Assisted Software Comprehension

Final report

Russell Wood

June 2003

Supervisor	Susan Eisenbach
Second Marker	Jeff Magee

Abstract

Software comprehension is the process of gaining an understanding of how a software system functions. Common development practices – working in large teams, code reviews, use of third-party libraries and a diverse range of languages – mean comprehension activities play major part in modern software projects. The need to gather an understanding of a software system is only likely to become more regular an activity, as the size and number of software systems increases.

Gaining an understanding of a software system requires the practitioner to draw on information scattered throughout the source code, a time consuming and laborious task. If, to assist the practitioner, the information can be gathered automatically and presented in a manner that highlights relevant information, the time and effort required can be reduced.

This document presents a framework for the analysis and visualisation of source code, with the intention of facilitating understanding and related tasks. The framework is designed to promote reuse of analysis and visualisation code across multiple programming languages.

A frontend is presented that demonstrates a subset of the features provided by the framework.

Acknowledgements

I would like to thank my supervisor Susan Eisenbach for her help and advice throughout the course of this project.

I would also like to thank my family and friends for their support, advice and tea breaks.

Finally I would like to thank Annys Green for her proof-reading and her patience.

Contents

List of Figures

Introduction

> The contents of the world are not just there for the knowing but have to be grasped with suitable mental machinery.
>
> *How the Mind Works*
> Stephen Pinker,
> Peter de Florez Professor of Psychology,
> MIT

1.1 Overview

Software developers have to continually cope with complexity in their work – it is a major factor affecting their productivity. For this reason there have been many efforts over the past decades to combat complexity through new programming paradigms, software processes and verification methods.

Program source code tends not to be the most immediately informative representation when it comes to understanding a program; There is a lot of information contained within code that is either periphery or obscured by other elements. One intuitively useful approach to facilitating program understanding is to extract the information that is most important.

Although the use of "best practice" or of verification during development goes a long way towards easing understanding, processing programs to generate alternative representations is the most far-reaching solution for a number of reasons:

- the existence of legacy or poor quality software,
- the continual increase in system size,
- poor adoption of "best practice" or verification during new developments,

- the high staff turn over (demise of the "Job for Life").

This project focusses on provision of alternative program representations to aid the comprehension of source code, in a manner that promotes reuse across different source languages.

1.2 Project scope

The objectives of this project are:

1. To investigate ways in which program understanding can be facilitated, focussing on alternative program representations.

2. To investigate ways in which these procedures can be applied to multiple source languages.

3. To produce a framework that demonstrates multiple techniques for easing program understanding, allows multiple developers to collaborate on the understanding process and is designed to support multiple source languages.

This project develops a framework for operations designed to ease the understanding of complex software systems, in such a way that support can be easily added for further source languages. The framework includes substantial built in functionality for easing understanding, and can be extended with other understanding-promoting techniques. The framework is designed to support remote access for collaboration between developers. A basic frontend is provided to demonstrate the features of the framework.

Figure 1.1 provides an analogy to the behaviour of the software product of this project: undigestible software sources are processed to another form, which is transformed and transferred over a network for the use of a developer, who benefits from the more digestible form. It is the intention that the framework (the meat-grinder) can process sources in a variety of languages, and (with the help of assorted food-processing attachments) generate a variety of different types of output, as described in the background section.

For the purposes of demonstration, the framework includes the ability to transform source code in the Kenya language into an analysable form. Kenya is a teaching language, developed and used at Imperial College, that provides a simplified Java-like syntax. Source browsing functionality is provided as well as a number of visualisations, covered in detail the background section, including call and caller graphs and source overviews. In addition the system allows annotations to be added to the source code. To minimise the cost of keeping up to date with a changing source base, the framework makes use of inter-file dependencies to determine required changes.

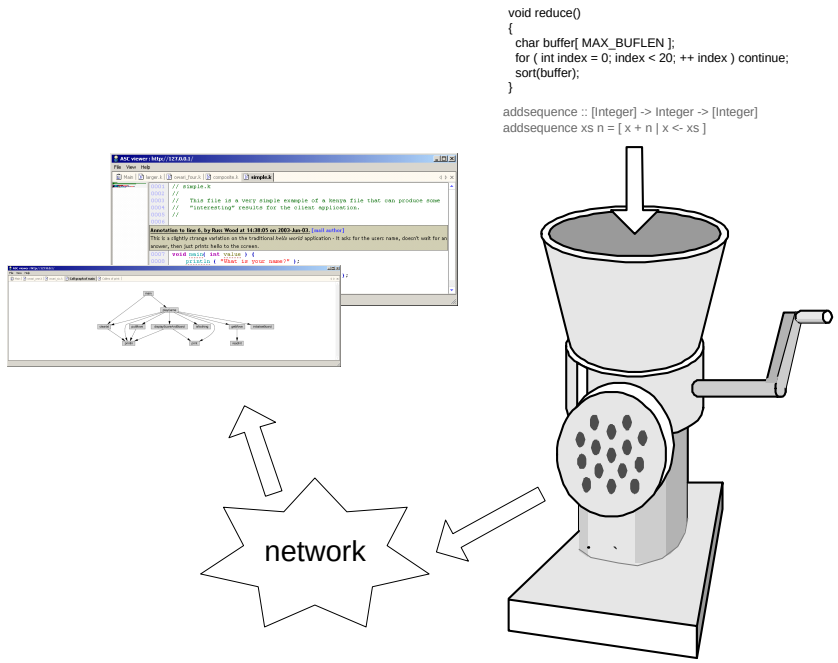

Figure 1.1: Analogy to the behaviour the framework developed

1.3 Report structure

There are five major sections to this report. The background section provides a motivation for software comprehension and outlines a number of techniques that can be used for the purposes of easing understanding. Additionally, it briefly examines existing program comprehension tools and covers existing research into application of software comprehension techniques to multiple source languages.

The discussion section builds on the background research by examining a number of requirements and potential problems in the production of a framework for software comprehension. A specification for the framework is produced.

The design and implementation sections detail the design decisions made, and explain the motivation for those decisions. A medium level design is presented along with the implementation of that design.

The report ends with an evaluation of the project, in terms of the specification, user opinion and perceived achievement of project goals. Potential further work is discussed, both on the framework and in the problem domain in general.

Background

> The most costly outlay is time.
>
> Antiphon, Athens, circa 479-411 B.C.

In addition to providing a motivation for this work and software comprehension in general, this section aims to cover three points:

- How can program understanding be facilitated?
- What is the state of program understanding in research and commercial environments?
- How can techniques be implemented so as to apply to multiple languages?

2.1 A motivation for software comprehension

Maintenance is one of the major constituents of the software life cycle. It is not surprising then that a significant amount of money is spent every year by companies that simply want to prevent the devaluation of their investment in in-house systems. The systems produced by the industry today will be the focus of maintenance activity tomorrow, meaning that as time passes this expenditure is likely to increase.

It is apparent that when dealing with a complex software system, it is difficult to make changes without an understanding of the interactions that go on within that system. The maxim *Time is Money*[1] is one of the main reasons why the ability to quickly build up an understanding of a piece of software is desirable in a commercial setting. Often the staff who originally developed the system are no longer with the company, or as is increasingly the case, portions of the software

[1] The epigraph at the beginning of this chapter, by an Athenian orator named Antiphon, is believed to be the origin of this phrase

may be licensed from a third party; in both of these situations in order for the code to evolve it is necessary for someone to take the time to understand the system. The financial implications of this are manyfold, some sources indicate that in large organisations, fifty percent of the development activity is devoted to maintenance tasks[BLT78, LS80]; improvements in this area could drastically reduce costs as a consequence of the reduced man hours.

Alternative approaches that do not require a great understanding of a product have been suggested, the most well known of these being Refactoring[Fow00]. Refactoring is a process intended to improve the design and maintainability of programs through sets of rules for performing code transformations safely. Many of the transformations lend themselves to semi-automatic operation, leading to refactoring browsers that perform the transformations for you. The original refactoring browser was produced for Smalltalk[BR] and more recently browsers have been produced for more widely used languages such as Java[Ben02].

Although refactoring provides many benefits such as safe (meaning preserving) transformations, it still has a number of issues that lead back to the need to understand the system being worked on. It is not always simple to apply, as this excerpt from Refactoring[Fow00] indicates:

> C++ has language features, most notably its static type checking, that simplify some program analysis and refactoring tasks. On the other hand, the C++ language is complex, largely because of its history and evolution from the C programming language. Some programming styles allowable in C++ make it difficult to refactor and evolve a program.

In addition to this there is the problem of knowing when to apply a particular refactoring to the program. Even when experienced in applying the transformations, it is necessary to examine the software to determine where transformations are appropriate and useful. It could therefore be said that refactoring is an end result oriented process (the software *will* become more maintainable and robust) that is complemented well by the ability to understand the software being transformed. For more information on the pros and cons of refactoring, the reader is referred to Chapter 13 of the aforementioned book by Martin Fowler.

2.2 Facilitating program understanding

Obtaining an understanding of a piece of software in the absence of suitable design information has traditionally been an issue of the developer sitting with the source code and an editor, searching through to find the information needed for the task at hand. Such an approach makes it very difficult to gain an overall view of the software structure and reduces the usefulness of the experience carried over to the next maintenance task.

If the maintenance task is a one-off procedure, such as finding why a particular variable has an invalid value, this may be the fastest way to solve the problem; ho-

wever, the presence of issues like this is symptomatic of a software product that may have many underlying problems. Whenever there is the potential for the software to require many of these "quick fixes" and repairs, it may be a better option to simplify each procedure by re-engineering all or part of the software. To re-engineer, it is necessary to understand how different software sections interact, how the software is structured and, most of all, what effect any changes may have.

There are many approaches to making software more understandable. This project observes that the techniques can be roughly divided into groups and suggests the following taxonomy for program understanding techniques:

Abstraction based approaches aim to highlight a particular aspect of the software by discarding information not closely related to that aspect.

Summary based approaches try to provide some insight into where to work on code to make it more understandable, by providing metrics that allow comparison of some factor affecting complexity or ease of understanding.

Embellishment techniques involve adding detail to the software without changing its representation, for example adding cross reference or type information to the code.

Transformational approaches, such as refactoring, aim to improve the design and in doing so make it more understandable.

Of the categories listed above, this project will concentrate on the first three - Abstraction, Summary and Embellishment.

While there are systems in existence that provide comprehension related functionality, these tend to be designed to cater for a single language. There are classes of language that are similar, for example languages such as C and Pascal, C++, C# and Java; producing a framework for software comprehension that can work with a number of similar languages should be possible, and is one aim of this project.

From the systems mentioned later in this section and others mentioned on [the], it is clear that there are many approaches to producing software to assist with program understanding. These offerings all appear to be commercially successful[2], which suggests that even basic comprehension functionality is considered sufficiently useful to justify buying the product.

The commercial software examined always contains syntax highlighting and some degree of cross referencing. Most provide pattern based searches and class hierarchy diagram generation. The more elaborate representations tend to be present in the in-house software[BE96] or research implementations[LA93]. The interfaces provided are very variable and appear in most cases to have been a secondary concern. On the whole they are cluttered or tend to use outdated toolkits, producing an unfamiliar interface.

There are many potential end-user visible features of a code comprehension tool, a number of which are covered in the following sections. The following subsections

[2] those which are marketed as commercial products rather than given away as free software

```
// algorithm.hpp
//   This is a file in which to place stl-algorithm style
//   code that may be useful somewhere in the project.
//
// revisions:
//   27/12/2002 - added fill_from
//
#ifndef SUPPORT_ALGORITHM_HPP_
#define SUPPORT_ALGORITHM_HPP_

namespace support
{
    // fill_from <>
    //   provides the functionality of std::copy for when
    //   the input iterator type does not provide operator+
    //   ( cf. std::istream_iterator ). The intended use it
    //   to pre-allocate a string length using resize() and
    //   then copy in to the allocated memory.
    //
    // exceptions:
    //   std::bad_alloc in addition to any that may result
    //   from access to the underlying container via the
    //   iterators.
    //
    // pre:
    //   At least as many items can be read from the sequence
    //   of _i as there are in the range |_begin,_end)
    //
    template < typename It1, typename It2 >
    inline void fill_from ( It1& _i, It2& _begin, It2 _end )
    {
        while ( _begin != _end )
        {
            *_begin = *_i;
            ++_i;
            ++_begin;
        }
    }
}

#endif // SUPPORT_ALGORITHM_HPP_
```

```
// algorithm.hpp
//   This is a file in which to place stl-algorithm style
//   code that may be useful somewhere in the project.
//
// revisions:
//   27/12/2002 - added fill_from
//
#ifndef SUPPORT_ALGORITHM_HPP_
#define SUPPORT_ALGORITHM_HPP_

namespace support
{
    // fill_from <>
    //   provides the functionality of std::copy for when
    //   the input iterator type does not provide operator+
    //   ( cf. std::istream_iterator ). The intended use it
    //   to pre-allocate a string length using resize() and
    //   then copy in to the allocated memory.
    //
    // exceptions:
    //   std::bad_alloc in addition to any that may result
    //   from access to the underlying container via the
    //   iterators.
    //
    // pre:
    //   At least as many items can be read from the sequence
    //   of _i as there are in the range |_begin,_end)
    //
    template < typename It1, typename It2 >
    inline void fill_from ( It1& _i, It2& _begin, It2 _end )
    {
        while ( _begin != _end )
        {
            *_begin = *_i;
            ++_i;
            ++_begin;
        }
    }
}

#endif // SUPPORT_ALGORITHM_HPP_
```

Figure 2.1: Comparison of listings with and without highlighting

aim to present a selection of these features in an approximate order of expected familiarity and complexity.

2.2.1 Syntax highlighting

Syntax highlighting is the use of variation in colour or typeface to distinguish between elements of a structured document such as a program source file. It is one of the simplest ways in which source code can be made more understandable. Whilst not presenting the information in a radically different way, syntax highlighting draws the eyes to key language elements which makes it easier to pick these elements out. Almost every modern programmer's editor or code browser supports customisable syntax highlighting as it is relatively simple to do so using regular expressions or even just keyword highlighting.

If a parse tree is available for a program, syntax highlighting is a simple matter of assigning extremal nodes a style based on their type and reflecting these styles in the source text.

The benefits of syntax highlighting are obvious in the comparison figure 2.1. In my opinion, it is far easier to differentiate between elements in the right hand (syntax highlighted) listing than in the left.

2.2.2 Cross referencing

It's often the case that when looking through a section of code, a developer may wish to examine the declaration of a type, or determine where a particular variable is used. Cross referencing the source code can provide definition-use and use-definition

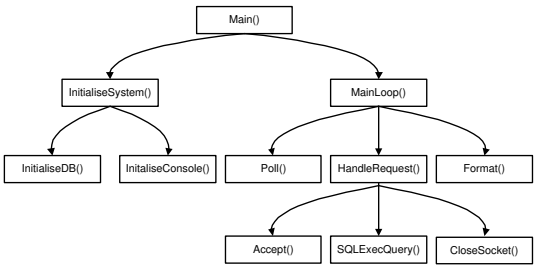

Figure 2.2: An example call graph

linkage for variables, functions and types within a program. In addition to building up a general understanding of the source, cross referencing is particularly helpful when tracking down problems in a large program. The information gathered to perform cross referencing can also be used as a basis for producing crosscutting diagrams and source overviews with usage highlights.

In order to provide cross referencing it is necessary to understand the source language. As a consequence, general purpose editors tend not to provide this functionality. The traditional approach to adding cross reference support to an editor was to use an external tool such as CTags to extract symbol information, or to make use of cross reference output provided by a compiler. More complex modern development environments tend to incorporate some level of understanding of the source languages, for example Microsoft Visual Studio allows the developer to make use of cross references through their Browse Information system.

The features that can be provided through cross reference information include:

- jump to function, variable or type definition,
- list uses of function, variable or type,
- determination of dependencies between separate source files.

2.2.3 Call graphs

A call graph represents a flow of control across the functions in a program. Control flow graph representation allows the user to see the general processing that occurs in the program, particularly in the presence of well-chosen function names. This is useful for extracting algorithm and design information when examining a program with intent to re-engineer. Call graphs can be derived from static or dynamic information to produce static or dynamic call graphs respectively. A static call graph (such as the example above) shows potential calls, whereas the dynamic call graph shows which calls actually occurred on a particular run of the program. Static call graphs may not show all potential calls, particularly in the case where functions are first class objects of the programming language used - this requires data flow and alias analysis to be used.

Call graph is commonly used to refer to a representation that records functions that may be called. It is also useful when understanding and maintaining programs to be able to view the reverse call graph (or caller graph) of a function. The reverse call graph answers the question, "How could I reach this function from elsewhere in the program?", particularly useful when trying to track how a function could receive an erroneous value. It is generally necessary to provide some level of control over call graph generation, as there is a tendency for them to get very complex very quickly. This is particularly the case for object oriented languages, where there tend to be a very large number of small functions with much delegation to methods of other objects.

A detailed survey and discussion can be found in the paper "*An Empirical Study of Static Call Graph Extractors*" by Murphy Et Al[MNGL98].

2.2.4 Metrics

Software metrics can be useful for analysis purposes, but don't tend to help with understanding much more than providing a general indicator of where might be a useful place to start making changes to improve understandability. They are generally quite cheap to provide.

Metrics such as these can also be presented visually, as discussed in [BE95, BE96]. A sense of scale is often more easily obtained from a diagram than a numeric representation, and extremal values of the metric are easier to spot.

Simple metrics

There are a number of metrics that are very simple to calculate and that have traditionally been (mis)used as measures of code quality and development effort. They include:

- **SLOC** stands for Source Lines of Code, and is the number of lines of non-empty lines in a source file that contain something other than comment text.
- **CLOC** is analogous to SLOC but counts non-empty lines containing comments.
- **Comments/Line** ratio has been used as a measure of code quality in the past, with some companies enforcing lower bounds on this value among their employees.
- **Size measures** such as the total number of lines, or size in bytes.
- **Average Function Length** is the average number of lines per function. This can usefully be backed up by a standard deviation to describe the distribution of lengths.
- **Avg/Min/Max name length**

Cyclomatic complexity

Cyclomatic complexity, due to McCabe[MB89] is a measure of the number of independent paths through a program module. It is calculated based on the control flow graph, as:

$$CC = E - N + p$$

where E = The number of edges in the graph.
N = The number of nodes in the graph.
p = The number of connected components.

