Remove Document
List Documents
Create Collection
Delete Collection
List Collections
craig@mcs.vuw.ac.nz
Disclaimer
Browse...
submit reset
File Upload
Open Cancel
Done

http://st-james.mcs.vuw.ac.nz:8888/db/createcollection.jsp

XDS

Create Collection

Create a new collection in the NXDB. Schemas and DTDs for input are **optional**.

Collection to **create**: PAL

XML Schema or DTD filename: /am/state-opera/home1/craig/comp489/traces/pal/pal.dtd

Home | Support | Query Document | Add Document | Remove Documen

Look in: /u/students/craig/comp489/traces/pal

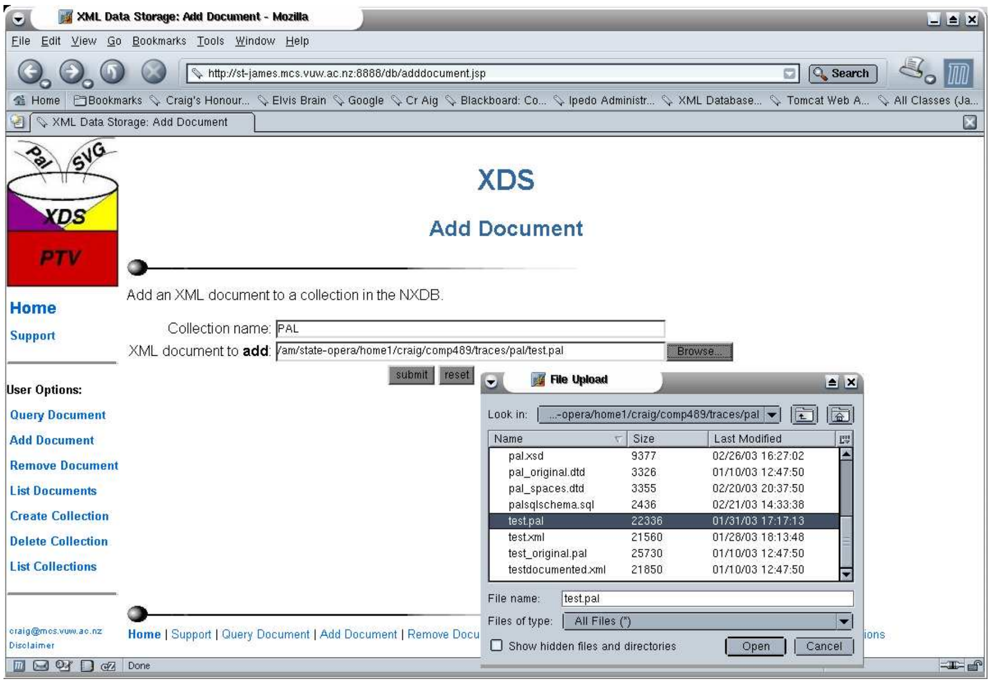
| Name | Size | Last Modified |
|---|---|---|
| blur.xml | 2581058 | 01/28/03 12:46:01 |
| blur_original.pal | 2584404 | 01/10/03 12:47:51 |
| eventhierarchy | 4296 | 02/21/03 13:08:47 |
| pal.dtd | 3325 | 01/27/03 12:34:16 |
| pal.xsd | 9377 | 02/26/03 16:27:02 |
| pal_original.dtd | 3326 | 01/10/03 12:47:50 |
| pal_spaces.dtd | 3355 | 02/20/03 20:37:50 |
| palsqlschema.sql | 2436 | 02/21/03 14:33:38 |

File name: pal.dtd

Files of type: All Files (*)

Show hidden files and directories

Figure 6.5: Create a PAL collection in XDS.


XML Data Storage: Add Document - Mozilla
File Edit View Go Bookmarks Tools Window Help
Search
Home Bookmarks Craig's Honour... Elvis Brain Google Cr Aig Blackboard: Co... Ipedo Administr... XML Database... Tomcat Web A... All Classes (Ja...
XML Data Storage: Add Document
Pal SVG XDS PTV
Home
Support
User Options:
Query Document
Add Document
Remove Document
List Documents
Create Collection
Delete Collection
List Collections
craig@mcs.vuw.ac.nz
Disclaimer
Browse...
submit reset
File Upload
Open Cancel
Done

http://st-james.mcs.vuw.ac.nz:8888/db/adddocument.jsp

XDS

Add Document

Add an XML document to a collection in the NXDB.

Collection name: PAL

XML document to **add**: /am/state-opera/home1/craig/comp489/traces/pal/test.pal

Home | Support | Query Document | Add Document | Remove Docu

Look in: ...-opera/home1/craig/comp489/traces/pal

| Name | Size | Last Modified |
|---|---|---|
| pal.xsd | 9377 | 02/26/03 16:27:02 |
| pal_original.dtd | 3326 | 01/10/03 12:47:50 |
| pal_spaces.dtd | 3355 | 02/20/03 20:37:50 |
| palsqlschema.sql | 2436 | 02/21/03 14:33:38 |
| test.pal | 22336 | 01/31/03 17:17:13 |
| test.xml | 21560 | 01/28/03 18:13:48 |
| test_original.pal | 25730 | 01/10/03 12:47:50 |
| testdocumented.xml | 21850 | 01/10/03 12:47:50 |

File name: test.pal

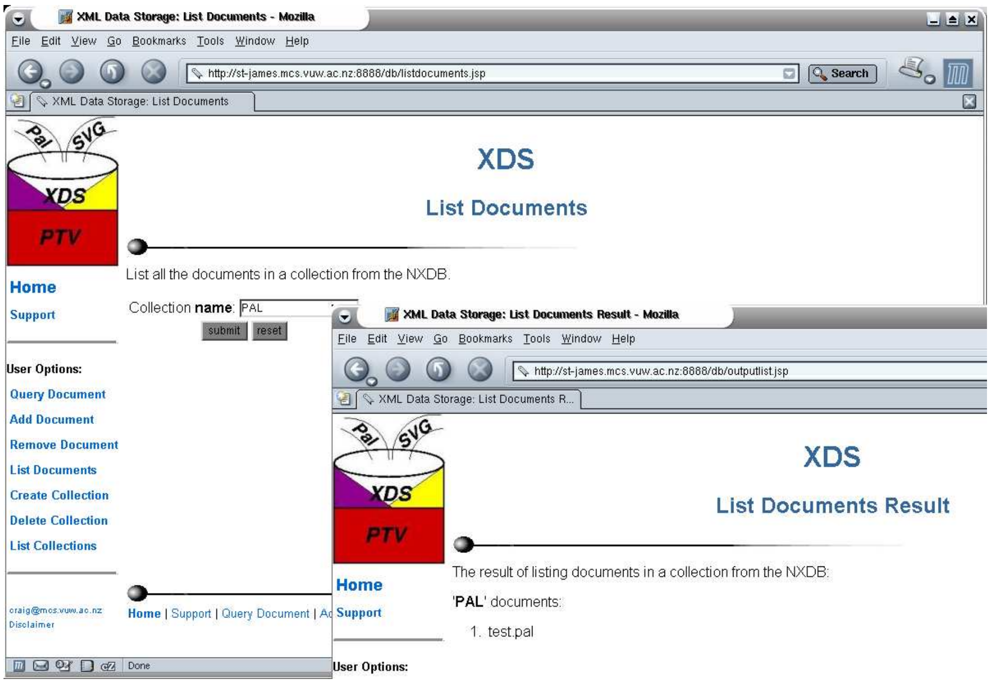
Files of type: All Files (*)

Show hidden files and directories

Figure 6.6: Add a document to the PAL collection.

### 4. List the Documents in the PAL Collection

Figure 6.7 lists the documents in the PAL collection. The user first types the name of the collection, and then clicks the submit button. Once submitted, the list of documents that are in the PAL collection are displayed. This can be seen in the bottom right hand corner of figure 6.7. It lists the recently added document, `test.pal`.


XML Data Storage: List Documents - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://st-james.mcs.vuw.ac.nz:8888/db/listdocuments.jsp
Search
XML Data Storage: List Documents
Pal SVG XDS PTV
Home
Support
User Options:
Query Document
Add Document
Remove Document
List Documents
Create Collection
Delete Collection
List Collections
craig@mcs.vuw.ac.nz
Disclaimer
XDS
List Documents
submit
reset
Home | Support | Query Document | Ad
Done
XML Data Storage: List Documents Result - Mozilla
http://st-james.mcs.vuw.ac.nz:8888/db/outputlist.jsp
XML Data Storage: List Documents R...
XDS
List Documents Result

List all the documents in a collection from the NXDB.

Collection **name**: PAL

The result of listing documents in a collection from the NXDB:

'**PAL**' documents:

1. test.pal

Figure 6.7: List the documents in the PAL collection.

