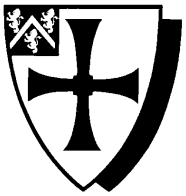
Tools To Support Education

by

Jon Dowland



Submitted in conformity with the requirements
for the degree of Computer Science
Department of Computer Science
University of Durham

Word count: 17,278

Acknowledgements

Special thanks to the faculty of Durham University Computer Science Department for their support.

Thanks also to the the Free Software Foundation; Juho Santeri Paavolainen; Sun Microsystems; the Eclipse Foundation; Trolltech and Don Knuth for their software.

Contents

List of Figures

List of Tables

Nomenclature

Annotation An annotation in this project is a fragment of documentation, relating to a specific portion of source code.

Association An association in this project is a record of a portion of source code and a corresponding annotation, plus optional information to help track source code change.

Documentation Directory The documentation directory for a software project is the common parent directory within which all the files which comprise the annotations for that project are located.

DTD **D**ocument **T**ype **D**efinition. A means of defining a particular dialect of XML, against which an XML document can be checked for validity.

False Negative Schemes to detect out-of-date annotations could suffer from false negatives - failing to report an annotation as out-of-date, when a semantic change has occurred to the source code concerned.

False Positive Schemes to detect out-of-date annotations could suffer from false positives - reporting that an annotation is out-of-date when it is not.

Haskell Haskell is a pure, strict functional programming language, based loosely on the syntax of ML.

HTML Hyper-Text Markup Language (HTML) is a markup language designed for use on the world-wide web.

IDE An Integrated/Interactive Development Environment (IDE) is a unified environment for the development of Software. The processes of reading, writing, compiling, testing and running programs are brought together.

Project Directory The project directory for a software project is the common parent directory within which all the files which comprise the software project are located.

Repository The repository for the systems described in this project is a file within which a list of associations are stored, in conjunction with information required to identify the relevant software project they relate to: the project directory, the documentation directory and the tracking scheme in use.

Valid XML A valid XML document conforms to a Document Type Definition.

Well Formed XML Well-Formed XML has an equal number of start and end tags, which are paired. Start and end tags are nested in a tree arrangement, so `<a><b></b></a>` is well-formed, whilst `<a><b></a></b>` is not.

XHTML XHTML is a recast of HTML in XML. XHTML was introduced so that web pages could migrate across to an XML-compatible format and take advantage of the XML tools available.

XMLHaskell XMLHaskell is the name given to Haskell source code marked up using the XML application defined in Appendix B.2.

XML E**X**tendable **M**arkup **L**anguage. A syntactic framework for defining data and data relationships.

Part I

Introduction

Chapter 1

Overview

1.1 Definition of the Problem

1.1.1 Origins

During the summer of 2003 I was working as an intern for the Storage Solutions department of IBM. In this role I was required to read and comprehend a large volume of source code.

In order to support my comprehension, I made notes and annotations about the code I was reading. In some cases I was able to do this directly in the source code using the language's 'comment' mechanism. Sometimes the size of my annotations was such that including them as in-line comments had a negative impact on the readability of the source code: trying to read the comments was fine, but the flow of the source code around the comments was fragmented.

Many programming concepts and data structures can be described effectively with the use of diagrams. However, most comment mechanisms in programming languages only allow plain text. It is most often not possible to insert diagrams and other rich forms of annotation into the source code.

Additionally, restrictions in place on some source code prevented me from even adding textual comments directly. For example, updates to the Linux kernel are distributed in 'patch' format, which relies on source code files locally being identical to those on the machine which produced the patches.

For these reasons, I developed a paper-based annotation system. This system was far from ideal: looking up annotations was a laborious process, and where I couldn't add a comment into a file to indicate that I had made a paper annotation,

I had to rely on memory.

This system is described in more detail in Section 3.2.

1.1.2 A Wider Problem

The difficulties related to program comprehension have long been studied and documented. Whilst the need for rich annotations[Bro95] and the negative impact of substantial in-line documentation have been recognised, they are only small parts of a larger problem.

Early attitudes towards software development, and models of the development process, did not place a great deal of importance on the stages that succeed delivery of a product[YK93]. As a result, little emphasis has been placed on the necessity for clear, readable source code, or documentation aimed at future maintainers.

Where source code has been documented during development, little effort has generally been expended on keeping this up-to-date as maintenance changes are enacted[UDCT93].

It has since been proposed that post-release development (maintenance) is in fact a very important part of the software life cycle[Sta84]. Some work suggests that it is in fact the only stage of the life cycle and that all development can be considered maintenance, starting with the special zero-state of software.

1.2 Objectives

- To explore the impact of computer-aided annotation management on the comprehension process.
- To develop a computer program to replace the paper-based system described in 1.1.1.
- To extend this program with the facility to identify annotations which could have potentially misleading content based on the evolution of the source code that they are related with.
- To determine how the granularity and reliability of this are effected by considering the semantics of the source code.

1.3 Deliverables

1.3.1 Basic

B1

A tool to manage the addition, editing, subtraction and retrieval of associations between source code and annotations, from a repository.

The source code should be unmodified by such associations so that the system could be used in situations where modifying the source code is not possible, as described in section 1.1.1. Diagrams and rich content must be supported.

B2

An interactive environment to support the tool described above, which will visually associate annotations with the respective areas of source code.

1.3.2 Intermediate

I1

Extend the tool to monitor change in the underlying source code, and identify annotations which potentially are out of date and misleading, marking them to be verified.

1.3.3 Advanced

A1

Extend the tool to understand the semantics of a programming language. Using this extension, improve the granularity of change that can be observed.

1.4 Project Plan

1.4.1 Dates

A table of deadlines for phases of the project is available in Table 1.1 and a GANTT chart of my plan for the duration of this project is available in Appendix A.

Item	Date
Basic Implementation	18/01/04
Project Presentation	23/01/04
Preliminary Demonstration	13/02/04
Final Deadline	04/05/04

Table 1.1: Table of Dates

1.5 Report Overview

- This chapter outlines the background to the project, a description of what I have attempted to do, and a breakdown of the rest of the report.
- Chapter 2 contains a review of the literature and tools available in the fields covered by this project, in particular that of Program Comprehension.
- Chapter 3 details the decomposition of the project objectives and requirements into the design used for the software deliverable.

  The existing paper-based system is analysed, desired improvements are developed and an architectural model is decided upon.
- Chapter 4 details changes made to the design; a description of re-used software components; a description of the development and testing environment, including software tools employed; a review of interesting technical details.
- Chapter 5 covers how the project is evaluated and how successfully objectives have been met, using a variety of techniques including a walk-through and an experiment.
- Chapter 6 considers how successful the project has been, identifying areas for improvement and possible future work in the field.

Chapter 2

Theory and Literature

2.1 Overview

Program comprehension is a long-studied problem, with multiple academic roots.

Within the field of computer science, a large amount of work has been done towards 'fixing' the process of developing software, in order to remove the problems evident today. The contrasting approach is to develop tools, techniques and methodologies to cope with the large body of existing source code which has been engineered without consideration of its long-term use.

A variety of such solutions are explored, with the fields of Literate Programming, Program Slicing, Documentation and Source Code representation explored in greater detail.

2.2 Program Comprehension

Programming and program comprehension are both forms of information exchange between two types of system: A formal computer system and the human brain.

2.2.1 The Problem

Bugs occur when a concept is incorrectly translated from the representation in the brain to the representation on the computer. Bugs are common and often not discovered when they are created but much later.

Program Comprehension involves the reverse of this process: concepts must be translated from the formal, hierarchical representation in the computer into a mental

model [Nel].

A failure in translation is often much more apparent than during programming, but not always.

These two operations are difficult because we do not understand the human brain particularly well. The precise 'input language' to use varies from person to person, there is an inconsistent efficiency and the brain is heavily effected by environmental factors.

Additionally, even a well-structured system designed with future comprehension in mind can be degraded by maintenance which did not use such considerations [MDJ+97].

2.2.2 Models of Comprehension

Models of comprehension are attempts at modelling the process of comprehension in order to study it, and to determine the best ways of supporting it.

There have been two 'pure' models of comprehension proposed:

Bottom-up Comprehension Bottom-up comprehension builds a cognitive model of the system from scratch based on information accumulated from low-level sources. This model is supported by the fact that most developer's must resort to the lowest-level source of information available about a program: Its source code [Shn80].

Small patterns are recognised at a low-level, and merged into the bigger understanding as the process continues. This process is called 'chunking' [Knu84].

Top-Down Comprehension A conflicting model of this process is that of Top-Down Comprehension[Bro83]. In this model, the cognitive model of the system takes the form of an 'initial hypothesis', formed from initial high-level information.

This initial hypothesis is then verified or refined as information is accumulated using 'beacons'.

Beacons are clues which support a particular pattern occurring. The maintainer must have prior experience with a particular pattern to recognise beacons which indicate its presence.

Combined Models

These two approaches conflict with one another, yet both fit observations of the comprehension process.

It has been noted that the appropriate model of comprehension in use at a given time is heavily dependent on the specific nature of the problem being studied.

Letovsky proposed Opportunistic Comprehension [Let86], where the maintainer makes use of both top-down and bottom-up comprehension at different stages. The technique to use is chosen based on *cues* in the source code.

As a consequence, tools which have been developed with only one model of comprehension in mind risk stifling the process of comprehension, rather than aiding it [SWM00].

Conceptual Building Blocks

The bottom-up model of comprehension makes reference to patterns in the information which are recognised, and then incorporated into the maintainer's cognitive model.

The top-down model of comprehension refers to beacons: aspects of the information source which verify or contradict the hypothesis.

The opportunistic model considers the concept to be a cue.

Further work on comprehension identifies commonly occurring solutions to problems. These have been called clichés, and developed into the field of design patterns [GHJV93].

These concepts all described a common conceptual building block, from which algorithms are composed [YK93].

Other Models

The process of comprehension is not perfectly understood, and the models that have been developed are not without their drawbacks.

It is likely that further models or refinements will be devised as understanding of the process increases.

Any tool or approach towards comprehension which alters the way a person comprehends a system in a positive way is of benefit to the study, as it sheds new light on the 'true' process being undertaken.

2.2.3 Solutions

There are a variety of approaches and tools designed to aid program comprehension.

Frameworks

Based on the continuing study of comprehension, some guidelines have been written for environments and frameworks that are to support it.

It is of general agreement that using tools to support comprehension should be no additional burden on the reader [FM87, LS86].

The absorption of information during comprehension generates further information [GO97], which ideally should be captured as easily as possible [FM87].

Tools which will allow this information to be recorded should support incremental documentation [YK93].

Paradigm Shifts

Many approaches involve altering the way we develop computer programs in order to make future comprehension easier.

In the past there has not been a great emphasis placed on the phases of software development after delivery [YK93].

However, it has come to light that the post-release phase of maintenance is one of the most significant and expensive stages in the software life cycle[Sta84].

The importance attributed to it nowadays is so large that it has become a widely accepted premise that all development is in fact maintenance, starting from the 'zero state' software program.

Literate Programming is one such paradigm, and is described in detail in Section 2.3.

The problem with alternative development paradigms is that they must be embraced in order to have effect. If a given paradigm was universally adopted it is possible that maintenance of software would be significantly easier: maintainers could be trained exclusively with this paradigm and be prepared for all software they might face.

There exists, and there is always likely to exist, a large volume of mission-critical legacy software which was not developed with future maintenance in mind [MR87]. There is thus always going to be a need for 'raw' comprehension, unaided by the decade's current paradigm of choice.

Information Hiding

Information Hiding is the art of extracting relevant sections of information from a larger set, in order to prevent information irrelevant to a particular context from distracting the reader.

Program Slicing is one example of an Information Hiding technique. Program Slicing is described in Section 2.6.

2.3 Literate Programming

Literate Programming is the name of a programming paradigm introduced by Don Knuth in 1984[Knu84].

Literate Programming aims to change the development process so that the primary audience of code is a human being rather than a computer, and so that code is documented at the same time as it is written.

Knuth found that by writing natural language descriptions of code at the same time as the code itself, the quality of his programming improved. By immediately expounding about the intentions of the code, mistakes are detected more easily [Thi86].

The author of a literate program writes a WEB file, which is a composite of both source code and natural language text describing the source code.

The program '`TANGLE`' extracts source code and the program '`WEAVE`' extracts typesetter markup from the WEB file. Figure 2.1 illustrates the journey of information from a WEB file to object code and documentation.

Additionally, the WEB system is a macro processor of sorts which allows code to be arranged into 'chunks', which need not appear in the same order as the resulting source code [Knu84]. This allows the author to present the program at varying levels of abstraction as the program develops, rather than being forced into an order by language restrictions.[1]

2.3.1 Implementations

The original implementation of the WEB system used the `PASCAL` programming language and the TEX markup language.

[1]For example, you may decide to introduce a variable in the middle of a block of code, but the language may require you to have defined the variable at the start of the block. This prior definition is contrary to the order in which you composed the program mentally.

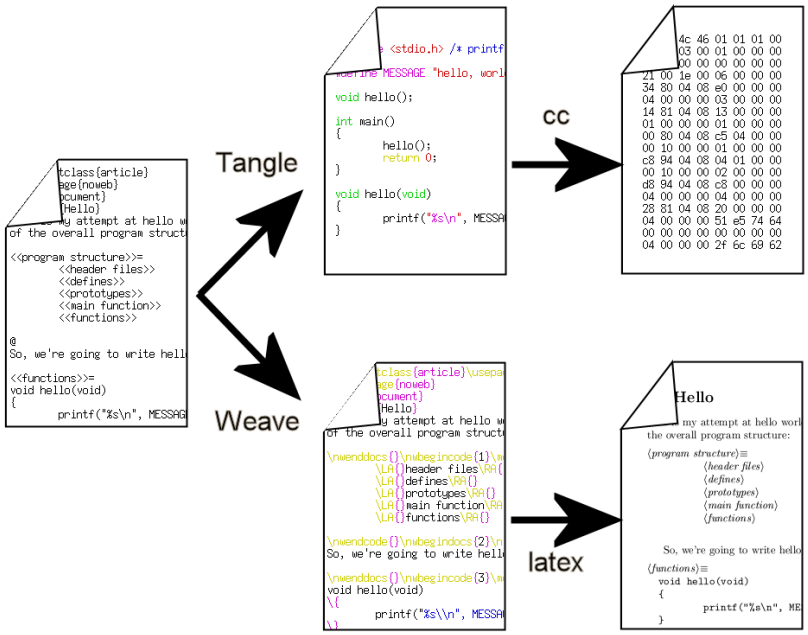


Figure 2.1: Information flow in the WEB system

`cweb` [Thi86] is an implementation which interleaves the C programming language with the troff typesetting system.