Cyclomatic complexity is one of the most widely used static software metrics and has found applications in risk analysis, re-engineering and test planning.

Halstead metric

The Halstead metric is calculated as:

$$H = n_1 \log n_1 + n_2 \log n_2$$

where n_1 = The number of distinct operators in the program.
n_2 = The number of distinct operands in the program.

An operand is a data object. An operator is a program keyword, including scope-defining keywords such as BEGIN. . . END (counted as a single keyword). While this is one of the most established metrics, it has problems stemming from the lack of context information involved in its calculation; for example, both nesting depth and relative complexity of operators are not taken in to account.

2.2.5 Annotations

Abstraction is one of the most powerful tools available to the developer. Through the implementation of a product, some of the abstraction present in the original design is often lost. Good commenting practice can help to retain the information that would be lost by describing the intention of the code where it is not immediately apparent.

Unfortunately it is a well known fact that code produced is often poorly commented, a factor of tight deadlines or over-familiarity with the code. When a developer subsequently comes to look at the code, the comments may not adequately convey what they need to know, or may be absent entirely. As a result it can be a lot more time consuming to build up an understanding.

Annotations attempt to fill this gap by allowing the user to virtually add comments to the code as they work through it. This has the advantage of enabling them to

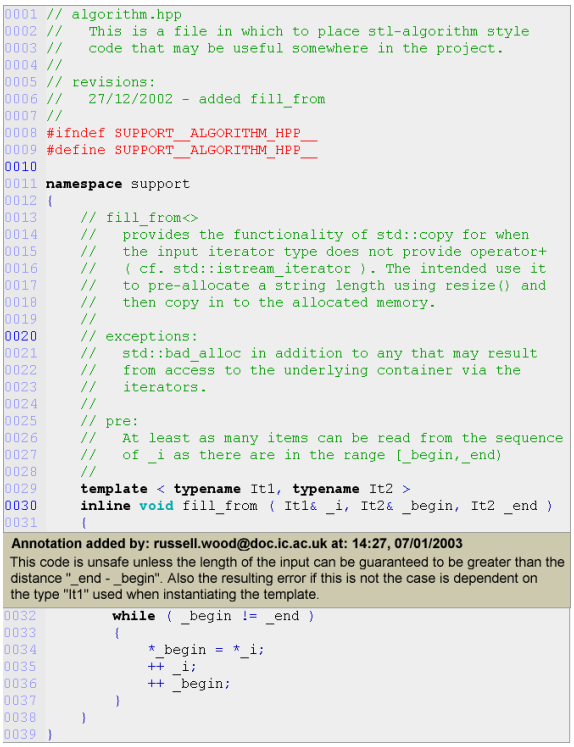

Figure 2.3: A mockup source listing containing annotations

quickly look back over something if they forget its purpose, as well as simplifying task of subsequent users who need to understand the code.

It may be suggested that simply adding comments to the original code would be simpler than providing a "virtual" commenting system. However annotations of this form have two advantages over traditional comments - they don't require write access to the source code and the developer can add as many comments as they like without swamping the code. Figure 2.3 shows how an annotation could look.

2.2.6 Source overview diagrams

Document viewers such as Adobe Acrobat Reader® provide a thumbnail view of a document to allow the reader to quickly see the document structure. There are many ways in which a similar overview can be provided for source code - by using a smaller font or by discarding the textual information completely and representing each character as a single pixel[BE96]. Such diagrams are used to good effect in [KH01], where an overview diagram is used to make obvious the locations of certain code within a file.

Such overview diagrams would be useful in two forms to assist code understanding:

- providing a simple syntax highlighted overview, allowing areas of source to be identified based on the appearance of the source in that area,
- providing search functionality, highlighting matches within the file.

Their usefulness would be enhanced by the ability to use the diagram to "index" the source file - so a click on the diagram would jump the viewer to the corresponding source line.

2.2.7 Cause and Effect determination

Although it is possible to determine globally where a particular variable may be written by simple observation, this does not take account of the flow of control. When information about program flow is included in the analysis it becomes possible to apply ordering constraints to a search for accesses. Why is this useful?

- When aiming to understand a program for the purposes of debugging, the ability to determine exactly which statements could have affected the value of a variable at a particular program point is beneficial, as it narrows the search for the cause of a problem.
- When performing comprehension related tasks as part of a process of change management, it is useful to be able to ask "What could this change affect?".

Two similar analyses can be used to aid both of these tasks – Program slicing and Ripple analysis.

Program slicing

De Lucia[Luc01] introduces program slicing concisely as:

> "Program slicing is a viable method to restrict the focus of a task to specific sub-components of a program. Example applications include debugging, testing, program comprehension, restructuring, downsizing and parallelization"

A program slice with respect to a given variable, v, is a set of variables the values of which can influence that of v. This is of use chiefly when debugging; however program slices can still be informative when trying to understand a particular section of code. The technique was originally presented by Weiser in [Wei79] and since then many methods for producing slices have been devised [Tip95, Luc01].

Program slices are useful for the purposes of debugging software - if a value of a variable can be seen to be erroneous, a program slice can narrow the search for the cause of the error. An extension to program slicing, known as dicing, can be used

to narrow the search further still. However, it is not overly useful within the scope of program understanding, so will not be discussed here.

The information from program slicing can be used to produce overview diagrams similar to those used by Kiczales to show crosscutting within a program[KH01]. Some additional information on program slicing is provided by [LR92]

Ripple analysis

Ripple analysis is designed to identify the code that will be affected by changes in a given statement. Ripple analysis is Program Slicing performed in a forwards direction; instead of asking "What could have affected this variable's value?", it asks "What can the value of this variable affect?".

Whereas program slicing is particularly useful when it comes to debugging, ripple analysis is helpful for change management. As such it is more useful in the context of this project, as most software comprehension activity is undertaken with the intention of eventually making changes to the source code.

The first mention of the concept of Ripple analysis the author was able to find was by Horowitz et al in [HRB88], referred to as "Forward Slicing". As Ripple analysis, it is referred to in a series of papers by Lividas et al related to the Ghinsu framework[LR92, LA93, LS94].

2.3 Designing for reuse

It is increasingly common for a software product to be developed using multiple languages. This necessitates support for more than one source language within a comprehension tool project. For the tool developer it is obviously undesirable to have to produce all of the analysis and visualisation code once for each supported language. Though a procedural language and a functional language share little in terms of syntax and semantics, there is sufficient commonality between some languages to suggest that re-use is possible. Hayes discusses this and a number of related issues in his masters thesis[Hay]. The issue of maintaining language-independent operation within analysis procedures is examined and a number of methods are presented:

Text-based analysis discards semantic information and treats the source text as a series of tokens for pattern matching against.

Common representation based analysis involves mapping multiple input languages to a single internal representation. The potential representations include abstract syntax trees, three address code and others commonly found in compilers texts.

Shared interface based approach involves defining an interface to access a program representation through, and defining all of the analyses in terms of that

interface.

Text-based analysis is the simplest method presented, however there are limits on the amount and type of information that can be extracted, as well as being more error prone. This method has been used to provide syntax highlighting and simple cross referencing of source code. It will not be considered further in this document.

There are two prominent issues with the common representation approach: firstly information loss when generating the representation must be minimised; and secondly it is important to ensure that the translation does not change the meaning of the original source.

The focus of [Hay] was on the redesign of an existing program analysis tool to provide support for multiple languages. The approach taken was the shared-interface approach, defining an interface to access the source representation and then re-implementing the existing analysis procedures through it. Adapting the tool to a new language became a matter of identifying an existing representation for the language concerned, and creating a module that provided the analysis interface based on that representation.

Although this approach is conceptually simple, it can be complicated to implement the bridging between dissimilar representations. In addition to this, fixing the interface through which the code can be examined places constraints on how, and possibly what analyses can be written, based on the information provided through the interface.

2.4 Existing program understanding systems

This section briefly details a number of existing software systems that provide code understanding features. It doesn't attempt to be complete, as this would serve little purpose for this project, as well as being time consuming to produce. In addition to the information provided in this section, there is a web site containing links to these and other comprehension support tools at [the].

2.4.1 Commercial products

WindRiver SNiFF+™

SNiFF+ is one of the better known commercial offerings, designed to aid code understanding and generally make the developers life easier. Its features include:

- support for projects whose code is managed by version control systems,
- support for projects in a variety of languages, including C, C++, Fortran, Ada and Java, including projects containing more than one language,

- generation of build scripts for a number of build systems,
- cross referencing of symbols in the code,
- syntax highlighting,
- general project information - file types, symbol summaries,
- object hierarchy display,
- production of overview documentation.

It appears to take the approach of constructing a database of information when a project is first created. This leads to an initial delay whilst it works through the source code, although it appears to be remarkably fast unless there are particularly complicated template based code sections.

More information is available from www.windriver.com/products/sniff_plus.

RedHat Source-Navigator™

Previously known as Cygnus Source-Navigator, before their acquisition by RedHat, Source-Navigator is an open source project that provides many of the features of SNiFF+. Although it appears to have a similar feature set, the interface is produced using TCL and feels heavier than that of SNiFF+:

- cross referencing of source code,
- class hierarchy and inclusion browsers,
- syntax Highlighting,
- pattern based searches for symbols,
- version Control management interface.

More information is available from sources.redhat.com/sourcenav/.

Source Dynamics' Source Insight™

Produced by Source Dynamics, Source Insight is an editor with facilities to present context information on the fly for a number of different languages, including some .NET languages. In addition to presenting this information, it provides a variety of graphical representations, including:

- reference trees,
- inheritance diagrams,

- call graphs,
- "syntax formatting", a variation on syntax highlighting that varies more than font colour and style, and does so based on semantic information about a symbol.

It also provides a wide range of other features that aren't particularly related to code comprehension, such as context sensitive search and replace which allow some simple refactoring to be performed.

More information is available from www.sourcedyn.org.

2.4.2 Research products

There are a wide range of research products for the investigation of source code visualisation, this section details mentions a few.

Ghinsu

Ghinsu is a program understanding framework described in[LA93, LS94], designed to work with ANSI-C source code. It uses a parse-tree like representation it terms the *System Dependence Graph*, on which all of its analysis procedures operate. It provides the ability to perform a swathe of program analysis techniques as well as browsing functionality.

Unlike most of the commercial offerings examined, Ghinsu appears to be file rather than project orientated in operation and has an outdated interface style.

SeeSoft

SeeSoft[ESJ92] is a tool for visualising software statistics from a variety of sources, including:

- version control systems,
- static analysis (call sites),
- dynamic analysis (profiling).

The statistics are displayed using a colour coded reduced representation, intended to allow patterns to be spotted easily. The visual representation allows the side-by-side comparison of a number of source files.

2.5 Summary

- The ability to quickly build up an understanding of a complex software system has not been made redundant by better software development processes or semi-automated design improvement tools.
- It can be a difficult and slow process to build up an overview of a complex system without tool support.
- A variety of techniques can be applied to ease understanding, those that:
  - highlight particular aspects by abstracting away from the program text,
  - target areas of code for improvement by providing a measure of their size, quality or complexity,
  - gather scattered information and place it at the developer's fingertips.
- As software systems are increasingly made up of sources in multiple languages, it is desirable to be able to support multiple source languages in a tool for easing understanding.
- Supporting multiple languages can be achieved in a number of ways, each with inherent trade offs in flexibility and potential usage.

Discussion

> A true history of human events would show that a far larger proportion of our acts are the results of sudden impulses and accident, than of the reason of which we so much boast
>
> Albert Cooper

At the start of this document, a stated project goal was to produce a framework for software understanding. This section highlights key problems for the development of such a framework, and discusses potential solutions. The contents of this section provide the motivation for the majority of the specification.

The subjects discussed in this section are:

Projects: the management of information extracted from a software system.

Support for collaboration: allowing multiple users to simultaneously access extracted source code information requires access to be made possible over a network.

Information presentation: providing easy access to a variety of different forms of data (graphs, diagrams, and listings) to simplify the task of finding that which is useful.

Program representations: choosing a suitable method for accessing program representations, usable with multiple source languages.

These problems are looked at in a general sense, rather than specifying exactly what will be done in the implementation, which will be presented in the design and implementation sections.

3.1 Projects

This section introduces the concept of a project in the context of the framework, and discusses the tasks involved in the maintenance of framework projects.

In the context of the framework, a project is the body of information extracted from a software system. A project will include such information as:

- a record of the sources that make up the project,
- the data extracted from the source to provide the understanding functionality.

There are a number of properties that a framework project should have:

- It should be possible to determine whether there is information within the project that was obtained from a given file (ie. is the file part of the project).
- For each file that is part of the project it should be possible to determine whether the information about that file is up to date, or whether the source contained in that file has been changed since the information was extracted.
- It should be possible to reconstruct the source code of any file in the project from information within the project (ie. without referring back to the source file, which may no longer exist).

It must be possible to create a project to store the results of analysis of a software systems' source. Project creation would entail determining which files are part of the project, processing those files and storing the results in such a way that they can be retrieved at a later date. The project creation process should not be restricted to working with source trees stored on a local disk, as this allows later expansion to work with software sources stored in version control systems. Working with source stored in a version control system does not have any immediate benefits, but it provides the potential to perform analysis of the evolution of the software (such as is performed by Seesoft[Eic94]).

The process of extracting information from source code may be quite time consuming. As a consequence it is desirable that the amount of information extraction that occurs is minimised. The source code of the software system being worked on may be changing, particularly in a commercial environment, resulting in the stored information becoming out-of-synchronisation with the source. Clearly it is undesirable to have to re-process every file in the project when a change is made: the ability to update information only for those sources that have changed or been affected by change is required.

Time stamp information on files and revision tags in version control systems are sufficient to allow automatic determination of whether a source has been changed. Automatically working out which files need to be updated reduces the burden on the developer, who would otherwise have to ensure that the information for each source is up to date.

3.2 Support for collaboration

Annotations were mentioned in the background section as a way of making source code more understandable for others without modification. The ability to add annotations through virtual post-it notes to the source not only makes it simple for one developer to refresh their memory, it allows others to leverage the work of that developer. When the program understanding is undertaken as a precursor to adaptive maintenance[1], many developers may be working on the project. For them to collaborate it is necessary that each can view the others annotations.

There are three ways in which this can be done:

- rely on underlying operating system support for file sharing, and allow all of the developers to access a shared database of software information, including the annotations,
- provide a client-server architecture, with a central server storing the project information and clients accessing it through a well defined interface, possibly via HTTP or SOAP,
- provide a peer-to-peer system, allowing each developer to connect to each other.

Reliance on operating system support is the simplest method to implement, but complicates the procedure of setting up the system to allow multi-user access to the software information. Of the remaining two options, a client-server approach is the simpler. The general reason for choosing a peer-to-peer architecture over a client-server one is that the former has a greater tolerance to load and lacks a single point of failure. In the case of a system such as this, it is unlikely that there will be enough developers working simultaneously to overload a reasonably written server. A client-server system also avoids any problems with synchronisation between peers, so appears to be the best choice for providing information sharing support.

Further motivation for the use of a central server comes from the cost of extracting and storing information from a large software project. The data extracted from a large body of source can require many times the amount of storage required to hold the source itself, making it undesirable to have to create multiple copies of the information. The time taken to perform the information extraction is also an issue, particularly when dealing with a large source base.

3.3 An environment for information presentation

The mark of success for a user interface is when the user doesn't notice that it's there. This is the motivation behind the use of common themes, interface paradigms

[1] Adaptive maintenance tasks involve updating the system to keep pace with a changing environment

and the principle of least surprise; past experience allows a user to work on the task at hand without wondering how to achieve that task through the interface.

The intention of the interface will be to tie together all of the different visualisations into a coherent environment, that makes use of semantic information about the software. This requires that:

- The interface is able to differentiate between different source code elements and adjust its behaviour accordingly.
- Appropriate actions for a source element can be obtained by clicking on that element, allowing visualisations and common information look-up tasks to be accessed in a logical manner.

To provide a sense of familiarity the most appropriate user interface concept in this situation is that of a web browser. The reason for this is that the user's interaction with the interface involves the viewing of a stylised document, navigating to other information associated with document elements.

It is possible that the interface could be a web browser, with web technologies being used to view styled and interactive documents. This approach has a number of advantages:

- flexible text styling – the colour, size and style of the text can all be varied to highlight different elements,
- built-in support for cross referencing between documents and document elements,
- support for accessing remote entities via HTTP provided, simplifying the provision of collaboration facilities.

The main problem with such an approach is that a number of incompatibilities exist between different web browsers and the web standards, and as such it is difficult to produce an interface that will work in the majority of scenarios. Ignoring the cross-browser compatibility issues, a secondary problem is the difficulty in producing a user interface using HTML.

Both of these problems can be worked around by embedding an existing browser in an interface produced by some other means. Given the advantages listed above, use of an embedded browser appears to be the best approach.

3.4 Program representations

Two feasible alternatives were mentioned when discussing the reuse of analysis procedures: The use of common representations, and interfaces to existing representations. Table 3.1 presents some perceived advantages and disadvantages of each approach.

Common representation approach	Using existing representation
Allows the most design flexibility	Restricted to an extent by the existing representation
Information loss can more easily be managed	Information availability is governed by existing representation
Potentially slower to develop, and more CPU and memory intensive.	May be faster to develop, runtime properties unknown.
More easily changed at a later date to accommodate further language features	Restricted by other implementations of the interface.
Less complicated development	May be difficult interface with provider of existing representation.
	Use of existing representation reduces the potential number of errors.

Table 3.1: Comparison of program representation approaches

Although the interfacing approach allows existing compiler-like software to be leveraged, it is the authors opinion that this will not reduce the cost of providing analysis for a new language significantly in the long term. This is because the development task changes for each new language representation, and there are additional inherent difficulties due to the information available in the representation.

Taking the common representation approach requires a parser for the language and a translation scheme that maps the parse tree on to the associated representation. Although writing a parser can be time consuming for complicated languages, this is not an issue as parsers already exist and can be licensed; for example, the EDG C++ and Fortran frontends[EDG].