### 5. Query the PAL Document from the PAL Collection

Figure 6.8 shows querying a document in the PAL collection. The user has two options to enter an XQuery query, either:

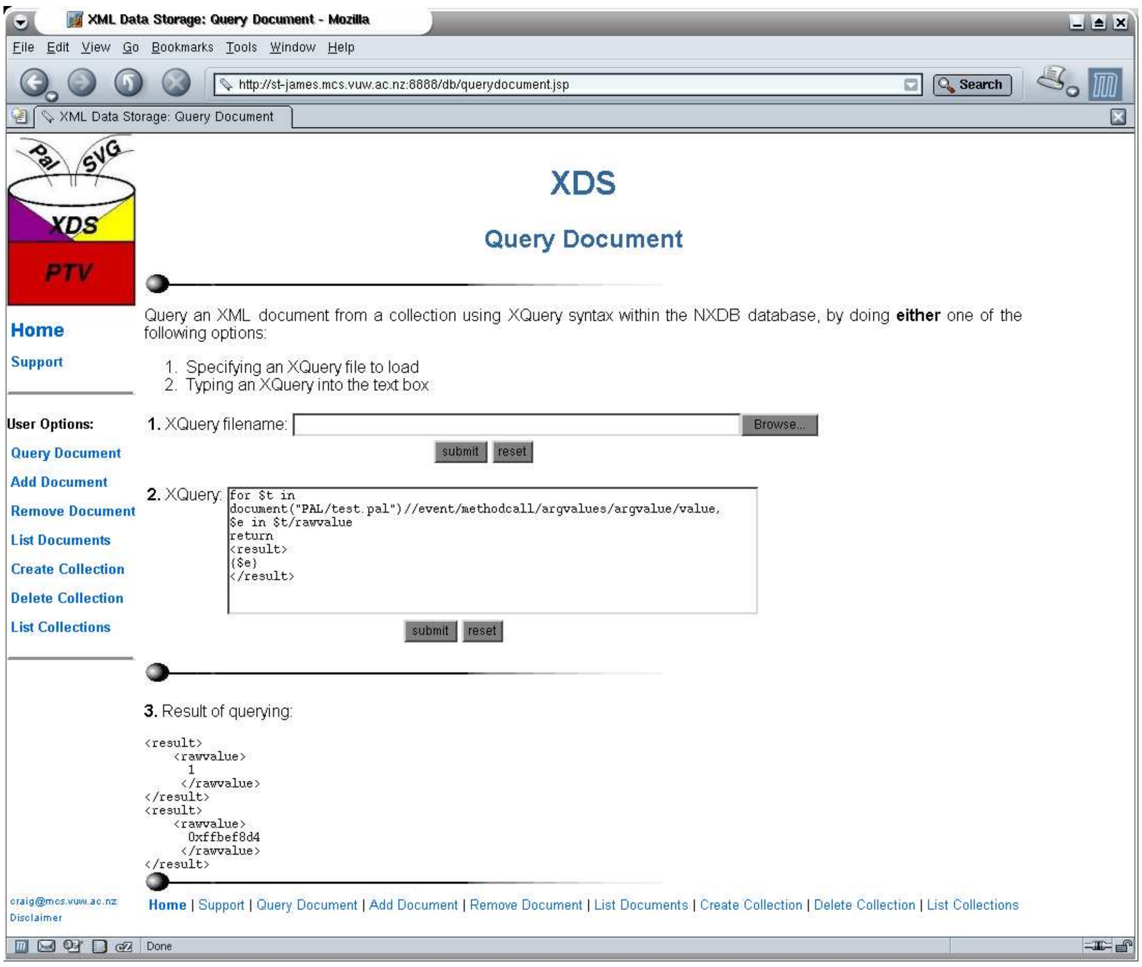
1. Upload the XQuery file.
2. Type the Xquery query into the text box.

Once the user clicks the submit button the query is executed and the results are displayed further down the page. This is so that another query can be generated without going to another page. The following query is used in figure 6.8. It retrieves all the `rawvalue` elements from the `test.pal` file. The result of the query, is displayed as well.

```
for $t in document("PAL/test.pal")//event/methodcall/argvalues/argvalue/value,
$e in $t/rawvalue
return
<result>
{$e}
</result>
```

```
Result of query:

<result>
    <rawvalue>
      1
     </rawvalue>
</result>
<result>
    <rawvalue>
      0xffbef8d4
     </rawvalue>
</result>
```



XDS

Query Document

Query an XML document from a collection using XQuery syntax within the NXDB database, by doing **either** one of the following options:

1. Specifying an XQuery file to load
2. Typing an XQuery into the text box

**1.** XQuery filename:

**2.** XQuery:

```
for $t in
document("PAL/test.pal")//event/methodcall/argvalues/argvalue/value,
$e in $t/rawvalue
return
<result>
{$e}
</result>
```

**3.** Result of querying:

```
<result>
    <rawvalue>
      1
     </rawvalue>
</result>
<result>
    <rawvalue>
      0xffbef8d4
     </rawvalue>
</result>
```

Home | Support | Query Document | Add Document | Remove Document | List Documents | Create Collection | Delete Collection | List Collections

Figure 6.8: Query a document from the PAL Collection.

## 6.3 VARE Integration

This section describes how XDS can be integrated with VARE. It then discusses how it could be linked up to other existing VARE tools.

### 6.3.1 SOAP Control

One of the underlying technologies of VARE is to use SOAP to transport data over a network. All the functionality that is provided in section 6.2.2 uses SOAP to transfer XML data. Figure 6.9 is an example of using SOAP in the XDS environment. It shows a SOAP request message, for a user who is "craig" with password "craig" to query a document by invoking the `executeXQuery` method. This is the same query from figure 6.8. The query has been encoded using SOAP. It shows the HTTP header (lines 1-5), the SOAP envelope (lines 8-28), and the SOAP body (lines 12-27). The SOAP message was captured using the nc (netcat) program. It uses network sockets from the command line to listen to incoming messages on a port.

```
1.  POST /soap HTTP/1.0
2.  Host: wakefield
3.  Content-Type: text/xml; charset=utf-8
4.  Content-Length: 703
5.  SOAPAction: ""
6.
7.  <?xml version='1.0' encoding='UTF-8'?>
8.  <SOAP-ENV:Envelope
9.   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
10.  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
11.  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
12. <SOAP-ENV:Body>
13. <ns1:executeXQuery
14.  xmlns:ns1="urn:IXSOAPServer"
15.  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
16. <XQueryStr xsi:type="xsd:string">
17. for $t in document("PAL/test.pal")//event/methodcall/argvalues/argvalue/value,
18. $e in $t/rawvalue
19. return
20. <result>
21. {$e}
22. </result>
23. </XQueryStr>
24. <user xsi:type="xsd:string">craig</user>
25. <password xsi:type="xsd:string">craig</password>
26. </ns1:executeXQuery>
27. </SOAP-ENV:Body>
28. </SOAP-ENV:Envelope>
```

Figure 6.9: A SOAP request to query test.pal in the PAL collection.

Figure 6.10 shows the response of the SOAP request, from figure 6.9. In this case the HTTP header (lines 1-6) was “OK”, meaning that there was a response. The content of the encoded SOAP message will determine if the method invocation was successful or not. The result of the query is shown in the SOAP body (lines 13-30).

```
1.  HTTP/1.1 200 OK
2.  Content-Type: text/xml; charset=utf-8
3.  Content-Length: 634
4.  Date: Mon, 24 Feb 2003 00:32:56 GMT
5.  Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)
6.  Set-Cookie: JSESSIONID=C2B99C944885E2262F3FAB56F54CF622;Path=/soap
7.
8.  <?xml version='1.0' encoding='UTF-8'?>
9.  <SOAP-ENV:Envelope
10. xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
11. xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
12. xmlns:xsd="http://www.w3.org/1999/XMLSchema">
13. <SOAP-ENV:Body>
14. <ns1:executeXQueryResponse
15.  xmlns:ns1="urn:IXSOAPServer"
16.  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
17. <return xsi:type="xsd:string">
18. <result>
19.     <rawvalue>
20.       1
21.      </rawvalue>
22. </result>
23. <result>
24.     <rawvalue>
25.       0xffbef8d4
26.      </rawvalue>
27. </result>
28. </return>
29. </ns1:executeXQueryResponse>
30. </SOAP-ENV:Body>
31. </SOAP-ENV:Envelope>
```

Figure 6.10: A SOAP response of querying test.pal in the PAL collection.

### 6.3.2 Information For Visualisations

There are various types of information that could be extracted from the database using XQuery. The advantage of this approach, is that only relevant information is required from the native XML database to create a visualisation. The whole program trace file is not required. This may mean parsing the results of a query, to create a visualisation will improve performance. This is because the XML fragment is smaller than the entire XML document.