`noweb` [Ram94] is another implementation which attempts to support multiple programming languages.

There are many published examples of literate programs, including [BKM86, Knu86b, Knu86a, VHG87, Ham88].

2.3.2 Drawbacks

Existing Software

Literate Programming is a paradigm applied at the development stage and suffers from the general drawbacks of this (see section 2.2.3).

However, some efforts have been made to develop automated, re-documenting Literate Programming tools for existing software [Win98].

Code Walk-through

Literate Programming recognises that the top-down arrangement of a program's source code may well not be the best arrangement for comprehension and allows the author to impose another arrangement upon the reader.

However, a software maintainer will most likely adopt an as-needed strategy [SI86] towards comprehension, and thus different maintenance jobs will require different slices of a program [Wei84] to be examined. The arrangement imposed by the author will be a 'total' one, encapsulating the entire program and from a single point of view.

Documentation Target

The output of the `WEAVE` command is oriented towards paper representation [Knu84]. This is due to the fact that Literate Programming was developed at a time when computer displays were relatively inferior, and graphical environments had only just started to develop.

Nowadays, a more on-line presentation would be more appropriate as on-line cross referencing and similar technologies could be exploited [Nør00].

2.4 Elucidative Programming

Elucidative Programming [Nør00] is a paradigm based on Literate Programming, designed to overcome some of its drawbacks.

Elucidative Programming attempts to capture the understanding a developer has of the code as it is being written.

The inclusion of documentation and code in Literate Programming means the code that the compiler sees is not immediately accessible to the author. Any errors discovered and reported by the compiler will refer to line numbers in the generated code, rather than where the code resides in the WEB file.

The major difference between the Literate and Elucidative paradigms is the elucidative programming works on the principle that the source code and the documentation should be stored separately.

2.5 Documentation

Documentation is crucially important to properly understand a software system [Win98].

However, the process of producing software documentation is not considered a high-priority part of development. Most software documentation available for existing systems is incomplete, out-of-date or simply missing [FM88].

2.5.1 Types of Documentation

Varieties of documentation can be broadly categorised as follows [vZ93]:

Development Development documentation is created during development of the system.

User User documentation is aimed at the end-user and details the 'public' interface to the system.

Technical Technical documentation is that which can be extracted from the source code by automated tools, for example dependency graphs, flow control diagrams, and statistical information.

2.5.2 Re-documentation

Re-documentation is the process of extracting documentation from source code, either to replace or augment existing documentation.

Re-documentation can be both automated and conducted by hand [Fle88].

Incremental Re-documentation

During comprehension, the reader will generate new information from the system. Ideally, this information should be captured and managed with the existing system documentation.

This information is generated incrementally - that is, piece by piece, as more and more of the system's internals are uncovered.

A re-documentation tool must allow a reader to continually add newly generated information to an existing 'stockpile' [FM87].

2.5.3 Management

Javadoc

'Javadoc' is the name of a documentation system developed by Sun Micro-systems for their Java Programming Language.

The Javadoc system [Fri95] is designed to provide high-quality, cross-referenced documentation for the Core Java Application Programming Interface (API).

One of the core design goals of Javadoc was that the documentation should be generated from the respective Source Code. The Javadoc designers opted to use Hyper-Text Markup Language (HTML)[Con] as the format for presenting the documentation. HTML provides cross-referencing in the form of hyper-links, and is the de facto standard format for cross-platform information exchange.

Most desktop computers come complete with a Web Browser capable of interpreting HTML documents, and a plain-text editor capable of authoring them.

Without explicit support in the form of special tags embedded into source code comments, Javadoc can create documentation containing inheritance trees, with known types cross-referenced to one another.

With these special tags however, the author can provide Javadoc with English text descriptions of classes and methods, enriching the resulting documentation.

The doc-comments support embedded HTML, which can be used to include multi-media content in the resulting documentation, or code samples. However, the code samples are not generated automatically from the source code that is being described: they must be explicitly copied into the comment, and kept in sync by the author.

The drawback of this, and embedded documentation in general, is the increase in 'signal to noise' ratio with respect to the source code: the code can be extremely de-localised [LS86].

2.6 Program Slicing

Program slicing is an information hiding technique which was formalised from observations of the practises of expert maintainers [Wei84].

During maintenance, only a subset of the program being maintained is relevant. For example, when correcting a bug, only the aspects of the program which contribute towards the incorrect behaviour are of concern.

A slice is a set of statements from a program, with respect to a slicing criterion. A slicing criterion is a set of variables and a statement. What is of interest is the values assigned to those variables at the time that the statement is to be executed.

There are two requirements for a slice [Wei84]:

1. A slice must have been obtained from the original program by statement deletion, and necessarily be smaller than the program.

2. The behaviour of the slice and the program as observed through the window of the slicing criterion must be indistinguishable.

So if a maintainer can specify the context of their problem in terms of a slicing criterion; a slicing tool obeying these requirements can provide a slice of the program, removing many of the statements which are irrelevant for that context, relieving the maintainer of that burden.

2.6.1 Ideal Slice

It is generally impossible to calculate the smallest possible slice for a given program and criterion [Wei84].

There may be statements present in the program which modify the variables in the slicing criterion, but are never executed. This may be due to a portion of code preceding this statement resulting in an infinite loop.

However, it has been proved that no general algorithm can exist to determining that a given portion of code never terminates is undecidable [Tur37].

2.6.2 Taxonomy

There are a number of useful applications for program slicing, each of which has a different set of required characteristics, in addition to the fundamental requirements above.

Executable Weiser's original definition of slicing included the need for slices to be *executable*. So, in addition to statements that effect the variables in the slicing criterion, the tool must also include statements that are paired with such lines, and necessary for syntactically correct programs.

For example, consider the code in Figure 2.2. If the variable `c` was in the slicing criterion, then it would be necessary to include line 1.

However if an executable slice was necessary, then line 5 is required to have a syntactically valid program - the `if` branch requires the closing brace.

```
1          if(b == a) {
2               ...
3               c = c + 3;
4               ...
5          } else {
6           ...
7          }
```

Figure 2.2: A Slicing Example

Dynamic Smaller slices can be computed with some knowledge of the program's dynamic input [KL88]. Considering figure 2.2; if `c` is in the slicing criterion, then a static slice must include line 1 as described above.

However, with dynamic information, it might be possible to eliminate such conditional branches due to an understanding that `b` does not equal `a` for the given input.

2.7 Software Representation

Source Code is traditionally represented as plain text. This has the advantage that it can be easily read by a human being, relative to a more complex representation.

When the source code is translated into a representation for manipulation by a computer, either by a compiler converting it into a system binary, or by an interpreter converting it into memory structures at run-time, the flat text representation goes through a complex process of transformation.

2.7.1 Analysis

Source code stored in such a way can be analysed using different techniques, each providing different details of information.

The simplest method of analysis, and that which provides the least amount of information, is lexical analysis.

Lexical analysis considers the syntactic structure of the source code, but not its semantic meaning.

Lexical analysis is powerful enough for purposes such as syntax highlighting in a source code editor.

More complex means of analysis consider the semantic meaning of the code. One task that requires an absolute understanding of the semantics is compilation.

2.7.2 Compilation

Compilation of source code typically goes through a number of stages [App98]. The 'front end' of the compiler consists of the stages which translate the flat text representation into an Abstract Syntax Tree (AST) structure in memory.

The first stage is lexical analysis. The text file is read in by the lexer component and searched for 'tokens'. The searching process is usually aided using regular expressions.

Once the lexer is complete, a stream of tokens is ready for the parser component, which will translate these tokens into the AST.

Much of the work needed for building the lexer and parser components can be simplified by using automatic lexer and parser generation tools, such as lex[Les75] and yacc[Joh79].

Such tools take as input the formal grammar of the language, and produce the source code for a lexer and parser.

In order to work around limitations in the parsing algorithm employed, such as the LR parsing algorithm[Tom87], the grammar that is used in conjunction with the lexer and parser generators is often different from the grammar that defines the program language.

The grammar used typically differs in order to remove ambiguity in the original grammar. This can occur with infix operators, where the relative precedence and associativity needs to be known.

As a result, compilers typically have a post-parser stage where the syntax-tree built in memory is traversed and translated into a representation which is true to the original grammar. Once the precedence and associativity have been encoded into the tree, the changes made to the grammar are no longer necessary, and later stages of the compiler can work with the simpler, original grammar.

2.7.3 Re-Engineering Tools

Many re-engineering tools benefit from analysis of the source code beyond that possible with simple lexical approaches. For example, semantics preserving program transformations operate on an Abstract Syntax Tree, derived from parsing.

However, many re-engineering tools opt not to use such analysis. The amount of work required to incorporate a compiler front-end into the tool is deemed to be too great[Bad00].

There are a number of reasons why this may be the case. One of these reasons is that the front-end of a real-world compiler is cumbersome. Deviations and in-house extensions to language specifications require a complex grammar.

Formal grammars are not well suited to evolution, and can quickly become unmanageable[dBSV98].

It is often possible to develop a re-engineering tool that can suffice with mere lexical analysis of the underlying source code, and the advantages that deeper analysis would bring are outweighed by the complexity that the lexer and parser would entail.

2.7.4 Alternatives to Plain Text

There have been a number of efforts to devise an alternative format for the representation of source code.

Some of the more promising work recently has involved formulating a representation using XML.

XML

E**X**tendable **M**arkup **L**anguage (XML) is a markup language for representing structured data.

By encoding data in XML markup, programmers can make use of existing XML parsing libraries, considerably simplifying the job of parsing data.

There exist applications such as e**X**tensible **S**tyle **L**anguage which automate the translation of one XML application into another.

One consequence of this is two or more independently designed programs can share information: as long as a translation can be written from one program's XML application to others.

There are a large number of tools for manipulating and querying XML documents, including editors, navigators, tools to manage transformations, and tools to convert data from non-XML to XML formats (and vice versa).

XML-based Source Code

A number of efforts have been made to use XML as the basis for representing computer programs, including JavaML[Bad00], srcML[CMM], GXL[HWS00], and Grax[EKW99].

Most of these approaches attempt to augment existing source code XML tags, which encapsulate semantic information.

The primary advantage of such an approach is the wealth of existing tools for manipulating XML documents that can be leveraged for semantic analysis [Bad00].

XML has been designed from the ground up to be easily incorporated into applications, and so XML parsers could take the place of compiler front-ends as a means for re-engineering tools to gain a semantic understanding of source code.

An example of renaming an identifier using an XSL transformation is given in [Bad00]. The result catches both definition and use of the identifier, without replacing occurrences of the name in other locations.

This is a considerable advantage over a traditional lexical approach, which would replace occurrences of the identifier within literal strings and comments. Lexical search-and-replace would therefore be impossible with common short variables such as '`i`'.

What to Mark-Up

The approaches listed have different philosophical ideas about what should be preserved or augmented when converting source code to an XML representation.

Some people believe that the original source code should be obtainable from the XML representation. This is essential for existing compilers and re-engineering tools to continue to work, un-modified.

However, the exact definition of 'original source code' is disputed.

JavaML [Bad00] and srcML [CMM] both preserve source-code comments in the marked-up representations.

However, srcML goes further in ensuring the retrieved source code retains the exact indentation and white-space arrangements as the original source code.

2.8 Conclusion

Software comprehension is a difficult task. Code is not often written with future comprehension in mind : the primary audience of source code is most often a computer rather than a human being.

There are a large variety of approaches towards aiding comprehension, which apply to various specific situations. Some approaches require changes to the development process. Literate programming is an extreme example of this, where the primary audience of the code is changed. Other techniques need support in the form of embedded keywords in comments or similar code-alterations, such as Javadoc.

Tools to support comprehension which are not aided by changes to the development process are often limited in their effectiveness, due to the difficulties in incorporating source code parsers. Such tools resort to the simpler technique of lexical analysis.

A fundamental disadvantage with the development-changing approach is that there will always exist code which was not developed with future comprehension in mind. Under these circumstances, more general comprehension strategies must be employed.

Part II

Development

Chapter 3

Design

3.1 Overview

This chapter describes the design of the system.

- The original paper-based system is detailed and desired extensions listed.
- An architecture model is adopted for the system and the components are developed based on the requirements.
- Each component's interaction with its peers is considered, and testing strategies that could be employed are discussed.

3.2 Existing System

The original paper-based system for managing annotations was introduced in Section 1.1.1.

Source code is read using a traditional text editor of the user's choice. Annotations for source code are developed on paper, along with a reference to the area of source code they applied to. There is no indication from the source code whether or not an annotation has been prepared for the section being studied.

Annotations can refer to a variety of different objects, such as lines in files, procedures in modules, or directories within the source code project. The inconsistent nature of this association makes recalling the location of annotations whilst browsing the source code more difficult.

The associations are not ordered or organised in any way. This is partly due to the lack of a consistent granularity of association. As a consequence, searching for and removing annotations are potentially exhaustive operations.

3.3 Desired Extensions

In addition to recasting the system described above in a software system, further functionality is desired.

Deliverables I1 (Section 1.3.2) and A1 (Section 1.3.3) describe a system for detecting out-of-date annotations.

I1 states that this should be done by detecting file change, whilst A1 states that the programming language of the underlying source code should be understood.

3.3.1 Properties

A scheme for determining out of date annotations could miss a change in the source code (False Negative), or indicate that an annotation was out of date when it wasn't (False Positive).

It is essential for a scheme to not miss any negatives in order to be useful. However, false positives are tolerable, albeit undesirable.

Given a choice, it is better for a scheme to err on the side of caution and suffer from false positives than to make uncertain decisions and suffer from false negatives, however infrequent.

3.3.2 Operations

Presented with a possibly out-of-date annotation a user of the system would want to verify this, updating the annotation to reflect changes in the source code where necessary.

If the annotation is in fact not out of date, or if the annotation has been updated to correspond to the source code, the user would want to inform the system that the annotation is not out-of-date.

