A Reverse Engineering Tool for the Analysis and Comprehension of Source Code

Nicola Louzado

BSc (Hons) in Computer Information Systems
University of Bath

May, 2005
A Reverse Engineering Tool for the Analysis and Comprehension of Source Code

Submitted by Nicola Louzado

COPYRIGHT

Attention is drawn to the fact that the copyright of this thesis rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed ………………………………..

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed ………………………………..
Abstract
Computer Aided Software Engineering technologies can support the lifecycle of design and implementation from end-to-end. Their all-embracing nature is appropriate for large-scale projects but imposes a significant overhead for more compact developments. This project examines CASE technology in the context of undergraduate computer science final year projects, and reports a reverse engineering system. It generates relevant system abstractions from components and dependencies as they exist in source code. The system is intended for use by someone other than the original developer but here primary users are both student developers and their project supervisors. It aids the comprehension of Java source code by graphically exposing its abstract structure alongside a structured parsed form of the code text itself, together with an annotation facility for the abstractions it creates. Evaluation suggests scope for further work including links between the textual and graphical representation with, for example, colour and direct manipulation to direct attention to relevant information.
Acknowledgements

I would like to thank my supervisor Dr Leon Watts firstly, for his willingness to take on the project mid way and for all of his support and continuous enthusiasm. I also thank Dr Carsten Fuhrmann for the initial project idea and his help during the early stages. I wish to thank Jeni Prout for proof reading and her constant support throughout the year.
Contents

1. Project Overview ........................................................................................................... 7
  1.1 Problem Description ............................................................................................... 7
  1.2 Chapter Overview ................................................................................................. 8

2. Literature Review: CASE Technology, Program Understanding and Reverse Engineering ................................................................................. 10
  2.1 Software Engineering Processes .......................................................................... 10
  2.2 CASE tools .......................................................................................................... 11
  2.3 CASE tool usage .................................................................................................. 16
  2.4 Program Comprehension ...................................................................................... 18
  2.5 Reverse Engineering ............................................................................................ 19
  2.6 Object – Oriented Programming ......................................................................... 22
  2.7 Software Visualisation ......................................................................................... 22
  2.8 Conclusion ............................................................................................................. 26

3. Requirements Analysis and Specification ......................................................... 28
  3.1 User Profile ........................................................................................................... 28
    3.1.1 Who are the users of the system? ................................................................. 28
    3.1.2 What are the needs of the system users? ...................................................... 28
  3.2 Requirements Specification ................................................................................ 29
    3.2.1 Format of the requirements: ................................................................. 30
    3.2.2 FUNCTIONAL REQUIREMENTS ......................................................... 30
    3.2.3 NON-FUNCTIONAL REQUIREMENTS .................................................. 32

4. Design ......................................................................................................................... 34
  4.1 Scenarios .............................................................................................................. 34
    4.1.1 Scenario 1 ........................................................................................................ 34
    4.1.2 Scenario 2 ....................................................................................................... 35
    4.1.3 Scenario 3 ....................................................................................................... 36
  4.2 High Level Design Decisions ............................................................................. 36
    Design Focus ............................................................................................................ 36
    4.2.1 Limitations .................................................................................................... 37
  4.2.2 Data Structures ............................................................................................... 38
  4.3 Modules Overview .............................................................................................. 39
    4.3.1 Source code parser: .................................................................................... 39
    4.3.2 Textual summary parser: .............................................................................. 40
    4.3.3 Layout Manager: ......................................................................................... 40

5. User Interface Design ............................................................................................ 42
  5.1 Object-Oriented Program Inspection Tasks .................................................... 42
  5.2 User input and output .......................................................................................... 43
  5.3 User Studies ......................................................................................................... 43
    5.3.1 Pilot Study .................................................................................................... 44
    5.3.2 User Study Plan ............................................................................................ 45
1. Project Overview

1.1 Problem Description
CASE stands for Computer Aided Software Engineering. Software engineering is a discipline which involves all aspects of software production; specification, design, implementation, validation and evolution. These activities are conducted according to a form of software process model such as the ‘waterfall model’ or ‘evolutionary development’. Process models make design and development activities systematic and contribute to the management of software engineering projects. CASE tools include a wide variety of programs which support software process activities. They aim to help the software engineer to organise and carry out activities and can also produce documentation and the code outline.

Typical student projects are a specific kind of software project whose development follows well-known patterns when compared to the commercial software development situation. It is an individual project, without the overhead of team coordination and delegation. It typically has an exploratory nature without contractual obligations or a specific client for whom to create a software solution. The exploratory nature creates uncertainty and a continual need to reflect. The final document produced is a dissertation with certain standard chapters, the deadline is always known from the start and an integral part of the process involves regular meetings with a supervisor.

Software development consists of three main areas; conceptualisation, representation and implementation. This project is most concerned with the area of representation which is the software documentation. Representation is important in all software development to enable maintenance and further development. It is particularly valuable to the success of student projects for the following two reasons. The final assessment and grade is mainly derived from the write up of the project: the project dissertation. Also, the documentation is a way of conveying ideas and findings to the project supervisor. This communication is very important as the supervisor needs to have a good understanding of each stage of the project to be able to provide comments and suggestions for further work. During the design and implementation the supervisor will be presented with code several times. He/she is expected to understand it although they will have little time to analyse it. It is likely that the meeting with the student is the first time that they will see the code. Also they may not have a very good understanding of the implementation domain as there are many routes that the student may take.

Reverse engineering is defined as ‘analysing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information’ (Chikofsky and Cross, 1990). The aim of this project is to design and implement a tool for student projects which can produce simple documentation by reverse engineering of source code. The system will have two main uses:

1. To aid the communication between computer science students and their project supervisors. The summaries produced by the system could be used to aid explanations and discussions in project meetings.
2. To help students understand the structure of their own source code.
1.2 Chapter Overview

Chapter II: Literature Review: CASE Technology, Program Understanding and Reverse Engineering
This chapter reviews the literature on CASE tools to establish a specific area within CASE technology which could be useful in the context of student projects. The review covers literature in the following areas:
- the software engineering process of student projects.
- the main difficulties faced by students and their supervisors.
- the suitability of CASE tool usage in student projects.
- the range and functionality of current Integrated Development Environments (IDEs) available and the different support they provide.
- the techniques used in program understanding.
- the use and method of reverse engineering.
- the main concepts of object-oriented programming.
- the techniques of software visualisation.

Chapter III: Requirements Analysis and Specification
In the requirements analysis, user profiles are developed to identify the needs of the user by understanding as much as possible about them, their work, and the context of their work. The identified needs are then developed into a set of initial requirements which are used as a basis for the design. The requirements are refined and built upon during the design stage until a final requirements specification is produced.

Chapter IV: Design
The initial requirements are developed into a conceptual design for a reverse engineering and code comprehension tool. This chapter describes the key decisions made in the design stage of the project including the subset of features in the conceptual design to be implemented, and the design the main data structure. The structure of the system is described in terms of each module followed by an explanation of other remaining design decisions. Informal event scenarios are used to clarify the specific uses of the system.

Chapter V: User Interface Design
The user interface design is developed iteratively through design, evaluation and redesign involving users. This chapter describes the prototyping activities through which the initial designs were developed and how the user studies were carried out. The user studies involved semi-structured interviews with computer science students and project supervisors. It describes the nature of the feedback gained from these activities and how the initial designs were refined and improved.

Chapter VI: Implementation
This chapter describes how the system is implemented. It discusses important choices of implementation techniques, the methods used and any significant problems that were encountered.

Chapter VII: System Testing
The testing strategy used to test and evaluate the implementation of the system is describe in this chapter. Functional testing is carried out to check how well the system conforms to the
final requirement specification. Structural testing is used to test the system output. Scenarios are carried out to evaluate the overall performance of the system. User tests are carried out to gain user feedback and evaluate the usability of the system. The results are discussed highlighting any areas of interest.

**Chapter VIII: Conclusions**
The implemented system meets the critical and essential requirements defined for the system given the constraints on the prototype that was implemented. It also meets some desirable features. This chapter evaluates the success of the implemented system and describes a number of additions and improvements that could be made.

**Chapter IX: Bibliography**
A list of citations used during the project.

**Chapter X: Appendix**
Supplementary project material. This includes paper prototypes, diagrams, class descriptions, screen shots, system tests, test data and the source code of the implemented system.
2. Literature Review: CASE Technology, Program Understanding and Reverse Engineering

The aim of this literature review is to gain a better knowledge of the subject of CASE tools and to find a more specific area within CASE technology which could be useful in the context of student projects. Relevant literature is analysed and evaluated to identify key concepts and ideas that relate to the project.

The review begins with literature in the area of software engineering processes to establish any trends which are specific to student projects. A set of CASE tools are tested and reviewed to discover the features considered to be important in CASE technology, followed by a discussion on the reasons for the low usage of current CASE tools. The review then covers the issues of program comprehension, reverse engineering, object-oriented programming and software visualisation which identify and focus on an area of interest for the project. The conclusion of the review discusses a possible system which could be supportive in the software process of students project.

2.1 Software Engineering Processes

The basic elements of software development are project conceptualisation, representation and implementation. Conceptualisation can be thought of as a method of organising the developers’ thoughts and discussions of the project (Stiller and LeBlanc, 2002). Documenting the conceptualisation is the project representation. In student projects the representation is a very important element for two reasons. Firstly it is the basis of the project assessment and therefore the success of the project depends on it. Secondly, it is a way of communicating project ideas from the student to the supervisor and vice versa. Constant communication between the student and supervisor is vital to enable the project to progress to its full potential. Both textual and graphical documentation are both important to representation and must be unambiguous. It must be interpreted in the same way by the writer and the reader. This can be aided by using specific formal notation. The information needs to be unaffected by background knowledge. Stiller and LeBlanc (2002) say a fundamental objective is that the viewers’ knowledge does not affect the interpretation of the information. A more important factor seems to be that the writer does not make unfounded assumptions about the viewers’ knowledge. The representation must mean the same thing to both a domain expert and a technical expert.

Activities common to all software processes are software specification, design, implementation, validation and evolution.

Agile software development methods are intended to reduce the complexity of software engineering. Agile software development follows four main principles (Abrahamsson et al, 2002):

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer Collaboration over contract negotiation
4. Responding to change over following a plan
The first principle of the agile movement focuses on the relationships and communications of the software team. In student projects this can be applied to the constant feedback required between the student and the supervisor. Secondly, agile methods are constantly producing new releases of working software. Minimal documentation is required by using simple, straightforward code. This is less applicable to student projects as documentation is very important and is usually what the project is assessed on. The third rule is also not applicable as there are no customers involved. However a good relationship needs to be maintained between the student and supervisor. The fourth rule refers to being open to changing requirements and implementation methods. The method ‘welcomes changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage’ (Beck et al, 2001). This is very relevant to student projects as they have an exploratory nature and new requirements and better methods may be discovered half way through the project. Flexibility is needed to allow for potentially better solutions.

Process iteration has been identified as the general process most applicable to student projects. Sommerville (2001) describes this as a hybrid model of other software process models. He discusses two types of process iteration: incremental development and spiral development. Mills et al (1980) first suggested the incremental approach as a means of delaying decisions and detailed requirements until more experience of the system is gained. This means that time spent on reworking earlier decisions is reduced. Student projects involve continuous learning throughout the project life cycle as it is often a new and unknown subject to the student. Incremental development seems to be the closest model to the software process of student projects.