#### Class Diagram

Figure 6.11 shows a query of relevant class information, to create a UML class diagram from an RCD file. It includes the name of the class, its super-classes, methods, and fields. Arguments for the methods and the types of the fields could be retrieved as well. Figure 6.12 shows the results from exeucting the XQuery query. For each XQuery query that uses an RCD or XTE file, has to declare the appropriate namespace. The results show three classes which could be used to create the UML class diagram from figure 5.11.

```
namespace rcd = "http://www.mcs.vuw.ac.nz/renata/rcd"
for \$t in document("RCD/test_classes.rcd")//rcd:class
return
<class>
 {$t/rcd:classname}
 {$t/rcd:superclassname}
 {$t//rcd:methodname}
 {$t//rcd:fieldname}
</class>
```

Figure 6.11: Information for a class digram

### 6.3.3 Engine Tools

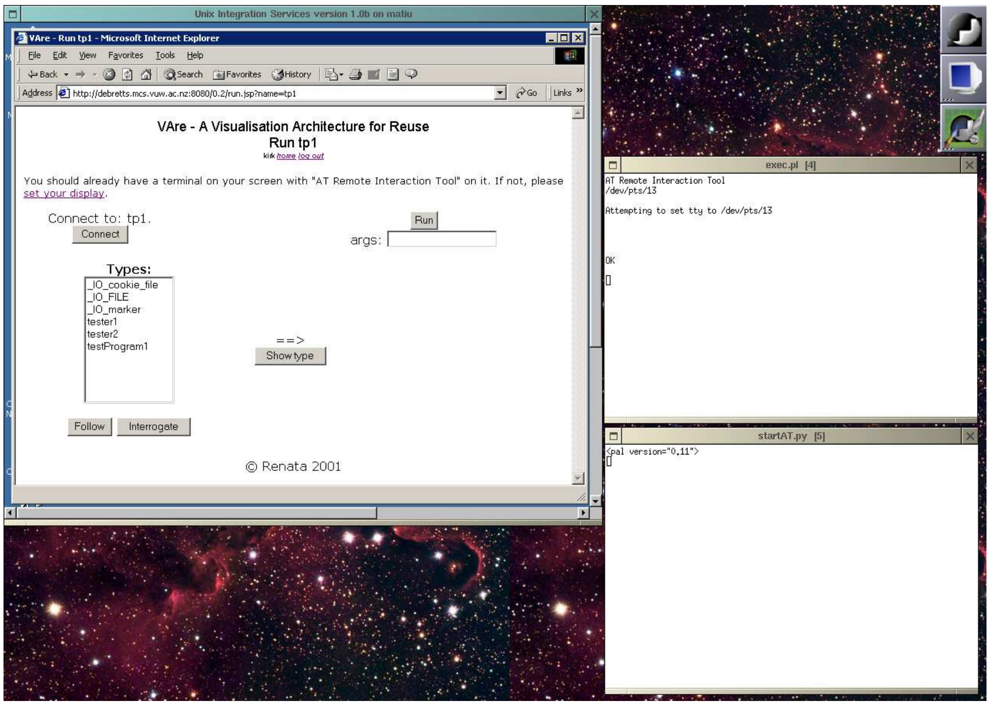
Abstraction Tool (AT) is an Engine from the VARE architecture, designed by Mike McGavin for his Honours Project [59]. It is a prototype utility that has been developed to extract information from applications and present the information, using PAL, so that visualisations tools can visually display the information to a developer. AT examines programs written in C++, using the GNU Debugger (GDB). It is written in the Python scripting language. The main tasks of AT are to drive GDB, and to output XML based on what was seen during execution. AT also uses SOAP for remote method invocation, to allow AT to be controlled by another application.

The left hand side of figure 6.13 shows a user that has selected a program to explore, and there are two windows on the right hand side of the screen. The upper window is used to allow the user to interact with the program being examined, if it has output or requires input. The lower window is the PAL output.

AT could be integrated into XDS by having a built in controller to send the SOAP messages to the XDS database, since it also has a SOAP layer. The other option is for the developer to save the PAL files to the local file-system and then interact with XDS through the web interface. The first option would be preferred. AT is written in Python, while XDS is in Java, so some configuration would be required for them to communicate togther. Once the first option is implemented, then less hand holding or user interaction would be required from the developers aspect to create a program traces for visualisation.

```
<class xmlns:rcd="http://www.mcs.vuw.ac.nz/renata/rcd">
    <rcd:classname>MainClass</rcd:classname>
    <rcd:superclassname>java.lang.Object</rcd:superclassname>
    <rcd:methodname>init</rcd:methodname>
    <rcd:methodname>doOperation</rcd:methodname>
    <rcd:methodname>close</rcd:methodname>
    <rcd:methodname>main</rcd:methodname>
    <rcd:fieldname>_field</rcd:fieldname>
    <rcd:fieldname>_thread</rcd:fieldname>
</class>
<class xmlns:rcd="http://www.mcs.vuw.ac.nz/renata/rcd">
    <rcd:classname>TestClass2</rcd:classname>
    <rcd:superclassname>java.lang.Object</rcd:superclassname>
    <rcd:methodname>method1</rcd:methodname>
    <rcd:methodname>getSize</rcd:methodname>
    <rcd:methodname>myRcd:StaticMethod</rcd:methodname>
    <rcd:fieldname>_field1</rcd:fieldname>
    <rcd:fieldname>_field2</rcd:fieldname>
    <rcd:fieldname>_field3</rcd:fieldname>
</class>
<class xmlns:rcd="http://www.mcs.vuw.ac.nz/renata/rcd">
    <rcd:classname>MyThreadClass</rcd:classname>
    <rcd:superclassname>java.lang.Thread</rcd:superclassname>
    <rcd:methodname>run</rcd:methodname>
    <rcd:methodname>stopThread</rcd:methodname>
    <rcd:methodname>continueThread</rcd:methodname>
    <rcd:methodname>getCount</rcd:methodname>
    <rcd:fieldname>_count</rcd:fieldname>
    <rcd:fieldname>_message</rcd:fieldname>
    <rcd:fieldname>_continue</rcd:fieldname>
</class>
```

Figure 6.12: Result information for a class diagram.


Unix Integration Services version 1.0b on matiu
VAre - Run tp1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://debretts.mcs.vuw.ac.nz:8080/0.2/run.jsp?name=tp1
Go Links
Connect
Run
Show type
Follow
Interrogate
exec.pl [4]
startAT.py [5]

VAre - A Visualisation Architecture for Reuse
Run tp1
kirk home log out

You should already have a terminal on your screen with "AT Remote Interaction Tool" on it. If not, please set your display.

Connect to: tp1.

args:

Types:
_IO_cookie_file
_IO_FILE
_IO_marker
tester1
tester2
testProgram1

==>

© Renata 2001

```
AT Remote Interaction Tool
/dev/pts/13

Attempting to set tty to /dev/pts/13


OK
```

```
<pal version="0.11">
```

Figure 6.13: Abstraction Tool (AT) interfaces.

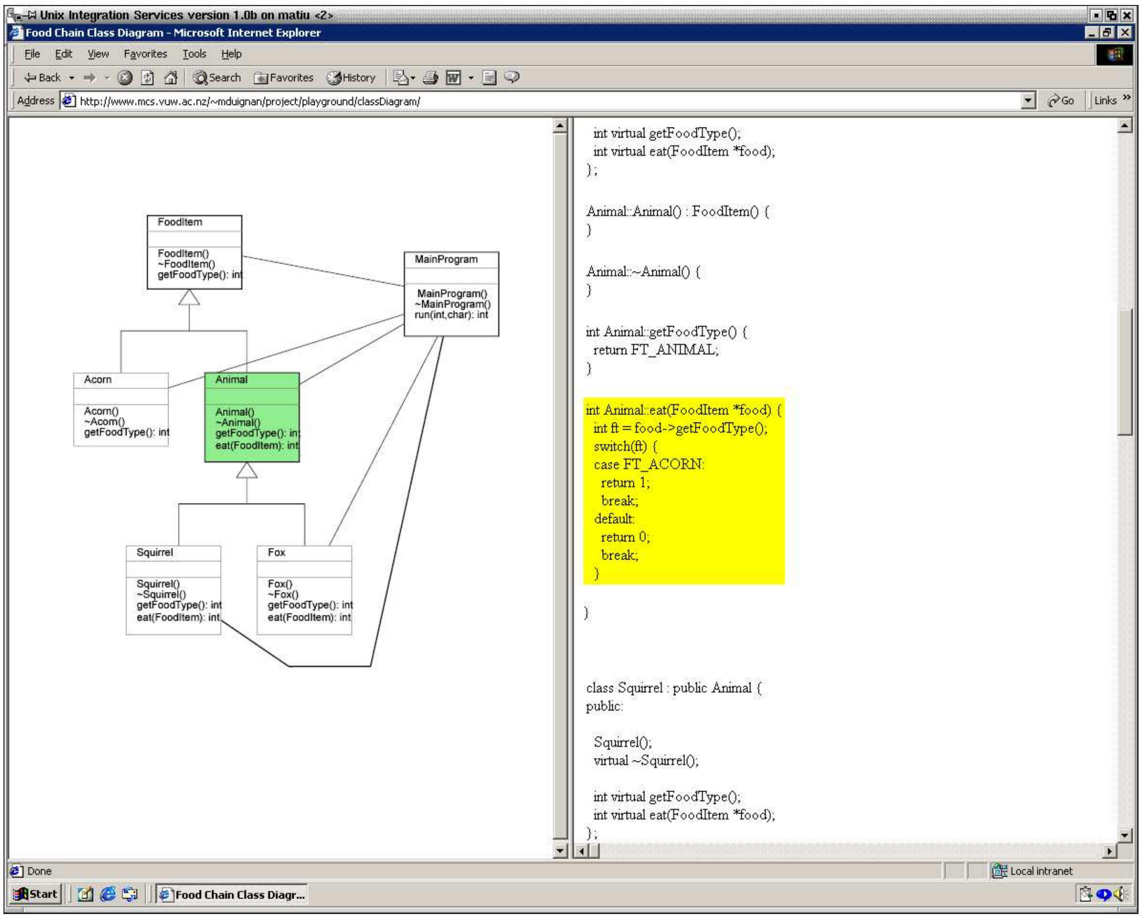
### 6.3.4 Visualisation Tools

Blur is a Transformer from the VARE architecture and was designed by Matthew Duignan for his Masters Thesis [23]. Blur takes a PAL file and transforms it into SVG visualisations. It is implemented as a Java Servlet running in Apache Tomcat and can be accessed over the Internet.