The system should provide a facility for verifying suspect annotations.

3.4 Architecture

The system will use the Model-View-Controller (MVC) architecture where the Model, the View and the Controller are the three principal components.

This architecture was first introduced as part of the Smalltalk-80 Object-Oriented development system [KP88].

The large, heavy-outlined items Figure 3.1 are the main components of the MVC model.

The Model encapsulates the core data structures of the project; the View manages the representation of the Model to the user via the user interface; and the Controller manipulates the Model according to user input.

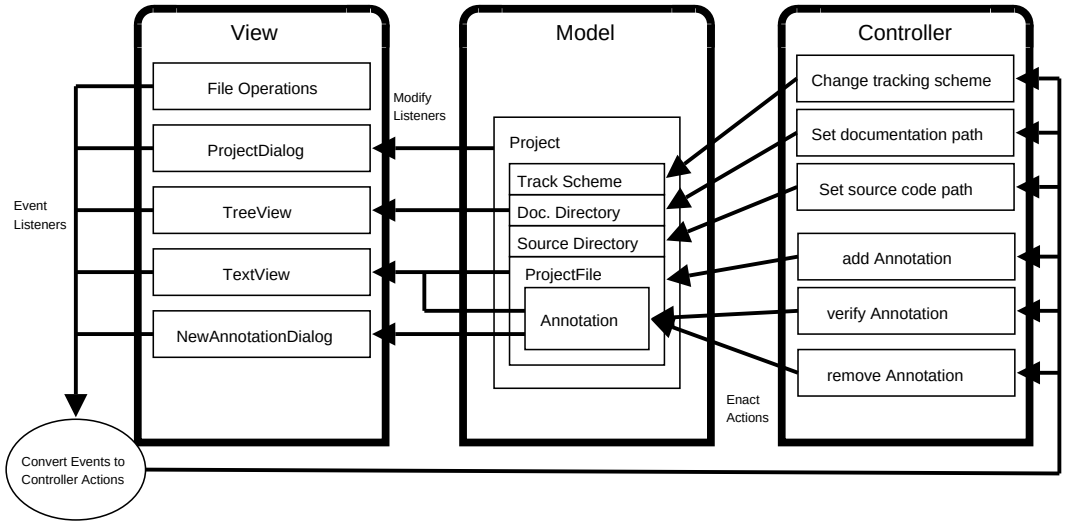


Figure 3.1: Architecture Diagram

Each component interacts with the others via strict interfaces which are independent of implementation specifics. This is so that each component can be changed significantly or substituted for alternatives with minimum impact on the other components.

This arrangement allows each unit to be developed and tested in isolation.

The interfaces between the components are described in terms of object oriented programming terms.

3.4.1 View/Controller Interface

The View receives input from the user and translates that input into instructions for the controller to carry out.

The View will maintain a reference to the Controller object, and by interpreting the user's instructions, call methods in the Controller.

The Controller's public interface is independent of the View's implementation specifics.

3.4.2 Controller/Model Interface

The Controller will maintain a reference to the Model and manipulate it according to the instructions provided from the Controller's caller. This may be the View component, or alternative 'driver' software written for testing purposes.

3.4.3 Model/View Interface

The Model will use the Listener Design Pattern to inform the View (and any other interested parties) about changes to its structure.

Each portion of the Model which is liable to change will provide a Listener mechanism. This means that listening components will only be informed about events related to the portion of the model that is relevant.

3.5 Model

The model component contains the data structures that are being manipulated.

The system must operate on a repository of association information, linking files containing annotations to source code.

The data structures must encode enough information about the source code and the annotation files for the operations listed in deliverable B1 (Section 1.3.1) to be fulfilled.

In order for this information to be preserved between instances of the program and for it to be distributed and shared with multiple users, the data structures must be translated into a representation suitable for storage and retrieval from an off-line storage device.

In order for the system to track file change, as required by deliverables I1 (1.3.2) and A1 (1.3.3), information gathered about files being annotated will need to be

stored when the annotations are created. The system will then retrieve this and compare it to the same information gathered when the annotation is displayed.

3.5.1 Source Code

Software Projects are a collection of files organised into a hierarchical structure on the file system of the host computer.

These files comprise the program's source code, as well as supporting documentation, licences and conditions of use, build instructions, test cases and test scripts.

A software project can therefore be defined in terms of the common directory within which all these files are stored. This directory will be referred to as the 'project directory'.

However, if the software project is shared amongst multiple users it is certain to be stored in a different location for each user.

For example, the code may be stored in the location `/home/jon/code` for a user on a UNIX machine, and `c:\programming` for a user on a Microsoft Windows machine.

The absolute location of the project directory at a given time is therefore distinct from the relative location of source code files within this directory. Changes to one do not affect the other.

For example, a file `src/main.c` relative to the project directory remains in this same relative place, even if copied from `/home/jon/code` on one computer to the location `c:\programming` on another.

Similarly, moving the file `main.c` from the sub-directory `src` to the subdirectory `code` within the project directory does not affect its absolute location.

Deliverable B1 (Section 1.3.1) states that source code must not be modified by the manipulation of annotations. This can be expanded to a design requirement:

The system will not alter files that reside within the project directory.

3.5.2 Monitoring File Change

Deliverable I1 (Section 1.3.2) requires the system to monitor changes to the software project, in order to mark annotations that could have become out-of-date.

Reading files under the project hierarchy may be necessary to achieve this.

File change can be discovered using number of different approaches. Each approach may suffer from false negatives or false positives as discussed in Section 3.3.

File Size The file that has changed may have either increased or decreased in size. The size of the file will be recorded at association creation time, and compared when the association was being visualised.

However, it is possible for a file to remain the same size after being modified, if the number of bytes added to the file match the number of bytes removed.

Such a scheme is therefore susceptible to false negatives.

Last Modified Value Most operating systems maintain a set of meta-data relating to files. The 'Last Modified' value contains the date and time of the last modification to the file.

This scheme will record the last modified value at association creation time, and compare it to the value when the association is being visualised, in the same manner as the File Size scheme.

This value can be updated by common operations that do not modify the contents of files, in addition to those that do. An example of this is the POSIX '`cp`' (copy) command, which updates the Last Modified value for the copy by default. A scheme relying on the this value would therefore be liable to produce false positives.

Caching A portion of the file could be recorded elsewhere, and tested using a comparison method such as lexicographic or alphanumeric comparison, to determine whether the cached portion of the file differs from the original portion.

Depending on the amount of the source file cached, this could result in a large amount of storage space being used up.

If a portion was recorded as being at a particular offset into the file, then the addition of material above this position would result in the offset being displaced. If this was not corrected for, then comparison of the cached portion to the master file would result in false positives.

Message Digests A message digest is a fixed length string of digits, produced from arbitrary input data to a one-way hash function.

A perfect one-way hash function guarantees that the output can only be produced from the original input - there are cases where two distinct inputs will generate the same message digest.

In practise, message digest algorithms provide only a 'computationally infeasible' guarantee, which states that the work needed to find pair of inputs that produce the same output is beyond the feasibility of modern computer systems.

A file could be converted into a message-digest string, which is then recorded. Typical message digest algorithms have fixed-length digests[1]. A collection of such digests could be stored with little storage overhead.

File change will be detected by comparing the stored digest with a newly-computed one.

The scheme to be employed must be recorded so that the system knows which scheme to use for comparisons when retrieving saved information. Each scheme described also requires information to be stored for each association.

3.5.3 Documentation

Authoring Annotations

Electronic annotations need to be composed in order to be associated with code. Ideally the system would allow easy, rapid authoring of annotations within the same framework as controlling their association.

However, it is outside the scope of this project to develop a format for storing such annotations, and to implement a visual authoring tool. The focus of this work is on managing the associations specifically.

The system will allow a user to specify an external program for viewing annotations, and an external program for editing them. No restriction will be imposed on the specifics of these.

HTML

Hyper Text Markup Language (HTML) is a prolific markup language for presenting text, diagrams and rich content - the types of information that annotations will be composed from.

Most desktop computer systems include both a Web Browser which facilities displaying HTML documents, and a plain-text editor, which is suitable for editing such documents.

[1] The widely used MD5 algorithm has a fixed size digest of 36 bytes.

For these reasons, HTML was chosen as the descriptive language of choice for documentation in the Javadoc system (see Section 2.5.3). For the same reasons, HTML is a good candidate language for describing annotations in this system.

Identifying Annotations

The model's data structures must identify the location of annotations on the host computer. There are a number of different situations which may define where annotation files are stored:

No Existing Documentation The annotations are composed from scratch within the system, and do not exist as documents already. The system could handle storage of all annotations within a directory under its control.

Existing Documentation Existing documentation is being enriched with associations using the system. The existing documentation may be a part of the source code distribution, or may be from another source, such as automated re-documentation tools.

User is a Source Code Author The user might be manipulating a master copy of the source code, and wishing to include association information into the main distribution. In this case, the design requirement specified in Section 3.5.1 would be unnecessary.

As there can be no assurance as to the exact location of the annotations, combined with the fact that the system is to delegate viewing and editing responsibilities to external programs (Section 3.5.3); no greater control can be assumed over storage of annotations than that of the source code itself.

Annotations will be identified as a parent directory, and relative paths from this to the specific files being referenced.

3.5.4 Association

Annotations are to be associated with source code, but how large an area of source code should an annotation be associated with? The original system (Section 3.2) had no consistent granularity and as a consequence, sorting and searching for annotations was troublesome. There are a number of candidates:

File Each association could be tied to distinct files in the project. The problem with this approach is that source code files can be very large[2] and typically contain a large number of program 'plans' (Section 2.2.2). It is conceivable that the code reader would want to attach more than one annotation to a given file.

Region A region in source code could be defined as a starting and ending character in a file, with the restriction that the starting character must precede the ending character. This provides a degree of flexibility to the annotator: the operation of annotating would bear a strong resemblance with that of highlighting a printed page. However it would also introduce complexity, as the system would need a policy for dealing with overlapping regions.

Line Annotations could be associated with lines in source code. Although many languages are structured so that end-of-line characters and whitespace in general are ignored by the parser, lines of code is still considered a significant figure to represent a program's complexity; and programmers naturally arrange code into multiple lines to aid readability.

This seems to be a good compromise between the granularity and complexity of the previous two candidates.

3.5.5 Storage and Retrieval

It was mentioned in Section 3.5 that the model must be translated to and from a representation suitable for storage and retrieval from off-line storage.

XML

XML (introduced in Section 2.7.4) is a suitable format for storing the model because the availability of XML parser libraries simplifies the work needed to parse the stored repository and rebuild the model in memory.

In addition, the use of XML means that other programs which use XML as a storage format could have their data imported for use in the system. Examples of such tools include some Integrated Development Environments (IDEs) and build tools such as Ant[Foua].

[2]For example, the average length of a source code file in the Linux Kernel (version 2.4.25) is 429 lines; the maximum length 31,597 lines.

Document Type Definition

A **D**ocument **T**ype **D**efinition (DTD) is a means of formally specifying an XML application. A DTD lists the allowed structural elements and what each structural element is permitted to contain.

By using a DTD, a program can be more confident that a conforming XML file contains the required data.

Figure B.1 in Appendix B lists a DTD for repositories based on the design decisions made in this section. This DTD can be used to ensure that an XML document intended to describe an instance of the model is valid.

3.5.6 Testing

Heisenbugs

For the Model to be tested, it must be observed. It must either be read directly from memory and interpreted using an external program such as Data Display Debugger [ZL96]; or translated into a different representation for displaying on screen or writing to disk.

The former approach requires manual operation and would be a time consuming task to repeat during implementation. The latter approach runs the risk of introducing 'heisenbugs': bugs which only exist or are discovered when the model is being debugged.

Offline Representation

Section 3.5.5 described a method of defining an XML application.

An XML file which does not conform to the DTD listed in Appendix B should be rejected by the system, as it can not be guaranteed that the document describes a consistent model.

Parser

The robustness of the system's parser can be tested by attempting to open a set of XML files, each of which violates precisely one rule from the DTD, such that the set encompasses all possible violations.

There are three main ways of violating the DTD:

missing tag Where the DTD states that a particular tag must be present within another tag, provide an XML document where the tag is missing.

There are 10 specified tags in the repository DTD. However: the root tag cannot be omitted or the XML document would not be well-formed; the <`file`> and <`annotation`> tags are defined as required to occur zero or more times; and the <`checksum`> tag is optional. This leaves 6 test cases.

extra tag For each tag described, provide a document where the tag contains an extra, unspecified tag.

This provides an additional 10 test cases.

empty character data For each tag which must contain character data, provide a document where the tag is empty.

There are 7 tags specified to contain character data. Of these, the contents of <`track`> and <`checksum`> could be empty, resulting in a further 5 test cases.

Results of these tests on the final system are available in Appendix C.1.

3.5.7 Summary

Figure 3.2 contains an Entity-Relationship diagram based on the development of the definition of a software project, the decisions made about association granularity, and the details of annotation authoring.

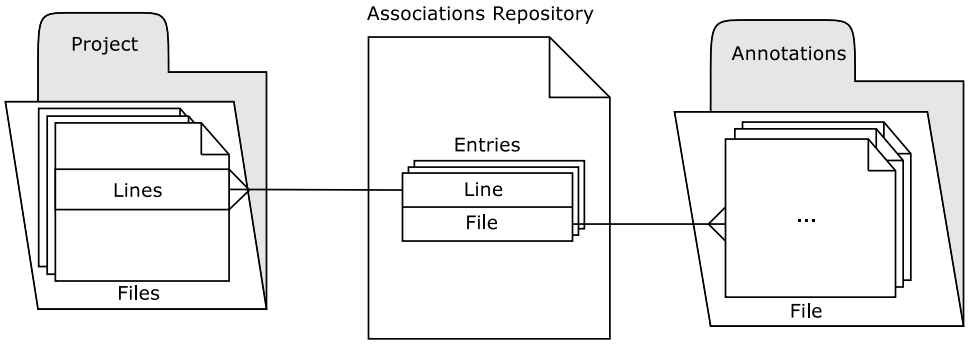


Figure 3.2: Entity-Relationship Diagram

3.6 View

Figure 3.3 describes the User Interface for the system.