The student and supervisor outline the services to be provided by the system. The student can then identify which of the services are most important. There may be more than one possible project end point in the project plan. Several delivery increments can then be defined, some vital to the project and others that can be thought of as further work if the project does progress that far. The most important increments are given highest priority and are worked on first. The same process does not necessarily have to be used for each increment. This model means that there is lower risk for overall project failure as even if there are problems with certain increments, it is likely that some parts will be successfully completed.

2.2 CASE tools

Computer Aided Software Engineering is a broad term that covers any program that supports software process activities such as requirements analysis, system modelling, debugging and
Tools which support the early stages of the software process such as analysis and design are known as upper-CASE tools, whereas tools focusing on implementation and testing are named lower-CASE tools. If a tool supports all stages it is called a CASE environment. CASE tools can aid planning, editing, documentation, change management, prototyping, program analysis and debugging. This section reviews a set of current CASE tools to expose the range of features considered to be important and what they try to achieve for software developers.

QSEE SuperLite (2001-2004) is one CASE tool that was analysed. It is a modelling environment and is known as a multi CASE tool which supports many tasks. First impressions of the tool were that it has a large range of functionality which can seem very confusing to a new or rare user. However the help pages are displayed automatically when a new project is begun which reduces confusion. There is also a ‘context sensitive help’ which when requested, displays the help document on whichever task is being carried out at the time.

To get a feel for the tool, it was used to create an entity-relationship diagram. The diagram was fairly intuitive to create with the vast functionality being the only aspect which slowed the process down. The tool includes a code generator which creates SQL code from the diagram. This function is very useful, creating whatever code possible from relatively simple diagrams. It then states the tables that could not be created due to missing information. Html pages can be created with information on each entity and each relationship. Although this is a nice feature, it seems to be partly redundant as it only repeats information from the original diagram. The resulting diagram is fairly untidy and the aesthetics could be improved upon.

Figure 2.2 shows a screen shot of the entity relationship diagram created using the tool. It also gives a view of the possible functionality of this feature.
The tool can perform very useful model checks which give warnings about missing information. Another good feature which supports group work is the ability to email projects straight from the tool. The UML model is very comprehensive and includes models which support classes, use cases, deployment, components, sequences, collaboration, state charts and activities. For the purpose of student projects this is unnecessary and more features than a single project would ever use. The brainstorming tool is a nice feature which allows the creation of very simple, clear brainstorms. This provides a good way of writing up individual and group brainstorms. The tool also offers reverse engineering, creating a model from an existing database. It uses Open Database Connectivity (ODBC) to connect to and extract from a database.

Visual Paradigm for UML (2004) seems a very comprehensive tool. Even during installation the user is asked about their customisation preferences for the look and feel of the tool. The community edition is discussed in this literature review, however the professional edition allows the option for integration with Eclipse, JBuilder, NetBeans, IntelliJ Idea, JDeveloper and WebLogic workshop. This capability is useful, allowing more scope for support of the whole project. The graphical user interface is aesthetically pleasing which makes the vast functionality less intimidating. The tool can be used to create the following diagrams: use case, class, sequence, collaboration, state, activity, component, deployment, CRC card, composite structure, timing, textual analysis and business workflow.

A use case diagram was created in order to test the tool. The diagram function has many customisation options, including a choice of colours for the fill colour of the lines and components, and choices for the shape of connection points for relationships. At first the diagram was fairly intuitive to create, however it was difficult to add a relationship between a use case and an existing actor. The usual method of creating a relationship seemed to be to create an actor or a use case at the same time as the relationship. The help document was slightly frustrating as it was not very user friendly but after becoming more familiar with it the solution was found. The tool does offer some nice features for diagrams; it prevents the creation of invalid diagrams because it includes continuous syntax checking and can produce html and pdf files which are helpful for project documentation.

Figure 2.3 shows the simple use case diagram that was created using the tool.

![Use case diagram](image)
The other diagrams in the tool all seemed to have good functionality but a lot of time would be needed to learn how to use the tool properly. There is much attention to detail such as the option for anti-aliasing of the text and graphics in the diagrams.

The tool includes the facility for code generation and also reverse engineering. As the tool offers code generation and code reverse engineering, it is called round trip engineering. It also supports real time round trip engineering which means that a change to a diagram is reflected in the code automatically. There is the option for customisation of the code generation function so that to a small extent, the user’s coding style can be reflected. An example of this is the option of where to position the start and end braces for a class, an operation and an inner class. The possible options are at the end of a line, on the next line or indented on the next line. A class diagram was created to test the code generation function. The diagram was extremely quick and easy to create as the tool was already familiar. The code generation did work and produced very simple code using all the information available from the diagram. However unless the diagram was very comprehensive, this function could only be used as a starting point for coding.

BlueJ - The Interactive Java Environment (2001-2004) is a different type of CASE tool. It is a Java development environment aimed at teaching Java at an introductory level. It is a very comprehensive tool with several useful features. The main use of the tool is for Java coding.

The tool has a very simple interface as shown in Figure 2.4. This is an appropriate design as the system does not require a complicated interface, and so it is left uncluttered. This means there is a large space to work with the diagrams.

![Figure 2.4 BlueJ user interface](image)

The design of the tool encourages the development process to be centred around class diagrams. When the user begins a new project, they can draw a class inheritance diagram of the system. The tool then creates a code framework for the diagram that has been drawn. The framework takes information from the diagram and reflects the inheritance relationships of the classes. The code can be accessed, modified and compiled by clicking on the blocks of the diagram. This feature is shown in Figure 2.5.
When a syntax error is found in the code, the system highlights the line where the error has occurred and sets the cursor to the correct place to allow the user to change it. The user can also get further information on compiler error messages. In addition, the system includes a debugging tool which checks the internal states of objects.

Java files that have been created using a different tool can be opened by selecting the ‘Open Non-BlueJ’ option on the ‘Project’ menu. The tool creates an inheritance diagram of the classes contained in the files. This is a useful feature however when given fairly complex code, the layout is not very good and the diagrams can be complicated and cluttered. This may be because the tool is aimed at Java beginners and so does not support large pieces of code. Also the diagrams do not contain any data about the classes which would normally be displayed in a class diagram such as variables and methods. This can only be accessed by clicking on the blocks and viewing the source code. The feature of creating diagrams from existing code is helpful and could be expanded upon in this project. It would also be useful to be able to create different types of diagrams.

The tool appears to be missing an in built help system. Currently the documentation is available separately from the internet and a tutorial is available in PDF format. The lack of an easily accessible help system slows down the use of the system.

Overall, the BlueJ environment seems to be a very good tool and supports the task of Java development well.

JCreator is an integrated development environment for Java. It provides the functionality of project management, templates, class browsers, code completion, debugging, syntax highlighting, wizards and a fully customizable user interface. The tool does not offer functionality to produce diagrams as it is focused on actual coding, however there is a very useful analysis function offering a ‘ClassView’ which displays a tree of the imported libraries, aggregate classes, methods, and variables in the selected class as shown in figure
2.6. This function is valuable in refreshing the user’s memory of the code included in the class.

![Figure 2.6 'ClassView' taken from JCreator](image)

2.3 CASE tool usage

‘Why are case tools dearly bought but sparsely used?’ (Jarzabek, and Huang, 1998).

In theory CASE tools reduce the time and cost of software development and enhance the quality of the resulting systems. However research has shown that few organisations use CASE, many organisations abandon CASE during projects and many developers within organisations do not use CASE (Lending and Chervany, 1998). The people that do use CASE have said that it improves quality but is time consuming and difficult to use. The reasons for abandonment include cost, the lack of measurable returns and unrealistic expectations. CASE tools do not meet the expectations of increased productivity, improved quality, easier maintenance and making software development more enjoyable (Jarzabek and Huang, 1998). Research has shown that management often think that more people in their organisation are using CASE than actually are (Lending and Chervany, 1998). This implies that at a theoretical level, CASE seems like a very good idea but in practice there are reasons why current tools are not well used.

Jarzabek and Huang (1998) ask the question: ‘Do people who use a CASE tools use the same methodologies that systems developers who do not use a CASE tool use?’.

They found that 75% of those who use CASE use the life cycle methodology compared to only 56% of non-CASE users. This is a significant difference and suggests that CASE tools are somewhat restrictive in the process that they support. They also found that few advanced CASE features are used. One developer said that her main use of CASE was as a communication tool between systems developers. This is relevant to this project because as well as aiding communication in group work, a CASE tool could be used as one form of communication between a student and their supervisor.

Companies need to spend approximately one dollar on training for every dollar invested in a CASE tool (Daneva, 1999). This is a huge amount and reinforces the idea that current case
tools are too complex and have steep learning curve. The time period until the investment begins to pay off can be very long and therefore the worth of the tool would be questioned.

Daneva (1999) discusses a best practice approach to CASE tool selection. Best practice in terms of software is a means of supporting software improvement goals. The use of CASE tools are considered to be one of the most important ways of implementing best practices by process optimisation and by increasing the scope and means of improvement.

Software benchmarking is ‘the process of continually searching for the best practice experience, indicating software standards to be achieved in day-to-day company operations, adopting or adapting the best in class practices and implementing them to become the best’ (Daneva, 1999). Daneva goes on to discuss the best practices for CASE tool selection. The characteristics that she identifies as qualities that a CASE tool should possess are: pertinence - achieving the overall goal, functional coverage, ease of use and reproducibility of results. These are all very important factors and are all relevant to a CASE tool for student projects. Daneva put together a benchmarking set for the functionality of CASE tools which consists of:

- Teamwork
- Analysis Support
- Enterprise Modelling Capabilities
- Implementation
- Documentation Development
- Design Support
- Software Estimation and Controlling
- Configuration Management
- Project Management
- Quality Assurance
- Coordination Support
- Project Performance Database

All CASE functionality can be classified under one of these headings and then each core benchmark can be given a weighting. This does not seem to be an effective way to choose and assess a CASE tool as it only assesses functionality and does not look at usability. The result could be a very high rating for a tool with broad functionality but this could mean that the user is overwhelmed and confused by the tool. Psychological tests have shown that human ability to cope with complexity is highly restrictive (Jarzabek and Huang, 1998). Therefore a tool which displays large lists of functionality can confuse the user. Ease of use is very important, otherwise many users will be discouraged and will abandon the tool. As the complexity of a system is a high factor for low usability, even an expert developer gains little from being exposed to many functions at once.

The design of a tool ideally should be based on the behaviour of a software developer making it easy to merge in to the software development process. CASE tools based on a programmers mental model of the project are more user centred, rather than those that are method centred (Jarzabek and Huang, 1998). Current tools are too oriented on software modelling and construction methods. They very rarely support creative thinking, understanding or other ‘soft’ aspects of software development. In the context of exploratory, uncertain, lone developer type projects such as a student project, creative thinking is at a premium. These missing features mean a tool is unable to support the whole development process.
A well-designed user interface should create a metaphor that bridges the conceptual gap between a computer system and human thinking’ (Leonhardt et al, 1995). A CASE environment should allow for problem solving and the expression of ideas. It should support goal, decision, reason, consequence, impact and variant solutions. Some tools do exist which support thinking and idea generation. They do not follow a certain model resulting in very flexible systems, allowing users to do anything at any stage.

The idea of an active tool is one that guides the user. It provides more support to novices than to experienced users and can suggest functions that are relevant to a particular task. To some extent the context sensitive help function on the QSEE tool is an active function. It offers help which is useful to the task being carried out. Active tools can also check that software models are consistent with one another, however it can be useful for a tool to tolerate some inconsistencies as this allows more flexibility in the design. Consistency checks are generally a complex task.