Figure 6.14 shows a SVG UML interactive class diagram of figure 5.2. When the mouse covers a piece of code in the right hand side frame, the left hand side highlights the appropriate class or method in the UML diagram. This is a helpful tool for developers, because they can have a clear understanding of the class diagram, and how the code works.

Figure 6.15 shows a SVG UML sequence diagram of figure 5.2. This is also helpful for the developer, as they can see the sequence of interactions during the execution of the program.

Blur could be integrated into XDS by having a SOAP mechanism at the Blur side, to retrieve the program trace data from the database. Currently Blur uses all of the program trace file to generate a visualisation. This is not an efficient process because if the program trace file is very large, it will take a long time to parse, and send across the network. The better approach would be to select smaller parts of the program trace file, send them across the network using SOAP, and then generate visualisations. Blur also does not store any of its visualisations. If Blur could communicate with XDS then it could store the SVG files in the native XML database. The SVG files that Blur can create, do not require a DTD or an XML Schema.



```
  int virtual getFoodType();
  int virtual eat(FoodItem *food);
};

Animal::Animal() : FoodItem() {
}

Animal::~Animal() {
}

int Animal::getFoodType() {
  return FT_ANIMAL;
}

int Animal::eat(FoodItem *food) {
  int ft = food->getFoodType();
  switch(ft) {
  case FT_ACORN:
    return 1;
    break;
  default:
    return 0;
    break;
  }

}



class Squirrel : public Animal {
public:

  Squirrel();
  virtual ~Squirrel();

  int virtual getFoodType();
  int virtual eat(FoodItem *food);
};
```

Figure 6.14: SVG interactive class diagram of figure 5.2.

## 6.4 Summary

In this chapter we have shown that it is not worthwhile using a relational database to store or retrieve XML program trace files. We have presented XDS, an XML Data Storage environment for storing and retrieving XML based program traces. We have described its architecture, and user interface design. Next we demonstrated XDS in action, by creating a PAL collection, adding a program trace file to the PAL collection, and then querying that document, for relevant information. We then showed how XDS can be integrated into the VARE architecture. We showed how XDS sends and receives data, using SOAP requests and responses. Finally, we presented some of VAREs Engine and Tranformer tools. We then described, how XDS could be combined with these tools.

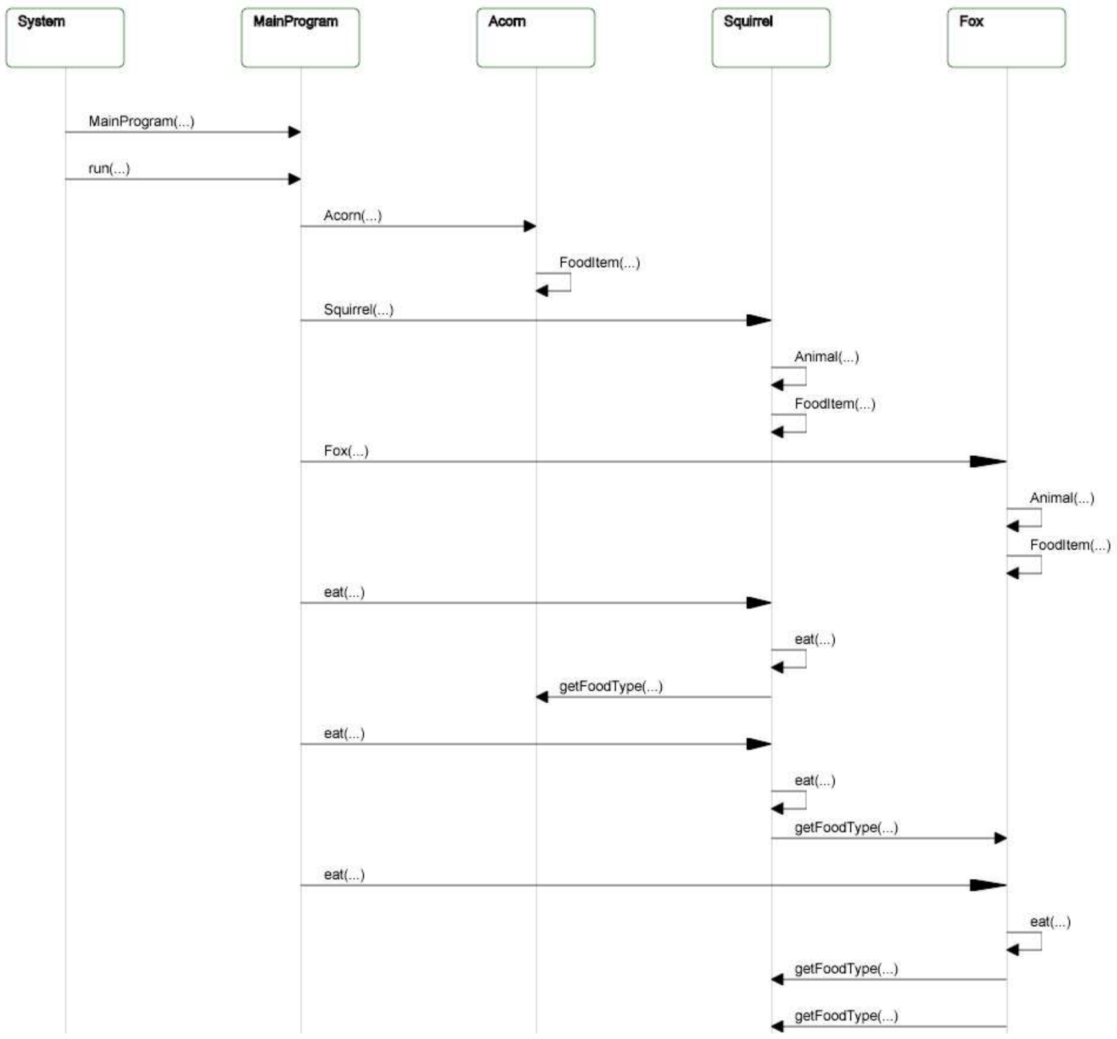


Figure 6.15: SVG sequence diagram of figure 5.2.

# Chapter 7

# Conclusion

In this report we have discussed supporting code reuse with web-based repositories and software visualisation, and presented XDS: An XML Data Storage environment. XDS is used for storing and retrieving XML based program traces of reusable software components in a native XML database over the web. Program traces can then be transformed into appropriate visualisations for a developer to understand how a component works, and whether or not it can be reused in their software program.

## 7.1 Contributions

We first identified a list of the types of information for visualising reusable components. This was followed by a set of requirements that described what a program trace should support for software visualisation of reusable components.

Next we evaluated a number of XML technologies that can be used in VARE and concluded that an XML Schema is a better approach than a DTD for document structure. An XML Schema has more functionality than a DTD and makes use of namespaces. It is best to store XML oriented data in a native XML database because it is based on the XML data model, and provides XML functionality such as XQuery. This is in contrast to storing XML data in a file system, as BLOBs or in a relational database, because they do not support any XML functionality. There are also many proprietary native XML databases that exist. Three were evaluated and provided similar functionality, but the open source native XML databases are further behind in development. Although native XML database development is in its infancy we expect there will be more powerful native XML databases with better functionality in years to come.

We then improved our existing trace language PAL. We had realised that there were major costs in using PAL. To overcome this we created version 1.1. We also developed RCD 2.0 and XTE 2.0, that use DTDs and namespaces. XML Schemas were developed for PAL, RCD and XTE so that future program traces can support this superior schema language. Relational schemas for each of the program traces were created manually by hand, but it did not prove worthwhile converting a DTD into a relational schema. The cost of inserting XML documents into relational databases and then querying the program traces was extremely high. If a relational database was desired, it would be best to design the relational program trace schema from scratch.

Finally, we implemented and demonstrated the XDS environment where we used the Ipedo

Native XML Database server to store the program traces. We also created a SOAP client that could communicate with the database server over the web. The client has functionality to create, delete and list collections. It could also add, remove, list and query documents. It uses the defacto XQuery standard for querying information in the database.

## 7.2 Future Work

In the future, we plan to make the XDS environment integrated and linked to Engines and Transformers from VARE, as it would provide an overall system for a user. Making XDS generalised would mean that it could also be linked to other lightweight web-based Computer Aided Software Engineering (CASE) tools such as Seek[48] for creating UML sequence diagrams and NutCase[54] for creating UML class diagrams.

We want to be able to create UML diagrams of the PAL, RCD and XTE program trace DTDs, and XML Schemas using another approach [80, 9]. Having completed this step will enable other users of VARE besides ourselves to understand how these program trace languages work. It would also be beneficial to visualise our own system for understanding and using the same method.

We plan to add extra functionality to XDS so that it can create automatic relational database schemas from either an XML DTD or an XML Schema, rather than manually computing it. We want to generate program traces that can be validated against an XML Schema. Using indexes on commonly used program trace files in the native XML database would improve the performance in querying the XML documents, hence the transformation process could be completed faster. Having built in queries that a user could select, like class diagrams and sequence diagrams, would also help performance.