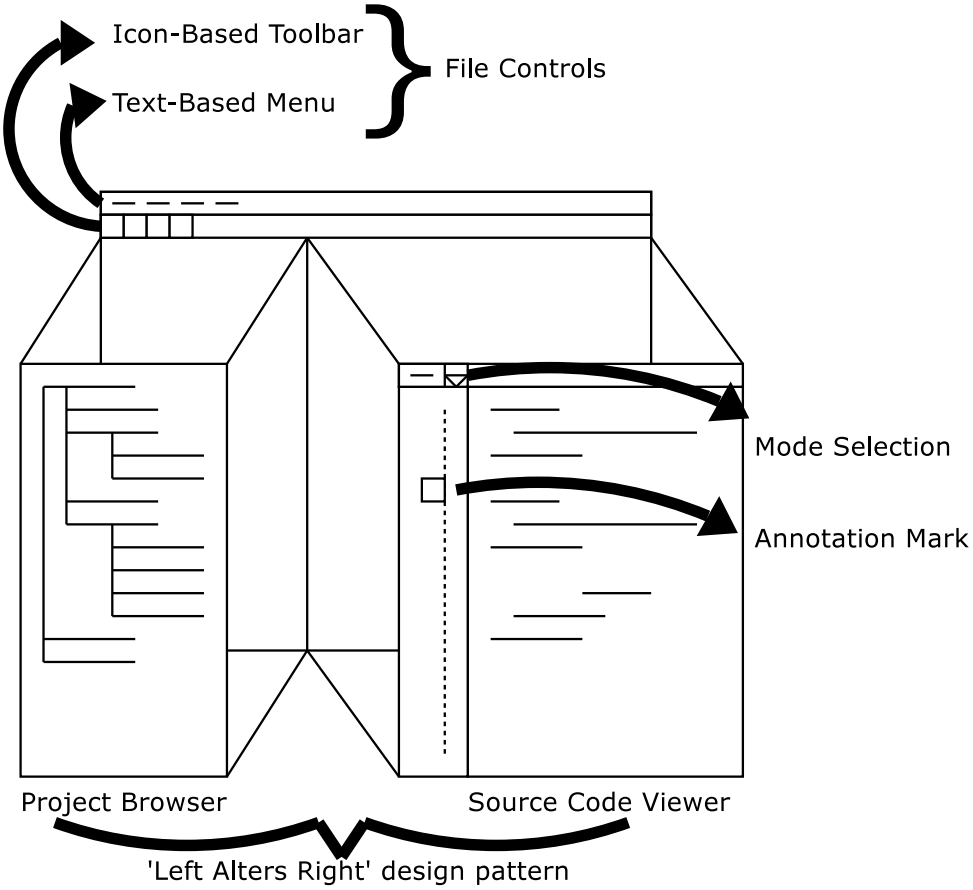


Figure 3.3: User Interface

3.6.1 High-Level Breakdown

The user interface consists of three main components: the File Controls; the Project Browser and the Source Code Viewer. Each component is described in detail below.

The Project Browser and Source Code Viewer are arranged in accordance with the 'Left alters Right' design pattern, popularised by file managers such as XTree and Windows Explorer.

Actions performed in the left pane result in changes to the right pane.

3.6.2 File Operations

The file describing the associations needs to be stored and retrieved from disk.

There are a set of common operations that many programs employ to manipulate files. These operations are:

New To create a new, empty file.

Open To open an existing file.

Save To commit changes to the file structure to disk.

Save As To save the current state to a new file.

Close To close the currently opened file.

Familiarity with one program which employs these operations means that the user can carry across some experience to an entirely new program, even if the files they manipulate are radically different.

People from different cultural backgrounds are better suited to different types of interface control. A study has shown that people from a Chinese cultural background are more suited towards controls with an iconic representation [YYC98], whereas people from an American cultural background are more suited towards textual controls. Expert users of a system can be expected to memorise keyboard accelerators to speed up common operations.

The system supports all three modes of operation. For the same reasons of familiarity as with the file operations chosen, the keyboard accelerators chosen are in common with many other programs.

3.6.3 Project Browser

The Project Browser must allow the user to navigate around the file system hierarchy underneath the Project Directory (Section 3.5.1) and select source code files for display in the Source Code Viewer described below.

A common widget in User Interface toolkits is the interactive tree. The tree is represented as a set of lists, where each non-leaf node can be expanded, and the child nodes are drawn as an indented list under the parent.

The hierarchy underneath the Project Directory is a tree-structure. The user can expand and collapse portions of the tree structure at will, ensuring that the display is not cluttered with irrelevant information.

3.6.4 Source Code Viewer

Visual Mechanism

The program must augment the basic browsing facility with a visual indication as to the presence of an annotation (Section 1.3.1). Annotations are associated with lines of source code (Section 3.5.4) and so the Source Code Viewer shall indicate the presence of an annotation by placing a discrete marker into the margin of the line concerned.

The marker should be an informative colour such as yellow, which can be considered a metaphor for informative items in the stationary world, such as highlight-pens and post-it notes.

Out-Of-Date Annotations

Deliverables I1 (Section 1.3.2) and A1 (Section 1.3.3) require annotations that are potentially out-of-date to be marked for verification.

The colour red is often associated with warnings or danger. As a result, red items are very eye-catching. Changing the colour of an annotation to a red hue indicates that there is a potential problem and draws the users attention.

Simplified Association Operations

Deliverable B1 (Section 1.3.1) states that a user must be able to *add*, *subtract*, *edit* and *retrieve* annotations. It was decided in Section 3.3 that a *verify* operation should be supplied to inform the system that an association is up-to-date.

The *add* operation only works in conjunction with a line not previously annotated, whereas *edit*, *delete*, *view* and *verify* work only with previously annotated lines.

add and *edit* are constructive operations which happen to be mutually exclusive. It is therefore possible to replace them with an operation *author* which encapsulates their functionality. This reduces the complexity of the interface presented to the user.

This simplified set of operations could be selected as the current mode that the Source Code Viewer is operating in. Selecting an annotation (E.g. via mouse clicks) would perform the selected operation on it.

Changing an Annotation's File

If the *author* operation results in *edit* behaviour when applied to an existing association, how can the user alter the details of which file is to be associated with that line?

The same functionality can be achieved by removing the existing annotation and authoring a new one in its place.

3.6.5 Dialogue Boxes

A frequently used metaphor in user interface design is that of the 'record card'.

In a traditional punch-card database, each record would have a single punch card containing its data.

In computer systems, the punch-card is replaced with a dialogue box, which allows the user to edit the fields of the entry. The dialogue box visually resembles a punch card.

A dialogue box shall be used to allow the user to alter the specifics of the current project, detailed in Section 3.5.

When authoring a new association, the program will also present the user with a dialogue box prompting the user to select the file which describes the annotation.

As the system is required to run on more than one operating system, the programs used to edit and view annotations will differ. A dialogue box will allow the user to specify the location of the delegate programs.

3.6.6 Testing

Due to the of the difficulty of automating interaction with a graphical user interface, The visual component is the most difficult of the three to test.

There are tools which attempt to record user input from peripherals such as mice and keyboards, and replay the input at a later time. These rely on graphical elements being positions at playback time in precisely the same location as when the input was recorded. An example of such a tool is GNU Xnee[Fouc].

The View relies on events generated by the Model component to display anything interesting. The majority of controls which could be manipulated on the View are responsible for instructing the Controller component to act; and without the Controller are ineffective.

As a result it is difficult to consider the View component in isolation. Testing of the View will be part of ensuring that the complete system works as desired, performed manually.

3.7 Controller

The Controller is responsible for performing operations on the Model. The user of the system will be manipulating the Controller portion of the system via the View.

For example, removing an annotation will be performed by selecting the annotation in the Source Code Viewer, whilst the mode of operation is set to 'Remove'.

This interaction would be translated into a call to the Controller. The Controller's method will not be tied to the specifics of the View: the method could be called by testing code as appose to explicit user interaction. This arrangement was described in Section 3.4.

3.7.1 Manipulating Paths

In Sections 3.5.1 and 3.5.3 it was determined that the Source Code Project and the location of annotations would be identified by a common parent directory each.

These two values may not be accurate for the current system. For example, the project may have been copied from one machine to another, as in the example given in Section 3.5.1, or simply re-organised on the same machine.

The controller should therefore provide methods for changing these values.

Set Source Code Path

input: Desired path.

output: Listeners notified of change. Listeners will include the Project Browser component of the View.

Set Documentation Path

input: Desired path.

output: Listeners notified of change.

3.7.2 Tracking Schemes

The model also maintains a reference to the tracking scheme being employed (Section 3.5.2).

The user of the system may wish to switch to another tracking scheme.

Change Tracking Scheme

input: Desired tracking method.

output: Listeners notified of change.

3.7.3 Manipulating Annotations

The system will need to remove and introduce annotations to the system.

Add Annotation

input: Line to be annotated; path to annotation; source code file to be associated.

output: Listeners notified of change.

Remove Annotation

input: Annotation to be removed.

output: Listeners notified of change.

Verify Annotation

input: Annotation to be verified.

output: Listeners notified of change.

3.7.4 Testing

Due to the low cohesion between the Controller and View components, the Controller can be tested without the presence of a graphical user interface.

A small software driver could be written to manipulate the controller, and test the results of the Controller's output for consistency.

3.8 Programming Language Semantics

Deliverable A1 (Section 1.3.3) states that the system should 'understand' the semantics of a programming language. This should improve the granularity of change that can be observed.

It was noted in Section 3.5.2 that the existing schemes could suffer from false positives and false negatives. These are the result of the difference between file change and semantic change.

By basing a decision on the semantics of the source code, a scheme should suffer from comparatively few false positives, and no false negatives.

3.8.1 Semantic Components

What components of a programming language should be identified and considered when assessing whether annotations may be out of date?

Imperative Languages

Imperative languages have the concept of a 'variable' which are used to maintain a handle on a particular value.

Imperative program slices take a program and a variable name, and return all the statements within the program that effect the variable. Slicing is discussed in Section 2.6.

If the set of statements that are returned by a slicer for a variable change, then the underlying program has been modified.

If annotations were restricted to program variables, a program slicer could be used as part of a semantically-aware tracking scheme.

Functional Languages

Pure functional programming languages do not have the concept of a program variable. The basic building block of a functional program is a function in the mathematical sense - where the result of a function with identical input does not vary between invocations[3]. This allows the mathematical technique of equational reasoning to be applied to programs.

A function is defined in terms of other functions, and operates on data types.

[3]This is called referential transparency.

A given function's semantic meaning is dependent on that of the functions and data types it is composed from and how they are combined.

3.8.2 Program Representation

Section 2.7.4 described ways for programs to be represented and the difficulties re-engineering tools face with parsing such representations.

XML was presented as a possible means to add semantic cues to source code, lightening the burden of parsing on re-engineering tools.

By enriching source code with XML tags, existing XML parsers can be re-used to extract semantic information from the code.

XML Markup

```
sum :: [Integer] -> Integer
sum x:xs = x + (sum xs)
```

Figure 3.4: A sample implementation of 'sum' in Haskell

Consider the Haskell [Teaa] program fragment in Figure 3.4. The entire fragment is a declaration of a function, `sum`. The second line (the implementation) makes reference to the (same) function, `sum`, in a calling context. It also calls the infix operators `cons` (`:`) and `plus` (`+`).

In each case, the operators and functions have a name component and a list of arguments.

Attempting to capture all this information in XML markup results in the code segment in Figure 3.5.

This is very verbose and harder to read than the original definition. A great deal more information could be captured and presented in XML: the specification and the implementation portions of the definition; the pattern-matching portion of the implementation and the resulting code on the right-hand side of the equals sign.

With a more complex function, the result would be an enormous increase in verbosity.

For the purposes of determining the dependencies between functions, only two pieces of information are necessary: the definition and calling of functions. However, more information is required to detect changes in function definitions.

```
<definition>
  sum :: [Integer] -> Integer
  sum <call>
        <name>:</name>
        <argument>x</argument>
        <argument>xs</argument>
      </call> =
        <call>
          <name>+</name>
          <argument>x</argument>
          <argument>
            <call>
              <name>sum</name>
              <argument>xs</argument>
            </call>
          </argument>
        </call>
</definition>
```

Figure 3.5: 'sum' With XML Markup

3.8.3 Chosen Strategy

The advanced tracking scheme will track dependencies between functions in a functional programming language, with the aid of semantic information captured in XML tags. This allows the implementation to reuse code written for parsing the repository file.

More information is required to detect change in function definitions. In order to keep the implementation simple, the function definition will be converted into a 'digest' representation.

The MD5 algorithm is already required for the message digest tracking scheme described in Section 3.5.2, and could also be used for this purpose.

The disadvantage of this approach is the advanced tracking scheme would be susceptible to false positives in much the same way as the naive schemes described in Section 3.5.2. However this weakness would be limited to within the scope of a function which the annotation depended upon.

3.9 Summary

This section described the evolution of the system's design. The Model-View-Controller architecture was adopted, which imposed a breakdown of the system into three main components: The Model, the View and the Controller.

The data structures that must describe the system are determined, based on the existing system and the desired extensions to it. A format for describing the data structure off-line is designed. These developments are part of the Model component.

The operations that must be applied to this data structure are determined and refined. These form the Controller component.

The interface to these operations is designed, based on existing software trends and user interface principles. This is a description of the view component.

Each component is detailed independently, and the inter-relations considered. Testing and the issues which affect it are discussed.

Finally an advanced tracking scheme was developed which operates on semantic information from a functional programming language embedded in XML.

Chapter 4

Implementation & Testing

4.1 Overview

This chapter describes the implementation and testing phases of the project, including changes to the design developed in Chapter 3; concrete realisations of the User Interface; a detailed look at some of the more interesting implementation challenges and an in-depth description of the development and testing environment, including software tools employed.

4.2 Choice of Language

The project specification does not require a particular language to be used. However, Deliverable B2 (1.3.1) requires a language that supports building a graphical user interface. In addition, the project must be demonstrated on a University lab computer.

4.2.1 Graphical User Interface

Most modern programming languages have support for graphical user interfaces.

Graphical toolkits such as Qt[Tro], GTK+[Pro] and Tk[Teab] boast large collections of language bindings. GTK+ claims to have bindings in 30 programming languages.

4.2.2 University Facilities

The University provides lab computers running the Microsoft Windows XP operating system. The university also provides time sharing servers running the Sun Solaris UNIX operating system.