Neural networks are an overlap of artificial intelligence and approximation algorithms. The algorithm for a neural network is based on the human brain and how it processes information. When used in a CASE tool, the tool can learn from its environment and find non-evident dependencies and relationships. It can be self-learning, self-improving and can work with incomplete information. Active tools use neural networks. They have a certain knowledge base of standard solutions to well-known problems and have the ability to offer them when appropriate. They can identify common design patterns and can offer templates which might be useful if reused. However, do users find second-guessing of their thoughts and next movements useful or annoying? Does it aid progress or slow it down? Like all tools, artificial intelligence is useful when it works correctly but a hindrance if errors occur. Artificial intelligence can have problems working with imperfect data and adapting to change. If a tool fills in gaps in incomplete data with the wrong information, the process is slowed down as the user has to correct the mistakes of the tool. However, when an artificial intelligence tool works correctly it can be very useful and can save time.

2.4 Program Comprehension

A critical hurdle for exploratory projects is evolving an understanding of the nature of the project itself and how this is realised in code. So program comprehension seems to be a vital issue for projects of this kind.

To have a good understanding of a program, one must understand the application domain functionality, the high level architecture and the myriad implementation details. The essence of program understanding is identifying artefacts, discovering relationships and generating abstractions (Tilley and Huang, 2003). It can also be thought of as mental pattern recognition at various abstraction levels. Program understanding is very complex as several different programming languages may be used, there may be a complex third-party infrastructure and all systems are heterogeneous (Tilley and Huang, 2003). Also as the system is developed, there may be several ‘hacks’ added to the code which can stray from the program structure making it difficult to understand.

Program understanding is a relevant topic to student projects because the supervisor may be presented with the design and source code of the program several times during the course of
the project and is expected to understand it and be able to make comments after little inspection or analysis.

‘Maintenance programmers spend almost half of their time analysing code for general comprehension’ (Jarzabek and Huang, 1998). This is a large amount of time and a project supervisor has very little time to spend on understanding a student’s code. Normally the first time that the supervisor will have a chance to look at a piece of code that needs to be understood will be in the meeting with the student.

Koeneman and Robertson (1991) carried out an experiment to identify the methods used by software developers to achieve program understanding. They showed that programmers generally use two approaches to program comprehension; an as-needed approach or a systematic strategy. The as–needed approach is also known as bottom–up comprehension where only localized, directly relevant code is looked at. It can mean that important aspects are missed and the understanding achieved can be insufficient. This is the general method that a project supervisor is forced to use unless there is other documentation available which summarizes the code. Systematic understanding involves trying to get a deeper understanding of the program by looking at data and control flow execution between subroutines. In the experiment, most subjects used an ‘as-needed’ approach but did not manage to gain an overall understanding of the program. A certain level of overall understanding was attained to enable them to search for specifically relevant code. It was found that the use of relevant variable and procedure names aided understanding greatly and meant that subjects could guess the names of procedures that they were looking for. Searching through code is very time consuming and can be reduced by using simple diagrams to represent and summarize the code. Line numbers relating to the parts of the source code that a diagram represents would also be helpful. After the experiment, participants gave feedback on their actions. When subjects were finding it difficult to understand a piece of code they needed to switch between different documentation to find further information. The combination of different types of data and different representations aided understanding greatly. However, it was also found that apart from flow charts, documentation was only used as a last resort when the source code did not provide all the information needed.

The approach to program understanding depends on ones familiarity with the application domain, cognitive preferences and abilities, and the support facilities provided by the environment (Tilley and Huang, 2003). A project supervisor will generally have a good knowledge of the application domain because they would have agreed on the project topic. However they may not necessarily be very familiar with the implementation domain as there are many different routes that a student can take.

Tilley and Huang (2003) mention three groups of support mechanisms for program understanding: unaided browsing, leverage corporate knowledge and experience, and computer aided techniques such as reverse engineering.

2.5 Reverse Engineering

Reverse engineering is a method of extracting high level information from low-level artefacts and is defined as ‘analysing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information’ (Chikofsky and Cross, 1990). It is most widely used in the process of understanding legacy
systems to produce different abstractions to aid understanding for the tasks of maintenance, modifications and re-use. It can produce functional descriptions of architecture, object based designs and documentation. In the case of a student project, reverse engineering could be a good aid to communication to increase the supervisor’s knowledge of the system.

The three main components of a reverse engineering tool are the parser, the repository and the visualisation engine. A parser is a program that takes a set of sentences as input and identifies the structure of the sentences according to a given grammar. In the case of a reverse engineering tool, it would recognize important parts of the source code and discard any lines which are no longer needed. The repository is a store of items needed to perform the task. The visualisation engine is used to transform the data collected into a useful format, usually a graphical representation of the system.

Reverse engineering research by Muller et al (2000) has shown that there are several ways of understanding code:

- subsystem decomposition,
- concept synthesis,
- design, program and change pattern matching,
- program slicing and dicing,
- analysis of static and dynamic dependencies,
- object-oriented metrics,
- software exploration,
- visualisation.

One problem is that source code does not hold all information about a program. Gaps can occur due to a lack of information about the architecture, design trade-offs, engineering constraints and the application domain. This is not a huge problem in student projects as the system is designed and implemented in one go and the student is usually at hand to answer any further questions.

Problems can occur when reverse engineering is used in a ‘big bang’ attempt to regain information. It is important to carry out reverse engineering in increments as much human input is needed to make the information useful (Muller et al, 2000). Continuous program understanding is very important to have effective comprehension. Given a design module, it should be possible to identify the source code that goes with it and vice versa. This is known as forward and backward traceability. It is important to maintain mappings between different levels of abstraction (Muller et al, 2000).

Code reverse engineering focuses on aiding human understanding of how information is processed. Data reverse engineering looks at what information is stored and how it can be used for different purposes in a different context. It is a good indicator of the quality of the software as the data held in a system is vital to the end goals. Design flaws in the data structures is an indicator of a poorly implemented system.

Reverse engineering of the main data structure using a database management system is more specifically known as database reverse engineering. There are two main activities in database reverse engineering: data analysis and conceptual abstraction. The aim of data analysis is to recover an up-to-date data model that is structurally complete and semantically annotated (Muller et al, 2000). However the information that is recovered is always incomplete and so
the process does need human input and can never be fully automated. Another problem is that there are many different possible hardware and software platforms available and it is difficult to make a tool that accounts for all combinations. Despite these problems, reverse engineering can significantly reduce the time and effort spent on this phase. Conceptual abstraction aims to map the logical data model from the analysis, to the equivalent conceptual design. It normally uses entity-relationship diagrams or an object-oriented model (Muller et al, 2000). Most tools are of limited use for the following reasons. The diagrams produced from the analysis need to be added to and altered as new knowledge is gained about the system and current tools do not support this iteration process. Using the data model to form a data abstraction is a bottom-up data abstraction process. In reality there is usually some form of pre-existing design of the data structure available which would allow for a top down approach. Ideally a hybrid bottom up / top down process is required. This is known as a bi-directional mapping process which is not possible with most current systems (Muller et al, 2000).

The ‘InSight’ Reverse Engineer tool as described by Rajala et al (1999), has several interesting features. It can extract ‘indirect relationships’ as well as direct and also process-function relationships which are used for a dynamic structure diagram. The tool can create two sub-views of a system. The first is a hierarchy view in the form of a tree diagram which shows all containment arrangements. The second is an architecture view which shows each level of architecture separately. This view clearly shows relationships but hides containment details. InSight also has features for code inspections and a user support environment which is an online documentation tool. This supports text and basic graphics, links to web resources and has the potential to include animations, videos and sound clips.

The most relevant feature of the InSight tool seems to be the ‘Insight flowchart and server editor’. It displays flowcharts with all decision points and paths in the system. It can produce a chart of the general flow through the system. These views would be very useful for a project supervisor as a quick and simple overview of the system.

Reverse engineering is not commonly used for several reasons. There is a lack of awareness of the capabilities of reverse engineering tools. People also lack the skills and foundational knowledge of program analysis techniques needed to use the tools properly. Also there is not enough integration between reverse engineering tools and more common software tools. Short projects such as a student project, lack the time to learn how to use a new tool or to move to a new tool mid-way through. When a long training period is required this is a strong disincentive. These are all issues facing people in industry as well as students. For student projects, it should be possible to produce a tool which is fairly simple to use, a tool which produces simplified source code representations which can be given to a supervisor to aid communication and can also be used for documentation in the project write-up. Keeping functionality to a minimum with only the most important features included would decrease the amount of training and skills needed.

The idea of ‘good enough’ or ‘just in time’ understanding (Muller et al, 2000) when using a program is interesting. Research has shown that the 80/20 rule seems to apply to patterns followed when using a program. 80% of the time only 20% of a tool’s capabilities are used. The 20% of capabilities is the most commonly used functionality. If the most frequently used features of reverse engineering programs were integrated into other tools perhaps the use of
reverse engineering would increase. The QSEE tool adopts this concept and does have a simple reverse engineering feature incorporated into it.

Evaluating reverse engineering tools is challenging as it is difficult to measure how much a user’s understanding has improved. The techniques used are user studies, field observations, case studies and surveys. Testing on the ‘InSight’ Reverse Engineer tool showed that the training period for new designers was shortened and they felt it was a better way of understanding programs (Rajala et al, 1999).

‘Reverse Engineering: A Roadmap’ (Muller et al, 2000) mentions that ‘the most critical issue for the next decade is to teach students about software evolution’. This will increase the knowledge and skills involved in analysis, maintenance and evolution and therefore increase the awareness of reverse engineering technologies.

2.6 Object – Oriented Programming

Object-oriented programming is the main paradigm for contemporary software engineering. Object-oriented languages require the construction and association of several software components or objects. ‘An object is a software bundle of related variables and methods. Software objects are often used to model real-world objects you find in everyday life’ (Sun Microsystems, 1995-2005). A class defines variables and methods that are common to a certain type of object. Two key technologies of object-oriented programming are inheritance and aggregation.

Deitel and Deital (1999) describe inheritance as ‘a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors and embellishing these with capabilities the new classes require’. It is a technique of dealing with software complexity that allows the reuse of proven and debugged high quality software, saving time and reducing problems when the system comes into use. When writing a new class, the programmer can specify that the new class inherits the variables and methods of a previously defined class. Inheritance is known as an ‘is-a’ relationship. For example, a circle is a round shape and so has all the attributes of a round shape as well as some new ones of its own. Java does not support multiple inheritance (like C++), meaning that a class can not inherit from more than one other class. Inheritance is specified using the keyword ‘extends’. Java does support the concept of interfaces which allows many of the advantages of the multiple inheritance without the complications. The keyword ‘implements’ is used in the class declaration to specify that the class implements an interface. The class must then define every method from the interface.

Aggregation is the notion of an aggregate object being made up of other components and the components being part of the aggregate. This is known as a ‘has-a’ relationship, for example a car has a steering wheel. In the source code, this occurs when one class is specified within another class. There is no explicit keyword for aggregation.

2.7 Software Visualisation

Software Visualisation is the use of graphics and animation to represent computer programs, processes and algorithms. It tries to ‘find simplicity in a complex artefact (e.g., thousand-line
code), to produce a selective representation of a complex abstraction’. (Petre et al, 1998). One way to find this ‘simplicity’ is to produce a representation that allows the user to see to whole system in a single view, and thus see how the components relate to the whole.

Petre et al (1998) discuss the following uses of software visualisation:

- ‘To provide tools for thinking’ by producing supplementary representations which include factors that need to be considered as well as alternative views of the problem. This could also include helping the user to ‘keep track of the relationship of a sub-problem to the whole and to other parts’.
- ‘To provide a focus for communication between experts’.
- ‘To externalise images of thought’, using the visualisation as an external memory.