One other key important aspect that will be incorporated into the XDS environment is the ability to update XML documents, so that they can be changed and modified in the future. Currently in principle there should be no need to change a program trace once it has been created, but it may be useful to improve a visualisation. This can either be done by using the proprietary API for updates that most native XML databases provide, or by waiting until the update functionality is added to XQuery, which most native XML databases now support.

# Appendix A

# Proccess Abstraction Language

This is a definition of the Process Abstraction Language (PAL) Document Type Definition (DTD) version 1.1, the previous version was 1.0. The changes are located in section A.1. The next sections describe the following aspects of PAL, the top level element (section A.2.1), static information elements (section A.2.2) and dynamic information elements (section A.2.3).

## A.1 PAL DTD Version 1.1 Changes

The changes that have been made to PAL version 0.14 by Matthew Duignan, Stuart Marshall and myself are as follows:

- Elements (unlike attributes) have to be #PCDATA (parsed character data) and not #CDATA (character data):

  ```
  <!ELEMENT rawvalue (#CDATA)>
  ```

  is changed to

  ```
  <!ELEMENT rawvalue (#PCDATA)>
  ```

- Enumerated type attributes needed to be defined without quotations and the #DEFAULT removed:

  ```
  <!ATTLIST typespec classification ("class"|"alias"|"generic") #DEFAULT "generic")>
  ```

  is changed to

  ```
  <!ATTLIST typespec classification (class|alias|generic) "generic")>
  ```

- Attribute values of type ID must conform to the Name production rule:

  ```
  Name    ::=    (Letter | '_' | ':') (NameChar?)*
  ```

- Empty elements need to be defined in the dtd without brackets:

  ```
  <!ELEMENT modifier (EMPTY)>
  ```

is changed to

```
<!ELEMENT modifier EMPTY>
```

- The attribute `callerpositionref` in `methodcall` is being output as callpositionref by AT and it doesn't seem to be an IDREF hence change it to CDATA:

  ```
  <!ATTLIST methodcall callerpositionref IDREF #IMPLIED>
  ```

  is changed to

  ```
  <!ATTLIST methodcall callerpositionref CDATA #IMPLIED>
  ```

- The attribute `methodidref` in `methodcall` needs to be changed because sometimes objects from language libraries are not required to be outputted hence the `methodcall` element sometimes has no id to refer to:

  ```
  <!ATTLIST methodcall methodidref IDREF #REQUIRED>
  ```

  is changed to

  ```
  <!ATTLIST methodcall methodidref IDREF #IMPLIED>
  ```

## A.2 PAL DTD Version 1.1

### A.2.1 Top Level Element

Every XML document requires a top level element to surround nested elements. The `pal` element is the top level element and it represents a PAL session which is an exeuction trace generated from an engine.

```
<!ELEMENT pal (typecollection?,(type|execution)*)>
<!ATTLIST pal version    CDATA #FIXED "1.1">
<!ATTLIST pal sessionid  CDATA #IMPLIED>
```

### A.2.2 Static Information Elements

This section describes the element which are used to explain static information about a program's execution.

```
<!ELEMENT typecollection (typespec*)>

<!ELEMENT typespec (context*)>
<!ATTLIST typespec idref IDREF #REQUIRED>
<!ATTLIST typespec name  CDATA #IMPLIED>
<!ATTLIST typespec classification (class|alias|generic) "generic">

<!ELEMENT type (context*,(aliasdata?|classdata?))>
<!ATTLIST type typeid ID    #REQUIRED>
```

```
<!ATTLIST type name   CDATA #IMPLIED>

<!ELEMENT context EMPTY>
<!ATTLIST context contextname  CDATA  #REQUIRED>
<!ATTLIST context contextvalue CDATA  #REQUIRED>

<!ELEMENT aliasdata (typeref)>

<!ELEMENT classdata (superclasses?,methods?,variables?)>

<!ELEMENT typeref (modifier*)>
<!ATTLIST typeref typeidref IDREF #REQUIRED>

<!ELEMENT superclasses (superclass+)>

<!ELEMENT methods (method+)>

<!ELEMENT variables (variable+)>

<!ELEMENT modifier EMPTY>
<!ATTLIST modifier name CDATA #REQUIRED>

<!ELEMENT superclass (typeref)>
<!ATTLIST superclass access (public|package|protected|private) "public">

<!ELEMENT method (modifier*,typeref?,argument*)>
<!ATTLIST method methodid ID     #REQUIRED>
<!ATTLIST method name     CDATA  #IMPLIED>
<!ATTLIST method access (public|package|protected|private) "public">

<!ELEMENT variable (modifier*, typeref?)>
<!ATTLIST variable variableid ID    #REQUIRED>
<!ATTLIST variable name       CDATA #IMPLIED>
<!ATTLIST variable access (public|package|protected|private) "private">

<!ELEMENT argument (modifier*,typeref?)>
<!ATTLIST argument argumentid ID    #REQUIRED>
<!ATTLIST argument name       CDATA #IMPLIED>
```

### A.2.3 Dynamic Information Elements

This section describes the element which are used to explain dynamic information about a program's execution. Dynamic information is made up of an execution element, containing events that occur during the process execution. An exection can also contain static type information.

```
<!ELEMENT execution ((type|event)*)>
<!ATTLIST execution execid ID #IMPLIED>

<!ELEMENT event
```

```
 (processbegin|processend|newclassinstance|
 deleteclassinstance|methodcall|methodreturn)>
<!ATTLIST event eventid ID #REQUIRED>

<!ELEMENT processbegin (argstring?)>

<!ELEMENT argstring (#PCDATA)>

<!ELEMENT processend EMPTY>

<!ELEMENT newclassinstance EMPTY>
<!ATTLIST newclassinstance classinstanceid    ID    #REQUIRED>
<!ATTLIST newclassinstance typeidref          IDREF #REQUIRED>
<!ATTLIST newclassinstance stackmethodidref   IDREF #IMPLIED>

<!ELEMENT deleteclassinstance EMPTY>
<!ATTLIST deleteclassinstance instanceidref   IDREF #REQUIRED>

<!ELEMENT methodcall (argvalues?)>
<!ATTLIST methodcall methodcallid             ID    #REQUIRED>
<!ATTLIST methodcall methodidref              IDREF #IMPLIED>
<!ATTLIST methodcall classinstanceidref       IDREF #IMPLIED>
<!ATTLIST methodcall callerclassinstanceidref IDREF #IMPLIED>
<!ATTLIST methodcall callermethodcallidref    IDREF #IMPLIED>
<!ATTLIST methodcall callermethodidref        IDREF #IMPLIED>
<!ATTLIST methodcall callerpositionref        CDATA #IMPLIED>
<!ATTLIST methodcall threadnum                CDATA #IMPLIED>

<!ELEMENT argvalues (argvalue*)>

<!ELEMENT argvalue (value)>
<!ATTLIST argvalue argumentidref IDREF #REQUIRED>

<!ELEMENT methodreturn (returnvalue?)>
<!ATTLIST methodreturn methodcallidref IDREF #REQUIRED>

<!ELEMENT returnvalue (value)>

<!ELEMENT value (abstractvalue|rawvalue)>

<!ELEMENT abstractvalue EMPTY>
<!ATTLIST abstractvalue classification (classref) #REQUIRED>
<!ATTLIST abstractvalue classinstanceidref IDREF  #REQUIRED>

<!ELEMENT rawvalue (#PCDATA)>
```

# Appendix B

# Reusable Component Descriptions

This is a definition of the Reusable Component Descriptions (RCD) DTD version 2.0. RCD documents store static information of a reusable component, where a component is defined as consisting of one or more packages, and each package having one or more classes.

## B.1 RCD DTD Version 2.0 Changes

The changes that have been made to RCD version 1.0 by Stuart Marshall and myself are as follows:

- Added namespaces `rcd:rcd` to each of the elements in the DTD file.
- Changed DOCTYPE declaration from:

  ```
  <!DOCTYPE RCD [    to
  <!DOCTYPE rcd:rcd [
  ```

- Added ? to version and creator attributes in the rcd element:

  ```
  <!ELEMENT rcd:rcd (rcd:version?, rcd:creator?, rcd:package*)>
  ```

- Changed the ordering of nested elements in the following elements: class, method, argument.
- Added a ? to initialiser in the methods element:

  ```
  <!ELEMENT rcd:methods (rcd:initialiser?, rcd:constructor*, rcd:method*)>
  ```

- Added fields element:

  ```
  <!ELEMENT rcd:fields (rcd:field*)>
  ```

- Added ? to initialiser in the methods element:

  ```
  <!ELEMENT rcd:methods (rcd:initialiser?, rcd:constructor*, rcd:method*)>
  ```

- Changed modifiers element from:

```
<!ELEMENT modifiers ()>    to
<!ELEMENT rcd:modifiers EMPTY>
```

- Added separate attributes for the modifiers element as previously they were all written in one element as if they happened to be in an RCD document:

```
<!ATTLIST rcd:modifiers rcd:access (package|private|public|protected) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:static (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:native (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:final (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:abstract (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:synchronized (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:strictFP (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:volatile (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:transient (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:interface (true|false) #REQUIRED>
```

- Changed initialiser element from:

```
<!ELEMENT initialiser ()>    to
<!ELEMENT rcd:initialiser (#PCDATA)>
```