At the time of writing, there is also the provision of an experimental service running Red Hat GNU/Linux. However, the availability and reliability of this service is not guaranteed.

4.2.3 Experience

A major factor in deciding upon the main programming language for implementation is the experience of the developer.

Languages which have been introduced to us as part of the Degree in Computer Science include C, C++, Java, Haskell, Perl and PHP.

4.2.4 Chosen Language

The Java programming language has been chosen as the primary implementation language.

The Java system differs from most programming systems in that the binary files produced for execution are machine-independent. That is, a Java class prepared on one operating system can be executed on another, provided that a Java run time interpreter is available for that system.

Java compilers and run time interpreters are available for all three of the university systems listed above.

Java has been the main language for education as part of the Computer Science degree, and I am comfortable using it for large scale development.

There are a number of graphical user interface toolkits available. The choice of toolkit for the user interface is described in Section 4.3.1.

4.3 Reused Components

A number of software components were reused during the implementation of this system.

4.3.1 User Interface Toolkit

Cross platform user interface toolkits for the Java programming language are built on top of the native functionality of the operating systems which Java supports.

Three such toolkits are Advanced Widget Toolkit (AWT), Swing and Simple Widget Toolkit (SWT) [Foud]. These toolkits are available for each of the three systems listed in Section 4.2.2.

Performance

Java provides two APIs for user interface development as part of the core API: AWT and Swing.

AWT is a relatively low-level API, providing access to a number of UI building-blocks (known as 'Widgets'). AWT takes a lowest-common denominator approach towards supporting widgets, providing a widget only if it is available natively across all platforms that Java officially supports.

Swing provides a more complete set, providing more complex widgets built from multiple AWT ones. Is even if a widget is available on the native platform that a Java application is running on, it may not be used if the same widget is not available on another supported platform.

The Standard Widget Toolkit (SWT) is an alternative API for user interface development. The approach of SWT is to always use a native widget if available.

A single native widget has a greater performance than a similar widget composed from multiple native ones. As a result, SWT in general has better performance than Swing and greater functionality than AWT.

Prototypes

Prior to implementing the system, a number of prototypes were developed to test the suitability of the various toolkits.

It was found that none of the proposed toolkits supplied a text widget with a margin, which is an aspect of the system's initial user interface design.

SWT provided a styled text widget which could have individual lines styled separately. Swing provided similar widgets, but styling individual portions of them was a complex affair.

Choice of Language

As a result of these experiments, SWT was adopted as the toolkit for developing the graphical user interface.

4.3.2 XML Parser

The decision was made in Section 3.5.5 to use XML as the storage format for association information. The two main approaches towards parsing XML are the Simple API for XML (SAX) and the Document Object Model (DOM).

Simple API for XML

SAX Parsers work on an event-based principle. A SAX Parser object is built, and calls methods in a Handler object for each 'event' that it generates whilst parsing the XML file. Events include:

startElement An XML element's start tag has been encountered.

endElement An XML element's end tag has been encountered.

characters Character data has been encountered.

SAX Parsers can be validating or non-validating. A validating parser will check that the document being traversed fits a Document Type Definition (DTD), and throw exceptions if it is found to deviate from the definition. Both varieties of parser will throw exceptions if the XML document is not well-formed.

SAX is suited towards the parsing of small documents. The responsibility for interpreting the contents of tags and building data structures is left to the programmer. SAX has a very minimal memory footprint, releasing memory used to store parts of the document being parsed once they have been traversed.

Document Object Model

The Document Object Model (DOM) is a means for applications to manipulate a parsed XML document. DOM has been designed independent of any language or platform.

Typically, a DOM library will be implemented on top of a SAX library, although the programmer will not be involved with the underlying parser.

DOM takes responsibility for building and maintaining a data structure corresponding to the XML document.

Chosen Approach

The official Java runtime available on University computers is bundled with the Apache Xerces XML parser, which implements both DOM and SAX.

SAX was chosen as the API of choice. Choosing DOM would require the Model's interface to be written in terms of the DOM, meaning that the other components would need to be XML-aware. this would deteriorate the architectural design of the system.

The implementation of the XML parsing portions of the system are covered in more detail in Section 4.6.

4.3.3 Message Digest Algorithm

Section 3.5.2 described the Message-Digest technique for detecting file change.

MD5 is a message-digest algorithm which is commonly used for verifying data integrity across insecure or unreliable network connections.

The system uses an implementation by Juho Santeri Paavolainen[1], which is distributed for use under the GNU Lesser General Public Licence (LGPL)[Foub].

4.4 Changes to Design

4.4.1 Visual Indication of Annotations

The mechanism for indicating the presence of an annotation was decided to be a mark in the margin to the left of the line in question (Section 3.6.4).

Due to limitations of the SWT toolkit, it was decided to instead change the background colour of the line in question.

Whilst implementing the margin is technically possible, it would require significantly more work to implement. Since the initial design decision was arbitrary, it was sensible to instead work on other aspects of the system.

[1] md5

4.5 User Interface

Figure 4.1 details the UI as implemented, for comparison with the original UI design (Figure 3.3).

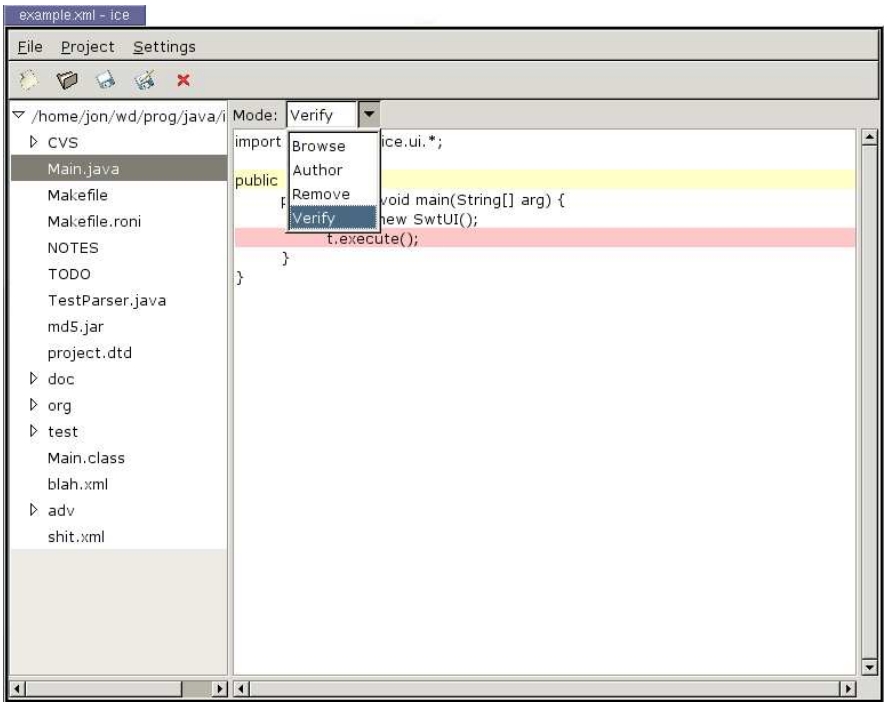


Figure 4.1: Implemented User Interface

This screen-shot demonstrates the Project Browser, the Source Code Viewer, the menu bar and the tool bar.

4.5.1 Project Browser

The Project Browser has the project's root directory expanded, revealing a number of files and sub-directories, including the file `Main.java` which is selected.

The user can expand and collapse sub-directories as required, to present only the information they require.

4.5.2 Source Code Viewer

The Source Code Viewer has the contents of the file `Main.java` on display. There are two annotations visible: one marked in a yellow colour; another in red.

The red colour is an indication that the file has been modified since the association was created, and the annotation may be out-of-date (see Section 3.6.4).

Also visible is a drop down 'combo box', containing the four modes of operation: *Browse*, *Author*, *Remove* and *Verify*.

4.5.3 Dialogues

Section 3.6.5 described the use of Dialogue Boxes as a User Interface metaphor for record cards. Figure 4.2 demonstrates the three dialogue boxes specified in the design.

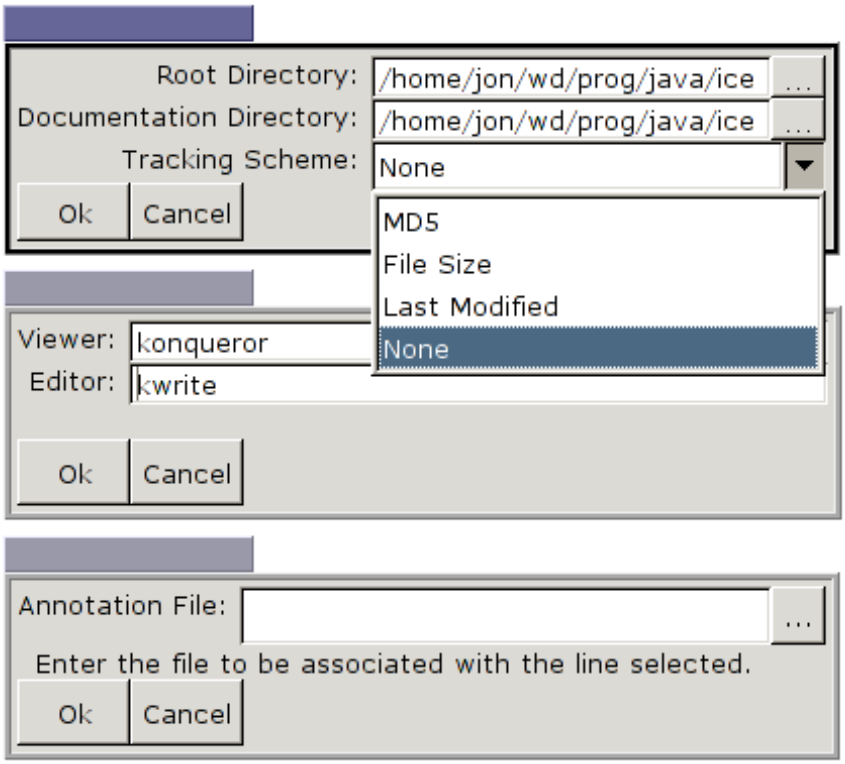


Figure 4.2: Dialogue boxes from top to bottom: Project details; Delegated programs; New Annotation.

In addition to these, a number of toolkit-supplied dialogues were used for selecting files and directories on the host computer.

4.6 XML Parser

The XML parsing approach used is the SAX method, as described in Section 4.6.

The parser bundled with the Java 1.4 API is non-validating. The job of determining whether or not the document meets the required definition is the burden of the programmer.

4.6.1 XML Project Syntax

The XML application used to store the details of associations described in Appendix B includes various tags, of which *dir*, *doc*, *track*, *name*, *line*, *filename* and *checksum* can contain character data.

For one class to properly handle such a document, each of the parser event methods `startElement`, `endElement` and `characters` would need to check to determine which tag resulted in the event being thrown, and whether or not that tag was expected.

The resulting state information would make such a class large and unwieldy.

A considerably more desirable arrangement would be for one class to handle each separate tag: the class handling the *project* tag would expect to encounter only *dir*, *doc*, *track* or *file* as start tags - anything else would be erroneous.

In order to use separate, specialised handler instances, a delegate handler class was devised.

Delegate Handler

The delegate handler class maintains a stack of potential delegate objects, and delegates all parsing events to the object at the top of the stack.

The stack is initialised to contain a `RootHandler` class, which is a specialised Handler for the root element.

When the `startElement` event method is triggered, a specialised handler will check to see what element was responsible, and whether or not that element was anticipated.

When an anticipated start tag is encountered, the specialised handlers create an instance of the handler required for that tag, and push it onto the stack. From this point, events triggered by the parsing API will be captured by the new handler.

The Class Diagram in Figure 4.3 details the relation of the delegate and specialised handler classes. Not shown is the inheritance relationship between these

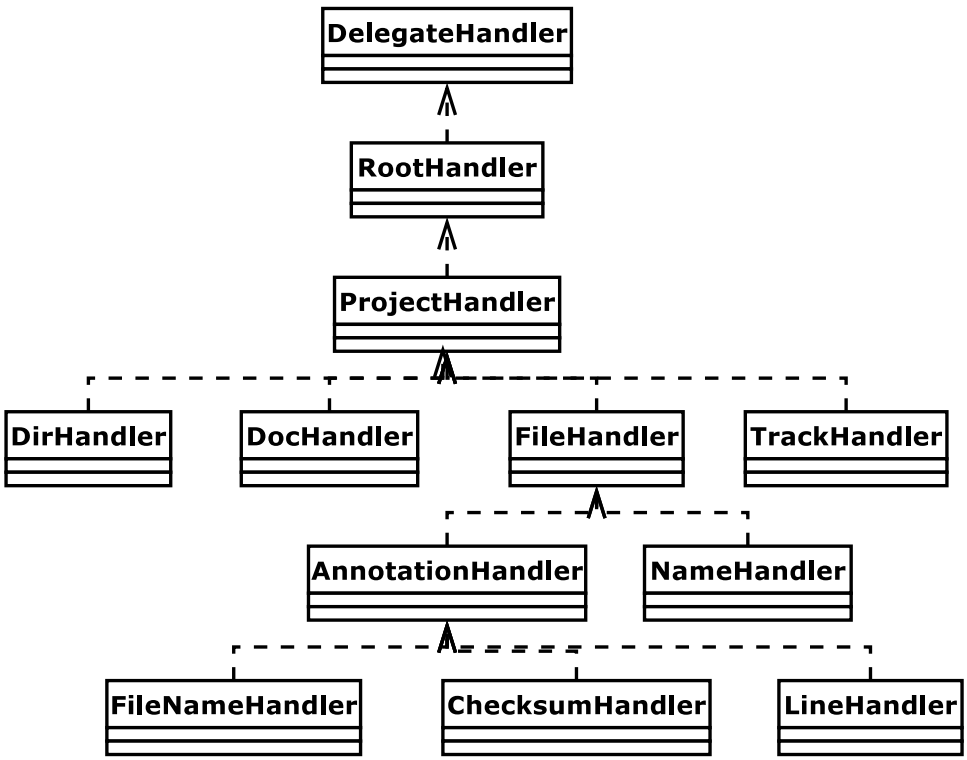


Figure 4.3: Parser Class Diagram

classes and the `DefaultHandler` parser class - this is omitted for clarity.