A suitable representation is not easy to decide upon. Hayes[Hay] mentions Abstract Syntax Trees and three address code. This issue is discussed in the context of compilers by Muchnick[Muc97]. Muchnick discusses multiple level representations, with the lower levels being used for some optimisation and register allocation. The lower level representations are generated from higher level ones, as a consequence it is desirable to have a higher level representation for program communication, as lower level representations can always be constructed if needed for particular analyses.

The association of a family of similar languages with a single representation suggests that it would be possible to create a tree-like representation based on the hierarchical structure of those languages. A representation for procedural languages would include structures for scope, functions, loops and statements, among other things (see figure 3.1). The benefits of keeping the representation as close to the source languages are that it simplifies the translation to the representation, and the mapping of information back to the source language elements.

As the translation to a form such as this discards information on the layout of the source code, it may be necessary to store additional information or another representation that keeps this information. Apart from the difficulty of providing a layout system (let alone a language independent one), it is important to maintain

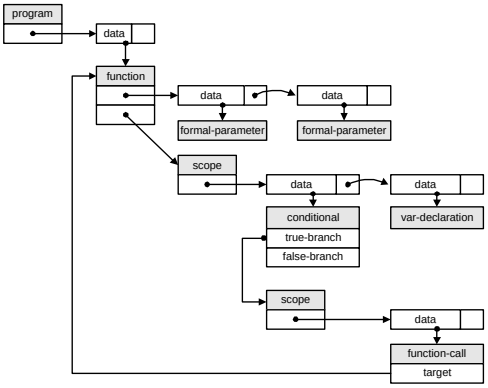

Figure 3.1: An example tree-like internal representation

the original formatting so that the source is presented in a style that the user is used to.

3.5 Summary

- Networked multi-user access allows a team of developers to simultaneously work on the same source base.
- A client-server architecture with a central server is the most appropriate approach to provide multi-user access.
- A user interface is required that makes use of semantic information to provide relevant operations based on the context.
- It is possible that a web based client could be provided, if an HTTP server interface to the framework was written. This allows layout and styling features to be provided by an standard component.
- Use of a web based interface brings problems in the form of compatibility between browsers and difficulty in the construction of the user interface.
- The internal representation used for programs affects how easily the framework can be adapted to a new language.

Specification

> I always wanted to be somebody, but I should have been more specific.
>
> Jane Wagner (Writer/Director)

This section contains the specification for the framework implementation. The specification restates the points raised in the discussion and specifies the core feature set to be implemented.

4.0.1 General requirements

A_0 The implementation should provide an interface for information presentation and request.

A_1 The implementation should perform information acquisition and publishing.

A_2 The information must be accessible from machines other than that on which it is resides.

A_3 The information must be accessible by multiple users simultaneously.

The intention is that the implementation of the framework be done in two sections - a frontend and a backend. The purpose of the frontend is to provide a usable interface to the information made available via the backend. The backend performs the information acquisition, and makes the information available to the clients. In this document the frontend may be additionally referred to as the interface or client, and the backend as the server.

The frontend is intended to be used in separation from the backend code, with the

possibility of the backend being in a separate location with communication via some network protocol. It is suggested that HTTP is used as the network protocol as this allows the use of existing components to speed the development of the frontend.

4.0.2 Interface functionality and operation

IF_0 The interface must allow the user to connect to a server in order to access a project.

IF_1 The interface must begin with a display of the project hierarchy, allowing individual files to be selected for browsing.

IF_2 The implementation must be able to produce syntax highlighted source code.

IF_3 It must be possible to jump to the declaration of a symbol from a usage in the source.

IF_4 It must be possible to generate a list of uses of a symbol.

IF_5 It must be possible to add annotations to a line of a file.

IF_6 The system must be able to produce source overview diagrams.

IF_7 The system must be able to produce some basic software metrics, possibly in diagrammatic form.

IF_8 The system must be able to display a forward and reverse static call graph on request for a function within the source code.

IF_9 The interface should be designed such that it is usable on a monitor of resolution 1024x768 pixels or above.

IF_{10} The interface must have a modern look and feel.

IF_{11} Keyboard shortcuts must be provided for common operations.

IF_{12} Considered use of focus placement must be used to minimise the requirement for mouse usage.

Although the requirements are placed on the interface, it is not necessary for the code that fulfils a particular requirement to be present in the client code. The client may be implemented in such a manner that the computation is performed by the server. This is to allow a thin-client to be provided in the form of a web-based interface.

In the case of call graphs, it is not required that data flow analysis be performed.

This means that the information provided by the call graph may be incomplete in the presence of functions called via functor types.

4.0.3 Backend functionality and operation

The backend is responsible for the construction of source code comprehension projects. The project refers to the information extracted from the source code, processed and stored for later use.

BF_0 The backend must have a command line interface that allows:

- project creation,
- complete project deletion,
- the publishing of a project on a specified port,
- project updates to reflect source code changes.

BF_1 The backend must store sufficient information that a change in a single source file does not require the entire source base to be reprocessed.

BF_2 On update of a project, the backend must determine which files require reprocessing and perform that activity.

BF_3 The backend should not require any third party software systems to be set up that it cannot automatically do itself.

Note that no mention is made of storage requirements; it is acceptable for the extracted information to require more stable storage space than the original source code to allow for reasonable query speeds. It is expected that an embedded database system be used to provide the storage, however this is left unspecified.

4.0.4 Miscellaneous requirements

M_0 Annotations placed by one client in a file should be accessible to every client on the next occasion that they browse to that file.

M_1 The implementation must be documented and implemented in a manner that facilitates further work.

M_2 Where the user is required to deal with large quantities of information, some form of pattern based searching must be provided.

4.0.5 Language support

L_0 The system must be designed with the intention of supporting multiple languages.

L_1 The implementation must be capable of supporting at least one language as an example.

L_2 The report must detail the steps required to add support for similar languages.

L_3 Similar languages should use the same analysis code to maximise re-use wherever possible.

It is suggested that a simple language such as Kenya, BASIC or Pascal be chosen as an initial language for implementation, as the small size of the language simplifies the task of implementing something that is purely intended to demonstrate the framework.

4.0.6 Extensions

X_0 The system should provide the ability to calculate program slices.

X_1 The system should provide the ability to perform ripple analysis.

X_2 On creation of a project, the backend must perform sufficient information extraction and manipulation to reduce the cost of simple queries below five seconds, excluding network based delays.

Design

> But Mousie, thou are no thy-lane,
> In proving foresight may be vain:
> The best laid schemes o' Mice an' Men
> Gang aft agley,
> An' lea'e us nought but grief an' pain,
> For promis'd joy!
>
> *To A Mouse*
> Robert Burns, 1785

This section details the transition from specification to outline system design. It begins with a restatement of the aims of the project with respect to the implementation, followed by a brief presentation of the overall design decided upon[1], finally covering each of the main areas of the system in more detail.

The intended software result of this project is a usable framework for easing code comprehension, with the additional properties that:

- It is designed to facilitate the addition of new languages in such a way that code comprehension functionality does not have to be provided anew for each language.
- It allows those working on building up an understanding to share information between themselves.
- It provides a client that demonstrates the functionality provided by the framework.

In order to satisfy A_2 and A_3, a client-server architecture was suggested as the best approach (§3.2). Figure 5.1 is a block diagram showing the main system

[1] The excerpt from Burns was thought to be a comment applicable to design in general, and should not be considered to be an indication that the design set out in this chapter was not realised in the implementation, or turned out to be flawed.

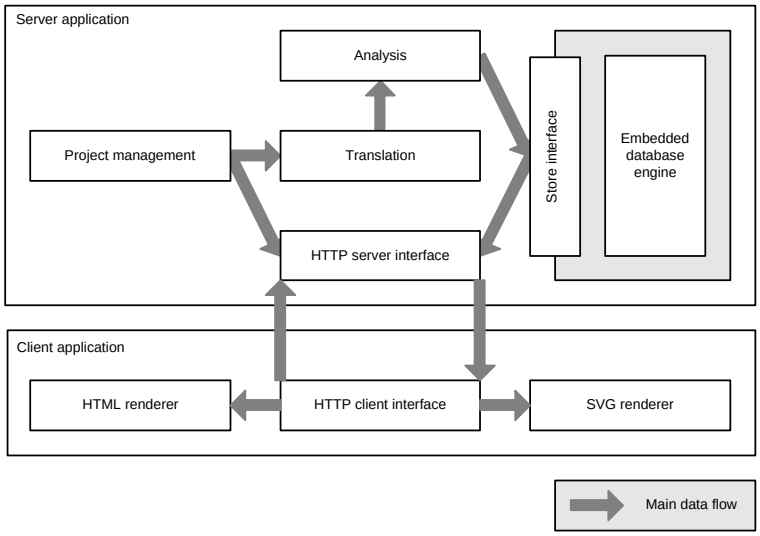

Figure 5.1: An overview of the framework design

components and data flows at a high level. The discussion of the design will be split into two sections, treating the server and the client separately.

This project aims to support the reuse of analysis code. Based on the argument in section 3.4, the approach taken to achieve this within the framework will be the use of shared program representations. This decision motivates most of the framework design:

- In order to analyse a source file, a representation of its contents in a form supported by the framework is required. As a result there must be a mechanism for translating the program source language into that form. This is the translation stage shown in figures 5.1 and 5.2.
- For each program representation, there will be a number of analysis procedures that can be performed with that representation as input. An analysis module groups together all of the analysis procedures that can be performed to a particular form of program representation. The analysis module provides an interface that can be used to perform each of the individual analyses on a given program.

A large amount of information can be extracted from program sources. The storage requirements can increase further if data constructed from extracted information is stored to reduce the runtime cost of requests for that data. There are three issues that will govern the choice of storage mechanism.

1. Speed – does the use of a particular storage mechanism have overheads in the form of network based communication, query construction or parsing, or does it simply perform poorly?

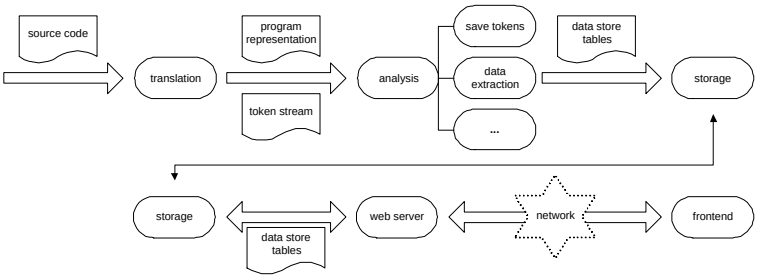

Figure 5.2: Data flow from file to frontend

2. Interface – does the interface to the storage mechanism allow the required operations to be expressed concisely?
3. Self containment – if the use of a particular mechanism requires separate software to be set up manually, it provides more potential for error. BF_3 states that any third party software required should be set up by the backend application for this reason.

For the purposes of satisfying points one and three, an embedded database engine would suffice. Point two is harder to satisfy with an embedded database engine, as the provided interfaces vary greatly from system to system. However, the interface can be wrapped to provide one that is easier to deal with.

As can be seen in figure 5.1, an embedded database engine is used in the framework design, for reasons commented on further in section 5.1.5

5.1 Framework design

5.1.1 General design rationale

This section comments on decisions made when designing the application that are not part of a particular sub-system.

A certain amount of information is shared and needs to be accessed in scattered locations throughout the backend application. In general there are three ways in which this can be achieved: The singleton pattern, the context-object idiom, and the use of global variables. There is some disagreement over whether singletons are "better" than context objects from a design perspective[autb, auta]. This project uses a context object, for three reasons:

- There is no overhead in this context as the object can be passed with the project, and therefore does not leave the implementation subject to the usual problem with context objects - that each method signature has to include the context object.

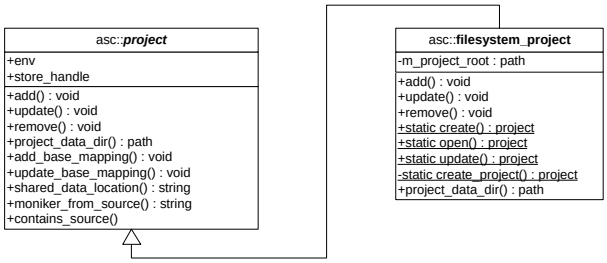

Figure 5.3: Project management classes

- In this context it places fewer restrictions on the evolution of the design; If it was later decided to extend the system to serve multiple projects at once, it would be possible to have multiple context objects. It would not easily be possible to have multiple singletons.
- The code for accessing shared objects is briefer and less costly at runtime.

5.1.2 Project management

The project class hierarchy (Figure 5.3) is the part of the design that deals with project management. There are a number of features that are to be provided by the project management system, which will be discussed in this section:

- insulation of the remainder of the framework from the storage type of the source code (local filesystem or version control system),
- creation of new projects and access to existing projects,
- source file addition, update and removal from the project,
- update of project information to reflect changes in the source code.

Storage type isolation

The framework is isolated from the knowledge of how the source code is stored by ensuring that all requests to access a file pass through the project. In reality, the only occasion on which this occurs is during the translation process, required for addition or update of a source file. On this occasion, the project will generate a local filesystem path that can be used to access the source code.

The decision to allow the translation process to work on a file on disk does not have a detrimental effect on the isolation – the majority of version control systems require a file to be "checked out" to a local file in order to view its contents. The alternative approach would have been to provide access to an in-memory copy of the file, which would restrict how the translation process could be written.

On all other occasions, a file is referred to by its moniker (see section 5.1.5). A moniker is an object with a textual representation (for storage purposes) that uniquely identifies a particular version of a source file. Only the project subclass that created the moniker is able to translate it to a file name.

The implementation of the framework will contain a single project subclass, that handles the situation where the source files are stored on a local filesystem. This is the *filesystem_project* class that can be seen in figure 5.3.

Creating a new project

All subclasses of the project class have a *create* method that creates a new project. Project information is always stored on the local file system, in a user specified location. The project creation process contains five steps:

1. Create the directory to contain the project information.
2. Create an empty data store.
3. Create an instance of the project subclass.
4. Initialise the data store contents.
5. Add all source files that should be part of the project.

The first four steps are straightforward, with the initialisation of the data store contents being performed by the project class rather than the subclass[2].

The source files that should be part of the project are determined through the application of a set of rules specified by the user. The user will supply a sequence of rules of the form:

rule	::=	*action ':' '[' path ']'*
action	::=	*("include"|"exclude") "-subtreee"?*
path	::=	*(alnum|punct)**

The rules will then be evaluated (by the project subclass) in the order that they are written, to generate a set of source files that should be part of the project. Each of these files will then be added to the project using the *add* method described below.

Opening an existing project

To open an existing project will simply be a matter of creating a data store handle (see section 5.1.5) that references the previously created store.

If the project does not exist, the attempt to open the project will fail rather than creating a new project in the location where the project should have been.

[2]For details of how the data store is initialised, see section 5.1.5

Adding files to a project

For a file to be added to a project, there are two pre-requisites:

- There must be a translator module available through the translator manager, that is capable of handling the source language.
- There must be an analysis module that is capable of analysing the output of the translator module for the file, available through the analysis manager.

The *add* method will generate the moniker that uniquely represents the file being processed, and add an entry to the store to record the presence of the file. A translator module will be looked up, and a translation produced for the file. This translation will be used to look up an analysis module that will then be used to analyse the translation, populating the data store with information relating to the source code.

Updating stored information

There may be many different locations throughout the framework that extract information from the source code. As a consequence there is no way of determining what information pertains to a certain file. From a design perspective, this is desirable, as it avoids the need to store extra information or to have knowledge of what information is created. However, it is still necessary to be able to update this information. This will be achieved through the use of an event subscription and notification system.

A known source file can be in three states when the project is updated – unchanged, changed or removed. When updating stored information, each of these will be handled differently.

A removed file may have a number of dependent source files within the project. These files will always need to be updated, regardless of whether they have changed. The update process must find these dependencies within the project, and use them to determine which files must be reprocessed. A file that is unchanged will not be reprocessed unless it is dependent on a file that has changed or been removed.

Event notification will be used for a number of purposes:

- to inform interested parties that a file has been removed or changed,
- to gather a set of dependencies between sources

There are a variety of tasks that would need to be performed when code is changed or removed: the flushing of caches, the removal of generated files and the removal of information stored in the data store.

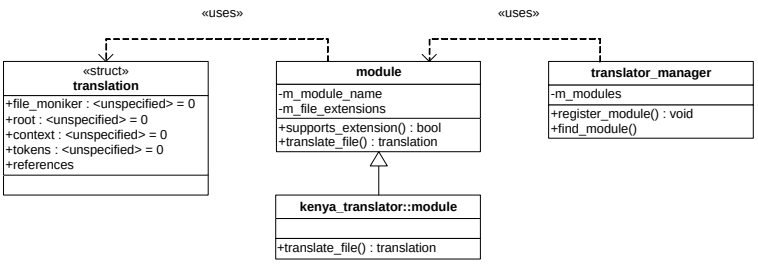

Figure 5.4: Classes involved in the translation of source code

As the analysis that occurs depends on the source language, different information will be generated from file to file. As a result it is necessary to distinguish removal of information generated by one analysis from that generated by another, and similarly for updates and dependency information. To achieve this, the event is prefixed with the name of the analysis module that was used to generate the information.

5.1.3 Language translation

It can be seen in figure 5.1 that the information stored about a project goes through two stages - translation and analysis. The purpose of the translation stage is to generate one of the internal program representations from the contents of a source file. A number of translator modules can exist in the system, each accepts input in a single language and outputs a translation. A translation contains two representations of the source, described in detail in §5.1.4:

- A tree-like representation whose root node type is used to identify the appropriate analysis module.
- A token representation that can be used to reconstruct the input source code. This also contains tags that can be used to associate some tokens with stored data, such as variable use or function call information.

The translator modules provide two pieces of information that are used to associate a translator module with a source file:

- a predicate over file extensions that determines support for a file based on its name,
- a unique name that is stored with extracted information to allow the origin of the information to be determined.

The translator modules will be managed by the translator manager, which allows modules to be looked up using a source file name or a module name. A single translator manager exists and is held in the project environment, which is a context object accessible through the project (see §5.1.1).