They also draw attention to the fact that the value of visualisation is subject to individual perception and interpretation skills. In relation to a computer science student project, this is a minor issue as all users are experts in the field and know what to expect, where to look, and what to look for when analysing well-known formats of source code representations.

Research by Petre et al (1998) has shown that programmers at all levels describe graphical representations as:

- ‘richer; providing more information with less clutter and less space,
- providing the ‘gestalt’ effect: providing an overview; making structure more visible; clearer,
- having a higher level of abstraction,
- a closer mapping to the problem domain,
- more accessible; easier to understand; faster to grasp,
- more comprehensible,
- more memorable,
- more fun
- ‘non-symbolic’; more formal’.

It appears that a graphical representation makes structures and relationships more accessible by displaying the relationships visually or spatially. This makes it easier to construct a mental model.

The Unified Modelling Language (UML) is a notation used to create a number of graphical diagrams. It can represent the system from several different perspectives at various levels of specificity (Stiller and LeBlanc, 2002). It allows representation in an abstract or progressively more detailed manner (as understanding develops). UML has become ‘the “de facto” standard for modelling modern object-oriented applications’(Tilley and Huang, 2003). In forward engineering the modelling process parallels the progress of understanding and therefore helps to control complexity. The diagrams can be thought of as a ‘blueprint for implementing the system’ (Stiller and LeBlanc, 2002). The diagrams are unambiguous and represent key artefacts using ‘things’, ‘relationships’ and ‘diagrams’. UML uses an overly simplistic representation, as more formal notation has been found to cause barriers to understanding for some readers (Tilley and Huang, 2003). The meaning of the diagrammatic building blocks is generally agreed and UML is supported by many commercial tools. The most commonly used UML diagrams are class diagrams and use case diagrams.
Class diagrams model one aspect of the system by the composition of its classes. A starting point to create classes is to make a list of all nouns from the requirements specification (Coad and Yourden, 1990). When the system is in use, the class definitions are static, therefore a class diagram is a static perspective of the system. It shows elements which are part of the class and also the relationships between the classes. The ‘building blocks’ for class diagrams are classes, interfaces, relationships and collaborations. The attributes and methods of a class may not be known when a class diagram is initially drawn but can be added later. The same class may appear in several different diagrams. An interface specifies when classes interact with each other and helps to ensure object oriented encapsulation. In forward engineering they are normally added to the diagrams in the design phase. There are three different types of relationship which can be modelled: associations, generalisations and dependencies. An association is a labelled relationship between two classes. This is demonstrated in figure 2.7 where a patron is associated with a resource by the ‘checks out’ relationship. A generalisation is a hierarchical relationship where one class is the parent of the other class. For example a lecturer is a particular type of university staff. This is also known as an inheritance relationship, as described earlier. A dependency relationship occurs when a class depends on another class so that changing one may require changing the other. For example, an online catalogue object is a collection of resource objects. If a resource changes then the online catalogue must change as well (Stiller and LeBlanc, 2002). The relationships are represented using different types of arrows. Elements called collaborations are used to help explain the system behaviour and specify when two elements must interact with each other to produce a certain action. Collaborations have a structural component and a behavioural component. Only the structural part is shown in a class diagram, represented by a dashed-line eclipse around the elements involved.

Figure 2.7 shows a simple class diagram for a library ‘CheckOutResource Collaboration’ taken from Stiller and LeBlanc (2002).

![Class diagram modelled using QSEE (2001 – 2004)](image)

Fig 2.7 Class diagram modelled using QSEE (2001 – 2004)

A use case is a description of how the system would behave in a particular situation. Figure 2.3 shown earlier in this chapter is an example of a simple use case diagram created using Visual Paradigm for UML (2004). Use case diagrams show how different behaviours interact.
with external actors and also with each other (Stiller and LeBlanc, 2002). The same relationships are represented as in class diagrams: dependency, association and generalization. Use case diagrams also show use cases (represented in an oval bubble) and actors (represented as stick men).

An experiment to determine how much a set of UML diagrams (using UML 1.0) could inform a developer about an existing system shows UML to have the following areas of limitation (Tilley and Huang, 2003): syntax and semantics, spatial layout and domain knowledge.

Some syntax and semantics for the advanced features are not clearly specified. In reverse engineering different tools may model the same thing in different ways. The problem is accentuated because the information may be incomplete. Syntactic ambiguity can lead to misunderstanding. Semantics can cause a problem for users as it takes a long time to learn complete UML. This means that reverse engineering tools must produce the appropriate diagrams for the targeted users to enable understanding. The experiment showed that even experienced users with a good knowledge of UML found it difficult to tell if errors were real system errors or if they were interpreting the diagram incorrectly.

The spatial layout of a diagram is very important for ease of use. The physical layout of the spatial relationships between entities play a role in understanding. Layout engines can be used to produce diagrams, however this does not work on saved data which may not have the corresponding layout information available. Also, some UML diagrams are fairly information dense and do not compress well when printed.

UML is suited to modelling high-level design, some low-level design, implementation guidelines and dynamic information related to message passing and method invocation. This is all application knowledge. Some domain knowledge is necessary to understand a system. Knowledge of the implementation domain and the development and deployment infrastructure is also needed. This information cannot easily be shown with any form of graphical representation, including UML. Also reverse engineering tools cannot extract this information as it is not available in the main body of source code.

‘A Lightweight Web-Based CASE Tool for UML Class Diagrams’ (Rajala et al, 1999) discusses a useful and well designed tool named NutCASE. It is called a lightweight tool because it does not have the vast functionality of heavyweight UML tools. This is to avoid usability problems caused by too much functionality. The designers believe that functionality should not restrict users or clutter the interface; using a tool ‘should be as easy and productive as pen and paper’ (Rajala et al, 1999). A lightweight tool supports the quick and easy production of models. It has the core features of a heavyweight tool ‘without the overheads’ (Rajala et al, 1999). A simple tool of this kind with only core functionality would be well suited to a student project.

The functionality of ‘NutCASE’ includes:
- creation, deletion, editing and searching classes.
- packaging classes so that certain groups can be displayed at one time
- reading in classes from a file.
The article goes on to discuss the advantages that the tool has over pen and paper. Editing diagrams is simple and non-disruptive, classes are laid out automatically, there are search facilities and the possibility of hyperlinks. These points are all real advantages and some tools are certainly more productive than pen and paper. For example the text editor used to write this document saves time by allowing easy deletion and rearrangement of text and the final product is of a higher quality than could be achieved with pen and paper. NutCASE seems to achieve these advantages better than other CASE tools because of its simplicity. A simple tool can bring these advantages without needing long training periods which counter the benefits that the tool brings.

2.8 Conclusion

The area of software development that this project is most concerned with is representation, which is the software documentation. Representation is particularly important in student projects for two reasons. The final grade given is mainly derived from the write up of the project: the project dissertation. Also, the documentation is a way of conveying ideas and findings to the project supervisor. This communication is very important as the supervisor needs to have a good understanding of each stage of the project to be able to provide comments and suggestions for further work. During the design and implementation the supervisor may be presented with code several times. He / she is expected to understand it although they will have little time to spend analysing it. It is likely that the meeting with the student is the first time that he / she will see the code. Also they may not have a very good understanding of the chosen implementation domain as student projects are exploratory and therefore have many possible routes.

Having reviewed existing literature, an area of interest has been found for this project. Reverse engineering is defined as “analysing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information” (Chikofsky and Cross, 1990). A useful tool for student projects would be one that can produce simple, clear documentation by reverse engineering of source code. The problem of missing information in reverse engineering, such as a lack of details of architecture, design trade-offs, engineering constraints and the application domain is not a problem as the student is at hand to discuss these topics and to annotate the resulting documentation. This is usually not the case when trying to understand legacy systems which is currently the main use of reverse engineering. The system would aim to produce a software visualisation which tries to ‘find simplicity in a complex artefact (e.g., thousand-line code)’ (Petre et al, 1998). Diagrams produced by reverse engineering could be used to provide focus for communication between students and their project supervisors. Visualisations of the code could facilitate discussions of the project and the implementation domain and could also be used in the project write up.

Existing systems in a similar domain do not satisfy the specific requirements for a tool to aid student projects, and the documentation produced is not always satisfactory. The tool must be easy to use and only include core functionality. Students projects have a relatively short time scale and so time can not be wasted on learning how the tool works. None of the existing systems produce a summary of all the source code classes at once, as well of a representation of the code structure. Figure 2.8 is a diagram explaining the function of the tool.
A grammar for the chosen language e.g. Java, will be used to create a parser using a parser generator, for example the ‘CUP Parser Generator for Java’ (Hudson et al, 1999). The parser will be implemented so that it takes Java source code and outputs a simplified form of the code. This parsed, textual summary could consist of the names of all the classes, methods and variables in the code. To allow a selective representation, the user will then be able to delete any data that is irrelevant to a desired diagram. This will allow the users to control the focus of the visualisation. A layout manager will then be implemented to use the textual summary to produce a diagram of the system. This will enable a summary of the code to be viewed in conjunction with a representation of the code structure.

![Diagram](image)

*Figure 2.8 System idea for a CASE tool for student projects*
3. Requirements Analysis and Specification

The purpose of this chapter is to establish an initial set of requirements for a source code comprehension tool. The design chapters build on these requirements, iteratively refining them until a final requirements specification is produced in section 5.4. The process of requirements analysis is to build an understanding of what the system should do. In this chapter a user profile is developed as it is essential to understand who the users of the system are, and what they want to achieve. This is followed by an initial set of requirements which can be used to begin the process of design.

3.1 User Profile

Before establishing requirements, the needs of the users must be identified. It is important to understand the users, their work and the context of their work to enable the development of a system to support them in achieving their goals. A user profile is a collection of attributes of a ‘typical user’ (Preece et al 2002).

3.1.1 Who are the users of the system?

Using categories defined by Eason (1987), the different types of user are identified:

Primary users (Users who are likely to be frequent hands-on users of the system, who interact directly with the system):
1. Final year computer science students – This set of users will use the system mainly as an aid when discussing and explaining their source code to others. They may also use it to aid their own understanding of the code or to help them when documenting the code.

2. Computer science project supervisors – This set of users will use the system to aid their understanding of code. The system will provide a summary of the code so that they are familiar with it and able to discuss it and advise students during project meetings.

Secondary users (Occasional users or those who use the system through an intermediary):
1. Computer science project supervisors – Some supervisors may not use the system directly but will use diagrams that their students produce using the system. The diagrams could be used as discussion aids during project meetings.

3.1.2 What are the needs of the system users?

The following questions need to be answered in order gain an understanding on the needs of the users of the system.

What are the characteristics and capabilities of the user?
All users of the system will be assumed to be capable computer users due to their expertise in computer science. However students are likely to be first time users of this particular system as they may not have worked on a large project prior to their final year. Project supervisors may be first time users or may have used the system during previous projects. Both sets of users are likely to have experience using similar systems to complete tasks such as creating
UML diagrams. Both sets of users are unlikely to be willing to spend time learning how to use a new tool. This means that the system needs to be very quick and easy to use.

**What are users of the system trying to achieve?**

**Computer Science Students**
- to explain and discuss their source code
- to gain new ideas for their project
- to refresh their own memory of the content and structure of the code

**Project Supervisors**
- to understand and comment on the work of their students
- to have a summary of the work of the students to use as a reminder before or during project meetings.