Leaf Nodes

Each specialised handler requires the constructing 'parent' handler to pass a reference to itself in as a constructor argument.

When a specialised handler detects that its work is done (via an anticipated `endElement` event), it instructs the parent delegate to pop it from the stack.

The parent handler removes the delegate from the stack, and integrates the information gathered by the delegate into its own.

Valid XML

This arrangement also simplifies the job of testing that the document being parsed conforms to the Document Type Definition.

Each delegate throws a `SAXParserException` as a result of the three situations

outlined in Section 3.5.6.

4.7 Development Environment

A number of tools were used repeatedly to aid the development process. These included `cvs`, `make`, `find`, `grep`, `diff`, `patch`, `perl`, and `xmllint`. Their use is described in the following sections.

4.7.1 Configuration Management

The configuration management tool CVS [Fog99] was used to maintain the source code for the project[2].

With CVS, the master copy of source code is kept in a CVS repository. The code is checked out into a working-directory when in use. Changes to the working directory are only incorporated into the repository when the CVS command is invoked with the instructions to commit changes.

The main advantage of such an arrangement is the aiding concurrent development of software. However, CVS is also useful for stand-alone development: the state of the source code project can be reverted back to the state it was in at any commit during development.

This ability to undo severe changes encourages the developer not to be wary of making radical changes, which improves productivity.

4.7.2 Compilation and Running

The project was implemented using the Java programming language. The program `javac` was used to compile the source code files into executable class files, and the program `java` to execute them.

Both of these programs are part of the Java 2 Software Development Kit, version 1.5.0.

4.7.3 Build Management

The build process was managed by the `make` utility [Fel79]. This allowed common rules and tasks to be defined once, and executed multiple times.

[2]CVS was also used to maintain this document!

The Makefile was responsible for cleaning stale binary objects (including `.class` files) from the project directory; running automated tests; gleaning statistical information from the source files and managing the execution of both `java` and `javac`.

4.7.4 Statistics

During development, gathering certain statistics aided decision making.

The UNIX tools `find` and `grep` provide a powerful means of searching for keywords and expressions across multiple files.

The keyword 'TODO' was used in much of the source code to indicate incomplete functionality, or potential sources of bugs. Counting the number of 'TODO' occurrences in the source code and determining their concentration helped to ensure that no part of the system was neglected.

Additional Advantage

A common problem with the CVS system is that files newly introduced into a working directory are not automatically incorporated into the repository on the next commit - they need to be explicitly added with the command `cvs add`.

One way of solving this problem is to always insert the RCS keyword `$Id$` into each source code file. The CVS system will replace this string with an expanded identification string for each file committed.

`find` and `grep` can be used to search for files containing the unaltered string `$Id$`, which can only be present in source code files which are not part of the repository.

4.7.5 Tests and Experiments

A variety of tools were used to aid both automated testing and to perform of experiments.

Testing

Testing of the parser component was automated. A set of purposefully invalid XML documents were produced in accordance with the problems listed in Section 3.5.6. A driver class was written using Java which attempted to parse each document. This used the parser component of the model.

This driver class would return an indication of success or failure to the operating system. A `perl` script collated these return values and compared them against a table of required values. The results of this comparison were then transformed into a table for inclusion in this document.

The table of results for the last build of the system are in Appendix C.1.

Evaluation

Section 5.4 describes the procedure used to evaluate the tracking schemes used to detect file change. This evaluation was aided by many of the same tools used to test the parsing component. For each scheme, the following was performed:

1. A file `Main.java` is generated. This file is listed in Figure D.1.
2. An XML file is generated which describes a source code project in the current working directory. An annotation is associated with a key line in `Main.java`. This line is indicated alongside the file in Appendix D.1.
3. The test is enacted. For example, the touch test runs the POSIX program `touch` on `Main.java`, updated its Last Modified field. A full description of the tests is available in Section 4.7.5.
4. A driver program reads in the XML file generated previously and reports on whether the association is tainted.
5. This result is captured and compared with the desired result: Some tests perform a semantic change and so the association should be tainted whilst others do not.
6. The results are transformed into an entry in Table 5.2.

Steps 2 and 4 are performed using small Java programs.

The program used for step 2 uses the controller to create a new project, and write it out to disk. Step 4 uses the controller to read in this file, parse it to build the model, and check the annotations within.

The execution of these programs and the collection and arrangement of data in Steps 5 and 6 are performed using `perl` scripts.

Each test performs a different action to enact change. The precise details of this change differ from test to test. Most tests make use of the program `patch` to apply a prepared patch-set to the file. The tests are described in Section 5.4.

4.8 Semantic Tracking Scheme

The semantic tracking scheme for the advanced deliverable was introduced in Section 3.8. The decision was made to base the tracking on dependencies between functions in a functional programming language and to enrich the language with XML tags, encapsulating the semantic information.

4.8.1 Annotation Granularity

In order for the advanced tracking scheme to be used alongside the existing tracking systems, the annotation mechanism still works on a line basis. As a result, the advanced tracking scheme needs to maintain a mapping of line numbers to function calls and definitions.

The SAX API (Section 4.3.2) provides a mechanism for determining the line number of the file when an event is generated.

4.8.2 Example Source Code

XMLHaskell is the name given to Haskell source code marked up using the XML application defined in Appendix B.2. An example of an XMLHaskell program is available in Figure E.1.

This program has the following XML tags:

<program> XML requires a single root tag to encapsulate the entire document. The <program> tag serves this purpose.

<def> The <def> tag defines a function definition. The attribute 'name' informs the XML parser of the name of the function, removing a need to mark up the name explicitly.

<call> the <call> tag represents the calling of a function.

Figure 4.4 details the dependencies between the functions in `project.lhs`. This graph was generated by piping the output of the parser in a suitable format to `neato`, a graph drawing tool.

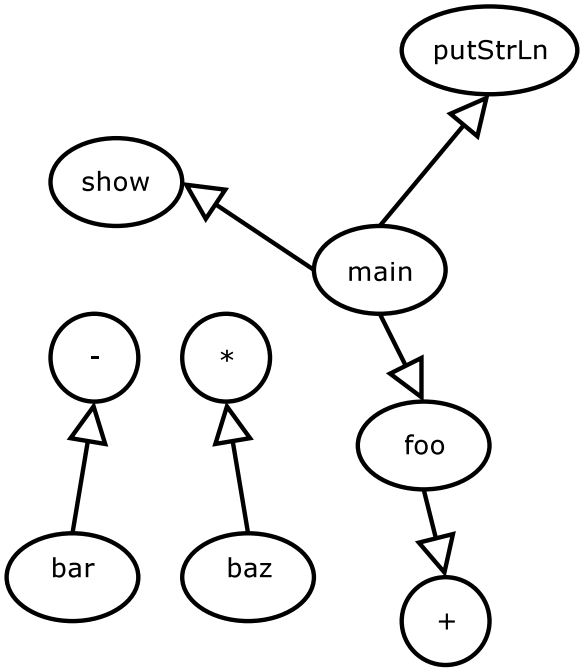


Figure 4.4: Dependency Graph for `project.lhs`

4.8.3 XML Parser

A delegating, stack-based XML parser was built to create a dependency graph from the marked-up source code, in much the same way as the repository files (Section 4.3.2).

This hierarchy of classes consisted of a chief stack-delegate and a distinct delegate class for each of the three tags described above.

When a <`call`> handler is finished, the parent <`def`> handler adds the name of the called function to a list which the currently defined function depends on. <`call`> tags can only occur within <`def`> tags.

4.9 Testing

Testing is used to determine how successful the project deliverable is in meeting its objectives. Methods of testing the system have been considered since the Design Chapter: Sections 3.5.6, 3.7.4 and 3.6.6 for the Model, Controller and View respectively.

There are a number of different strategies that can be employed to test a program.

It is generally accepted that testing can never prove that a program is entirely 'correct', as exhaustive testing is infeasible for all but the most trivial of programs.

However the focus for this project is for the system's basic functionality to work correctly in normal circumstances, so that the impact made by the system on the comprehension process can be measured. Bugs which only manifest themselves in obscure or unusual circumstances, whilst undesirable, are of a lower priority.

4.9.1 Strategies

Black Box Testing

Black box testing is a strategy whereby the components of a software program are tested in isolation from one another.

Each component is considered only by its external interfaces: What output should be observed for a given input. The implementation details are not taken into account.

White Box Testing

White box testing aims to test every logical path of execution through a program's statements. The principal behind this is that every line of code is tested at least once.

This can be very time consuming as for even a relatively small program there can be an enormous number of separate paths of execution.

In addition, the precise behaviour of a program statement depends upon the state of the program's structures at the time that the statement is executed. A statement may be tested in one state and deemed correct, whilst a bug exists which is only evident in a different state.

Chosen Strategy

The program must be tested sufficiently for operation during experiments and demonstrations.

Each of the separate architectural components could be tested in isolation due to the MVC architecture model adopted (see Section 3.4).

4.9.2 Parser

The parser component was tested by taking each requirement from the DTD, and formulating an XML document which negated that rule. This was covered in Section 3.5.6.

A driver class was written to construct and execute the Parser, in isolation from the other components of the system. A list of all such test documents, combined with the result of attempting to parse them are available in Appendix C.1.

Part III

Results

Chapter 5

Results & Evaluation

5.1 Overview

This chapter details the evaluation of the project.

The drawbacks and limitations of measuring the impact on program comprehension are discussed; a walk-through is used to demonstrate the system's satisfaction of deliverables B1 (Section 1.3.1), B2 (Section 1.3.1) and I1 (Section 1.3.2) and the performances of the schemes mentioned in Sections 3.5.2 and 3.8 for detecting file change are evaluated.

5.2 The Impact On Program Comprehension

If this project has been successful, the process of comprehension should be aided by the use of the system, either by an improved quality of understanding, or by the same quality of understanding being reached in a shorter time.

It is feasible to measure the time taken for a test subject to be comfortable with the purpose of a source code segment. It is less feasible to try and judge how thorough and accurate a test subject's understanding of a source code segment might be, as this is highly subjective.

One possible approach towards evaluation would be to time how long each of a set of test subjects provided with sample source code took to reach a state of understanding that they were satisfied with. The time for each subject would be recorded.

Some of the test subjects would be able to use the project deliverable. If those with the aid of the system reached a satisfactory state of understanding more quickly,

the improvement could be attributed to the system. However there are a number of problems with this approach:

Documentation The program deliverable is effective only when documentation exists to be associated to the source code.

There are no effective ways of determining whether two sources of documentation are of equal benefit to the comprehension process: it is heavily dependent on personal bias towards language and structure.

If the test subjects without the system were not provided with the same documentation as those with it, then any improvement measured could be attributed to the documentation alone, and not the system.

All test subjects must therefore have access to the same documentation.

Equivalence of Code Complexity There exist metrics that can be used to determine the complexity of source code. One such metric is Cyclomatic complexity [McC76].

Such metrics however server only as a guide, and cannot provide any guarantees as to how comprehensible a sample of code is.

Domain knowledge (in the case of code samples with an identifiable domain) of the test subjects and experience with particular clichés employed in the code samples would introduce a bias to the results.

Equivalence of Test Subject Skill Test subjects would have a varying degree of experience with programming, code reading and domain knowledge of the code samples which would influence their performance in the experiment. This difference in aptitude would have to be corrected for, and therefore quantified.

One solution would be for each test subject to perform the experiment with a control sample. The differences in performance on this could be used to adjust performance scores in other code segments.

As a result of these shortcomings, this evaluation will not be pursued.

5.2.1 Time Improvements

In Section 3.2 it was noted that searching for associations in the original paper-based system could potentially be an exhaustive operation. This was a result of the associations not having a consistent granularity or any organisation.

The implemented system solves both of these problems. A consistent granularity of one line per annotation was decided upon (Section 3.5.4). Searching for annotations is no longer necessary, as they are presented to the user as a highlighted line on top of the source code.

As a result, using the system to aid comprehension is quicker and more versatile.

5.3 Walk-Through

The system developed allows a user to define a repository of association information for a software project and add, retrieve and delete association information from this repository using an interactive environment.

The annotations themselves are viewed and edited by external programs – the system allows the user to specify which programs and calls them when the user performs browse and author operations.

The software project operated on is not modified by the system during operation.

Annotations are visually indicated in the user interface and annotations which are flagged as potentially out-of-date are indicated using a distinct colour.

In order to demonstrate that these objectives have been met a walk through was performed. In order to reduce duplication reference is made to screen shots provided in the previous chapter where necessary.

5.3.1 Creating a Repository

Before the system can be properly tested, a repository of associations is needed. The first action is to create one. A sample source code project is copied to a new location on the host computer, and the system started.

'New' is selected from the tool bar, which presents the user with the New Project dialogue. This is the first dialogue in Figure 4.2 (previous chapter).

The location of the source code project is entered into the 'Root Directory' text field. A temporary directory is selected for 'Documentation Directory'. 'Tracking Scheme' is left as *None* for now, although it will be changed later.

5.3.2 Authoring & Browsing Annotations

The system now looks like Figure 5.1. The Project Browser contains a single item, the root directory of the project. Expanding this reveals the immediate children

including a file `proxy.c` . Clicking on this entry displays the contents of `proxy.c` in the Source Code Viewer.

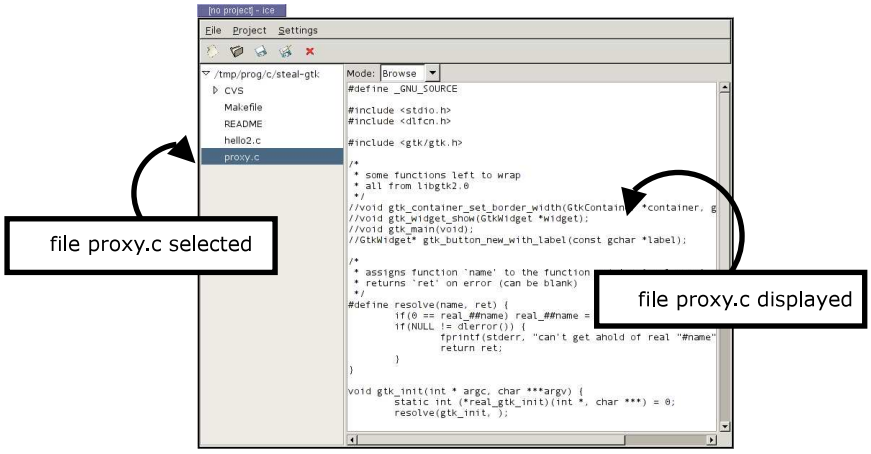


Figure 5.1: The file `proxy.c` in the Source Code Viewer

The Source Code Viewer's drop down box is opened and the operational mode 'Author' selected[1]. A particular line within the file is selected with the left mouse button. A 'New Annotation' dialogue is presented. This is the third dialogue in Figure 4.2. From here the path to an annotation is entered.