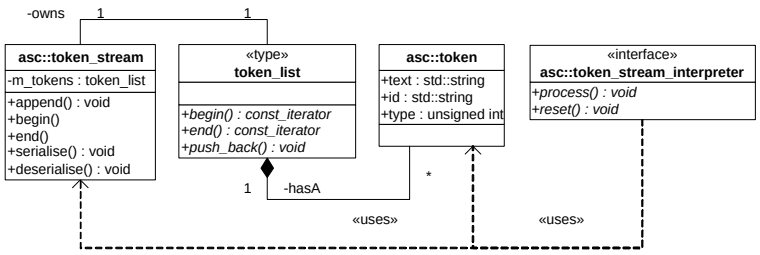

Figure 5.5: The token stream representation

No restrictions are placed on how the translator modules generate the internal representations. The initial design includes a single translator module that takes the teaching language Kenya as input, and outputs in the form of the procedural representation (§5.1.4). This translator module uses a parser written using Spirit, which generates a parse tree presentation that is then walked to generate the translation.

5.1.4 Program representations

The internal representations are intended to provide a translation target for multiple source languages. This section will cover the design of a high-level program representation for simple procedural languages and a low level token representation.

Token stream representation

The source analysis stage of a compiler is sometimes divided into two stages - lexical analysis and parsing. The lexical analysis stage would transform the character stream into a sequence of tokens - groups of characters with associated types like *integer* or *identifier*. In a compiler this stage tends to eliminate inconsequential tokens from the output stream, however in this application there are a number of tasks that require the original source text. If no tokens are discarded then it is possible to reconstruct the original source text from the sequence of tokens.

One of the goals of the project was to isolate the knowledge of how the source code is stored - whether on disk or in some form of version control system. This is not possible if parts of the program refer back to the source text. To avoid this, access to the original representation of the source is performed through a project subclass during the translation stage, with any later requirement for the source text satisfied by the token stream.

The token stream is generated as part of the translation output by a translator module. The token stream for each file is saved in the project data directory and can be loaded when the user requests a representation of the code that needs the original source.

Figure 5.5 shows the design of the token stream. A token contains three items of

information; the text of the token, a type identifier and a textual identifier. The original source can be reproduced by outputting the text of each token in order. The numeric and textual identifiers provide semantic information for the text.

The integer identifier associates the token with a node type of the tree-like representation or one of a set of common types such as *new-line* or *whitespace*. The common types allow language-independent output of the text in an environment where line breaks are represented in a different way (such as HTML).

The textual identifier allows the token to be associated with a specific program element, such as a variable or function.

The diagram also contains an interface named token_stream_interpreter. A token stream interpreter is an object that performs some task based on the tokens between two token stream iterators. There are a number of classes that implement this interface to generate alternative representations of the source code; the syntax highlighting and overview diagram generation are done by token stream iterators.

Procedural internal representation

All internal representations will add to the class hierarchy rooted in *ir::node*, which allows the representation to be passed through the program without management-related components needing to know the exact type of the representation.

The design of the procedural language representation is intended to have a form similar to a partially flattened abstract syntax tree. It provides support for:

- name and unnamed block scopes,
- function, variable and composite object declarations,
- variable usage and function invocation,
- initialisers,
- looping constructs,
- loop and function exit points,
- conditional execution of statements.

Figure 5.7 shows the types that make up the procedural program representation. At run time the object graph representing a program will always be dominated by a *procedural::file_scope* element, the dynamic type of which is used to identify the representation.

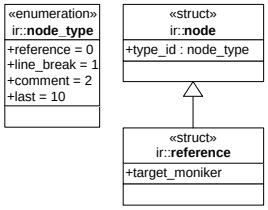

Figure 5.6: The core internal representation types

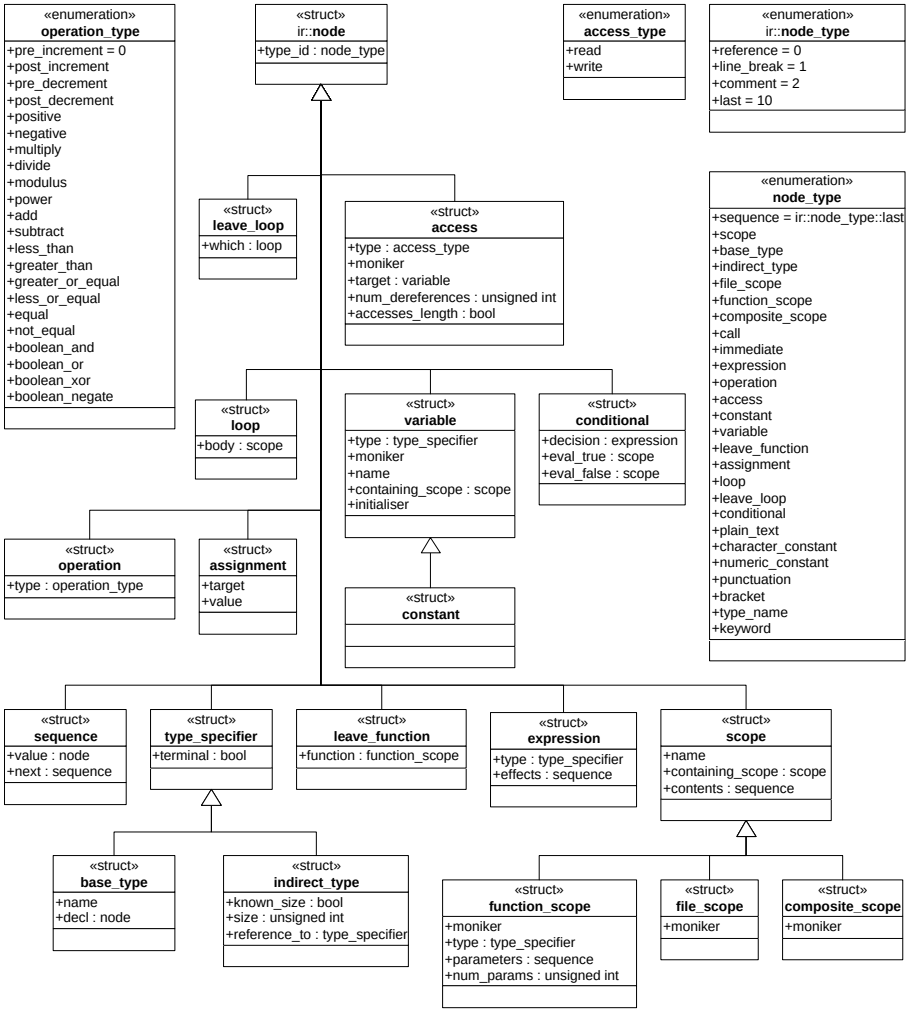

Figure 5.7: The procedural representation

5.1.5 Information storage

This section attempts to cover three points:

- What information about the source needs to be stored to achieve the goals set out in the specification?
- How can the data be stored?
- How can a suitable interface for data access be provided?

Identifying source regions

A number of operations need the ability to map between a region of source code and a higher level entity such as a function or variable use. A source file can be treated as a linear sequence of characters, or as a sequence of character rows separated by line breaks. The former treatment allows a point in a file to be represented very compactly by its integer offset from the start of the file. The latter treatment would represent a location as a pair (line, column), requiring more storage space. Although the compact representation can offer space savings, it has a number of problems related to the use of information:

- It is less suitable for a human reader than the (line, column) representation commonly used in editors.
- Transformation to (line, column) form is expensive, requiring either the offset of every line break be known, or the linear traversal of the original source file (which may not be available).

The (line, column) form allows a cheap implementation of commonly performed operations, such as *jump to definition* or *show variable uses*. As a consequence it is better (from a computational cost perspective) to use a (line, column) representation for points in the source, and a (line-min, line-max, col-min, col-max) representation for regions.

In addition to identifying a range of characters, it is necessary to identify the file. A file can be uniquely identified by the project in which it resides, its path and some form of version information.

In some situations it is necessary to identify a file, in others a region of that file. These requirements lead to the moniker representations used in the project. There are two types of moniker – partial and complete. A file is represented by a partial moniker, consisting of:

- a project identifier,
- a project-relative path to the file,
- a storage specific version stamp.

A version stamp is a class with a total ordering over instances, that can be serialised to give a textual representation which can be deserialised to produce an instance of the class. This system allows the use of version information specific to the storage used for the source code – a CVS file can use a revision tag, a file on the filesystem a modification time stamp. This is important as it allows the framework to be extended to support projects where the source is stored in a version control system rather than on the filesystem.

A complete moniker extends a partial moniker, adding region information to identify a range of characters within the file.

Both partial and complete monikers have a textual and a runtime representation. The purpose of the textual form is to allow the moniker to be stored in the data store, and to be passed to the client to allow it to refer back to elements of the source.

Providing data storage facilities

As was touched upon earlier, the data extracted from a large body of source can require many times the amount of storage required to hold the source code itself. When aiming to provide an interactive system, there is an interplay between the amount of information extracted in advance and the efficiency of extracting the information on demand, a problem discussed by Atkinson and Griswold[AG96].

It was decided that the disk resources used was not a concern for the implementation of this project, due in part to the greater modern disk sizes and also the opinion that providing access to information at interactive speeds is of more importance to the end user.

The type of information that needs to be stored in this situation fits the relational model of storage well. In addition to small relations it will be necessary to store large binary objects (sometimes referred to as BLOBs). A number of options for data storage were examined:

A custom file format was discarded immediately due to the existence of perfectly good database systems suited to this kind of data.

XML based storage was possible but would not be the most efficient system for handling queries.

PostgreSQL and MySQL were examined and although proven and widely used, have the overhead associated with communication via a network connection.

SleepyCat DB is a widely used high performance embedded database, however it stores its data key/value pairs which would increase the amount of work required to use it in this context.

The system chosen for the base of the data storage system was an embedded relation database called MetaKit[Wip]. The reasons for this decision were as follows:

- MetaKit is free to use, with very reasonable license terms.
- MetaKit has reasonable performance with large data sets.
- It is embedded, there is no need for a socket connection.
- It has some built in support required for queries (joins, unions, differences).
- It has a small and relatively self explanatory C++ API.

Providing data access facilities

MetaKit is lacking in some features that may be desirable at some point in the project. In order to ensure that these features can be added easily at a later date it will be necessary to access MetaKit through an interface that keeps it separate from the rest of the program. This decoupling will also allow the program and the database to vary independently. This interface will be referred to in the rest of this document as the data store interface or the store interface.

Both MetaKit and SleepCat DB lack a query language such as SQL. Although the use of a query language adds inefficiency in the form of additional parsing requirements, it allows the code using the database to be expressed more concisely. To avoid the overhead of providing a query language for the data store, higher level features, such as selection and set theoretic operations, will be provided as part of the store interface.

C++ supports a style of programming that involves deferring the evaluation of expressions through the creation of composite structures or object hierarchies that store the expression structure. The construction of a composite structure through the use of templates is a compile time technique due to Veldhuizen[Vel95]. In this project it is desirable to allow query structure to be determined at run time, so use of the static expression template technique is not possible. However, the basis of expression template techniques – operator overloading, can still be used. Instead of "free" compile time construction, an object hierarchy representing the expression can be created with a small runtime cost. A major benefit of this technique is that it allows queries in the source code to be represented concisely and with clear intention.

5.2 Frontend design

5.2.1 Presentation

The use of a web based interface was mentioned in §3.3 as a simple way of providing a styled interlinked document structure. Although it is simple enough to generate and serve web pages, it is complicated to provide a complete interface just using web pages. This suggests that a web browser embedded in a suitable interface

would be a better option; allowing the use of the browser as a layout engine and the interface toolkit to provide shortcuts, menus and dialogs.

Embedding a web browser inside a user interface can be done in a number of ways:

- By using the HTML rendering component provided by Java/Swing. The issues with this approach are that the HTML support is outdated, the rendered results aesthetically unappealing and that it would require the use of Java to write the interface.

- Using Mozilla and XUL to implement the interface is a further option. The main concerns with this approach are the excessive size of the libraries involved and the use of JavaScript for interface behaviour.

- Using COM to embed Internet Explorer into an interface. This technique is known to work, as Internet Explorer was designed as a reusable component, and can be seen in many Microsoft products. The issue with this approach is that the client is limited to the Windows platform.

As the purpose of the client is merely to demonstrate the features provided by the backend, only being able to run on the Windows platform is not considered a major issue. Consequently the web interface will be provided by an embedded Internet Explorer component.

Internet Explorer can be embedded in an application through the use of the WebBrowser COM component. COM allows the component to be used from multiple source languages, including C++, Visual Basic and C#. Of these languages both C# and Visual Basic are well suited to rapid user interface development. The authors greater familiarity with C# lead to its selection for the development of the client application.

The use of an ActiveX control[3] requires an adapter called a Runtime Callable Wrapper (commonly referred to as an RCW). The RCW provides marshalling of data at the language boundary – converting C# types into the C data types used to communicate with COM components. Tools are supplied with the .NET framework that can automatically generate these wrappers.

The WebBrowser component is immediately usable in a C# Windows Forms[4] application once an RCW has been created. However, the RCW does not allow complete customisation of the web browsers behaviour – for example, it is not possible to replace the context menu of Internet Explorer with your own. To implement this and other required customisations requires the implementation of a number of COM interfaces in the client application. These customisations are not explained in this document. A clear explanation is available in[Fai02].

[3] a COM interface component, such as the Internet Explorer *WebBrowser* control.
[4] The .NET framework user interface library for windows applications

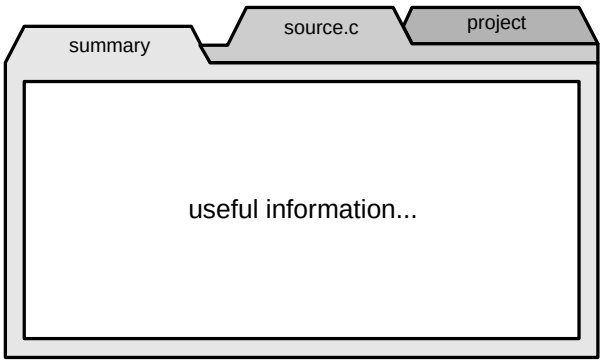

Figure 5.8: The tabbed window concept for information organisation

5.2.2 Organising information

When building up an understanding of a large software system, the developer tends to want to refer to multiple sources of information to accomplish the task. The standard way of allowing this to occur[5] is to provide an MDI[6] layout.

It is the opinion of the author that this approach can be improved upon through the use of a technique often seen in modern web browsers – tabbed windows.

A tabbed window presents information in a manner similar to index sheets used in a conventional filing cabinet – each piece of information has a name that is written on a virtual sheet of card, which can then be clicked upon to access the associated information (see figure 5.8). Such an interface has a number of advantages over an MDI-style interface:

- The interface is uncluttered – only the information required for the task at hand is visible.
- The temptation (present with MDI) to spend copious amounts of time organising how the interface is arranged, and altering the arrangement with each new bit of information, is removed.
- Each piece of information is filed under its name, so is easy to find when required but not in the way when not required.

Although the use of tabs to organise information has all the advantages stated above, there are still a number of disadvantages. The most obvious disadvantage is that it is not possible to compare two files side by side.

Not being able to view multiple sources of information at once may not always be a disadvantage. The experiences of the author using Visual Studio .NET, a

[5]The term standard is used in this situation to refer to the way the problem is solved in the commercial offerings examined - there is no written standard.

[6]MDI : Multiple Document Interface. This divides up screen space into windows that can be resized.

development environment that supports tabbed browsing, showed that the use of tabs had the effect of concentrating the authors mind on the task at hand. Each tab was switched to with a particular task in mind, preventing the "wandering-eyes" process of looking for information, which often resulted in a loss of concentration.

The frontend will be implemented using tabs to present each piece of information for the reasons outlined above.

Implementation

> The physician can bury his mistakes, but the architect can only advise his client to plant vines - so they should go as far as possible from home to build their first buildings
>
> *New York Times, October 4, 1953*
> Frank Lloyd Wright

This section details how various parts of the framework and client were implemented and describes any issues that occurred during their implementation. The implementation is presented as a series of framework features, with the implementation of each discussed. Where appropriate, the discussion of a feature will include cross references to other features related to its implementation.

6.1 Visualisations

This section describes how the visualisations of the source code are created in the context of the framework design.

The visualisations fall into two classes – those that are independent of program type and those that rely on information specific to the internal representation used. The former category are written in a generic manner, operating over the token stream, allowing them to be used with any source language supported by the framework.

Those that operate on a token stream tend to be customisable through the use of a *token_mapper* passed as a parameter to the visualisation procedure. In general, a token mapper is a function mapping tokens to strings. In practice the string tends to be a style in the form of a CSS class[1]. The visualisation sub-system contains a number of subclasses of *token_mapper* that can be used to customize how the mapping occurs, affecting how the result of the visualisation appears.

[1] Cascading Style Sheet – A W3C recommendation for the styling of structured documents

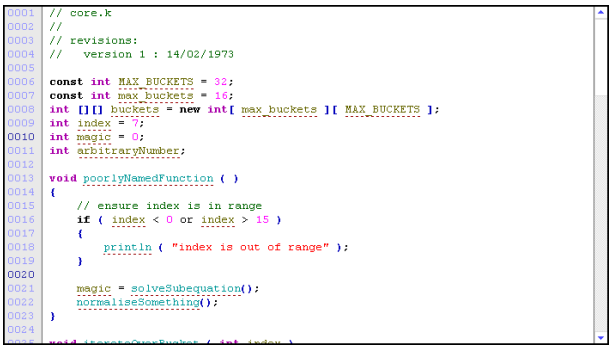

Figure 6.1: Syntax highlighting performed by the framework

token_type_mapper is the most frequently used subclass. It is used by the syntax highlighting procedure to style the tokens based on their type – variable use, language keyword, white space and so on.

regex_text_mapper allows the results of a pattern based search to be highlighted. Those tokens that match against the regular expression specified are styled differently to those that do not.

typed_regex_mapper allows a regular expression based search to be applied to all tokens of a certain type. This is used to support queries such as "*highlight all function uses matching* ^*get.*⋆", which would match methods whose name began with get.

id_set_mapper is the most general subclass, allowing the results of a more complex query to be highlighted. The style of a token depends on whether or not its identifier is a member of a particular set of identifiers.

6.1.1 Syntax highlighting

Syntax highlighting is an operation that works on the token stream of the source code. It is implemented as a token stream interpreter (the *syntax_highlight_interpreter*) that takes additional input in the form of a type-based style lookup table.