## B.2 RCD DTD Version 2.0

The RCD version 2.0 DTD is as follows:

```
<!DOCTYPE rcd:rcd [

<!ELEMENT rcd:rcd (rcd:version?, rcd:creator?, rcd:package*)>

<!ELEMENT rcd:package (rcd:packagename, rcd:class*)>

<!ELEMENT rcd:class (rcd:classname, rcd:superclassname*, rcd:packagename,
rcd:methods, rcd:fields, rcd:interface*, rcd:dependency*)>

<!ELEMENT rcd:methods (rcd:initialiser?, rcd:constructor*, rcd:method*)>

<!ELEMENT rcd:constructor (rcd:modifiers, rcd:argument*)>

<!ELEMENT rcd:method (rcd:type, rcd:modifiers, rcd:methodname, rcd:argument*)>

<!ELEMENT rcd:fields (rcd:field*)>

<!ELEMENT rcd:field (rcd:type, rcd:fieldname, rcd:modifiers)>

<!ELEMENT rcd:modifiers EMPTY>
<!ATTLIST rcd:modifiers rcd:access (package|private|public|protected) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:static (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:native (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:final (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:abstract (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:synchronized (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:strictFP (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:volatile (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:transient (true|false) #REQUIRED>
<!ATTLIST rcd:modifiers rcd:interface (true|false) #REQUIRED>

<!ELEMENT rcd:argument (rcd:argumentname, rcd:type)>

<!ELEMENT rcd:type (rcd:typename)>

<!ELEMENT rcd:initialiser (#PCDATA)>
<!ELEMENT rcd:creator (#PCDATA)>
<!ELEMENT rcd:version (#PCDATA)>
<!ELEMENT rcd:methodname (#PCDATA)>
<!ELEMENT rcd:argumentname (#PCDATA)>
<!ELEMENT rcd:typename (#PCDATA)>
<!ELEMENT rcd:fieldname (#PCDATA)>
<!ELEMENT rcd:classname (#PCDATA)>
<!ELEMENT rcd:packagename (#PCDATA)>
<!ELEMENT rcd:superclassname (#PCDATA)>
<!ELEMENT rcd:interface (#PCDATA)>
<!ELEMENT rcd:dependency (#PCDATA)>

]>
```

# Appendix C

# eXtensible Trace Executions

This is a definition of the eXtensible Trace Executions (XTE) DTD version 2.0. eXtensible Trace Executions (XTE) stores execution trace information derived dynamically from test driven reusable components.

## C.1 XTE DTD Version 2.0 Changes

The changes that have been made to XTE version 1.0 by Stuart Marshall and myself are as follows:

- Added namespaces `xte:xte` to each of the elements in the DTD file.
- Changed DOCTYPE declaration from:

  ```
  <!DOCTYPE XTE [    to
  <!DOCTYPE xte:xte [
  ```

- Changed the xte element from:

  ```
  <!ELEMENT xte (creator, date, details, execution, object*)> to
  <!ELEMENT xte:xte (xte:creator, xte:date, xte:details?, xte:object*,
  xte:thread*, xte:execution)>
  ```

- Added in a thread element:

  ```
  <!ELEMENT xte:thread (#PCDATA)>
  <!ATTLIST xte:thread xte:threadid ID #REQUIRED>
  ```

- Added in a field element:

  ```
  <!ELEMENT xte:field (xte:fieldname, %VALUE;)>
  ```

- Added separate attributes for the objectcreation element and added a threadid element:

  ```
  <!ELEMENT xte:objectcreation (xte:complextype, xte:parameter*)>
  <!ATTLIST xte:objectcreation xte:eventid ID #REQUIRED>
  <!ATTLIST xte:objectcreation xte:objectid IDREF #REQUIRED>
  <!ATTLIST xte:objectcreation xte:threadid IDREF #REQUIRED>
  ```

- Added separate attributes for the methodcall element and added a threadid element:

```
<!ELEMENT xte:methodcall (xte:methodname, xte:typename, xte:parameter*)>
<!ATTLIST xte:methodcall xte:eventid ID #REQUIRED>
<!ATTLIST xte:methodcall xte:senderid IDREF #REQUIRED>
<!ATTLIST xte:methodcall xte:receiverid IDREF #REQUIRED>
<!ATTLIST xte:methodcall xte:threadid IDREF #REQUIRED>
```

- Added separate attributes for the methodreturn element and added a threadid element, also changed the objectid to #IMPLIED from #REQUIRED as sometimes it was not a necessity to have in the output:

```
<!ELEMENT xte:methodreturn (xte:parameter*, %VALUE;, xte:methodname,
xte:typename)>
<!ATTLIST xte:methodreturn xte:eventid ID #REQUIRED>
<!ATTLIST xte:methodreturn xte:objectid IDREF #IMPLIED>
<!ATTLIST xte:methodreturn xte:threadid IDREF #REQUIRED>
```

- Added separate attributes for the fieldaccess element and added a threadid element:

```
<!ELEMENT xte:fieldaccess (xte:fieldname, %VALUE;)>
<!ATTLIST xte:fieldaccess xte:eventid ID #REQUIRED>
<!ATTLIST xte:fieldaccess xte:objectid IDREF #REQUIRED>
<!ATTLIST xte:fieldaccess xte:threadid IDREF #REQUIRED>
```

- Added separate attributes for the fieldmodification element and added a threadid element:

```
<!ELEMENT xte:fieldmodification (xte:fieldname, xte:oldvalue,
xte:newvalue)>
<!ATTLIST xte:fieldmodification xte:eventid ID #REQUIRED>
<!ATTLIST xte:fieldmodification xte:objectid IDREF #REQUIRED>
<!ATTLIST xte:fieldmodification xte:threadid IDREF #REQUIRED>
```

- Added separate attributes for the exceptionthrow element and added a threadid element:

```
<!ELEMENT xte:exceptionthrow (xte:location, xte:details)>
<!ATTLIST xte:exceptionthrow xte:eventid ID #REQUIRED>
<!ATTLIST xte:exceptionthrow xte:threadid IDREF #REQUIRED>
```

- Added separate attributes for the exceptionthrow element and added a threadid element:

```
<!ELEMENT xte:exceptioncatch (xte:location, xte:details)>
<!ATTLIST xte:exceptioncatch xte:eventid ID #REQUIRED>
<!ATTLIST xte:exceptioncatch xte:threadid IDREF #REQUIRED>
```

- Added separate attributes for the exceptionthrow element and added a threadid element:

```
<!ELEMENT xte:securitypermission (xte:location, xte:details,
xte:permission)>
<!ATTLIST xte:securitypermission xte:eventid ID #REQUIRED>
<!ATTLIST xte:securitypermission xte:threadid IDREF #REQUIRED>
```

- Changed the parameter elements nested elements order from:

```
<!ELEMENT parameter (%TYPE;, argumentname, %VALUE;)> to
<!ELEMENT xte:parameter (%TYPE;, %VALUE;, xte:parametername)>
```

- Changed the objectvalue element from:

```
<!ELEMENT objectvalue ()> to
<!ELEMENT xte:objectvalue EMPTY>
```

- Changed primitivevalue element from:

```
<!ELEMENT primitivevalue (#PCDATA)> to
<!ELEMENT xte:primitivevalue (xte:value)>
```