Once an annotation is chosen, the dialogue is dismissed and the system has marked the selected line with a yellow background (Figure 5.2)[2].

Changing the operational mode to 'Browse' and selecting the line results in a web browser displaying the annotation. Changing the operational mode back to 'Author' and selecting the line results in a text editor being called, with the annotation file open.

The annotation file in this case is written using HTML. A tag <`img`> is added, which references a prepared image on the host machine. This demonstrates the ability to use rich content in annotations.

5.3.3 File Change

The tracking scheme 'MD5' is selected for the project using the project dialogue, which is re-invoked via the menu.

[1]The operations were modified for the sake of interface simplicity so that *add* and *edit* are represented as the operation *author*. This is discussed in Section 3.6.4.

[2]During implementation, the means of indicating an annotation was changed – see Section 4.4.1.

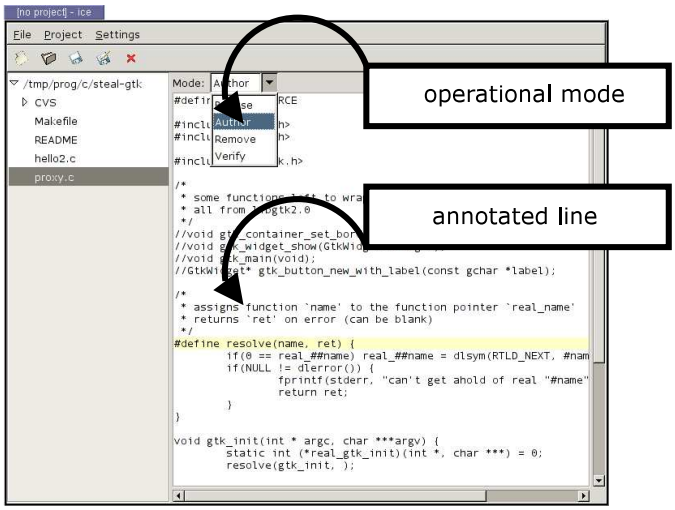


Figure 5.2: An Annotated Line

At this point, the repository is saved to disk, by calling 'Save' from the tool bar. Since the repository has not yet been saved, the result is 'Save As' behaviour. A dialogue is presented from where the user chooses a file to store the associations. Using a text editor, a small change is made to the file `proxy.c` . Selecting `proxy.c` in the Project Browser results in it being re-displayed in the Source Code Viewer. The annotation is now displayed in a red colour, indicating the tracking scheme's warning that the annotation may be out-of-date (Figure 5.3).

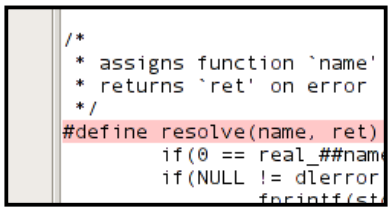


Figure 5.3: A Potentially Out-of-Date Annotation

A new association is made with a different line in `proxy.c` . It is not marked red, as the underlying file has not changed since *it* was created. A mixture of trusted and suspect annotations can exist for a single file.

Now, the change made earlier to the file `proxy.c` is undone, which returns it to its previous state.

`proxy.c` is now re-displayed in the Source Code Viewer. The first annotation has been restored to a yellow colour, as the stored MD5 hash corresponds to the current file. However, the new annotation, associated with the modified `proxy.c` , is now marked red (Figure 5.4).

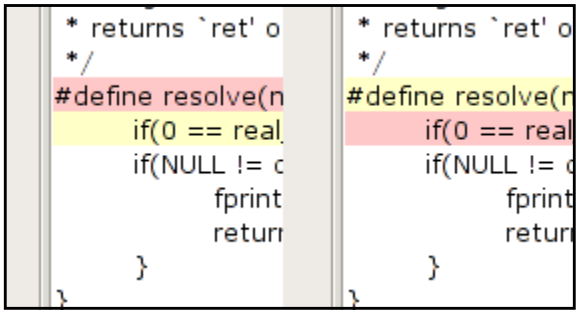


Figure 5.4: A Mixture of Suspect and Verified Annotations

5.3.4 Verification

Changing to the operational mode 'Verify' and selecting the suspect annotation results in the annotation changing colour to yellow.

An inspection of the repository file before and after performing this operation reveals the stored data for the MD5 scheme changing.

5.3.5 Removing Annotations

Changing to the operational mode 'Remove' and selecting one of the annotations results in its deletion. The line is no longer high-lighted, and inspecting the repository file proves that the entry has been removed.

5.3.6 Summary

The functionality of by deliverables B1, B2 and I1 has been achieved.

5.4 Tracking Schemes

Section 3.5.2 described four schemes for monitoring file change. The schemes *File Size*, *Line Cache*, *Last Modified* and *MD5* were implemented, in addition to *None*, which represents no attempt used to monitor file change.

5.4.1 Strategy

The file change tracking is designed to aid the detection of out-of-date and therefore potentially misleading documentation. An ideal scheme will only detect changes which result in a change in the semantic meaning of the source code which has been annotated. However, a realistically attainable scheme may suffer from false positives and false negatives, as discussed in Section 3.3.1.

The tests used for these schemes were written with two purposes in mind. One purpose is to ensure that the schemes work in the most rudimentary of situations: where no change has occurred, and where a semantic change has occurred.

The following tests were used to ensure predicted operation:

nc No change is made to the file at all. The desired response from the schemes would be to not indicate that change had occured.

cc A change is enacted on the file which results in a change of meaning for the line annotated. All of the schemes should report that change has occurred.

The second purpose was to determine how susceptible each scheme is to false negatives and false positives. These tests were designed with the implementation of the schemes in mind, so as to find inherent weaknesses.

	Scheme				
Desired Result	None	Last Modified	File Size	Line Cache	MD5
False Positive	N/A	touch	indent	indent	indent
False Negative	cc	*	ar2	ar2	?
Normal	nc	nc /cc	cc		cc

Table 5.1: Table of Test Strategies and Desired Outcomes

Table 5.1 lists the tests used to achieve a desired outcome for each scheme. Since *None* will never report a suspect annotation, it cannot suffer from a false positive. No method was conceived to make the MD5 scheme report a false negative.

In addition to these, a test multi is used to demonstrate the limitations of observing change at a file granularity.

touch The UNIX command `touch` was used to update the files' Last Modified field. This is designed to generate a false positive in the scheme which relies on the last modified field. Such an event is likely to occur in practise, as discussed in Section 3.5.2.

indent The GNU tool `indent` was ran over the file. This tool re-arranges the layout of the source code to meet coding conventions of various bodies - thus this is likely to occur in practise. However, the semantics of the code remain the same, and as such reports of change are false positives.

ar2 A plus character is substituted for a minus character. This does not result in a change in file size. The character was chosen such that the semantics of the code on the line which carries an association is changed, the association should be considered out-of-date.

multi The multi test revolves around two files, `main.c` (Figure D.1) and `hi.c` (Figure D.1). The former calls a function in the latter. The test modifies `hi.c` but not `main.c`. The annotation is associated with a line in `main.c` which calls the function in `hi.c`. The annotation is therefore out-of-date.

5.4.2 Performing the Tests

For each scheme, a project file is generated; change enacted and the project file parsed to see if the scheme detects a change. More details of the environment and tools used for this experiment are covered in Section 4.7.5.

5.4.3 Results

	Scheme				
Test	None	Last Modified	File Size	Line Cache	MD5
cc	N	pass	pass	N	pass
ar2	N	pass	N	N	pass
touch	pass	P	pass	pass	pass
multi	N	N	N	N	N
indent	pass	P	P	P	P
nc	pass	pass	pass	pass	pass

Key	
pass	correct result
P	false positive
N	false negative

Table 5.2: Results of Tests against Tracking Schemes

5.4.4 Evaluation

With the exception of the multi test, the *Last Modified* and *MD5* schemes were the only two to not suffer from a false negative. Of these, *MD5* suffered the fewest false positives.

Both schemes operate at a file-level granularity. If such schemes were to operate at a finer granularity, there is the possibility of fewer false positives. However, the schemes would need to know the relationship between regions of the files: which regions resulted in a change of semantic meaning in others. This is the approach adopted by the semantically-aware scheme, described in the next section.

The *Line Cache* scheme operates at a line-level of granularity. However this experiment has shown that such a scheme is unreliable, as the meaning of a given line of source code can depend entirely on another area. The multi test demonstrates that consideration needs to be made across files in a project.

A scheme which stripped out regions of source code which did not have a semantic meaning[3] and stored a digest of the result would not be sensitive to false positives when such things were modified. Such a scheme would be expected to pass the indent test. However a semantically-aware scheme can be expected to perform considerably better.

5.5 Semantic Tracking Schemes

The semantic tracking scheme described in Section 3.8 is tested in much the same way as more simple schemes. Indeed, most of the test from the previous section have been carried across, reworked to be based on the program `project.lhs` (Figure E.1, Appendix E).

The semantic tracking scheme is referred to as the *Advanced* scheme in the following sections.

5.5.1 Tests

The indent test described above was dropped for testing this scheme. The `indent` program was designed for languages derived from C, such as C, Objective C, C++, Java and C#. It does not provide support for Literate Haskell or XML, which are the two base syntaxes from which `project.lhs` is written.

[3]Such as comments and white-space in languages where white space is ignored.

Two new tests are introduced:

ncc This test performs a semantic change to a function which does not directly impact the annotated line.

w This test alters the white-space within a function which does directly affect the meaning of the annotated line. However the meaning of the function does not change.

Neither test should result in the annotation being marked as out of date.

5.5.2 Results

	Scheme					
Test	None	Last Modified	File Size	Line Cache	MD5	Advanced
w	pass	P	P	pass	P	P
ncc	pass	P	P	pass	P	pass
cc	N	pass	pass	N	pass	pass
ar2	N	pass	N	N	pass	pass
touch	pass	P	pass	pass	pass	pass
nc	pass	pass	pass	pass	pass	pass
multi	N	N	N	N	N	N

Please refer to Table 5.2 for a key to this table.

Table 5.3: Results of Semantic Tracking Schemes

Table 5.3 lists the result of performing these tests on the existing schemes, combined with the semantic analysis scheme.

5.5.3 Evaluation

The schemes tested in Section 5.4 performed identically as desired.

The 'ncc' test created a context change which did not affect the annotated line. The schemes which did not suffer a false negative in the previous section, *MD5* and *Last Modified*, both reported a false positive. However, the advanced scheme did not consider the modified function as relevant to the annotated line and correctly ignored the change.

The 'w' test was an adequate replacement for the missing 'indent' test, resulting in most of the schemes reporting false positives. Unlike the 'indent' test, the *Line*

Cache scheme did not report a false positive, as no change was made to the annotated line.

The advanced scheme reported a false positive for the 'w' test. This is a reflection on the limitations of the design and was predicted in Section 3.8.3.

Semantic information greatly improves the accuracy of a scheme to detect change in source code. However, the scheme presented is significantly more complex than the more naive schemes evaluated in the previous section. In addition, this scheme is only useful when used in conjunction with the specially prepared language XMLHaskell (4.8.2).

A production system would have to have explicit support for each programming language it was to be used in conjunction with. It would require a more complex compiler-front end in order to extract the semantic information, as most source code the user could be expected to face would not be augmented with XML tags.

Just like the more naive schemes, the advanced scheme failed to track change across file boundaries. A production system would need to consider multiple files in a project and possibly even beyond the project: candidates include standard libraries supplied with the programming language and third-party code.

5.6 Summary

- The impact of the system on the process of comprehension has been considered. It has been determined that the implemented system is significantly easier and faster to use than the original paper-based system, however there are significant practical limitations that hamper attempts to measure impact in more depth.
- The functionality required of the system by deliverables B1 (Section 1.3.1), B2 (Section 1.3.1) and I1 (Section 1.3.2) has been met and demonstrated using a walk-through of the system.
- The schemes for detecting file change (Section 3.5.2) have been compared against each other using examples of file change (Table 5.1). It has been determined that many of the schemes can suffer from false negatives and are therefore not practically useful.
- The advanced scheme (Section 3.8) has been compared to the more naive schemes, which has demonstrated the advantages of considering semantic information.

However the advanced strategy suffered from practical design limitations (Section 3.8.3) and relied on the presence of additional semantic markup, which may not be present in practise.

None of the implemented schemes considered change across multiple source code files either internal to the project or externally to supporting libraries. In a production environment, such consideration would be essential.

Chapter 6

Conclusion

6.1 Achievements

The objectives for this project listed in Section 1.2 have been met. A computer version of the paper-based annotations system described in Sections 1.1.1 and 3.2 was developed. This system provided all of the functionality of the paper system, whilst also presenting annotations in a visual manner alongside the source code they are associated with.

Evaluating the impact of the system on the process of comprehension was determined to be an involved, complex issue (Section 5.2) and as such was limited to considering the increased efficiency of the system over the paper-based alternative.

The goal of detecting out-of-date annotations was met by devising a variety of schemes (Section 3.5.2), criteria for evaluation (Section 3.3.1) and determining their shortcomings (Table 5.2).

An advanced scheme for detecting out-of-date annotations was developed which considered the semantic structure of the underlying source code. This was aided by enriching a functional programming language with XML tags describing a subset of the semantic structure (Section 4.8.2), allowing existing XML parsing tools to be re-used.

6.2 Extensions

6.2.1 Authoring Annotations

The responsibility for displaying and viewing annotations was delegated to external programs, as the focus of this project was on managing the associations (Section 3.5.3).

HTML was identified as a suitable representation for annotations for three reasons: it supports diagrams and rich content; HTML viewers are available on most desktop computers in the form of web browsers; HTML requires no special tools to write.