The styling table, an instance of *type_token_mapper*, maps each token to a CSS class name based on its type. The text of each token is output as part of an HTML document, with the class associated with that token. A shared stylesheet is linked to that defines the visual appearance of the each CSS class. This stylesheet can be changed to accommodate user preferences, to fit in with a company-wide style, or to mimic the behaviour of the developers favourite editor. It would be possible to allow the CSS style sheet to be changed within the client, however there has been insufficient time to implement this feature.

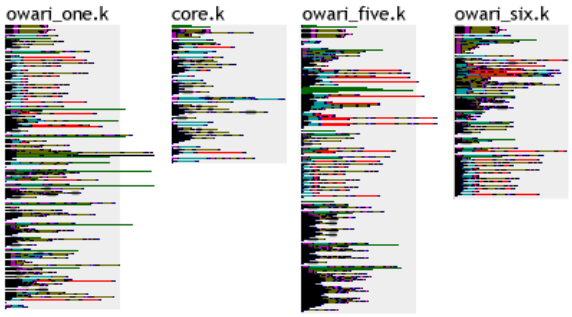

Figure 6.2: Overview diagrams for a number of files, created by the framework

6.1.2 Overview diagrams

The source overview diagrams are produced in a manner almost identical to the syntax highlighted listings. A *token_stream_interpreter* subclass is defined that also accepts a mapping from token to CSS class.

Unlike the syntax highlighting, the overview diagram generator does not always perform this mapping based on the type of the token. A different subclass of the *token_mapper* may be used to show crosscutting within the program or the results of a pattern based search, for example.

When this interpreter traverses the stream, it generates a simple SVG document. Each token is represented by a rectangle of unit height and the width of the token text, style with the CSS class associated with the token. If the same CSS style sheet is used as with the source listing (as it is by default) then the appearance of this diagram will be as a shrunken source listing, with each character represented by a unit square.

6.1.3 Call graphs and caller graphs

Call graph generation occurs in two stages, due to the use of the external graph layout tool GraphViz Dot.

The first stage is the construction of the graph description, in the graph description language given as input to Dot. This uses a helper class which allows the graph description to be constructed succinctly using methods such as *add_edge()* and performs book-keeping tasks to ensure that the script is correct.

A work-list style algorithm is used, with behaviour similar to the pseudo-code in figure 6.3. The generation of the caller graph description occurs in exactly the same way, except that the edges in the graph are defined by the reverse of the calls relationship.

The second stage of the call graph generation involves the execution of Dot to

input: $\hat{f}$ – the function for which the graph is being generated.

output: (N,E) – A graph such that:

$$N = \{n \mid n = \hat{f} \vee \exists n' \in N : n' \circ_{calls} n\}$$
$$E = \{(n, n') \mid n, n' \in N \wedge n \circ_{calls} n'\}$$

Where $\circ_{calls} : Function \times Function \mapsto \{true, false\}$

$f \circ_{calls} f' \iff$ A call exists from f to f'

W : set $= \{\hat{f}\}$
G : graph $= (\{\hat{f}\}, \emptyset)$
while $W \neq \emptyset$
do
    $W = W - f$ where $f \in W$
    for-each $f' : f \circ_{calls} f'$
        if $f \neq f'$ then $W = W \cup f'$
        G.addNode(f',nameOf(f'),linkTo(f'))
        G.addEdge(f,f')
    end for
done

Figure 6.3: Pseudo-code algorithm for call graph generation

perform the graph layout. The graph is output as an SVG diagram, making it possible for the developer to view very large call graphs on small monitor by zooming and panning the image.

The layout of the call graph can be very time consuming, particularly as the complexity of the graph increases. The cost of the generation is amortised over several uses by caching the generated graph.

6.1.4 Tabular data

Not all of the information presented to the user lends itself to a diagrammatic representation. Some types of information, such as the locations of variable uses for example, are better presented in a tabular form. There are two sources of data that are rendered in tabular form – property lists and store query results.

Property lists store key-value pairs of information. They are used in a number of situations to allow arbitrary properties to be associated with a data element. They are most often used to store calculated metrics. The transformation of a property list to a tabular form is achieved by application of templates to the pairs. For the purposes of the client application, the templates cause an HTML representation to be generated. However it is possible to supply templates that result in XML output, plain text or some other representation.

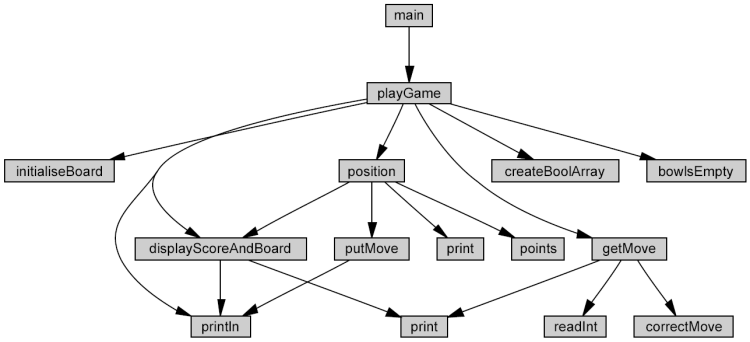

Figure 6.4: A call graph generated by the framework

Property lists contain no information about the types of the values that they store. Both the key and the value are treated as plain text. Data store tables do contain some type information – fields are known to have a certain type, such as a moniker, and so can be treated differently depending on their type.

When transforming data store tables, type information is exploited through the use of type-triggered transforms. The *field_formatter* class maintains a mapping from fields to functions that transform the data. For example, a moniker can be transformed to a link to the source location or file identified by that moniker. This allows cross referencing support to be easily provided through a formatted query script. Figure 6.5 presents a comparison transformation of property lists and data store tables.

6.2 HTTP interface

A method is required for mapping HTTP requests to visualisations and store look-ups. There are two options:

- encoding the request as a query string,
- encoding the request as a path.

Neither has any great advantage over the other. In the absence of any good reason to select either of them, the latter was chosen. The next issue was how to encode the request in the path. The information required to satisfy any one request can differ greatly from another, for example to browse to a file a file identifier is required, whereas to generate a call graph requires a function identifier, a call depth and a direction. Given this variation in type and number of parameters, it did not make sense to attempt to handle every request in the same way. Instead delegation can be used to allow requests to be handled flexibly, by a component that "knows" the the parameters it expects.

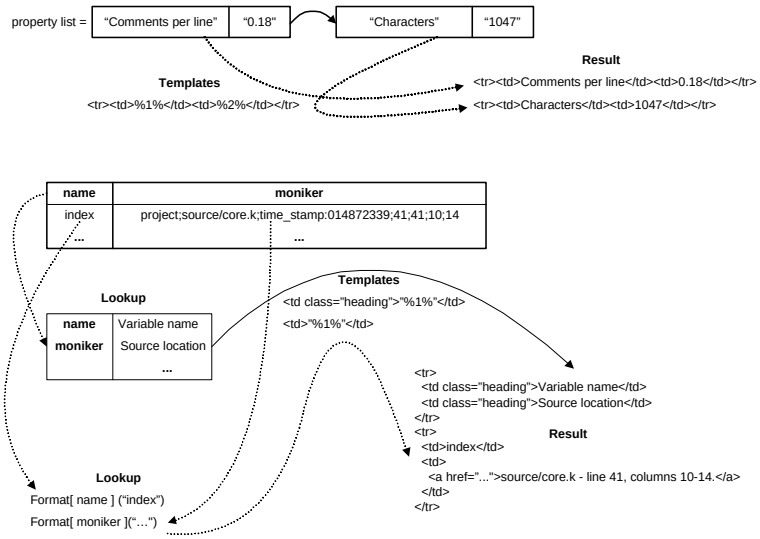

Figure 6.5: Comparison of the tabular formatting processes

The design for the HTTP interface includes a component termed the data filesystem. This component maps HTTP entity requests to functors. The first section of the entity path is treated as a key that is associated with a given functor in the data filesystem. This functor is invoked and receives the remainder of the entity path to handle the request.

6.2.1 Supporting multiple concurrent users

The specification states that the server must support multiple users. The simplest method of achieving this is to rely on the operating system queuing connection attempts to the server. Although this method provides a simple server design, it increases the request-response latency. An alternative approach is to handle the requests asynchronously, allowing the server to accept new requests as soon as they are made. This reduces the latency at the cost of requiring a parallelised HTTP server.

The standard approach is to maintain a control thread that receives the requests, and a pool of worker threads to which it distributes requests to be handled. The implementation devised for the server allows the thread pool management to be kept separate from the HTTP request handling code. Three basic tasks were identified that must be performed by an HTTP server:

- the receival of requests from remote clients,
- the handling of requests,
- the handling of errors that may occur during the process.

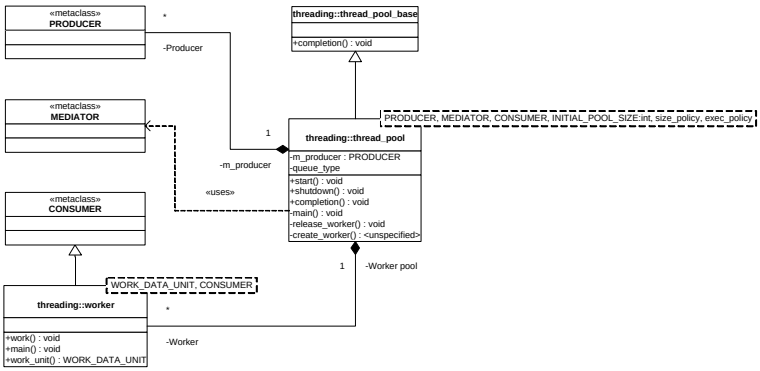

Figure 6.6: Abstract multi-threaded producer-consumer-mediator

These three tasks can be classified as Production, Consumption and Mediation. The producer-consumer relationship occurs quite frequently in computing applications with external input, suggesting that it may be worthwhile designing a component for managing a pool of consumers waiting for a producer, that can be re-used in other situations. Such a component was devised for this project.

The *threading::thread_pool* component shown in figure 6.6 provides:

- transport of work units from a producer to a consumer,
- asynchronous consumption of the work unit,
- transferral of work units to a mediator if a consumer reports an error,
- separation of thread pool management logic from work processing logic,
- flexible policy based management of thread pool behaviour.

6.2.2 Scripting the data store

A number of tasks that the client application may wish to carry out amount to little more than performing a query and displaying the results. Ideally all of these tasks could be carried out through a single interface, minimising the amount to which the backend is tailored to specific tasks or repeats common code.

The obvious solution would be to provide an interface that allowed arbitrary SQL queries to be executed, with the result displayed to the user in a tabular form. However, at the time of writing there was no SQL interface to MetaKit. As a result, some careful consideration went into how complex a scripting language for use within this framework should be.

Parsers for SQL and subsets thereof are freely available. However, there are a number of reasons why providing an SQL interface was not considered the best option in this situation:

```
script   ::=   ( '!' identifier '(' escaped_string ')'
               | '@' identifier )
               ( '.' | script )
where
 !i(p)   executes action i with parameter p
 @i      stores current result table under the name i
 .       finishes execution of the script
```

Figure 6.7: The scripting language used to access the store

- Only a subset of the features of SQL are required.
- Only a subset of the features of SQL are provided by the underlying database. A complete implementation would result in significant amounts of extra work for little gain.
- SQL is independent of domain concepts – concepts which could be leveraged to provide a more concise and intentional query language.

A simple language that allows the use of domain concepts and that can be extended to support new concepts was devised. Figure 6.7 shows the basic syntax of the language. This syntax allows much more succinct representation of queries than the equivalent SQL, which is desirable if the queries are to be transmitted across a network from a client application. For example, to find where a particular variable with the identifier *var:17* is used could be expressed in SQL as:

SELECT * FROM variable_uses WHERE references = *var:17*

Whilst the equivalent using this scripting system could feasibly be expressed as:

!variable_use("var:17").

A script action is a function that takes as input a table, an environment and a parameter, and produces a table. The table that each action receives is the result of the previously executed action, and can be ignored if not required. The output of the last action to be carried out before the period that terminates script execution is taken as the result of the script. The environment maintains a mapping between identifiers and tables. When the *@* command is used, a mapping is added from the identifier following the *@* to the current result table.

There are a number of built in actions provided by the scripting language, that support basic usage:

import loads the table named in its parameter from the data store to become the active table.

activate loads the table named in its parameter from the environment to the active table.

project discards columns of the table, keeping those specified as a comma-separated list in its parameter.

select performs a selection based on a disjunction of clauses specified as its parameter.

The HTTP interface allows the user to specify a query in this form, the results of which will be transformed to a table for presentation to the user as described in section 6.1.4. This scripting system is used to implement a number of the clients cross referencing features. For example, the locations of writes made to a given variable are found by the following script:

```
!import("var_usage")!select("references=var-id")
!select("modifies=1")!project("moniker").
```

6.3 Translating Kenya

Kenya is a programming language for teaching programming. It was designed with a simple syntax, with an easy progression to the Java language. The small size of Kenya makes it an ideal first language to use to demonstrate the features of this framework.

The grammar used by the Kenya compiler is freely available, written for the parser-generator SableCC. However, it cannot easily be used for this project for two reasons. Firstly, the grammar generates an abstract syntax tree, discarding the formatting present in the source code. Secondly, SableCC will only generate a parser in Java. As the rest of the framework implementation is in C++ this would cause difficulty integrating. Consequently it was decided to port the grammar for use with a different parser-generator.

Three alternative parsing systems were examined: Antlr[Par], Spirit[dGea] and the combination of Lex and Yacc.

The author has used both Lex and Yacc in the past for simple grammars, and whilst deciding which parser-generator to use felt that production of a reasonable sized grammar using this system could prove problematic. As an example that corroborates this opinion, the following was presented by MKS software[sof]:

```
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"
```

The pattern above is used to allow Lex to correctly match a C style comment – text enclosed within the delimiters /* and */. Debugging a grammar produced for use with Yacc can be very time consuming.

Antlr is a parser-generator able to create parsers in C++, C# and Java. It is reputed to be significantly more powerful that Yacc, although still with a steep learning curve. Antlr was considered the best option until Spirit was examined.

A description of Spirit is provided in its documentation:

> Spirit is an object oriented recursive descent parser generator framework implemented using template meta-programming techniques. Expression templates allow us to approximate the syntax of Extended Backus Normal Form (EBNF) completely in C++. Parser objects are composed through operator overloading and the result is a backtracking LL(∞) parser that is capable of parsing rather ambiguous grammars.

On experimentation with Spirit it was found to be easy to use and sufficient for the implementation of a parser for Kenya. Parsers written with Spirit can be very easy to understand, given a basic familiarity with EBNF.

Referring back to the design section detailing language translation (section 5.1.3), support for the translation of Kenya requires:

- a subclass of *translator::module* for the language, *kenya_translator::module*, used to request a translation,
- the production of the token representation of the source being translated,
- the production of the procedural IR of the source being translated.

The grammar for Kenya was ported from SableCC to Spirit with minor changes to avoid discarding white space and comments. The parser produces a parse tree output that is used for the generation of the required representations.

The production of the token representation is performed by walking the parse tree, outputting the text at the leaf nodes. Tokens that are associated with variables, functions and types have their identifier set to their location in the source code. As the parsing process does not discard any of the original source, the token text can be output to generate an exact copy of the source as required.

The production of the procedural representation is slightly more complex, occurring in two passes. The first pass involves the discovery of the nested scoping structure of the program and the population of symbol tables with type, variable and function declaration information. The second pass uses the information discovered in the first pass to generate the procedural representation with the variable uses, types and function calls correctly resolved.

Each of the passes involves a walk over the tree, matching on the language constructs of interest. The walk is a table-driven recursive process – as each node is visited an action is performed that handles that node. Table entries can be left unspecified, allowing only the program constructs of interest to be visited. The code that walks the tree relies on the parse-tree structure generated by Spirit, and as such is independent of the source language.

The translation from Kenya language constructs to procedural representation is straightforward, and most easily explained through an example. Figure 6.8 shows an example piece of source code and the equivalent object graph for its procedural program representation. For a more complete presentation, see appendix D.

```
1   // file: example.k
2
3   int main ( )
4   {
5       int value = seven();
6       if ( value > 10 )
7       {
8           return 0;
9       }
10      return 1;
11  }
12
13  int seven()
14  {
15      return 7;
16  }
```

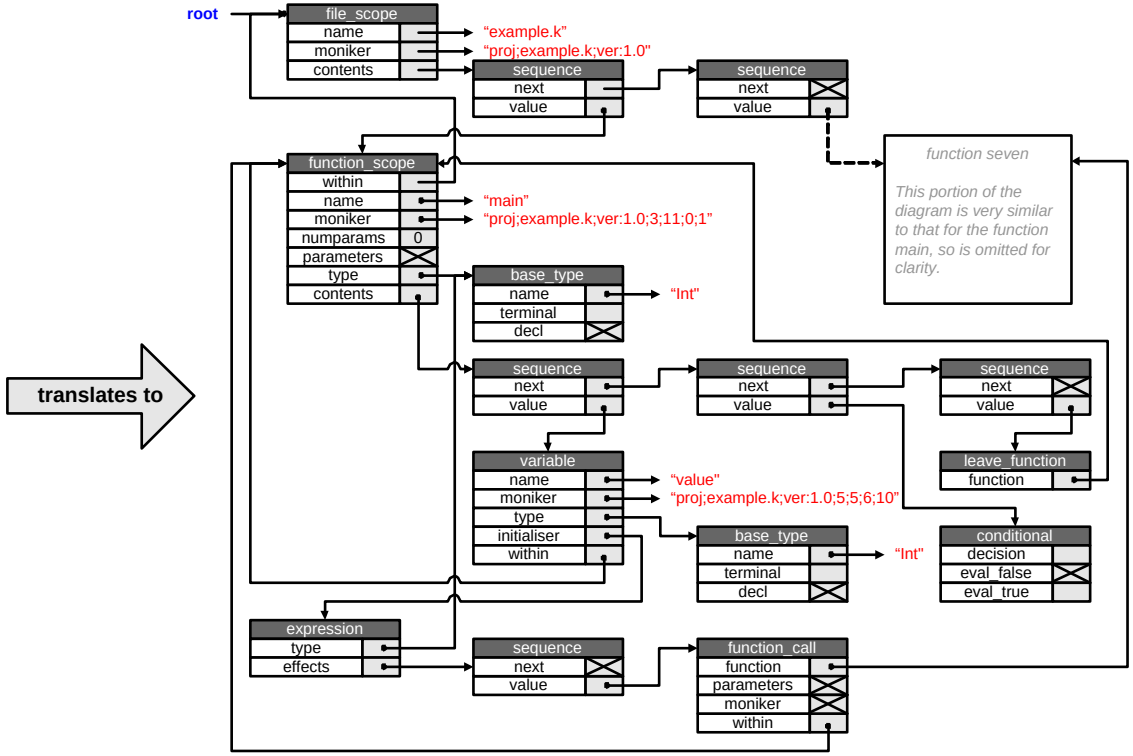

Figure 6.8: Example translation from Kenya to the shared "procedural" representation

6.4 Problems

6.4.1 Memory efficiency

The potential memory requirements for program representations were an implementation concern – large programs generate larger internal representations due to the cross-referencing that occurs.