**Would they achieve their goals more effectively if they were supported differently?**

A simple system to produce a code summary when required, will be useful to both students and supervisors. As long as the system is easy to use and does not affect the project life cycle, the tool could be an effective communication aid during project meetings and a method of refreshing the users’ memory of the code contents and structure. In terms of their main goals, it is of fundamental importance to be able to cast and recast representations of the system (code and abstractions) during discussion. This is how they build a robust understanding of a problem.

### 3.2 Requirements Specification

The requirements specification is essential in understanding what the system should do. It is an iterative process beginning with an initial set of requirements that are investigated and built upon to become a comprehensive specification.

A requirement is defined as ‘a statement about an intended product that specifies what it should do or how it should perform’ (Preece et al, 2002). The requirements for this system are primarily based on the research in the literature review. They have been written with the aim of taking the greatest features of existing systems combined with new ideas of extra features that would be useful for the specific purpose of this system. The usability problems of existing systems were also kept in mind and avoided where possible.

The main aim of the system is to aid the comprehension of Java source code written by someone other than the user. One specific use is to aid the project supervisor’s comprehension of a student’s source code, thus reducing the time wasted in project meetings trying to remember and understand the code details and structure.

The system will include a parser which takes source code files as input, and extracts and stores the main features of the code. The system will then display a textual summary of the source code and allow the user to delete any details that are not required in their particular, desired summary. A ‘graphical layout manager’ will then take the textual summary and produce a representation of the source code structure in the form of a diagram.
3.2.1 Format of the requirements:

The requirements have been split into functional and non-functional requirements. Functional requirements are what a system should do, describing the functionality or services that the system should provide. Non-functional requirements are not directly concerned with the functionality of the system. They describe the constraints that are on the system.

The non-functional requirements have also been categorised as:

- product requirements – requirements that specify behaviour of the system.
- organisational requirements – requirements from the policies and procedures of the customer’s and developer’s organisation, in this case the University of Bath, Computer Science Department.
- external requirements – requirements derived from factors which are external to the system.

Within these categories the requirements are listed in terms of importance, with the first requirement being the most important function of the system.

The requirements have also been classified as either critical, essential or desirable.

Critical requirements – These requirements are imperative and must met in order to satisfy the main goals of the system.

Essential requirements – These requirements are important to the system and must be satisfied if the system is to be a success.

Desirable requirements – These requirements are not vital to the basic system. They provide extra functionality. Due to the time constraints of the project some of the desirable features may not be captured in the implemented system.

Critical and essential requirements are indicated with a (1) and are distinguishable by an asterisk (1*) highlighting those that are critical. The word ‘shall’ or ‘must’ is used to describe critical and essential requirements. Desirable requirements are indicated with a (2) and the word ‘should’ is used to depict them.

3.2.2 FUNCTIONAL REQUIREMENTS

1. The system shall aid program understanding by abstracting information from the source files.(1*)
   1.1. The system shall aid program understanding for the project supervisor to optimise time usage in project meetings. (1)
   1.2. The system shall aid source code analysis for the student. (1)

2. The system shall take source code as input. (1*)
   2.1. The user shall be able to specify the location of input files containing source code.
   2.2. The system shall import the files and use the data it contains without modifying it in any way. (1)
3. The system shall produce multiple representations of source code as output. (1*).
   3.1. The system shall provide the functionality to display a list of the class names. 
   This list should also include details of inheritance and aggregation. (1)
   3.2. The system should have the functionality to display a list of method names. This list 
   should also include the arguments that the method take. (2)
   3.3. The system shall be able to produce a diagram of the classes and their inheritance 
   and aggregation relationships. (1)
   3.3.1. The diagram should include the variables and methods of the classes to give further information of the inheritance details. (2)

4. The system shall be made up of a parser and a graphical layout manager. (1)
   4.1. The parser shall parse the source code and output a set of stubs. (1)
   4.2. The graphical layout manager shall take the stubs and create a diagram of the system. (1)

5. The system shall allow the user to remove any unwanted classes, methods or variables 
   from the textual summary before it is put into a diagrammatic format. (2)

6. The class diagram shall consist of class blocks. (1)
   6.1. Each diagram shall include a rectangle box for each class mentioned in the textual summary. (1)
   6.2. The class blocks should include variables contained within the class. (2)
   6.3. The class blocks should include the methods defined within the class. (2)
   6.4. Arrows must be used to connect the class blocks to indicate relationships. (1)

7. The system shall include a print function. The user shall be able to print the source code 
   summaries created using the system. (1)

8. The user shall be able to open existing diagrams generated by the system. (1)

9. The system shall include a save function. The user shall be able to save a file created on the 
   system to the computer hard drive. (1)
   9.1. The user shall be able to save the stubs of the source code. (1)
   9.2. The user shall be able to save the final diagram. (1)

10. A method diagram shall consist of method blocks. (1)
    10.1. Each diagram shall include a rectangle box for each class mentioned in the textual summary. (1)
    10.2. The method blocks should include variables contained within the method. (2)
    10.3. Arrows will be used to connect the method blocks to indicate the method calling sequence. (1)

11. The system should produce diagrams of a suitable quality to be included in the project 
    documentation. (2)

12. The system should allow the user to modify the diagrams if desired. (2)
    12.1. The user should be able to ‘drag and drop’ the blocks of the diagram. (2)
    12.2. The user should be able to add connecting arrows to the diagram. (2)
12.3. The user should be able to delete blocks of the diagram. (2)

3.2.3 NON-FUNCTIONAL REQUIREMENTS

13. The system shall be implemented in Java. (2)

14. The parser shall be implemented using CUP Parser Generator for Java version 0.10k. (2)

Product Requirements

15. Usability: The system shall be designed around a clear user-centred conception of the process and thereby minimize the need for formal training. End-user validated task-based scenarios shall be employed as the mechanism to secure this clear conception. Users shall require a maximum of 15 minutes training before becoming a competent user. (1*)

15.1. The system shall only include core functions vital to the end task to keep training time as low as possible. (1*)

The functions included on the task bar shall consist of the options:
- Open (existing file),
- Import (source code),
- Save
- Print
- Produce Diagram

15.2. The system shall be self-consistent. System behaviour shall be similar for similar situations. (1)

15.2.1. The logical operations required for the user to achieve tasks shall be the same throughout the system. (1)

15.2.2. The user interface shall be self consistent. (1)

15.2.2.1. Buttons shall always have the same positions, colours, font. (1)

15.2.2.2. Labels shall always be in the same text font. (1)

15.2.2.3. Screens and message boxes shall use the same layout where possible. (1)

15.3. The system shall provide error recovery. (1)

15.3.1. The user shall be able to close a source code file if it is not required. (1)

15.3.2. The user shall be able to close a diagram if it is not required.

15.4. Only information which is essential to the task shall be displayed on screen to reduce navigation time. (1)

15.5. The system shall be have a similar format to generic formats and should therefore be intuitive. (1)

15.6. The system shall use icons where possible to aid navigation. The task bar shall have relevant, clear icons as well as text for each task. (1)

15.7. The system shall provide a simple help system. (1)

16. Performance: The system should perform at a suitable speed. (1)

16.1. The parser shall produce stubs within 10 seconds of taking the source code as input. (1)
16.2. The parser should produce stubs within 4 seconds of taking the source code as input. (2)
16.3. The graphical layout manager shall produce a diagram within 10 seconds of the stubs being inputted. (1)
16.4. The graphical layout manager should produce a diagram within 4 seconds of the stubs being inputted. (2)

17. Reliability: The system and the information it stores must be reliable. (1)
17.1. The stubs and diagrams produced shall be accurate representations of the original source code. (1)

18. Security: The system shall use the source code as input without modifying the file in any way. (1)

Organisational Requirements

19. Delivery: The project must be completed within the specified time. (1)
19.1. The whole project shall be completed by 16th May 2005. (1)
19.2. The project shall be fully and clearly documented. (1)
19.3. The documentation shall be self-consistent. (1)
   19.3.1. The same writing style shall be used throughout the documentation.(1)
   19.3.2. Diagrams and tables shall be in the same format throughout the documentation. (1)
19.4. The project shall include a comprehensive dissertation on the project. (1)

External Requirements

20. Legislative: The project must comply with all current laws. E.g. Copyright laws, privacy laws. (1)

21. Ethical: The project must respect all current society ethics. (1)
4. Design

This chapter uses scenarios as a basis for describing the key decisions made in the design stage of the project. This includes the overall conceptual design for a reverse engineering and code comprehension tool, and those features of the conceptual design that will be the focus of the implementation. It also describes the design of the main data structure. The structure of the system is described in terms of each module followed by an explanation of other remaining design decisions.

As part of the design process it is necessary to complete task descriptions to clarify the specific uses of the system. Following the scenario based approach to task analysis described by Sommerville (2001), informal event scenarios have been developed for the system. These describe alternative user activity involving the system.

4.1 Scenarios

Three scenarios of use are described here as a statement of the three generic activities in which users are expected to engage. In each case, they begin with a description of the task context to provide global constraints on the activity. They then detail the normal, exceptional and concurrent activities that an effective system should be able to deal with. Finally, they state the expected outcome on the part of the user, in terms of the end state of the anticipated system.

4.1.1 Scenario 1

A student finished the implementation of the basic system for his project three weeks ago. Since then he has not looked at his code and has been focusing his time on the project write up. He now wants to add a new feature to his implementation but is having trouble understanding the structure of his code.

Normal:
1. The user opens the CASE tool.
2. The user imports all of the Java source code files for his project.
3. The system produces a list of all classes, the methods they contain and the variables they contain.
4. The user chooses to create a diagram and is given a choice of diagram types.
5. The user chooses a diagram type.
6. The system displays the diagram showing the chosen relationships.

What can go wrong:
1. The user selects incorrect files to import. If this happens the user should be able to close the wrong files and select the correct ones.
2. The user doesn’t select all of the required files to import. If this happens the user should be able to go back and re-select all of the required files.
3. The user imports a file which is not a Java file. The user should be prevented from doing this.
4. The user chooses a diagram type which is not suitable for his code. The user should be able to close this diagram and select the correct type.
Other Activities:
1. The user may wish to make notes on certain classes so that they will be easier to understand next time he comes back to them.

System state on completion:
The system is displaying a textual summary of the code and a diagram of the system.

4.1.2 Scenario 2
A student has a project meeting with her tutor and wants to explain the overall structure of the project code. The meeting will last for no more than 30 minutes and the proportion of the meeting involving code review will last for no more than 15 minutes. The tutor does not remember the state of the implementation at the last meeting but remembers having made some comments about work to be done.

Initial Assumption: Either the student has the system on a laptop that can be taken to the project meeting or the system is online and the files can be accessed by both the student and the supervisor.

Normal:
Prior to the meeting: (The student uses the system to save a file containing the appropriate summary of her code.)
1. The user opens the CASE tool.
2. The user imports all of the Java source code files for her project.
3. The system produces a list of all classes, and the methods and variables they contain.
4. The user deletes any classes, methods or variables that are not relevant to the discussion.
5. The user chooses to create a diagram and is given a choice of diagram types.
6. The user chooses a diagram type.
7. The system displays the diagram showing the chosen relationships.
8. The user makes notes on issues regarding the whole system or about specific classes.

During the meeting:
1. The user opens the CASE tool.
2. The user opens the saved file.
3. The summary is used to aid discussions about the system implementation.
4. The user takes note of any comments that the project supervisor makes about the system implementation.
5. The user edits existing comments and adds new comments when necessary.
6. The user saves the file.