Ideally, authoring annotations would be a quick activity requiring as little effort as possible. The process of annotating code should not place an additional burden on the comprehension process [FM87].

Therefore, an extension of this system would be to provide authoring facilities internally. These should be natural and intuitive to use without requiring significant experience.

6.2.2 Incorporation of other Re-Engineering Tools

The system could be extended to provide the functionality of other re-engineering techniques, by incorporating new or existing re-engineering tools.

Bundling multiple re-engineering tools into one framework means that a user need only become familiar with one environment. In addition, the tools can make use of common functionality, simplifying implementation in much the same way development tools can in Integrated Development Environments.

6.2.3 Relation Strengths/Grades

In this project, associations between code and comments have all had the same 'strength'.

In the Elucidative programming model [Nør00], relationships between documentation entries and program bundles (the equivalents of our source code and annotations) are either *strong* or *weak*. Strong relationships are those where the documentation directly explains the code it is related with. Weak relationships are those where the associated code is only referenced in the documentation.

In this project, associations are displayed in the source code browser for the user

to manipulate at will. However, it could be desirable to have the version-tracking facility applied to other forms of documentation.

Howver if a large amount of non-annotation documentation was associated using the system the browser would become cluttered with highlighted lines, reducing the effectiveness of the browsing facility.

If associations could have different grades, then it could be arranged so that only those of a particular grade were visually indicated in the text browser. Documentation that was not strictly an annotation could be assigned a different grade, and harness the power of the change-tracking facilities without cluttering the visual interface.

A possible extension to this project would be to investigate how desirable it would be to use the change-tracking facilities in conjunction with other forms of documentation and means for supporting these without compromising the original aim.

6.3 Further Work

This project has touched on a number of avenues for further research.

6.3.1 Code Representation

In order to save time when developing the advanced deliverable, an XML-based source code representation was devised (Section 3.8.3). This allowed the reuse of existing tools for manipulating XML.

The shortcomings of 'traditional' flat text representations for source code have been researched and alternative representations have been invented (Section 2.7). These include a number of XML-based representations.

Further work in this area includes categorising the parsing needs of a variety of re-engineering tools and determining the feasibility of reusable parser components.

6.3.2 Program Slicing for Tracking Change

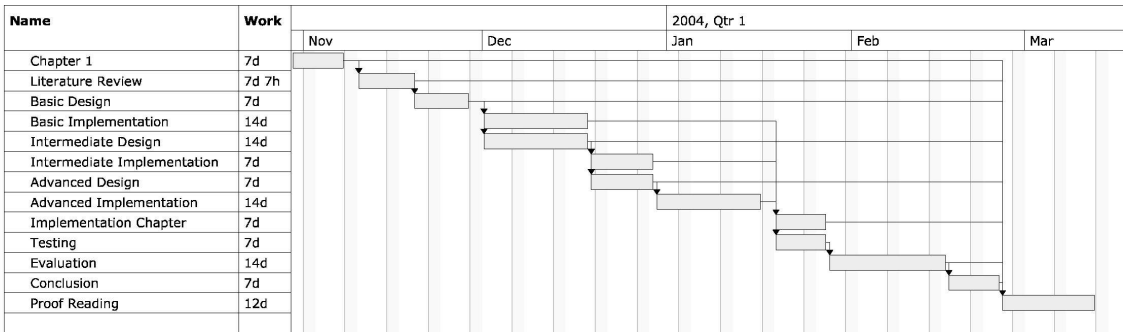
Section 3.8.1 introduced the possibility of using Program Slicing as a means of detecting out-of-date annotations.

Although this technique was not used during this project, this is an interesting and unexplored use of slicing technology, remaining an avenue of exploration.

Appendix A

GANTT Chart

Figure A.1: GANTT Chart



Appendix B

DTDs

```
<!--
  root element is project: has a root dir for each
  of documentation and source code, a true/false
  value for track change, and 0+ files
-->
<!ELEMENT project   (dir,doc,track,file*) >
<!ELEMENT dir       (#PCDATA) >
<!ELEMENT doc       (#PCDATA) >
<!ELEMENT track     (#PCDATA) >

<!--
  files have names (relative from dir above); 0+
  annotations
-->
<!ELEMENT file       (name,annotation*) >
<!ELEMENT name       (#PCDATA) >

<!--
  annotations have line numbers, filenames
  (relative from doc above); optional checksum
-->
<!ELEMENT annotation (line,filename,checksum?) >
<!ELEMENT line       (#PCDATA) >
<!ELEMENT filename   (#PCDATA) >
<!ELEMENT checksum   (#PCDATA) >
```

Figure B.1: DTD for the XML project dialect

```
<!ELEMENT program (def*)          >
<!ELEMENT def     (#PCDATA|call)* >
<!ATTLIST def name CDATA #REQUIRED>
<!ELEMENT call    (#PCDATA)       >
```

Figure B.2: DTD for XMLHaskell

Appendix C

Testing Results

C.1 Parser

Test	Result
empty <doc>	passed
empty <line>	passed
empty <dir>	passed
empty <filename>	passed
empty <name>	passed
extra tag inside <filename>	passed
spare tag inside <line>	passed
extra tag inside <annotation>	passed
spare tag inside <name>	passed
extra tag inside <track>	passed
spare tag inside root	passed
extra tag inside <doc>	passed
spare tag inside <checksum>	passed
spare tag inside <project>	passed
extra tag inside <dir>	passed
extra tag inside <file>	passed
missing <name>	passed
missing <doc>	passed
missing <line> in <annotation>	passed
missing <track>	passed
missing <dir>	passed
missing <filename> from <annotation>	passed

Table C.1: Table of Testing Results: Parser Component

Appendix D

Intermediate Evaluation

D.1 Sample Programs

The sample program in Figure D.1 was used as part of the evaluation of the Intermediate deliverable, Section 5.4.

```
 1  public class Main {
 2
 3        public static int sum(int [] a) {
 4                int i = 0;
 5                for(int ii = 0; ii < a.length; ++ii) i+=a[ii];
 6                return i;
 7        }
 8
 9        public static void main(String[] arg) {
10                int i[] = {1,2,3,4,5};
11                System.out.println("sum: " + sum(i));
12        }
13 }
```

Figure D.1: `Main.java`

Line 11 is associated with an annotation. This line was chosen because it calls the method `sum` defined on line 3. The plus operator in this method is subtituted for a minus in the test *cc*. This is a semantic change to the program, which effects the meaning of the code under the association.

Figures D.1 and D.1 list the files used as part of the 'multi' test. The annotation is associated with line 4 of `main.c`, which depends on the file hi.c.

```
#include "hi.c"

int main() {
        hi();
        return 0;
}
```

Figure D.2: `main.c`

```
#include <stdio.h>

void hi() {
        printf("hi\n");
}
```

Figure D.3: `hi.c`

D.2 Repository File

Figure D.4 lists a repository file that was generated for each tracking scheme during evaluation.

```
<?xml version="1.0" ?>
<project>
<dir>/home/jon/wd/prog/java/ice/test/track</dir>
<doc>/home/jon/wd/prog/java/ice/test/track</doc>
<track>None</track>
<file>
<name>Main.java</name>
<annotation>
<line>1</line>
<filename>Main.java</filename>
</annotation>
</file>
</project>
```

Figure D.4: A sample Repository File

Note that this example lists the tracking scheme as 'None' - the correct value for each tracking scheme was substituted in turn. In addition, the other tracking schemes include a <`checksum`> tag as part of the file entry.

Appendix E

Advanced Evaluation

The following XMLHaskell program was used to test the advanced deliverable.

```
<?xml version="1.0" ?>

<program>
  <def name="foo">
    foo :: [Integer] -> Integer
    foo (x:xs) = (<call>+</call>) (<call>foo</call> xs) x
    foo [] = 0
  </def>
  <def name="bar">
    bar :: [Integer] -> Integer
    bar (x:xs) = (<call>-</call>) (<call>bar</call> xs) x
    bar [] = 0
  </def>
  <def name="baz">
    baz :: [Integer] -> Integer
    baz (x:xs) = (<call>*</call>) (<call>baz</call> xs) x
    baz [] = 0
  </def>

  <def name="main">
    main = <call>putStrLn</call>
         $ <call>show</call>
         $ <call>foo</call> [1,2,3,4,5]
  </def>
</program>
```

Table E.1: Sample Program for Advanced Deliverable

Bibliography

[App98] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 1998.

[Bad00] Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.

[BKM86] Jon Bentley, Donald E. Knuth, and M. Douglas McIlroy. Programming pearls: A literate program: A `WEB` program for common words. *Communications of the Association for Computing Machinery*, 29(6):471–483, June 1986.

[Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[Bro95] Jr. Frederic P. Brooks. *The Mythical Man Month*. Addison–Wesley, anniversary edition, 1995. ISBN: 0–201–83595–9.

[CMM] Michael L. Collard, Jonathan I. Maletic, and Andrian Marcus. Supporting document and data views of source code.

[Con] World Wide Web Consortium. Hyper text mark-up language (html). http://www.w3.org/MarkUp/.

[dBSV98] M. Van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the 6th International Workshop on Program Comprehension*, page 108. IEEE Computer Society, 1998.

[EKW99] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. GraX – An Interchange Format for Reengineering Tools. Fachberichte Informatik 5–99,

Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1999.

[Fel79] Stuart I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–65, 1979.

[Fle88] Nigel T. Fletton. *Documentation for Software Maintenance and the Redocumentation of Existing Systems.* PhD thesis, School of Engineering and Applied Science, University of Durham, 1988.

[FM87] John R. Foster and Malcolm Munro. A documentation method based on cross-referencing. In *Proc. Conference on Software Maintenance*, pages 181–185, september 1987.

[FM88] N. T. Fletton and M. Munro. Redocumenting software systems using hypertext technology. In *Proceedings of the Proceedings of the International Conference on Software Maintenance 1988*, pages 54–59. IEEE, IEEE Computer Society Press, 1988.

[Fog99] Karl Franz Fogel. *Open Source Development with CVS.* Coriolis Group Books, 1999.

[Foua] Apache Foundation. Ant build tool. http://ant.apache.org/.

[Foub] Free Software Foundation. Gnu lesser public licence (gnu lgpl). http://www.gnu.org/copyleft/lesser.html.

[Fouc] Free Software Foundation. Gnu xnee. A tool to record, replay and distribute X events. http://www.gnu.org/software/xnee/.

[Foud] The Eclipse Foundation. The standard widget toolkit (swt). http://www.eclipse.org/.

[Fri95] Lisa Friendly. The design of distributed hyperlinked programming documentation. In S. Fraisse, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design, Montpellier, France, 1–2 June 1995*, pages 151–173, 1995.

[GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, Berlin, 1993. Springer-Verlag.

[GO97] Keith Gallagher and Liam O'Brien. Reducing visualization complexity using decomposition slices. In *Proc. Software Visualisation Work.*, pages 113–118, Adelaide, Australia, 11–12 1997. Department of Computer Science, Flinders University.

[Ham88] Eric Hamilton. Literate programming—expanding generalized regular expressions. *Communications of the Association for Computing Machinery*, 31(12):1376–1385, December 1988.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schrr. Gxl: Toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 162. IEEE Computer Society, 2000.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[Knu86a] Donald E. Knuth. *TEX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[Knu86b] Donald E. Knuth. METAFONT*: The Program*, volume D of *Computers & Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[Les75] M. E. Lesk. Lex - A Lexical Analyzer Generator. CSTR 39, Bell Laboratories, October 1975.

[Let86] S. Letovsky. Cognitive processes in program comprehension. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79. Ablex Publishing Corporation, 1986.

[LS86] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 41–40, May 1986.

[McC76] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[MDJ+97] Lehman M.M., Perry D.E., Fernandez-Ramil J., Turski W.M., and Wernick P.D. Metrics and laws of software evolution - the nineties view. In *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*. IEEE Computer Society, 1997.

[MR87] Malcolm Munro and David J. Robson. An interactive cross reference tool for use in software maintenance. In *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, pages 64–70, 1987.

[Nel] Michael L. Nelson. A survey of reverse engineering and program comprehension.

[Nør00] K. Nørmark. Requirements for an elucidative programming environment. In *Proceedings of IWPC'2000, Limerick, Ireland*, June 2000.

[Pro] The GNOME Project. Gtk+ user interface toolkit. http://www.gtk.org/.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[Shn80] Ben Shneiderman. *Software Psychology, Human Factors in Computer and Information Systems*. Winthrop Publishers, Inc., 1980.

[SI86] Elliot Soloway and Sitharama Iyengar. *Empirical Studies of Programmers*. ABLEX Publishing Corporation, Norwood, New Jersey, 1986.

[Sta84] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, September 1984.

[SWM00] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[Teaa] The Haskell Team. Haskell. http://www.haskell.org/.

[Teab] Tk Team. Tk graphical toolkit. http://tcl.sourceforge.net/.

[Thi86] Harold Thimbleby. Experiences of 'Literate Programming' using `cweb` (a variant of Knuth's `WEB`). *The Computer Journal*, 29(3):201–211, June 1986.

[Tom87] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1-2):31–46, 1987.

[Tro] Trolltech. The qt toolkit. http://www.trolltech.com/products/qt/.

[Tur37] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936–1937.

[UDCT93] G. A. Di Lucca U. De Carlini, A. De Lucia and G. Tortora. An integrated and interactive reverse engineering environment for existing software comprehension. In *International Workshop on Program Comprehension*, pages 128–137. IEEE Computer Society Press, 1993.

[VHG87] Christopher J. Van Wyk, David R. Hanson, and John Gilbert. Literate programming: Printing common words. *Communications of the Association for Computing Machinery*, 30(7):594–599, July 1987.

[vZ93] H.J. van Zuylen, editor. *The REDO Compendium*. John Wiley & Sons, 1993.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Win98] Nwe Nwe Win. Incremental redocumentation using literate programming. Master's thesis, University Of Durham, 1998.

[YK93] E.J. Younger and K.H.Bennett. Model-based tools to record program understanding. In *IEEE Second Workshop on Program Comprehension*, pages 87–85. IEEE, 1993.

[YYC98] Gavriel Salvendy Yee-Yin Choong. Design of icons for use by chinese in mainland china. *Interacting with Computers*, 9(4):417–430, 9 1998.

[ZL96] Andreas Zeller and Dorothea Lutkehaus. DDD - a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996.