The internal representation currently implemented is made up of a large number of small[2] structures. Small structures such as these can result in massive memory management overheads when used with standard memory allocators. This is a common problem for object oriented languages, where programs may consist of many classes with a small runtime representation, and it is solved through the use of a custom allocation strategy.

Many clever strategies have been devised for the efficient allocation of small objects. However, they are not required in this situation for a single reason – the lifetime of every object in the representation of a particular program fragment is the same. Pool based allocation is a technique where a large chunk of memory is allocated, from which subsequent requests for memory are satisfied. The common lifetime allows the following optimisations:

- There is no need to provide any compaction functionality (code that aggregates free space from holes created by object de-allocation).
- There is no need to store any management information. The objects can all be de-allocated simultaneously, so there is no need to mark sections of the pool as used or otherwise.

A further efficiency gain can be had if no destructors need to be called. In this case, destructors can be avoided.

Pool based allocation is used in the implementation for the allocation of internal representation objects. The implementation allocates a system page at a time. A side effect of this allocation behaviour is that all of the nodes of a program representation are tightly packed in to one or more pages. This has the potential to improve run time efficiency by reducing the number of page faults that occur, as well as reduced cache misses.

6.5 Libraries

The framework implementation relies on a number of third-party libraries.

AT&T GraphViz was used to layout the graphs produced by the framework. Although not easily integrated (at this point in time) as a library, it is relatively

[2] on average, around sixteen bytes in size

simple to interface with as an external process.

http://www.research.att.com/sw/tools/graphviz/

MetaKit is a multi-platform embedded database engine created by Jean-Claude Wippler.

http://www.equi4.com/

The Spirit parser framework was used in the implementation of the translator for the Kenya language. During the course of the project Spirit became a Boost library.

http://spirit.sourceforge.net/

Boost is a collection of peer-reviewed portable libraries covering a range of tasks. It was used throughout the project and allowed the framework implementation to be performed in the time available.

boost::filesystem was used to provide platform independent access to the file system.

boost::format was used as a safer *printf* function throughout, and forms the basis on which the template-based output generation is built.

boost::function and **boost::bind** are both used to allow a more functional style of programming.

boost::regex provides the all of the pattern based search functionality used within the framework.

boost::smart_ptr is used to facilitate the use of a number of idioms.

boost::thread is used to provide cross-platform multi-threading, used in the implementation of the HTTP server.

boost::utility is used to express the non-copyable behaviour of classes with clear intention.

All of the Boost libraries are available together from http://www.boost.org/.

Evaluation

> Results! Why, man, I have gotten a lot of results. I know several thousand things that won't work.
>
> Thomas A. Edison (1846–1931)

This section outlines the successes and the limitations of the project at the time of writing. It consists of:

- an evaluation against the stated project objectives,
- a point-by-point analysis of whether the specification was met,
- a critique of the approach taken to the problem.

In addition a small user trial is performed to evaluate some of the more subjective qualities of the implementation, such as ease of use.

7.1 Project objectives

Three objectives were specified at the start of the project:

1. to investigate ways in which program understanding can be facilitated, focussing on alternative program representations,
2. to investigate ways in which these procedures can be applied to multiple source languages,
3. to produce a framework that demonstrates multiple techniques for easing program understanding, allows multiple developers to collaborate on the understanding process and is designed to support multiple source languages.

Of the three objectives, only the last requires in-depth evaluation. The other two objectives can be seen to have occurred through this report and the framework implementation itself.

It is difficult to perform any kind of quantitative analysis of the framework implementation. The specification did state that, if possible, the execution time for simple queries should be kept below five seconds, excluding network based delays. This requirement on response time for requests is satisfied in most cases. Requests have been known to take longer when involving the generation of complex caller or call graphs. The graph layout is performed by an external program – AT&T GraphViz. The time taken to execute the application and for it to lay out the graph can exceed the stated five seconds for query handling time. The cost may be amortized over a multiple requests, as the generated graphs are cached and later requests satisfied from the cache. However it is not guaranteed that the graph will be accessed repeatedly so it is difficult to characterise the performance exactly.

To determine performance in the absence of network delay, the time taken to complete a selection of operations in the client was examined. Both the client and the server were run on the same machine, with tests performed to determine whether or not X_2 was satisfied. The time taken from the request being made to the results appearing an a usable form in the client was measured. The results were encouraging: in the case of a request to browse a source file, the operation took (on average) 2.2 seconds to complete with little deviation due to source file size. The production of a moderately sized call graph (containing 20 functions) took 2.8 seconds on the first request, with subsequent requests taking around 1.2 seconds.

It is possible to make a qualitative evaluation of whether the system succeeds in easing common tasks performed for the purposes of gaining an understanding of a software system. The usefulness of such a study is questionable, as the techniques implemented in the framework are known to ease understanding (as set out in the background section).

It was unfortunate that there was insufficient time before the deadline to provide support for a second language. This would have shown conclusively that the framework is capable of re-using analysis and visualisation code across languages. However, even in the absence of a concrete demonstration it should be apparent from this report and examination of the approach taken, that supporting further languages could be achieved without requiring significant portions of the framework to be re-implemented, particularly if the languages are suitable for translation to the procedural representation.

The body of code that makes up the framework (excluding the frontend) is approximately 36,000 lines of code, of which roughly 5000 provide the support for the Kenya language – around 14% of the code. Thus the amount of implementation required to add support for another language is not excessive, particularly if an existing parser is used rather than constructing one as was done for Kenya.

7.2 Specification features

This section details the evaluation of whether the project implementation satisfies the specification (see §4). For the specification items not covered in the user trials section, this section:

- states the conditions that must be satisfied in order to satisfy the specification item,
- states the tests performed to determine whether the each item was satisfied,
- states whether the specification item was satisfied.

The testing appendix contains a description of each of the tests performed. Where a test is used to evaluate the presence of a feature, its test identifier will be referenced and the results of the test stated.

7.2.1 General requirements

The general requirements (A_0 – A_3) can be seen to be satisfied by observation. The presence of the client application satisfies A_0, and the backend application A_1. The requirements A_2 and A_3 were checked through test T_0 and T_1.

It was possible to connect to the server from a machine other than that running the server. No abnormal behaviour was experienced whilst accessing the server. T_1 was also performed successfully, with several requests being made for a period of a number of minutes, showing that it is possible for a number of users to access a hosted project at the same time.

Given the design of the web server, it is possible to handle five requests concurrently by default, although this can be easily altered[1]. The requests are short lived, only lasting for the length of time it takes to generate and transfer the information requested. As a result it is not expected to be necessary to be able to handle a greater number of concurrent requests, even if there are many more users. If a request is made when there are five active requests, the request will be queued until a worker thread is available.

7.2.2 Interface functionality and operation

This section evaluates any features of the user interface specification that can be seen to be present or absent.

It is possible to connect to a server, as was done in the evaluation of the general specification, satisfying IF_0. In addition to being able to connect, the task is

[1] See the HTTP server design section, specifically the pool size policy

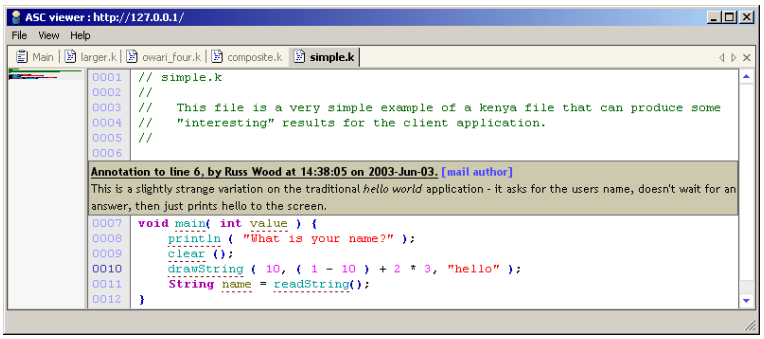

Figure 7.1: The client application viewing an annotated source file

simplified by the client keeping a history of previous projects that were connected to and providing access to them through the file menu and in the connection dialog. Access to the connection dialog is accelerated by the control sequence *Ctrl-N*, and the dialog design allows the last used project to be connected to quickly by *Ctrl-N,Enter*.

When the client connects to the server initially, it receives a page providing general information about the project. The page shows the hierarchy of sub-projects and allows each of the files within the project to be accessed by a click to their name. In addition to this it also provides access to summary information for each file within the project. The summary information includes metrics calculated for the file, information on the contents of the file and links that allow the user to jump to points of variable and function declarations within the file. This satisfies IF_1 and IF_7.

Test T_9 was performed, browsing to a source file within the project. The source code for the file was displayed and can be seen to be syntax highlighted (see figure 7.1) as required by IF_2. When a right-click occurs over a variable or function usage in the source code, a context menu appears that allows the user to jump to the declaration of, or to list all usage locations of the variable or function. The elements that can be clicked are underlined to highlight them in the source code. In addition to this, the cursor will become a hand when an element can be clicked. Thus requirements IF_3 and IF_4 are satisfied.

From the successful performance of test T_{13}, it is possible to add annotations to the source code as required by IF_3. If a line number is right-clicked upon, a menu appears that allows the user to annotate the line. Selecting this item shows a dialog box into which the user can enter the annotation text. It is possible to use HTML markup to add emphasis to certain words or phrases, to link to an external document or even an image. Although not part of the specification, each annotation that is added includes the user name and email address set in the applications preferences dialog. Figure 7.1 shows an annotation added to line six, similar in appearance to the mock-up that was provided at the start of this document.

Source overview diagrams are provided alongside the source listings, as can be seen

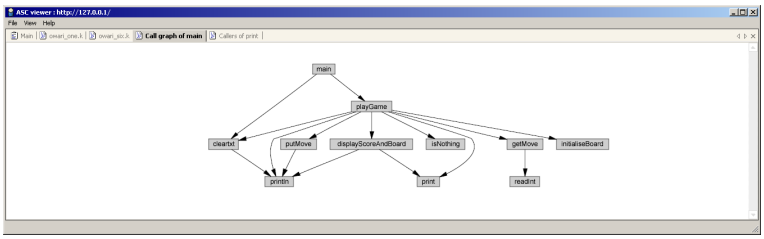

Figure 7.2: Example of a generated forward call graph for main

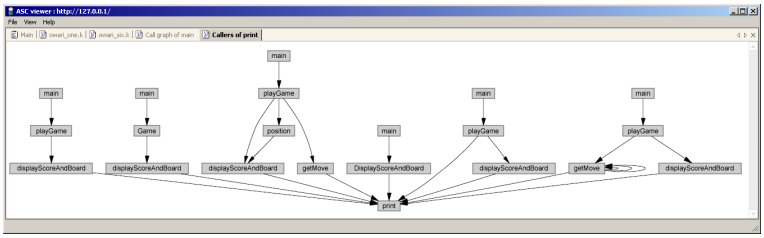

Figure 7.3: Example of a generated reverse call graph for print

in figure 7.1. The original intention with this placement was for the overview to be used to jump to points in the file. However due to issues with communication with the Adobe SVG renderer, it has not been possible so far to enable this behaviour. Alternatives have been examined but there has not been sufficient time to implement this feature. Test T_{15}, the test dealing with overview diagrams, was successful.

It is possible to access call graphs through the same pop-up menu that provides cross-referencing. Both forward and reverse call graphs are supported, as shown in figures 7.2 and 7.3. For the same reason as the source overview diagram, although the call graph contains information to link the graph nodes with the source listing, it is not possible to follow those links. Although desirable, it was not a requirement to be able to jump from the call graph to the source listing, so IF_8 is satisfied.

The specification requires that the interface be usable at a screen resolution of 1024x768 pixels or above. The application is still usable at a resolution of 800x600 pixels, although the amount of source visible at any one time is obviously limited at this resolution. Once the screen resolution reaches 2048x1536 the application is not making the best possible use of screen space when viewing source code, as there is only a single column of source. When viewing complex call graphs at this resolution it becomes easier to see the entire structure as there is less need to pan over the call graph to see sections that would not fit in to the application window at a readable zoom level. If re-designing the interface a more flexible partitioning scheme would be used, so that multiple source files could be viewed simultaneously when there was sufficient screen space.

In addition to keyboard shortcuts for menu items, additional shortcuts have been added to allow quick navigation through the multiple tabs that may be open at any one time. When holding the control key, the active tab can be changed by using

the left and right keys, and closed using the up arrow key.

When constructing the user interface, care was taken to ensure that the keyboard focus was set to a suitable location when a new dialog appeared and that the tab key could be used to switch the focus between input elements in a logical order.

7.2.3 Backend functionality

The original specification of the background was very flexible over what was required of the implementation. As a result all of the points (BF_0–BF_3) in the specification have been satisfied. The functionality built in to the backend was driven by what was required of the frontend, as can be seen in the design discussion earlier in this document.

With respect to requiring any third party software to be set up, the backend can be distributed with all of the libraries that it requires in a single directory.

7.2.4 Language support

It is apparent from the design discussions in this report that the backend was designed with the intention of supporting multiple languages (L_0). It is currently possible to use source code written in the Kenya language with the system (L_1).

The information on the translation of the Kenya language explains the steps that were taken to support Kenya. From the implementation discussion it should be relatively clear how to extend the framework with support for a similar language.

7.2.5 Extensions

Due to the limited time for the project, it was not possible to implement the slicing or ripple analysis techniques. It is estimated that it would take a further week of work to provide these features.

The requirements on response time for requests are generally satisfied, as explained earlier in the evaluation.

7.3 User trials

To evaluate the more subjective elements of the specification and the general ease of use of the application, a small test has been prepared to determine:

- whether the application behaves as the user expects,

- how easily the user can accomplish tasks using the facilities provided by the system.

A solution to this test is also provided as part of a walk through showing the usage of the client application (see appendices).

The author's overall impression was the the participants found the application relatively easy to use. They did not tend to use any of the keyboard shortcuts designed to make the use of the application more efficient, but this can be put down to a lack of knowledge that the shortcuts exist.

All of the participants looked straight in the summary section to accomplish the task of finding out source metrics. However, when performing the other tasks there was a degree of experimentation and searching before methods were uncovered for the accomplishment of the task.

The greatest length of time taken to complete the tasks was just over five minutes from start to end. This is almost definitely faster than it would be achieved by the majority of people without tool support. The use of suitable abstractions and gathering of information allowed the noise (in the form of the program itself) to be eliminated from the problem.

7.4 Approach

This section contains an evaluation of how the project task was tackled and suggests ways in which the problem could have been approached differently that may have been better.

Without the choice of tools that was made, it is doubtful that the implementation would have reached the stage that it did. The use of C++, a language that the author is very familiar with, allowed the design of the application to be concentrated on rather than the code itself. The choice of C# allowed the use of many existing components that are not available with Java (for example) or that are more complicated to use with C++.

When discussing potential further work (section 8) an alternative design is suggested that appears to be more elegant on initial investigation. This was a design that came to mind as a result of work on this project, rather than being one that could have been devised by the author beforehand. However, even if it had been thought of initially, it is a design that may have proven too complicated to execute within the project period. The design that was used is more straightforward, allowing potential problems to be seen more easily.

Although the use of HTML was an efficient method (with respect to development time) of providing styled documents, it was far from the most efficient from the transport perspective. The problem is due to the number of elements that are marked up – surrounding each source element with an HTML *span* to associate it with a style resulted in a fifteen to twenty times increase in file size from the source

code to the generated HTML.

On a fast network, this isn't a major problem, but browsing could be made significantly more efficient through the use of an alternative transfer mechanism - such as transfer in the token form. The upshot of this would be a shift of a portion of the workload on to the client, which would then have to transform the tokens in to HTML for presentation.

7.5 Summary

- All of the observable mandatory parts of the specification were met.
- Subjective elements of the specification can be deemed to have been met from the results of the user trials
- Quantitative performance measures cannot easily be obtained and have little meaning due to the un-characterised performance of the methods used for graph layout, which are highly dependent on the the input.

Conclusions

> I have learned to use the word impossible with the greatest caution
>
> Wernher von Braun (1912–1977)

Although (in the opinion of the author) this project has succeeded in what it set out to do, the size of the problem presents many areas that offer opportunity for further work.

This section presents a selection of areas that could be worked on, divided in to three categories:

Additional features presents a number of ways in which the functionality of the framework as it stands could be built on, providing further support for code understanding.

A different approach examines the possibility of implementing the framework in a more elegant demand-driven manner.

A change in focus suggests areas of work where the ability to understand the code is required, but not the main task at hand.

8.1 Additional features

There are many further techniques that could be implemented within the framework:

- Some additional metrics could be added. There are software tools[1] in existence that simply provide the user with access to hundreds of different metrics.

[1] see McCabe & Associates : http://www.mccabe.com/

Although not all of these are useful from the perspective of easing code understanding, there are a number that would help with the maintenance tasks that tend to follow on from code comprehension tasks.

- Points-to analysis[GH98, Ste96] has been an active research area in the past. Though most commonly used in compilers for the purposes of optimisation, they also have applications for code comprehension – they allow program behaviour to be more easily visualised by determining what cannot be affected by an operation.
- Data flow analyses can be used to obtain more accurate results in some cases (variable write locations using du-chains to highlight individual statements that could have written to a variable) and more complete results in others (call graphs in languages where functions are first-class).
- Design recovery techniques can be used to automatically or semi-automatically obtain high level architectural information from the source code. Recovered information can be presented to the user in the form of UML diagrams[2]
- Documentation extraction can use similar techniques to design recovery, and the association of comments with program entities to produce documentation for a software system. Even when there is no commenting in the source code, skeleton documentation can be generated including cross referencing and usage information. This is immediately useful and can be extended by the developer during investigation of the source code.