What can go wrong:
1. The user selects incorrect files to import. If this happens the user should be able to close the current wrong files and select the correct ones.
2. The user does not select all of the required files to be imported. If this occurs the user should be able to go back and re-select all of the required files.
3. The user imports a file which is not a Java file. The user should be prevented from doing this.
4. The user makes a mistake when editing the textual summary. It must not be simple to edit the text unintentionally. If a mistake is made, the user should be able to re-enter any data that has been deleted.

5. The user chooses a diagram type which is not suitable for her code. The user should be able to close this diagram and select the correct type.

System state on completion: The files containing the code summary are saved. The file is still open on the system displaying the textual summary and the code diagram.

4.1.3 Scenario 3
A project tutor wants to refresh his memory about a student project prior to a project meeting. The tutor has had an idea in the intervening period between meetings for changing the architecture of the programme for greater modularity.

Initial Assumption: The project tutor has a copy of the student’s source code either on his computer hard drive or on disk.

Normal:
same as scenario 1

What can go wrong:
same as scenario 1.

System state on completion:
same as scenario 1.

4.2 High Level Design Decisions

Design Focus
As discussed in section 3.2, the critical requirements for the intended system are:

1. The system shall aid program understanding by abstracting information from the source files. (1*)
2. The system shall take source code as input. (1*)
3. The system shall produce multiple representations of source code as output. (1*).
15. Usability: The system shall be designed around a clear user-centred conception of the process and thereby minimize the need for formal training. End-user validated task-based scenarios shall be employed as the mechanism to secure this clear conception. Users shall require a maximum of 15 minutes training before becoming a competent user. (1*)
   15.1 The system shall only include core functions vital to the end task to keep training time as low as possible. (1*)

The foregoing scenarios provide a backdrop for the implementation and a basis for tracking and testing its usage. Given the prioritization of requirements, the major thrust of this design must be to create a user-centred system that allows the simultaneous review of source code using multiple abstractions.
The system will offer two types of code summary: a textual summary and a diagrammatic summary. Offering different types of representations means that it is more likely to be of value to the user. As described by Petre et al (1998), ‘one representation may be more suitable for expressing the problem, but less congenial to the user, and so a second representation helps the user to reason about the first’. They also describe the notation of ‘heterogeneous inference’ (Stenning and Oberlander, 1995) where two forms of representation need to be used simultaneously for best results’. The user gains an understanding for the whole problem by alternating between different representations. Multiple representations also force a deeper level of thought and understanding in order to make correspondence between them. This can reveal gaps in understanding which would have otherwise been overlooked. This is supported by the findings of the studies on program comprehension discussed in section 2.4 of the literature review.

The textual summary will offer a list of the class, variable and method names. The names will be listed in the order that they appear when inputted in to the system. This can be beneficial because the order that they appear in the source code is likely have been a logical order for the programmer at the time when the code was written. This is relevant to method and variable names but less so for class names as these are prone to be in alphabetical order when saved on a computer hard drive. The aim of the textual summary is to display the source code extracts in a structured manner with a description of the relationships between the classes. The class, method and variable names will be clearly labelled using unambiguous keywords to describe their types. A textual summary is essentially a linear representation as text is read in a linear manner. Petre et al (1998) describe this as a one dimensional representation whilst a graphical representation is two dimensional. They also suggest that graphical representations are better for constructing a mental model of the problem. The diagrammatic summary will make use of the user groups knowledge of system models including inheritance and aggregation models.

The diagrammatic summary will display the classes at positions which represent the relationship that the diagram represents. For example, in an aggregation diagram, classes at the top level will be ‘parent’ classes and classes at lower levels will be ‘child’ classes. Two ‘sister’ classes may not have a relationship explicitly represented by an arrow between them, but the relationship will be evident as they will be on the same level and have the same ‘parent’ class.

4.2.1 Limitations
Due to the time constraints of the project, in particular the time constraints of the implementation stage of the project, the system will not be a final product but a high fidelity prototype. The prototype will be a functional demonstrator, rather than a conceptual mock up, since comprehension requires interaction. Functions such as the ‘open an existing file’ function, the save function and the print function will not be implemented. However the buttons for these functions will be included in the user interface to provide the complete look and feel of the system.

As discussed in chapter 3, casting and recasting data is of fundamental importance for a person to build a robust understanding of a system. In the terms of the HCI usability principles promoted by Dix et al., dialog initiative and multithreading underpin the necessary flexibility and observability of the system state its task robustness. The task here is comprehension of source code. Multiple representations and user-initiated recasting of code
are considered to be the critical design issues that capitalise on these principles. The main short fall of the prototype is that it will not provide the functionality for the user to produce all types of diagram. It will not produce diagrams based on the methods of the program.

Depending on the speed of the development some of the desirable requirements may not be implemented. However all essential requirements should be implemented and any desirable requirements that time permits.

These constraints should not affect the ability for the user to navigate through, and interact with the system. The user should be able to fully appreciate how the final system would behave.

A constraint put on the final system as well as the prototype is that it will only consider newly written code. It will not look at any imported libraries that are used in the program. This is because data taken from libraries will make the summaries and the diagrams unnecessarily complex, including large amounts of unused, irrelevant code. In general, the user will not be interested in this information.

**4.2.2 Data Structures**

The main data structure in the system is used to store the data about each class. An array has been chosen as the structure to store the data. This was chosen over using a database as it seems more natural to use a Java array in a Java program to store data from Java source code. It avoids impedance mismatch which occurs when two systems are connected that have very different conceptual bases. The most common example of this is the use of an SQL database with an object oriented program. Java and other object oriented languages result in encapsulated objects which when mapped to database tables makes the database fragile. This is because there are fewer constraints on the encapsulation representation of objects compared to a database’s use of public data.

Figure 4.1 shows a data model of the data stored in the class data array.

![Data model for 'class data' object](image)
4.3 Modules Overview

The system is split into modules. The possibility for modularisation was identified figure 2.7 of the literature review which shows a diagram of the system. There are two distinct modules, the parser and the layout manager. Important parts of the source code will be extracted and displayed to the user as a textual summary. The user then has the chance to modify the data which subsequently needs to be parsed again before it is displayed as a diagram.

Three modules have been identified; the source code parser, the textual summary parser and the layout manager. Two types of parser are needed because when the modified textual summary is parsed, the data will be in a more organised format than previously when in the source code files, without any unwanted text such as the class body. This means the method used can be simpler and faster.

4.3.1 Source code parser:
A system model for the source code parser is shown in figure 4.2.

Using figure 4.2, this section can be broken down into further modules:
Import file: This allows the user to specify the name and location of the source code file. It then searches for the file and checks that it is a Java file before opening it.

Tokeniser: This module takes the text for the file and splits it into individual words.

Source code parser: This module parses the source code. It extracts and saves data about each class.

Textual summary display: This module prints the stored data to the screen. It will be useful for this summary to remain available for when a diagram is produced. It may be valuable to the user to be able to view the textual summary as well as the diagrammatic summary at the same time.
4.3.2 Textual summary parser:
This module is similar to the source code parser but the parsing method is not as complex. This is because the format of the text will be known. It can be split further into the following modules:

- **Tokeniser:** same as Tokeniser in the source code parser
- **Textual summary parser:** This module parses the data that has been displayed and potentially modified by the user.

4.3.3 Layout Manager:
This module creates the diagrams of the source code, it is a major part of the system. The data from the textual summary parser is taken as input and displayed in a graphical representation. This involves grouping together all classes which are connected by the relationship that the diagram is displaying. It also needs to draw the class boxes containing the data at the correct size and correct position.

**Dynamic boxes**
The building blocks of the diagram need to display the information from the textual summary about each class. Each class will have a separate box. The information to be displayed in each box includes the class name, the variables created in that class and the methods declared in the class. The design of the class box is to be a generic design which is normally used in class diagrams. The design has been taken from Sommerville (2001). Figure 4.3 shows the design for a standard sized class box in a diagram.

![Figure 4.3 Design of a standard class box in a diagram](image)

Some classes will have more information to be displayed that others. The standard diagram boxes will have space to display the class name, three variables and one method (the method name will include the parameters, e.g. `myMethod(String name)`). If there are more variables or methods then the box height will increase. The box width will not increase when the strings are too long as some strings, especially method names may be very long and this will ruin the shape and aesthetics of the diagram. There will be a maximum length and strings that are longer will be shortened so that they fit in the standard box.

**Diagram Layout**
Two possible layouts for the diagrams were considered as shown in figure 4.4 below.

Layout 1 is more aesthetically pleasing as it is symmetrical, however in terms of implementation it is much more complex. Firstly, when drawing child nodes, you must consider whether the nodes are going to be positioned on the left hand side of the diagram or on the right hand side. If the node is on the left hand side, then any further nodes must be
drawn further to the left. If the node is on the right hand side then further nodes must be
drawn further to the right. The main difficulty with layout 1 is that to maintain a symmetrical
layout, you need to calculate the whole diagram layout before drawing any nodes. Consider
an instance when box 1 has a third child node that needs to be drawn. If it was to be added to
the diagram now next to box 3, then there would be no space directly below it for draw any
child nodes that it might have. A further example is if box 4 had three child nodes drawn.
This would mean that box 5 had no space to draw any child nodes directly below it, if it
should need to.

![Diagram 1](image1)

![Diagram 2](image2)

*Figure 4.4 Possible diagram layouts*

Layout 2 is drawn from left to right. Node 5 is only drawn after node 4 has been drawn and so
it can be drawn further to the right of it. This enables any diagram to be drawn with the layout
being a relatively small factor.

For these reasons layout 2 was chosen over layout 1.

**Aggregation and Inheritance Diagrams**

Each diagram will show all the classes in the source code files, not just those that have the
particular relationships that they are displaying. Classes that are not involved in the relevant
relationships will be drawn as a lone box. This is so that each diagram shows a complete
representation of the input source code files.

The format of the aggregation and inheritance diagrams will differ in the type of arrows that
they use. These are shown in figure 4.5.

![Aggregation and Inheritance Arrows](image3)

*Figure 4.5 Diagram arrows*
5. User Interface Design

This chapter explains how design decisions were made for the user interface. User studies were carried out to gain feedback on the initial ideas. The design was then modified and improved using the results of the studies.

The user interface is as important for the design of a CASE tool as for any other interactive system. As stated in requirement 13 of the requirements specification: ‘The system shall be simple to use without the need for formal training’. Simplicity has two connotations here. The first is that it must conform to a clear, iterative process of program inspection and code annotation. The second is that it must adhere to the broader constraints of supporting a software project meeting. This is imperative as the system is a tool to aid the project process, it will not be used if it either makes code inspection cumbersome or if it is overly intrusive in project meetings. It must be easy to understand and quick to use. The ISO-9241 definition usability casts it as: ‘The effectiveness, efficiency and satisfaction with which users accomplish tasks’. The user should be able to carry out tasks on the system with ease, with little or no training specific to the application. Navigation through the system shall involve a minimal number of steps and the possible routes shall be clear.

5.1 Object-Oriented Program Inspection Tasks

As described in Chapter 2, object-oriented programming is the main paradigm for contemporary software engineering. Object-oriented languages require the construction and association of several software components. Each component can contain both data, such as variables and arrays, and the means for its manipulation, as procedures or “methods”.

The overall structure of an object-oriented program is supposed to reflect the nature of the application domain. An appreciation of the relationships between classes provides an insight into the problem as it is conceived by the program designer, and of the operation of the program.

It is common for components to take advantage of one another by associating with their data or methods. Each of the program’s components or classes can be used and reused at runtime by one or more coordinating classes. For example, a new class, K, can be defined by the programmer to make use of a method that has already been defined in another class, L. This is valuable for saving coding effort but can come at the cost of needing to know which other classes are involved. For example, it may be that the method in class L itself makes use of another class, M. It is often important to be able to track these associations to understand the dependencies in the program.