- Added the value element:

```
<!ELEMENT xte:value (#PCDATA)>
```

## C.2 XTE DTD Version 2.0

The XTE version 2.0 DTD is as follows:

```
<!DOCTYPE XTE [

<!ELEMENT xte:xte (xte:creator, xte:date, xte:details?, xte:object*,
xte:thread*, xte:execution)>

<!ENTITY % TYPE "(xte:complextype | xte:primitivetype)">
<!ENTITY % VALUE "(xte:objectvalue | xte:primitivevalue)">

<!ELEMENT xte:object (xte:complextype, xte:field*)>
<!ATTLIST xte:object xte:objectid ID #REQUIRED>

<!ELEMENT xte:thread (#PCDATA)>
<!ATTLIST xte:thread xte:threadid ID #REQUIRED>

<!ELEMENT xte:field (xte:fieldname, %VALUE;)>

<!ELEMENT xte:execution (xte:objectcreation | xte:methodcall |
xte:methodreturn | xte:fieldaccess | xte:fieldmodification |
xte:exceptionthrow | xte:exceptioncatch | xte:securitypermission)*>

<!ELEMENT xte:objectcreation (xte:complextype, xte:parameter*)>
```

```
<!ATTLIST xte:objectcreation xte:eventid ID #REQUIRED>
<!ATTLIST xte:objectcreation xte:objectid IDREF #REQUIRED>
<!ATTLIST xte:objectcreation xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:methodcall (xte:methodname, xte:typename,
xte:parameter*)>
<!ATTLIST xte:methodcall xte:eventid ID #REQUIRED>
<!ATTLIST xte:methodcall xte:senderid IDREF #REQUIRED>
<!ATTLIST xte:methodcall xte:receiverid IDREF #REQUIRED>
<!ATTLIST xte:methodcall xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:methodreturn (xte:parameter*, %VALUE;, xte:methodname,
xte:typename)>
<!ATTLIST xte:methodreturn xte:eventid ID #REQUIRED>
<!ATTLIST xte:methodreturn xte:objectid IDREF #IMPLIED>
<!ATTLIST xte:methodreturn xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:fieldaccess (xte:fieldname, %VALUE;)>
<!ATTLIST xte:fieldaccess xte:eventid ID #REQUIRED>
<!ATTLIST xte:fieldaccess xte:objectid IDREF #REQUIRED>
<!ATTLIST xte:fieldaccess xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:fieldmodification (xte:fieldname, xte:oldvalue,
xte:newvalue)>
<!ATTLIST xte:fieldmodification xte:eventid ID #REQUIRED>
<!ATTLIST xte:fieldmodification xte:objectid IDREF #REQUIRED>
<!ATTLIST xte:fieldmodification xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:exceptionthrow (xte:location, xte:details)>
<!ATTLIST xte:exceptionthrow xte:eventid ID #REQUIRED>
<!ATTLIST xte:exceptionthrow xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:exceptioncatch (xte:location, xte:details)>
<!ATTLIST xte:exceptioncatch xte:eventid ID #REQUIRED>
<!ATTLIST xte:exceptioncatch xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:securitypermission (xte:location, xte:details,
xte:permission)>
<!ATTLIST xte:securitypermission xte:eventid ID #REQUIRED>
<!ATTLIST xte:securitypermission xte:threadid IDREF #REQUIRED>

<!ELEMENT xte:location (xte:complextype, xte:methodname)>

<!ELEMENT xte:complextype (xte:typename)>
<!ELEMENT xte:primitivetype (xte:typename)>

<!ELEMENT xte:parameter (%TYPE;, %VALUE;, xte:parametername)>

<!ELEMENT xte:oldvalue (%VALUE;)>
<!ELEMENT xte:newvalue (%VALUE;)>
```

```
<!ELEMENT xte:objectvalue EMPTY>
<!ATTLIST xte:objectvalue xte:objectid IDREF #REQUIRED>

<!ELEMENT xte:primitivevalue (xte:value)>

<!ELEMENT xte:creator (#PCDATA)>
<!ELEMENT xte:date (#PCDATA)>
<!ELEMENT xte:details (#PCDATA)>
<!ELEMENT xte:permission (#PCDATA)>
<!ELEMENT xte:typename (#PCDATA)>
<!ELEMENT xte:methodname (#PCDATA)>
<!ELEMENT xte:fieldname (#PCDATA)>
<!ELEMENT xte:parametername (#PCDATA)>
<!ELEMENT xte:value (#PCDATA)>

]>
```

# Bibliography

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[3] Alphaworks. Jinsight ibm alphaworks site, 2003. `http://www.alphaworks.ibm.com/tech/jinsight`.

[4] T. Atwood, D. Barry, J. Duhl, J. Eastman, G. Ferran, D. Jordan, M. Loomis, and D. Wade. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann, San Francisco CA, 1996.

[5] Robert Biddle, Stuart Marshall, John Miller-Williams, and Ewan Tempero. Reuse of debuggers for visualization of reuse. Technical Report CS-TR-98-10, School of Mathematical and Computing Sciences, Victoria University of Wellington, November 1998. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-98-10.abs.html`.

[6] Robert Biddle, John Miller-Williams, and Ewan Tempero. Tools to aid learning reusability. Technical Report CS-TR-98/3, School of Mathematical and Computing Sciences, Victoria University of Wellington, May 1998. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-98-3.abs.html`.

[7] Robert Biddle and Ewan Tempero. Towards tool support for reuse. Technical Report CS-TR-97/7, School of Mathematical and Computing Sciences, Victoria University of Wellington, November 1997. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-97-7.abs.html`.

[8] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. Xquery 1.0: An xml query language. Website, November 2002. World Wide Web Consortium (W3C) `http://www.w3.org/TR/xquery/`.

[9] Grady Booch, Magnus Christerson, Matthew Fuchs, and Jari Koistinen. Uml for xml schema mapping specification, 1999. `http://www.rational.com/media/uml/resources/media/uml_xmlschema33.pdf`.

[10] Ronald Bourret. Mapping dtds to databases. Website, May 2001. `http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html`.

[11] Ronald Bourret. Xml and databases. Website, January 2003. `http://www.rpbourret.com/xml/XMLAndDatabases.htm`.

[12] Ronald Bourret. Xml database products. Website, February 2003. `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`.

[13] Ronald Bourret. Xml namespaces faq. Website, February 2003. `http://www.rpbourret.com/xml/NamespacesFAQ.htm`.

[14] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, May 2000. World Wide Web Consortium (W3C) `http://www.w3.org/TR/SOAP/`.

[15] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (xml) 1.0 (second edition). Website, October 2000. World Wide Web Consortium (W3C) `http://www.w3.org/TR/REC-xml`.

[16] Bernd Bruegge and Allen Dutoit. *Object-Oriented Software Engineering, Conquering Complex and Changing Systems.* Prentice Hall, 2000.

[17] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. Xml query requirements. Website, February 2001. World Wide Web Consortium (W3C) `http://www.w3.org/TR/xmlquery-req`.

[18] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211. ACM Press, 1991. `http://doi.acm.org/10.1145/117954.117970`.

[19] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. Website, November 1999. World Wide Web Consortium (W3C) `http://www.w3.org/TR/xpath`.

[20] Software AG The XML Company. Tamino xml server 3.1. Website, 2003. `http://www.softwareag.com/tamino/`.

[21] Dan Connolly, editor. *XML Principles, Tools, and Techniques.* O'Reilly and Associates Inc, Sebastopol, California, 1997.

[22] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Xml-ql: A query language for xml. Website, August 1998. World Wide Web Consortium (W3C) `http://www.w3.org/TR/NOTE-xml-ql/`.

[23] Matthew Duignan. Evaluating scalable vector graphics for software visualisation. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, 2003.

[24] James Eagan, Mary Jean Harrold, James A. Jones, and John T. Stasko. Technical note: Visually encoding program test information to find faults in software. In *INFOVIS*, pages 33–36, 2001. `http://citeseer.nj.nec.com/eagan01technical.html`.

[25] eXcelon Corporation. extensible information server (xis) native xml data management system. Website, 2003. `http://www.exln.com/products/xis/`.

[26] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. Scalable vector graphics (svg) 1.1 specification. Website, 2003. World Wide Web Consortium (W3C) `http://www.w3.org/TR/SVG11/`.

[27] R. Ferri, R. Pratiwadi, L. Rivera, M. Shakir, J. Snyder, D. Thomas, Y. Chen, G. Fowler, B. Krishnamurthy, and K. Vo.

[28] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. Technical report. `citeseer.nj.nec.com/article/florescu99performance.html`.

[29] The Apache Software Foundation. Apache xindice. Website, 2002. `http://xml.apache.org/xindice/`.

[30] William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, 1996. `http://citeseer.nj.nec.com/frakes96software.html`.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[32] Amit Gandhi. Xml query languages, the search for an ideal set of features. Honours Report, School of Mathematical and Computing Sciences, Victoria University of Wellington, October 2002.

[33] Graham Glass. *Web Services: Building Blocks for Distributed Systems.* Prentice Hall, 2002.

[34] Charles F. Goldfarb and Paul Prescod. *The XML Handbook.* Prentice Hall, Upper Saddle River, New Jersey, 1998.

[35] Ian S. Graham and Liam Quin. *XML Specification Guide.* John Wiley and Sons Inc, New York, 1999.

[36] Mark Graves. *Designing XML Databases.* Prentice Hall, Upper Saddle River, New Jersey, 2002.

[37] PostgreSQL Global Development Group. Postgresql. Website, 2003. `http://www.postgresql.org/`.

[38] Steven Holzner. *Inside XML.* Prentice Hall, Indianapolis, Indiana, 2001.

[39] Steven Holzner. *Inside XSLT.* Prentice Hall, Indianapolis, Indiana, 2001.

[40] IBM and Alphaworks. Jinsight: A visual tool for optimizing and understanding java programs, 2002. `http://www.research.ibm.com/jinsight/`.

[41] Macromedia Inc. Macromedia flash mx. Website, 2003. `http://www.macromedia.com/software/flash/`.

[42] Ipedo. Ipedo xml database website. Website, 2003. `http://www.ipedo.com`.

[43] Kirk Jackson. Understanding frameworks through visualisation. Honours Report, School of Mathematical and Computing Sciences, Victoria University of Wellington, June 2000.

[44] Kirk Jackson, Robert Biddle, and Ewan Tempero. Understanding frameworks through viusalisation. Technical Report CS-TR-00/02, School of Mathematical and Computing Sciences, Victoria University of Wellington, November 2000. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-00-2.abs.html`.

[45] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering*, pages 360–370. ACM Press, 1997. `http://doi.acm.org/10.1145/253228.253356`.

[46] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. `http://www.laputan.org/drc/drc.html`.

[47] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report 91-1696, 1991. `citeseer.nj.nec.com/johnson91reusing.html`.

[48] Rilla Khaled, Dan MacKay, Robert Biddle, and James Noble. A lightweight web-based case tool for sequence diagrams. In *In Proceedings SIGCHI-NZ '02*, pages 55–60, Hamilton, NZ, 2002. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-02-10.abs.html`.

[49] Frank Kruchio. Native xml databases and indexing native xml databases. Honours Report, School of Mathematical and Computing Sciences, Victoria University of Wellington, October 2002.

[50] James F Kurose and Keith W Ross. *Computer Networking A Top-Down Approach Featuring the Internet*. Addison Wesley Longman, 2001.