  Production of skeleton documentation provides an interesting alternative interface concept to that used for this project's client. The user would be presented with the skeleton documentation for the source code, and would gradually build up a "book" of information as they browsed through it.

8.2 A different approach

During the development of the system, an alternative design was considered that may have been a better option with respect to re-use, and more interesting from a software design perspective. The design was a demand-driven pipeline based approach, making use of components with meta-data describing their cost, inputs and outputs. There would be three types of component:

Sources would provide basic information such as a token or control flow based representation of a program. Components may exist that build this representation from information stored previously in a data store, avoiding the need to refer to the original source information.

Sinks would provide output that may be desired by the user: a call graph visualisation, highlighted source code and so on.

Transforms would act as internal pipeline elements, translating one form of data to another.

[2] see Imagix Corporation : http://www.imagix.com/

These components would form a directed graph that would map potential paths from sources to sinks. Edges would be weighted based on the cost information from the meta-data. A user request for a certain output (a sink node in the graph) would result in the selection of the least cost path from the graph from the sink to a source, followed by the execution of that pipeline.

This is not a simple problem to solve, but it would be an interesting challenge. Similar problems to this are encountered in component based development and Grid computing.

8.3 A change in focus

Further work involving a slight change in focus from the production of a framework purely to support understanding, would be to study ways of supporting software evolution. Though the work carried out in the project and that suggested as further work thus far provide basic support for change management, there are other areas of research that directly support it:

- Cliché recognition[Wil96] is the discovery of common operations at a higher level of abstraction. Examples include the identification of a functions behaviour as sorting data, or of the discovery of a pattern implementation in the code. This information is particularly useful if the code is to be ported to another language in the absence of documentation.
- Discovery of program properties (such as invariants) can be used to aid understanding of a system. It can also support verification that behaviour of an extended system is equivalent to a previous version.

Bibliography

[AG96] Darren C. Atkinson and William G. Griswold. The design of whole program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27. IEEE Computer Society Press, 1996.

[auta] Various authors. C2 wiki : Context objects are evil.

[autb] Various authors. C2 wiki : Singletons are evil.

[BE95] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6(2):119–133, 1995.

[BE96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[Ben02] G. Bennet. A refactoring code browser for java. 2002. http://freefactor.sourceforge.net/.

[BLT78] E. Swanson B. Lientz and G. Tompkins. Characteristics of application software maintenance. 1978.

[BR] J. Brant and D. Roberts. Refactoring browser tool. http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser.

[dGea] Joel de Guzman et al. The spirit parser framework. http://spirit.sourceforge.net/.

[EDG] EDG. The edison design group : Compiler front ends for the oem market. http://www.edg.com/.

[Eic94] Stephen G. Eick. Graphically displaying text. *Journal of Computational and Graphical Statistics*, 3(2):127–142, 1994.

[ESJ92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. Seesoft (tm) – a tool for visualising line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[Fai02] Ted Faison. *Component based development with Visual C#*. M and T books, 2002.

[Fow00] Martin Fowler. *Refactoring : improving the design of existing code*. 2000.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[Hay] James Hayes. A method for adapting a program analysis tool to multiple source languages.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[KH01] G. Kiczales and E. Hillsdale. Aspect oriented programming with aspectj, 2001.

[LA93] Panos E. Livadas and Scott D. Alden. A toolset for program understanding. In Bruno Fadini and Vaclav Rajlich, editors, *Proceedings of the IEEE Second Workshop on Program Comprehension*, pages 110–118, 1993.

[LR92] Panos E. Livadas and Prabal K. Roy. Program dependence analysis. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 356–365. IEEE Computer Society Press, 1992.

[LS80] B. Lientz and E. Swanson. Software maintenance management, 1980.

[LS94] P. Livadas and D. Small. Understanding code containing preprocessor constructs, 1994.

[Luc01] Andrea De Lucia. Program slicing: Methods and applications. IEEE Computer Society Press, Los Alamitos, California, 2001.

[MB89] T. McCabe and C. Butler. Design complexity measurement and testing. *Communications of the ACM 32*, 1989.

[MNGL98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[Par] Terence Parr. The antlr translator generator. `http://www.antlr.org`.

[sof] MKS software. Using mks lex and yacc effectively. `http://www.mkssoftware.com/support/ly/ly_use.asp`.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[the] The codist – a survey of source browsing tools. `http://members.tripod.com/codist/`.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[Vel95] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[Wei79] M. Weiser. Program slices : formal, psychological and practical investigations of an automatic program extraction method. phd thesis. 1979.

[Wil96] Linda M. Wills. Using Attributed Flow Graph Parsing to Recognize Clichés in Programs. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073, pages 170–184. Springer-Verlag, 1996.

[Wip] Jean-Claude Wippler. Metakit embeddable database engine. http://www.equi4.com/.

User's guide

> The mediocre teacher tells. The good teacher explains. The superior teacher demonstrates. The great teacher inspires
>
> William A. Ward

A.1 Client walk-through

A simple users guide for the client is provided through the its help menu. It contains quick-start tutorials on basic usage of the user interface. It will be the most up to date source of information on usage.

This section will contain an introduction to the client application using a walk-through problem solving session, looking at the problems set in the test script used for the user trials (see appendix E).

To begin with, figures A.1 and A.2 introduce two views of the client application, highlighting main areas of interest.

1. The title bar. The title bar will show the location of the active project.
2. The tab area. This will only be visible once a project has been connected to. The tab area shows all of the open windows. Each new file loaded opens in a new tab.
3. The tab control icons. If there are many open tabs, these arrows can be used to navigate between them or to close the current tab.
4. A source file. Each source file in a project is listed as shown here. A click with the left mouse button will take you to the source listing for that file.

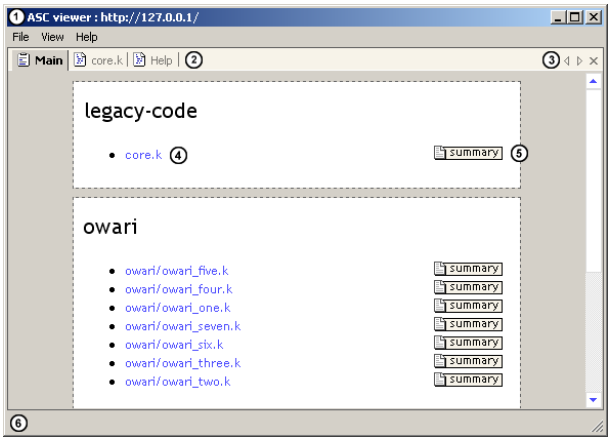

Figure A.1: The project view screen of the client

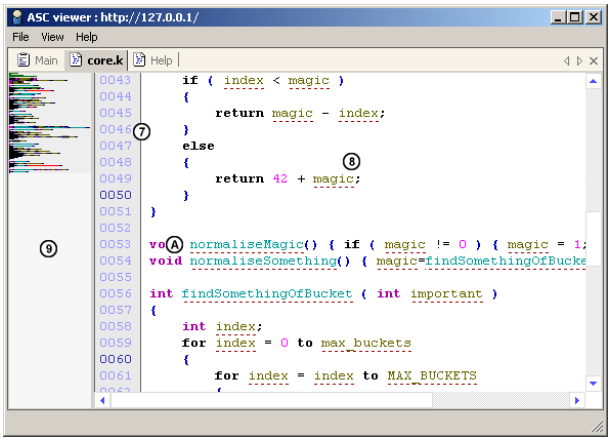

Figure A.2: The source view of the client application

5. The summary icon. Each file has a summary icon next to it. When this icon is clicked, a new tab page will open showing information about the file, including source code metrics and variable and function declaration information.

6. The status area. When browsing a source code listing, helpful comments may appear hear to explain briefly the effect of an action about to be performed.

7. The line numbering area. Each line of source code has its line number displayed along side. Right clicking on this number allows annotations to be added to that line of source code.

8. (and A), variables and functions. Parts of the source code listing that are underlined can be clicked on to bring up a menu of actions that can be carried out for that item.

9. The source overview diagram. Each listing has an accompanying diagram that shows the source code structure in a reduced form.

There now follows the walk through of the problems set in appendix E. The source files required for these tasks are contained in the example project supplied with the application source code. To begin with, it will be necessary to provide access to the results of analysing the source code. Assuming that the server is built, and the user has a command prompt in the project source directory, issue the following commands (assuming unix style path separators):

build/asc_manager /destroy example/example.scp to ensure that the next step is possible.

build/asc_manager /create example/example.scp to create the project, extracting information from the source code.

build/asc_manager /host example/example.scp to publish the project on the port specified in the project definition file, example/example.scp.

The first two steps can be skipped if the project has been created previously. After the third step, it will be possible to connect to the server on the port specified in the project definition file.

The first problem was set to gauge the where the user expected certain features to be. It is a relatively easy task. The task was presented as follows:

> *Your company is misguided and uses the comments/line ratio as a measure of code quality. As part of your initial assessment of the source code, your manager asks you to find the comments/line ratio for core.k.*

To begin with, the client application is run. To connect to the project server, select the *File-¿Connect to Project...* option from the main menu. Enter in the host name and port as a standard web address; for example, to connect to a project hosted on port eighty of the same machine as the client, enter *http://127.0.0.1:80/*.

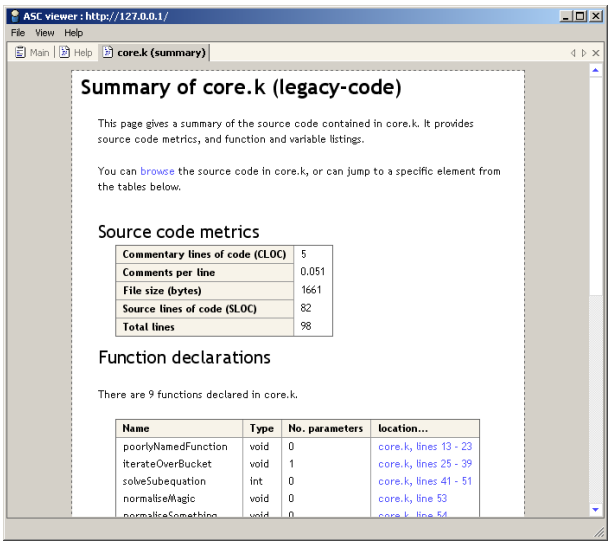

Figure A.3: Browsing the summary information of a file

Having connected to the server, a screen similar to figure A.1 should be seen. To find the metrics for the source file *core.k*, the summary icon alongside the file name is clicked. This presents the user with summary information, similar to figure A.3. The comments/line ratio can clearly be seen in the table in the centre of the figure. The second problem is to examine how they use the tools at hand, how they explore the interface, and whether they can see how to solve the problem. It does depend somewhat on prior knowledge of what some of the visualisation provide. The task is:

> *You have been informed that there is an optimisation that can be made to the function solveSubequation() if it is never called via normaliseSomething(). Is it possible to perform this optimisation?*
>
> *Note: "called via" should be taken to mean that there is no chain of function calls from an arbitrary point in the program to solveSubequation() that contains normaliseSomething().*

This is a case of checking whether or not the function *normaliseSomething* is the caller graph of *solveSubequation*. If it is present, then the optimisation cannot be performed.

The reverse call graph is obtained by browsing to the definition of *solveSubequation* in the source code of core.k. This can be done either by browsing the file, or through the function declaration list in the file summary of core.k.

Once the function has been found, right clicking on *solveSubequation* (which will appear underlined) will bring up a context menu. Selecting caller graph from that menu will open a new tab containing the graph. The result will be similar to figure A.4. Figure A.4 presents a very simple caller graph. The graph shows that

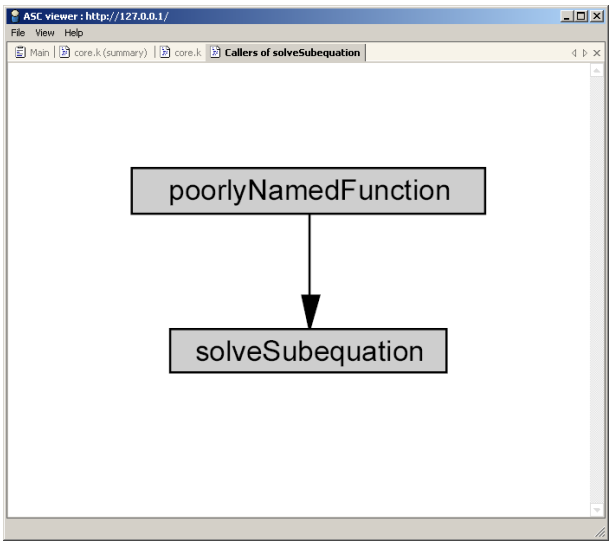

Figure A.4: Using the caller graph for problem solving

solveSubequation is called by one function only, and that function has no callers. As a consequence (in this project) it could be deemed safe to perform the optimisation.

The third question makes use of the cross referencing information provided by the framework to perform a debugging related task:

> *There is a bug that you believe is caused by an invalid value being assigned to the variable* **index** *before it is used in* **iterateAgain***(). Find where the value could have been assigned to* **index**.

The approach to solving this problem is similar to that for the previous one, beginning with navigating to the function *iterateAgain*. Having found the function, it is apparent that there is a single use of the variable *index*. Right clicking on the variable brings up a context menu, from which "Show writes" should be chosen. This will produce a list of the locations that variable is written to, each of which can be checked to see if it could be the source of the problem.

Being able to use cross referencing information in this way is far more powerful than a simple text based search, particularly if the code being examined contains a large number of similarly named identifiers. A simple search would highlight all instances of that name, not only the instances where that name refers to the problem variable.

The solution to this problem could be improved through the use of program slices, or in the absence of slicing functionality, the generated write information can be combined with the caller graph of *iterateAgain* to narrow down the search for the offending write.

The fourth and final task in the problem sheet was a short and rather open question:

You have noticed that the functionality provided by obsoleteCalculation() is no longer required. Which other parts of the source code can be removed?

To solve this question, a variety of the facilities provided by the framework are used. The approach taken by the author was as follows:

1. The function *obsoleteCalculation* was found in the source code.
2. The function was seen to contain only a call to *iterateAgain*, so the next step was to determine whether *iterateAgain* was called from anywhere else through the use of the caller graph.
3. Having seen from the caller graph that *iterateAgain* is not called from any functions other than that being removed, *iterateAgain* is marked down for removal.
4. It is obvious from the source that none of the functions called from *iterateAgain* can be removed – the only function called is the library function *println*. So each of the variables used in the function should be checked for usage. Using the "Show uses" option on the context menu for the variables *index* and *arbitraryNumber* shows that *index* is used in many locations throughout the source, but *arbitraryNumber* is only used once and so can be removed.
5. There are no other functions called directly or by transitivity, so there is no more to do.

The walk-through introduced a number of features supplied by the client application. As yet, the client does not supply access to all of the features provided by the framework. For further information on the operation of the client, the reader is encouraged to experiment.

A.2 Server application

The server application is currently a command line process, although it could easily embedded in a graphical user interface. All of the information needed for project management is provided through a project definition file (described below), allowing the command line options to be kept to a minimum.

The format of the command line is *asc_manager [option] project-definition-file*, where option is one of:

/help causes the application to print brief usage information to the console.

/create creates a new project based on the information in the project definition file. The definition file must specify a location that does not already exist.

/update updates the information in the project created by the previous call to **/create** for the definition file specified. The project must currently exist - an update on a non-existent project is not equivalent a create.

/destroy tells the system to remove all of the files that it generated in the process of creating or updating the project associated with the definition file.

/host tells the application to publish the project for access by (possibly remote) clients, using the port specified in the project definition file.

The project definition file provides:

- The port on which to publish the project information when the **/host** command line switch is used.
- The directory in which to place all generated project information.
- Sub-project specifications.

A sub-project specification is used to divide a project up into a logical grouping. A specification has a name, a root location, a set of extensions to match on and a set of rules to determine files to include. The way in which these rules are evaluated depends on the project instance being used. The filesystem based project treats the rules as a set of directories and directory trees to include or not include.

The example project provided with the project source in the *example* directory contains the project definition file *example.scp*, similar to the listing shown in figure A.5.

```
# example.scp - an example project definition file.
#
# Lines beginning with # are comments, blank lines are ignored.
#

# data-store-location:
# This is the location in which to place the results of source code analysis.
data-store-location: /Development/ASC/example/scproj

# publish-port:
# Identifies the port on which to publish the project.
publish-port: 8080

# define-project-tree and associated rules:
# These are used to specify what source files should be included in the analysis.
#
# The base-identifier is the name that maps to a particular project root directory.
#
# The include rule specifies that the contents of the given directory
# should be included, with the exclude rule asserting the opposite.
#
# The include-subtree and exclude-subtree recursively process subdirectories
# All of the directory inclusion and exclusion directives are processed in order,
# so that specifying include: a followed by exclude: a has effect of excluding a.
#
define-project-tree: base-identifier-1
        project-root: /Development/ASC/example
        include-subtree: /Development/ASC/example
        exclude-subtree: /Development/ASC/example/build
        exclude: /Development/ASC/example/documentation
        include: /Development/ASC/example/build/platform-specific
        use-extensions: .cpp;.hpp;.cxx;.hxx;.h
```

Figure A.5: An example project definition file

Developer's guide

> From the amoeba to Einstein, the growth of knowledge is always the same: we try to solve our problems, and to obtain, by a process of elimination, something approaching adequacy in our tentative solutions.
>
> *Objective Knowledge: An evolutionary approach*
> Karl Popper

The design itself has been covered in the earlier sections, this section will just add some information on the layout of the project and information on the build environment.