The motivation for this project is to give programmers in general a way of examining, understanding and improving object-oriented software projects. The specific focus is for final year students and for their project supervisor to do this.

Object oriented structure is not all about an abstract concept. The actual structure is realized in the code written by its programmer. Requirement 4 thus insists that the textual code summary be accessible alongside representations of its structure, setting up the inspection process as concurrently working between the outline of the code, and the structural information about it.
5.2 User input and output

It is important to think carefully about user input and output as the design of these features has a major effect on the system usability. Functionality that requires input must be clear and input must be easy to perform as recommended by the following design principle by Nielson (2001): ‘Recognition rather than recall – make objects, actions, and options visible’. Functionality providing output to the user must provide information in a way that is easy to interpret.

How will the user interact with the system?

Inputs:
- Selecting a source code file to import
- Modifying the textual summary - this is the only user input task that is not dominated by the mouse point and click method.
- Clicking any button available
- Selecting a location for the file to be saved

Outputs:
- Textual summary
- Diagrams

The system interface was designed with the general design principles described by Preece et al (2002) in mind.

5.3 User Studies

User studies were carried out with the aim of identifying further requirements and developing alternative designs to meet those requirements.

A Discount Usability Evaluation method (Nielson, 1993) was used. This is a low cost method carried out to identify usability problems. Users were interviewed with the intention of gaining explanations and opinions of the system design, and suggestions on possible modifications or additions.

The study had two main aims:
1. To gain an understanding of what information users perceive as being useful when extracted from source code and used to form a summary and also a diagrammatic representation.
2. To test the users understanding when presented with the user interface design

There were two categories of assumptions made prior to the design stage of the project as described below. One aim of the study was to test whether the assumptions made were correct. There are content assumptions about which features of the inputted source code the system needs to extract. There are also control assumptions about how the user would like to be able to interact with the system.
Assumptions
Content – features to extract from the source code
1. Class Names
2. Inheritance details
   - Whether the class extends another class and the name of the class that it extends.
3. Aggregation details
   - Whether the class is within another class and the name of the class it is within.
4. Whether the class implements an interface and the name of that interface
5. All variables defined in the class
6. All methods defined in the class

Control – how the user wants to interact with the system
1. The user should be able to specify a file containing the source code from a location on disk or on the computer hard drive.
2. After the source code has been parsed, the user should be able to modify the parsed data.
3. The user should be able to switch views between the original source code and the current screen.
4. The user should be able to drag and drop blocks within the end diagram.
5. The user should be able to add or delete connecting arrows within the end diagram.

The studies were carried out in the style of an informal interview. There are two categories of system users: students and project supervisors. Targeted students were final year computer science students using Java as the programming language for their project implementation stage. Student users were asked questions about how they communicate with their project supervisor (for example, have they spoken about specific details of the implementation stage of their project, have they used anything to aid their discussions) and also how they would have liked to communicate with their supervisor. Project supervisors were asked about useful explanation aids that students have used to describe their implementation stage. Both groups were also asked questions about the type of data that they would find advantageous in a summary of code and also the types of diagrams that they would find most useful.

The users were presented with static images of the screen designs and asked explain their understanding of each of the functions, and how they would perform the tasks that the system allows.

5.3.1 Pilot Study

A pilot study was carried out in order to test the planned format for the user studies and to identify any problems with the interviewing style or prototype design. The pilot study revealed that the planned questions worked well and some useful data was collected. However it also showed that the design of the paper prototype was inadequate and did not clearly demonstrate how the system will work.

The original paper prototype design displayed only one screen. It showed the main part of the system but did not explain how this state was reached and therefore many aspects of the system had to be explained to the user.
Figure 5.1 shows the paper prototype used in the pilot study. The buttons shown below the screen correspond to the menu bar at the top.

Following the pilot study, the paper prototypes were re-designed so that they consisted of four screens which represented a walk-through of the system more clearly. This allowed the user to use the prototypes with little explanation and therefore form their own opinions. The design of the paper prototypes can be found in Appendix I.

### 5.3.2 User Study Plan

A script was prepared to brief the users before the interview. This script was memorised by the interviewer so that each user received the same information about the system and the study. This was to minimise bias between different users.
5.3.3 Introduction to users:

My final year project is to design and implement a system to aid code comprehension. It is a system to help someone understand source code that they have not written themselves. More specifically, a system to aid the communication between computer science students and their project supervisors.

The aim of this research is to further investigate the type of data that target users would find useful if extracted from a source code file. I am also looking at different approaches to displaying this data in a meaningful way and how a user would like to be able to interact with the system to manipulate the data.

With your permission I will be recording your answers so that the conversation flow is not interrupted by pauses whilst I write notes. Any information about your project or project supervisor will be treated confidentially and will only be used for the purpose of this study. Your answers will be anonymous.

To aid the study I have made paper prototypes of a simple example scenario of the use of the system to gain feedback on improvements of the user interface design. The first page is the screen that will be displayed when the system is initially executed. At the bottom of each page are the buttons that are available from the menu bar at the top of the screen.

5.3.4 Paper Prototypes

The low-fidelity prototypes display ideas for the design of the user interface. One aim for the user study is to collect ideas for possible ways to improve the usability of the system. Paper mock ups were used as these have been proven to be the best way to encourage user reactions and modification. The designs are displayed in black and white. This is so that they appear ‘unpolished’ and do not seem like screen shots of the final system. This should encourage users to write their comments and criticisms on the designs.

Description of the prototypes

The users were not given a detailed description when presented with the prototypes. The users were given an explanation of the ideas and designs after they had made their own interpretation and given their comments. The paper prototypes can be found in Appendix I.

Screen 1: Opening screen with simple introduction and instructions.
Simple instructions are needed to explain the purpose of the system and how to begin using it. ‘FILE’ was chosen as the name for the first button menu due to its familiarity from other systems. The names and order of the buttons have been chosen for the same reason. There must be a relationship between controls and their effects and so the choice of button labels are very important. The design principle of ‘mapping’ described by Preece et al (2002) explains this.

Screen 2: When the button “NEW” is clicked the user begins a new project and is made to select a Java file to import. This will close the help instructions but they will still be available by clicking the ‘HELP’ button in case they need to be referred to again.
Screen 3: Once a file has been imported, the system extracts bits of the source code and displays a summary in the ‘CLASS VIEW’ column.

Screen 4: When a diagram button is selected, the relevant diagram is displayed in the remaining part of the window. In the case demonstrated in the paper prototype, the diagram is a class diagram. The class view remains on screen so that the textual summary and code structure diagram can be viewed simultaneously.

5.3.5 Guideline questions:

The following questions were used as guidelines during the interview. The interview was semi-structured, guided by the script, which allowed flexibility. Areas of focus varied with different users depending on their answers and interesting issues were explored in more depth. The users were always encouraged to think aloud and explain their comments.

Students:
Have you talked to your project supervisor about the implementation of your system? If so, did you use anything to aid your discussions, for example, sections of code, diagrams? If you needed to explain your code to someone else how would you go about doing this? What type of diagram would be most suitable to produce a summary of your project code?

Supervisors:
Do you talk to your students about the implementation stage of their project? What would you find useful as an aid to understanding the system, for example, sections of code, diagrams?

All users:
What detail would you find most useful in a summary of source code? What type of diagrams would be most useful to summarise code that you were trying to explain / understand? Are there any aspects of the prototypes that you do not understand? What modifications would you make to the paper prototypes?

5.3.6 Discussion and Redesign

The results of the user studies produced several good ideas for the system.

The development of the user study itself forced further thought into the user interface design. The pilot study revealed that the original paper prototypes were inadequate and did not explain the how to use the system. It also revealed the need for simple instructions when the program begins.

Five computer science student users were interviewed and one project supervisor. The students chosen were all final year students who were using Java as the programming language for their projects.

Have you talked to your project supervisor about the implementation of your system? If so, did you use anything to aid your discussions, for example, sections of code or diagrams?
Of the students users, only one had discussed his project implementation in detail with his supervisor. Some of the students had told their tutors about the general implementation concepts but this was mainly a one way explanation with the supervisors making few comments. All students said they would have liked to discuss their implementation techniques to gain some reassurance that they were on the right lines with the decisions they were making. The student that did discuss his code with his supervisor used his laptop as an aid in his discussions to demonstrate the source code how it executed. Overall feedback about the concept and aim of the system was positive, that the system would be useful and would encourage more discussions about the implementation process.

If you needed to explain your code to someone else how would you go about doing this? Answers included:
- printing the code with good comments
- listing the methods in pseudo code including which other methods each one calls and which methods call it.
- making the code available online

What type of diagram would be most suitable to produce a summary of your project code? The answers to this question were varied. Some users said that a class diagram would be useful as their code involves many classes that link together. Others said that their programming style is more procedural and a method diagram of the method calling sequence would be most useful. The functionality to create both types of diagram would be beneficial.

Suggestions for user interface design
Modifications were suggested by two users for the button names. Currently by clicking ‘NEW’ you are able to import a file. This is not logical and will be changed to ‘IMPORT FILE’ which better reflects the action. ‘OPEN’ is slightly ambiguous as a label, and needs further explanation such as ‘OPEN EXISTING’. The ‘CLOSE’ button suggests that it will close the current file whereas it actually closes the whole system. ‘EXIT’ is a more comprehensible label due to its use in other systems.

On screen 1 improvements were suggested for the help text. The ‘NEW’ button was mentioned in the text but is not visible on the screen as it is within the file menu. This can cause confusion and needs to be explained. A simple improvement is to begin the line with ‘On the file menu, NEW enables….’. Also the instructions on the lower half of the screen seem to be three bullet points but currently the text is not structured in this way and therefore does not flow well. This will be modified. The text will also be modified to ensure that the terms used are always consistent and that explanations are not redundant. For example the current instruction of ‘The DIAGRAM menu can then be used to produce relevant diagrams’ is not adequate as the word diagram is used to explain the same term. This will be changed to ‘The DIAGRAM menu can then be used to produce relevant graphical representations’.

A good improvement was suggested for the ‘CLASS VIEW’. Rather than the view consisting of one big list, it could begin as a list of the class names and then allow clicking on the names to ‘drill down’ for further detail of variable and method names. This could be done with a tree structure. A long list could become very confusing. This problem was mentioned by more than one user and is a beneficial improvement. Another suggestion for this module of the system is to use indentation to signify the aggregation between classes.
An interesting point raised in the interviews was the importance of comments in code comprehension. When the users were asked to describe the methods they would use to explain their code to someone else, several users said they would show the other person the source code, ensuring that it had good comments throughout. Comments were also mentioned in other parts of the interview.

In the literature review in chapter 2, the problem of insufficient information with reverse engineering was identified. This refers to the lack of information in the source code about the architecture, design trade-offs and engineering constraints. ‘Visualisations are not useful without insight about their use, about their significance and limitations’ (Petre et al, 1998). A useful extra feature would be the functionality to allow the user to add comments to the diagrams or to extract certain comments from the source code. This would create the opportunity to add any important information that is not contained in the source code.

One potential method is for the comments to be extracted to be indicated in a particular way, for example, extract all comments specified with the escape characters /*….*/ but ignore line comments indicated by ‘//’, or perhaps all comments from a specified place, for example before the class declaration. One further idea suggested was to be able to view comments by double clicking on the class box in the diagram. This provoked further thought in to code comments.