[51] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[52] Dan Livingston. *Essential XML For Web Professionals*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[53] Lyumrix. Dtd to xml schema: dtd2xs version 1.55. Website, February 2003. `http://puvogel.informatik.med.uni-giessen.de/lumrix/`.

[54] Dan Mackay, Robert Biddle, and James Noble. A lightwieght web-based case tool for uml class diagrams. In R. Biddle and B. Thomas, editors, *In Proceedings Fourth Australasian User Interface Conference (AUIC2003)*, volume 18, Adelaide, Australia, 2003. Australian Computer Science, Conferences in Research and Practice in Information Technology. `http://www.jrpit.flinders.edu.au/confpapers/CRPITV18Mackay.pdf`.

[55] Stuart Marshall. Understanding code for reuse. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, 1999.

[56] Stuart Marshall, Robert Biddle, and Ewan Tempero. Exploration and visualising of reusable components using java. Technical Report CS-TR-99/4, School of Mathematical and Computing Sciences, Victoria University of Wellington, April 1999. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-99-4.abs.html`.

[57] Stuart Marshall, Kirk Jackson, Craig Anslow, and Robert Biddle. Aspects to visualising reusable components. In Tim Pattison and Bruce Thomas, editors, *In Proc. Australian Symposium on Information Visualisation, (invis.au'03)*, volume 24, Adelaide, Australia, February 2003. Australian Computer Society, Conferences in Research and Practice in Information Technology. `http://www.jrpit.flinders.edu.au/CRPITVolume24.html`.

[58] Stuart Marshall, Kirk Jackson, Mike McGavin, Matthew Duignan, Robert Biddle, and Ewan Tempero. Visualising reusable software over the web. Technical Report CS-TR-01/12, School of Mathematical and Computing Sciences, Victoria University of Wellington, October 2001. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-01-12.abs.html`.

[59] Mike McGavin. Extracting software reuse information for visualisation tools. Honours Report, School of Mathematical and Computing Sciences, Victoria University of Wellington, October 2001.

[60] M. D. McIlroy. Mass produced software components. In P. and B. Randell, editors, *In Naur*, pages 138–150. Report on a Conference of the NATO Science Committee, 1968. `http://www.ericleach.com/massprod.htm`.

[61] David Megginson. The sax project. Website, January 2002. `http://www.saxproject.org/`.

[62] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Santa Barbara, CA, second edition, 1997. `http://archive.eiffel.com/doc/oosc/`.

[63] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *Software Engineering*, 21(6):528–562, 1995. `http://citeseer.nj.nec.com/mili95reusing.html`.

[64] John Miller-Williams. A program visualisation tool for emphasising the dynamic nature of reusability. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, 1995.

[65] John Miller-Williams, Robert Biddle, and Ewan Tempero. Rapid implementation of a progam visualisation system. Technical Report CS-TR-98/2, School of Mathematical and Computing Sciences, Victoria University of Wellington, May 1998. `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-98-2.abs.html`.

[66] Ann Navarro, Chuck White, and Linda Burman. *Mastering XML.* Sybex, San Francisco, California, 1996.

[67] James Noble. *Abstract Program Visualisation.* PhD thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, 1996.

[68] M. J. Oudshoorn, H. Widjaja, and S. K. Ellershaw. Aspects and taxonomy of program visualisation. In Peter D. Eades and Kang Zhang, editors, *Software Visualisation*, volume 7, pages 3–26. World Scientific, Singapore, 1996. `http://citeseer.nj.nec.com/oudshoorn96aspects.html`.

[69] M.J. Oudshoorn and H. Widjaja. Visor++: A visualisation tool for concurrent object-oriented programs. In *In Proceedings of the 8th International Conference on Computer Graphics and Visualization*, pages 287–294, Moscow, September 1998. http://citeseer.nj.nec.com/oudshoorn98visor.html.

[70] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society. `citeseer.nj.nec.com/papakonstantinou95object.html`.

[71] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *In Proceedings of OOPSLA '93*, pages 326–337, Washington D.C, 1993. Object-Oriented Programming, Systems, Languages, and Applications. `http://www.research.ibm.com/jinsight/papers/oopsla93.ps`.

[72] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Lecture Notes in Computer Science*, volume 821, pages 163–182, Bologna, Italy, July 1994. European Conference for Object Oriented Programming, Springer Verlag. `http://www.research.ibm.com/jinsight/papers/ecoop94.ps`.

[73] W. De Pauw, D. Kimelman, and J. Vlissides. *Software Visualization*, chapter Visualizing Object-Oriented Software Execution. MIT Press, 1997. `http://www.research.ibm.com/jinsight/papers/coots98.pdf`.

[74] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by analysis of running programs. In *Proceedings for Workshop on Software Visualization*, Toronto, Canada, May 2001. International Conference on Software Engineering. `http://www.research.ibm.com/jinsight/papers/ICSE2001.pdf`.

[75] W. De Pauw and G. Sevitski. Visualizing reference patterns for solving memory leaks in java. In *Lecture Notes in Computer Science*, volume 1628, pages 116–134, Lisbon, Portugal, June 1999. European Conference for Object Oriented Programming, Springer Verlag. `http://www.research.ibm.com/jinsight/papers/refpat.pdf`.

[76] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings in Concurrency: Practice and Experience 2000*, volume 12, pages 1431–1454. Concurrency: Practice and Experience, 2000. `http://www.research.ibm.com/jinsight/papers/CPE2000.pdf`.

[77] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Software Visualization International Seminar 2001*, pages 151–162. Springer, New York, 2001. `http://www.softvis.org/`.

[78] Wim De Pauw, Steven P. Reiss, and John T. Stasko. Icse workshop on software visualization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 758–759. IEEE Computer Society, 2001.

[79] Raghu Ramakrishnan and Johannnes Gehrke. *Database Management Systems*. McGraw-Hill, Singapore, second edition, 2000.

[80] Rational. Migrating from xml dtd to xml schema using uml, 2000. `http://www.rational.com/media/whitepapers/TP189draft.pdf`.

[81] Steven P. Reiss. Valley:3d visualization of program information, 1994. `http://www.darmstadt.gmd.de/~hemmje/Activities/Fadiva/seeheim-workshop/reiss.ps`.

[82] Steven P. Reiss. An engine for the 3d visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, 1995. `citeseer.nj.nec.com/29829.html`.

[83] Steven P. Reiss. Software visualization in the desert environment. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 59–66. ACM Press, 1998. `http://doi.acm.org/10.1145/277631.277643`.

[84] Steven P. Reiss. Bee/hive: A software visualization back end. In *ICSE Workshop on Software Visualization*, pages 49–54. IEEE Computer Society, 2001. `http://www.cs.brown.edu/research/softvis/papers/reiss.ps`.

[85] Steven P. Reiss. An overview of bloom. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–5. ACM Press, 2001. `http://doi.acm.org/10.1145/379605.379629`.

[86] Steven P. Reiss and Manos Renieris. Generating java trace data. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 71–77. ACM Press, 2000. `http://doi.acm.org/10.1145/337449.337481`.

[87] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Computer Society, 2001.

[88] Manos Renieris. Scripts for program trace visualization. In *ICSE Workshop on Software Visualization*, pages 23–26. IEEE Computer Society, 2001. `http://www.cs.brown.edu/research/softvis/papers/renieris.pdf`.

[89] Manos Renieris and Steven P. Reiss. Almost: exploring program traces. In *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM internation conference on Information and knowledge management*, pages 70–77. ACM Press, 1999. `http://doi.acm.org/10.1145/331770.331788`.

[90] Jonathan Robie, Don Chamberlin, and Daniela Florescu. Quilt: an xml query language. Website, March 2000. `http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html`.

[91] Jonathan Robie, Joe Lapp, and David Schach. Xml query language (xql). Website, September 1998. World Wide Web Consortium (W3C) `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[92] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12), December 1993.

[93] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *Proceedings for TOOLS Europe 2001*, Zurich, Switzerland, March 2001. Technology of Object-Oriented Languages and Systems (TOOLS) Conference Series. `http://www.research.ibm.com/jinsight/papers/toolseurope2001.pdf`.

[94] John E. Simpson. *Just XML.* Prentice Hall, Indianapolis, Indiana, 1999.

[95] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization Programming as a Multimedia Experience.* The MIT Press, Cambridge, Massachusetts, 1997.

[96] John T. Stasko. *Tango: A Framework and System for Algorithm Animation.* PhD thesis, Brown University, May 1989.

[97] John T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3), September 1990.

[98] Jake Sturm. *Developing XML Solutions.* Microsoft Press, Redmond, Washington, 2000.

[99] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

[100] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems Principles and Paradigms.* Prentice Hall, 2002.

[101] Linda van den Brink. Xml software. Website, 2003. `http://www.xmlsoftware.com/`.

[102] World Wide Web Consortium (W3C). Scalable vector graphics overview. Website, 2002. `http://www.w3.org/Graphics/SVG/Overview.htm8`.

[103] World Wide Web Consortium (W3C). Document object model (dom). Website, 2003. `http://www.w3.org/DOM/`.

[104] O'Reilly XML.com. Xml parsers. Website, 2003. `http://www.xml.com/pub/rg/XML_Parsers`.

[105] XML:DB. Xml:db initiative for xml databases. Website, 2000-2003. `http://www.xmldb.org/`.