B.1 Project layout

The project is made up of somewhere in the region of two hundred source files. To make working with this many files manageable the project source is split into many directories:

base/support contains reasonably generic code that supports the framework. The files in this directory mainly deal with I/O and debugging.

base/net contains the basic networking code. This includes code for sending and receiving over TCP sockets.

base/threading contains a generic multi-threaded producer-consumer-mediator relationship implementation that is instantiated to generate the project web server.

core contains the core data structures used for communication between various parts of the system.

analysis contains generic analysis code, as well as the analysis selection code and any specialised analyses.

analysis/procedural contains the analysis for the "procedural" language family.

host contains the implementation of the backend, including the server components and the command line processing.

language contains generic translation code and routines designed to ease the implementation of language-specific translators.

language/kenya contains the translator for the kenya language.

store contains the implementation of the data store for extracted information.

store/query contains code used to read information back from the data store.

store/update contains code used to update the information stored for each file.

store/script contains a miniature scripting system that can be used to extract information from the data store, intended to make it easy to add simple data manipulation features to the frontend.

store/utility contains code used for data transformation and formatting.

store/common contains code that is shared between other parts of the store.

visualisation contains code used to generate some of the alternative representations of the source code.

test contains some of the code that was developed for testing purposes.

shared contains shared files used by the server.

example contains an example project used for testing.

B.2 Build environment

The frontend is simple to build through the Visual Studio.NET workspace supplied with the project. Building the backend portion has the potential to be slightly more involved. It requires the following:

- MinGW (developed using GCC 3.2.2 version)
- GNU make.
- Boost 1.30.0
- AT&T GraphViz (dot binary in path is sufficient)
- Metakit 2.4.9.2

The library versions used during development are contained in the source tree in the ThirdParty directory. The build itself occurs through a single Makefile, which is commented enough that it should be relatively simple to change.

At the end of the build, prior to linking, all of the object files are copied into the "dump" directory. This is a work-around for a problem I encountered where the length of the link command line caused the shell to crash. It is obviously not an ideal solution, but it does work.

The link process expects libraries to be in certain places; When boost is compiled the library files required need to be copied into the ThirdParty/lib_prebuilt directory. This must be done separately, although the "fetchboost" script in the project directory provides some clues as to how this is done.

Once the libraries are built, the build process should be a simple matter of running make.

B.3 Known issues

An oversight in the parser produced for the Kenya language means that comments may not appear in certain positions that would be allowed by the Kenya compiler – positions where comments would rarely appear, but should be permitted all the same.

The client application does not provide a method to specify the depth of generated call or caller graphs. This can lead to extremely large graphs that may take many minutes to generate and even download – particularly in the case of caller graphs of commonly used library functions.

Another issue is the completeness of the client and the web-interface. The client application does not provide a user interface capable of using all of the features provided by the framework. In addition there are some framework features that are not yet accessible through the web interface, including (but not limited to):

- The ability to adjust the highlighting method used with the source overview diagrams. The framework allows a variety of different highlighting techniques to be used, some of which the client would use to highlight search results and provide pattern and type based searches.
- The ability to directly interact with the stored data – an interface component is partially complete that would allow queries on the store to be constructed from a set of basic operations.

As a result of the embedding of the WebBrowser component in the client for the purposes of HTML rendering, the Adobe SVG plugin no longer appears to react to user interaction in the same way. Each SVG element representing a function in a call graph has an associated URL that would allow the user to jump directly from the call graph to the declaration of the function. It should be possible to fix this

with a bit of experimentation with the generated runtime callable wrappers, or the COM interfaces implemented by the browser component. However, it is a minor issue that is not expected to greatly affect the operation of the tool.

Testing

> My pessimism extends to the point of even suspecting the sincerity of other pessimists.
>
> Jean Rostand (1894–1977)

This section describes the test setup that was used to determine whether the specification was satisfied. All of the tests described in this section are for the purposes of evaluation of the specification features only. Individual components that were developed for the framework were tested separately as they were developed.

For testing purposes, the set up shown in figure C.1 was used. Version 1.1 of Microsoft's .NET framework was installed for the execution of the client.

For the remainder of this section, Machine A will be referred to interchangeably as *A*, with similar shortenings being used for Machines B and C. Note that the majority of these tests are dependent on the presence of a usable network between the three machines. There is no point in performing these tests at all if there is a network firewall or some other obstacle in place that would prevent communication.

T_0 **Test procedure:**
A client run on Machine B is used to connect to the server running on Machine A.

The successful performance of this test shows that:

- It is possible to connect to the server.
- Connection is possible from a machine other than that running the server.

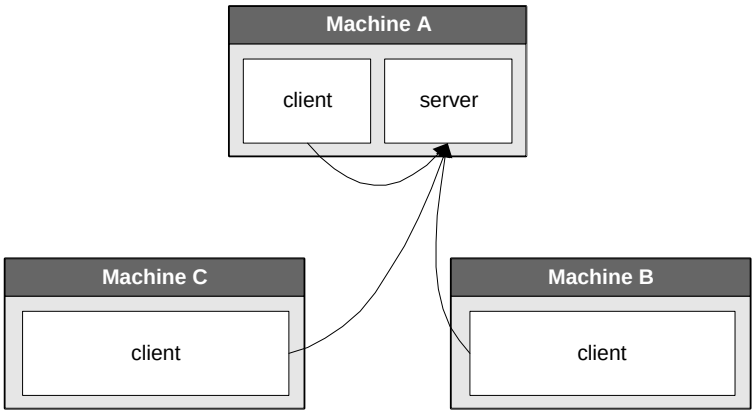

Figure C.1: The set up used for testing the framework implementation

T_1 **Test procedure:**
A client run on Machine C is used to connect to the server running on Machine A, whilst at the same time connections to the server on *A* are being made by *B*.

The successful performance of this test indicates that it is possible for the server to be used by more than one client at a time.

T_2 **Test procedure:**
The client application is run on any of *A*,*B* or *C* at a screen resolution of exactly 1024x768 pixels. The following conditions must hold:

- It must be possible to access all menu items.
- It must be possible to scroll the view if necessary to see the contents.

Completion of this test satisfies the requirement that the client software be able to run on machines with low resolution displays.

T_3 **Test procedure:**
For a reasonably sized framework project, a change should be made to a single source file in the project. The source file changed should have no dependencies within the project. Only the source file changed should be updated by the framework update process.

This test is part of evaluation of the behaviour of the framework when updating existing projects. When a single file with no dependencies has been changed, the only file that need be updated is that file.

T_4 **Test procedure:**
Using the same project as T_3, a change should be made to a single source file that does have dependencies within the project. The framework update process should be observed to update the altered file and all files dependent on it.

As T_3, this test is part of the update process evaluation. This is a test to determine whether dependent files are identified correctly.

T_5 **Test procedure:**
Using the same project as T_3, a file with no dependencies within the project should be removed. That file should be noted as removed by the framework during the update process, but require no user confirmation of removal.

This test is to identify whether the files that can be removed without affecting the information stored about other files are removed automatically.

T_6 **Test procedure:**
Using the same project as T_3, a file with dependencies within the project should be removed. On update the framework should be demand confirmation for the removal of that file.

This test is to ensure the correct behaviour occurs in the absence of complete information. It may be the case that functionality provided by a removed file to its dependents is now provided by another file. However, the framework may not know this, in which case it is desirable to check whether the file should be removed.

T_7 **Test procedure:**
For each dialog in the client application, cause the dialog to be shown and begin typing. Observe that the keyboard focus is initially set to a sensible position.

This shows that care has been taken in the construction of prototype client application with respect to usability issues associated with focus placement.

T_8 **Test procedure:**
Given a correct project definition file, the command line interface to the framework should be used to create a project from that definition file.

This test determines whether the it is possible to create a project using the command line interface.

T_9 **Test procedure:**
Using the same project definition file as with T_8, use the command line interface to the framework to delete the project. All files that were created by the performance of T_8 should be removed.

This test checks that the deletion functionality required by the specification has been provided (ie. whether the framework is capable of cleaning up after itself).

T_{10} **Test procedure:**
Given a project description specifying at least one source file, a project should be created on Machine A. A client running on any of the machines should connect to the project server running on *A* and should request to browse the syntax highlighted source code of a file in the project. The source code listing presented should contain the same source code as the file, but be syntax highlighted.

This test determines whether the system is able to produce syntax highlighted source code.

T_{11} **Test procedure:**
Machine A should publish a project containing a file with a number of function calls within. A client running on any machines should be used to connect to the server and browse to the file. A number of functions within that file should be selected and their forward and reverse call graphs requested. The produced graphs should be observably correct.

This test determines whether the system is able to produce forward and reverse call graphs.

T_{12} **Test procedure:**
Machine A should publish a project that contains a file with a number of variables declared that are used at some point in the project. The client should be used to request the locations at which those variables are used, written and read. The uses should be the union of the reads and writes. The locations of the reads and writes to each variable should be observably correct. Selecting an arbitrary read or write of a variable, request that the client jump to the point of declaration of that variable.

This test determines whether it is possible to obtain usage information for variables from the client. Additionally it tests whether it is possible to jump to the declaration of the variable.

T_{13} **Test procedure:**
Client applications on A and B should connect to a project served by machine C. The client running on A should be used to add an annotation to a particular line in a file. The client running on machine B should browse to that file. The annotation added by machine A's client should be visible on machine B. On a revisit to the file, the annotation should be visible to both clients. The server running on machine A should be shut down and restarted. The annotation must still be visible to both clients on reconnection.

This test determines whether annotations can be added to a line of source and whether annotations are preserved over server executions.

T_{14} **Test procedure:**
A client running on any machine should be able to connect to A and view a selection of source code metrics for any file in the hosted project.

This test determines whether source code metrics are provided.

T_{15} **Test procedure:**
A client running on any machine should be able to connect to A and view a source overview diagram for each file in the hosted project.

This test checks for the ability of the framework to provide source overview diagrams.

Translation of Kenya language constructs

> His eyes seemed to be popping out of his head. He wasn't certain if this was because they were trying to see more clearly, or if they simply wanted to leave at this point
>
> *Life, the Universe and Everything*
> Douglas Adams (1952–2001)

This appendix provides a description of the translation scheme from Kenya language constructs to the procedural program representation. This description is means to concisely communicate the transformation that occurs, rather than present a rigorous formal description of it. For a visual representation of this translation taking place, see figure 6.8

The symbol ℓ is used to represent the location in the source code of the Kenya construct being translated. The $\Uparrow_{\mathrm{T}}$ symbol is defined to mean the closest containing language construct with dynamic type T. If the parameter T is omitted, $\Uparrow$ will represent the closest parent scope. $++$ indicates concatenation.

For a description of the structures to which the Kenya language constructs are converted, see figure 5.7.

$$
\begin{array}{lcl}
[\![\texttt{type}]\!] & & \\
\quad [\![id]\!] & = & (\texttt{true}, id, \textbf{typeof}(id))_{\text{BASE\_TYPE}} \\
\quad [\![t[]]\!] & = & (\texttt{false}, \texttt{false}, 0, [\![t]\!])_{\text{INDIRECT\_TYPE}} \\
\quad [\![t[n]]\!] & = & (\texttt{false}, \texttt{true}, n, [\![t]\!])_{\text{INDIRECT\_TYPE}}
\end{array}
$$

Types consist of an identifier, which may be a built in type or user defined, and any number of array brackets. If the dimension of an array type is known, it is recorded.

$[\![\texttt{seq}]\!]$

$$
\begin{array}{lcl}
[\![s; tail]\!] & = & ([\![s]\!], [\![tail]\!])_{\text{SEQUENCE}} \\
[\![s;]\!] & = & ([\![s]\!], \texttt{null})_{\text{SEQUENCE}}
\end{array}
$$

The $[\![\texttt{seq}]\!]$ rule represents the translation of sequences of statements. A sequence takes the form of a singly linked list of translated statements.

$[\![\texttt{block}]\!]$

$$
[\![\{s\}]\!] \quad = \quad (\texttt{null}, \Uparrow, [\![s]\!])_{\text{SCOPE}}
$$

Anonymous block scopes are represented by a nameless scope structure. The translated contents of the block are stored within the scope. A chain of references to enclosing scopes is kept for navigation purposes.

$[\![\texttt{cond}]\!]$

$$
\begin{array}{lcl}
[\![\text{if } (e)\ b_{\texttt{true}}]\!] & = & ([\![e]\!], [\![b_{\texttt{true}}]\!], \texttt{null})_{\text{CONDITIONAL}} \\
[\![\text{if } (e)\ b_{\texttt{true}}\ b_{\texttt{false}}]\!] & = & ([\![e]\!], [\![b_{\texttt{true}}]\!], [\![b_{\texttt{false}}]\!])_{\text{CONDITIONAL}}
\end{array}
$$

Conditionals may or may not have an alternative branch (a branch to execute when the condition is not satisfied). The translation of a conditional contains the translations of the condition and any branches that are present.

$[\![\texttt{access}_{\texttt{read}}]\!]$

$$
\begin{array}{lcl}
[\![v]\!] & = & (\texttt{read}, \ell, \textbf{var}(v), \texttt{false})_{\text{ACCESS}} \\
[\![v.length]\!] & = & (\texttt{read}, \ell, \textbf{var}(v), \texttt{true})_{\text{ACCESS}}
\end{array}
$$

$[\![\texttt{access}_{\texttt{write}}]\!]$

$$
[\![v]\!] \quad = \quad (\texttt{write}, \ell, var(v), \texttt{false})_{\text{ACCESS}}
$$

Variable reads and writes are translated similarly, except that it is possible to read the length property of array variables, but not to write to it. An access structure stores a reference to the variable accessed, the type of access and the location. A flag is set to indicate whether the length property was read.

$[\![\texttt{var}]\!]$

$$
\begin{array}{lcl}
[\![t\ id]\!] & = & ([\![t]\!], \ell, id, \Uparrow, \texttt{null})_{\text{VARIABLE}} \\
[\![t\ id = e]\!] & = & ([\![t]\!], \ell, id, \Uparrow, [\![e]\!])_{\text{VARIABLE}} \\
[\![\text{const } t\ id]\!] & = & ([\![t]\!], \ell, id, \Uparrow, \texttt{null})_{\text{CONSTANT}} \\
[\![\text{const } t\ id = e]\!] & = & ([\![t]\!], \ell, id, \Uparrow, [\![e]\!])_{\text{CONSTANT}}
\end{array}
$$

The variable type represents the declaration of a variable. Each variable declaration stores the location and scope of declaration, the type and the name. If a variable has an initialiser, the expression that represents it is stored as well. Constants contain the same information as variables, but are not written to.

$[\![\texttt{fun}]\!]$

$$
[\![t\ id(p)\ b]\!] \quad = \quad (id, \Uparrow, [\![b]\!], \ell, [\![t]\!], [\![p]\!], length([\![p]\!]))_{\text{FUNCTION\_SCOPE}}
$$

$[\![\texttt{formalparams}]\!]$

$$
[\![t\ n, tail]\!] \quad = \quad ([\![t\ n]\!], [\![tail]\!])_{\text{SEQUENCE}}
$$

Function declaration structures store the location of the function, the translation of

the body and the formal parameter list. The number of formal parameters is also stored for later reference. Formal parameter lists are represented as a list of variable declarations in the scope of the function.

$[\![\texttt{return}]\!]$

$$[\![\text{return } e]\!] \quad = \quad (\Uparrow_{\text{FUNCTION\_SCOPE}})_{\text{LEAVE\_FUNCTION}}$$

The leave_function structure, generated by $[\![\texttt{return}]\!]$, represents an exit point from the closes function declaration.

$[\![\texttt{for}]\!]$

$$[\![\text{for}(e_0, e_1, e_2)\ b]\!] \quad = \quad [\![e_0, \text{while}(e_1)\ b + + e_2]\!]$$

$[\![\texttt{while}]\!]$

$$[\![\text{while}(e)\ b]\!] \quad = \quad ([\![\text{if}(e)\{\text{break};\}]\!] + + [\![b]\!])_{\text{LOOP}}$$

Both for and while loops are translated into the single looping construct. The loop exit test is moved within the body of the loop, and loop update (in the case of for loops) is appended to the loop body.

$[\![\texttt{break}]\!]$

$$[\![\text{break}]\!] \quad = \quad (\Uparrow_{\text{LOOP}})_{\text{LEAVE\_LOOP}}$$

The break keyword in Kenya causes the closest loop to be left. There is no way to specify an alternative loop, so the translation simply stores the closest loop construct as the target of the loop exit.

$[\![\texttt{expression}]\!]$

$$[\![e]\!] \quad = \quad (\textbf{typeof}(e), [\![\textbf{effects}(e)]\!])_{\text{EXPRESSION}}$$

An auxiliary function is defined for use in specification of the translation of expressions to simplify its presentation: **effects** is a function that accepts an expression, and returns a sequence containing the variable reads and writes, and the function calls of that expression.

$[\![\texttt{assignment}]\!]$

$$[\![v = e]\!] \quad = \quad (\textbf{var}(v), [\![e]\!])_{\text{ASSIGN}}$$

$[\![\texttt{struct}]\!]$

$$[\![\text{class } id\ b]\!] \quad = \quad (id, \Uparrow, [\![b]\!], \ell)_{\text{COMPOSITE\_SCOPE}}$$

Composite types in Kenya are similar to C structures – they contain only data members. A Kenya class is translated to a type declaration that includes its name, scope, location and the translation of its contents.

Test script

> I approach these questions unwillingly, as they are sore subjects, but no cure can be effected without touching upon and handling them.
>
> Titus Livius (59 BC–7 AD)

You are an employee at a large software contracting firm, who have just obtained a contract that involves maintaining a large piece of legacy software written in Kenya. To your delight, you are given the job of performing the maintenance tasks. The software is known to have a number of bugs, and comes without any documentation.

This test looks at how you tackle a series of problems using this project implementation.

1. Your company is misguided and uses the comments/line ratio as a measure of code quality. As part of your initial assessment of the source code, your manager asks you to find the comments/line ratio for *core.k*

Time taken	
Expected locations	
Comments	

2. You have been informed that there is an optimisation that can be made to the function *solveSubequation()* if it is never called via *normaliseSomething()*. Is it possible to perform this optimisation?

   Note: *called via* should be taken to mean that there is no chain of function calls from an arbitrary point in the program to *solveSubequation()* that

contains *normaliseSomething()*.

Time taken	
Expected locations	
Comments	

3. There is a bug that you believe is caused by an invalid value being assigned to the variable *index* before it is used in *iterateAgain()*. Find where the value could have been assigned to *index*.

Time taken	
Expected locations	
Comments	

4. You have noticed that the functionality provided by *obsoleteCalculation()* is no longer required. Which other parts of the source code can be removed?

Time taken	
Expected locations	
Comments	