Comments are very important to aid the maintainability and comprehension of code but are not meaningful without the source code to go with them. Raskin (2005) wrote an article entitled ‘Comments Are More Important Than Code’. He feels that comments are so important that the ‘use of escape characters to “escape” from code to comment is backwards. Ideally, comments should be the default, with a way to signal the occasional lines of code’. He also voices his opinions on automatic documentation generators. Although in theory he feels they are a good idea, in practice the documentation is of a poor standard. This is because they can not explain why the program has been written or explain the rationale behind using chosen techniques. This can not be derived from the code alone.

The aim of this system is not to produce full documentation but to produce a source code summary to aid code explanations and comprehension. However the ability to add comments in some form would be a useful function of the tool. This functionality will be added to the final redesign of the system.

There are several ways that comments could be incorporated in to the code summary.

1. Simply being able to add free text to the diagram area.
2. Having a small icon on every diagram class box which brings up a text window to add comments about that class.
3. Adding a separate ‘comments box’ below the class view where comments about each class can be written.

The suggested method of extracting certain types of comments or comments from a specific area of the source code is not appropriate for this system. This would require knowledge of the comment requirements prior to using the system or would suggest the need for the system to allow the user to add to the source code, whilst in the system. If the system included the functionality to modify the source code, the functionality to compile code would have to be added as well. This would be necessary to ensure that the source code still compiles after any
modifications have been made. This is outside the scope of this project as the purpose of the system is as an analytical tool rather than a development environment. Incorporating icons on each class box which open a text window to store comments seems to be the most robust method. This will be added to the final design.

Two users mentioned adding the functionality of being able to click on the class name or to drill down on the diagram to get the source code. This would be useful but is outside the scope of this project and would potentially extend the tool to an Integrated Development Tool (IDE). Similarly one user discussed the extra functionality of putting a ‘watch’ on certain variables to show where in the code they can be modified. This would be more appropriate in a debugging tool which is also beyond the scope of this project.

The final designs can be found as screen shots of the implemented system in Appendix IV.

5.4 Final Requirements Specification

The results obtained from the user studies developed a number of new requirements and modifications to the existing requirements. The new requirements are indicated with bold text.

5.4.1 FUNCTIONAL REQUIREMENTS

1. The system shall aid program understanding by abstracting information from the source files.(1*)
   1.1. The system shall aid program understanding for the project supervisor to optimise time usage in project meetings. (1)
   1.2. The system shall aid source code analysis for the student. (1)

2. The system shall take source code as input. (1*)
   2.1. The user shall be able to specify the location of input files containing source code.
   2.2. The system shall import the files and use the data it contains without modifying it in any way. (1)

3. The system shall produce multiple representations of source code as output. (1*).
   3.1. The system shall provide the functionality to display a textual summary of the source code.
       3.1.1. The list shall include the names of classes, methods and variables.
       3.1.2. This list shall include details of inheritance and aggregation. (1)
       3.1.3. The outline shall be displayed using a representation that maintains clarity even when displaying large amounts of data. (1)
   3.2. The system shall provide the functionality to produce a diagrammatical representation of the source code.
       3.2.1. The system shall be able to produce a diagram representing the inheritance relationships of the source code. (1)
3.2.2. The system shall be able to produce a diagram representing the aggregation relationships of the source code. (1)

3.2.3. The system should be able to produce a diagram representing both the inheritance and aggregation relationships of the source code. (2)

3.2.4. The system should be able to produce a diagram representing the method calling sequence of the source code. (2)

3.2.5. All diagrams should all display the classes, variables and methods in the source code. (2)

4. The system shall allow the user to view the code summary and the code diagrams simultaneously. (1)

5. The system shall allow the user to remove any unwanted classes, methods or variables from the textual summary before it is put into a diagrammatic format. (2)

5.1. The textual summary must be difficult to edit unintentionally. (2)

6. The class diagram shall consist of class blocks. (1)
   6.1. Each diagram shall include a rectangle box for each class mentioned in the textual summary. (1)
   6.2. The class blocks should include variables contained within the class. (2)
   6.3. The class blocks should include the methods defined within the class. (2)
   6.4. Arrows must be used to connect the class blocks to indicate relationships. (1)

7. The system shall include a print function. The user shall be able to print the source code summaries created using the system. (1)

8. The user shall be able to open existing diagrams generated by the system. (1)

9. The system shall include a save function. The user shall be able to save a file created on the system to the computer hard drive. (1)
   9.1. The user shall be able to save the stubs of the source code. (1)
   9.2. The user shall be able to save the final diagram. (1)

10. A method diagram shall consist of method blocks. (1)
    10.1. Each diagram shall include a rectangle box for each class mentioned in the textual summary. (1)
    10.2. The method blocks should include variables contained within the method. (2)
    10.3. Arrows will be used to connect the method blocks to indicate the method calling sequence. (1)

11. The system should produce diagrams of a suitable quality to be included in the project documentation. (2)

12. The system should allow the user to modify the diagrams if desired. (2)
    12.1. The user should be able to ‘drag and drop’ the blocks of the diagram. (2)
    12.2. The user should be able to add and remove connecting arrows to the diagram. (2)
    12.3. The user should be able to delete blocks of the diagram. (2)
12.4 The user should be able to resize the diagram. E.g. 50% of its original size.

13. The system shall allow integrated annotation the diagrams. (1*)
   13.1. The user shall be able to add comments regarding the system as a whole. (1)
   13.2. The user shall be able to add comments about individual classes. (1)
   13.3. The user should be able to access comment for different classes individually. (2)

14. The system should allow the user to view the original source code files along side the summary produced by the system. (2) This requirement is not stated as essential as the user always has the option of having the source code open in a separate text editor.

5.4.2 NON-FUNCTIONAL REQUIREMENTS

15. The system shall be implemented in Java. (2)

16. The parser shall be implemented using CUP Parser Generator for Java version 0.10k. (2)

Product Requirements

17. Usability: The system shall be designed around a clear user-centred conception of the process and thereby minimize the need for formal training. End-user validated task-based scenarios shall be employed as the mechanism to secure this clear conception. Users shall require a maximum of 15 minutes training before becoming a competent user. (1)
   17.1. The system shall only include core functions vital to the end task to keep training time as low as possible. (1)

   17.2. The system shall be self-consistent. System behaviour shall be similar for similar situations. (1)
   17.2.1. The logical operations required for the user to achieve tasks shall be the same throughout the system. (1)
   17.2.2. The user interface shall be self consistent. (1)
         17.2.2.1. Buttons shall always have the same positions, colours, font throughout the system. (1)
         17.2.2.2. Screens and message boxes shall use the same layout where possible. (1)

17.3. All buttons shall be labelled appropriately displaying a recognisable relationship between the control and the effect. (1)

17.4. The system shall provide error recovery. (1)
   17.4.1. The user shall be able to reselect source code input files if the original choice is incorrect. (1)
   17.4.2. The user shall be able to close a textual summary if it is not required. (1)
   17.4.3. The user shall be able to close a diagram if it is not required. (1)

17.5. Only information which is essential to the task shall be displayed on screen to reduce navigation time. (1)

17.6. The system shall be have a similar format to generic formats and should therefore be intuitive. (1)
17.7. The system shall use icons where possible to aid navigation. The task bar shall have relevant, clear icons as well as text for each task. (2)

17.8. The system shall provide a simple help system. (1)

18. Performance: The system should perform at a suitable speed. (1)
   18.1. The system shall display a textual summary within 10 seconds of taking the source code as input. (1)
   18.2. The system should display a textual summary within 3 seconds of taking the source code as input. (2)
   18.3. The graphical layout manager shall produce a diagram within 10 seconds of the diagram request. (1)
   18.4. The graphical layout manager should produce a diagram within 3 seconds of the diagram request. (2)

19. Reliability: The system and the information it stores must be reliable. (1)
   19.1. The source code representations shall be accurate representations of the original source code. (1)

Organisational Requirements

20. Delivery: The project must be completed within the specified time. (1)
   20.1. The whole project shall be completed by 16th May 2005. (1)
   20.2. The project shall be fully and clearly documented. (1)
   20.3. The documentation shall be self-consistent. (1)
       20.3.1. The same writing style shall be used throughout the documentation. (1)
       20.3.2. Diagrams and tables shall be in the same format throughout the documentation. (1)
   20.4. The project shall include a comprehensive dissertation on the project. (1)

External Requirements

21. Legislative: The project must comply with all current laws. E.g. Copyright laws, privacy laws. (1)

22. Ethical: The project must respect all current society ethics. (1)
6. Implementation

This chapter describes how the system was implemented. It discusses important choices of implementation techniques, the methods used and any significant problems that were encountered.

6.1 Programming Language

Java was chosen as the programming language for the system. One reason for this choice was because of its familiarity. Also, it seemed logical to implement the system in the same language as the source code that it would cater for. Java is a suitable language for the system to support as it is currently a popular programming language and is frequently one of the main languages taught in degree level computer science courses. Therefore, Java is a language that is likely to be chosen for use in final year computer science projects. It would not be a logical design decision to cater for an obscure language as this would limit the likelihood of use.

The performance of Java as a programming language is suitable for a system of this type. Java is compiled to an intermediate form known as byte codes (Flanagan, 1999). This means that Java programs are faster than programs written in interpreted languages but slower than programs written in languages which compile to native machine language, such as C or C++. However, improvements made in releases from Java 1.2 onwards, mean that performance is no longer a significant issue. Also these differences have little affect on the overall performance of an interactive system where delays are incurred while waiting for user interaction. Java combines performance and portability making it a suitable language for this system.

6.2 Implementation Process

Throughout the implementation process, explanatory names were used as much as possible for classes, variables and methods.

6.2.1 User Interface

A class named Interface was used to create the main user interface. This class has an aggregate component called ActionEventHandler which calls all other major classes in response to an event such as a button click.

The graphical user interface was implemented using the Javax.swing package. The interface uses a JDeskTopPane and JInternalFrames. This enables a multiple document interface allowing several windows to be open in parallel. As well as allowing a single textual summary and diagram to be opened and viewed simultaneously, several frames containing textual summaries and diagrams for different source code files could all be open at the same time and toggled between.

A class implementing a file filter is used so that only Java files can be selected. This makes the files that the user requires more accessible and easier to find. It also means that it is easier to select many files at once as users will almost certainly want to import a list of files. The filter means that the required files are likely to be available in a consecutive list.
The two modules that took the longest time to implement were the source code parser and the layout manager.

6.2.2 Source code parser
The use of ‘Cup Parser Generator for Java, version 0.10k’ by Hudsen et al (1999) was planned for the implementation of the source code parser. This involved using the parser generator on a Java specification to produce two files containing declarations for classes. The Java specification is also available from the Java CUP website so this task was fairly straightforward. However, when compiling the two files in hope of creating an executable file, compilation errors resulted. This problem was not overcome and a few weeks of the implementation period were wasted trying to understand how to use the Java CUP package. It was decided that this program was too complicated for the intended use and it would be simpler and more logical to create a parser from scratch containing only the required functionality for this system. The parser required for the system only needs to extract certain features of the code, it does not need to check for errors in the code or recognise all Java structures.

One benefit gained from considering Java CUP was that it did provide a full Java specification which was very useful in establishing the different possible syntax that needed to be searched for when searching for a class, variable or method declaration.

When a file is imported and the text is analysed, the text first needs to be split into words. The first method that was considered for this was from the class StringTokenizer, however the methods provided by this class were limited and restricted the flexibility of the parser. String.split( ) was chosen as the method to be used. The use of this combined with the class ‘Pattern’ allowed the text to be split into words and stored in an array. This provided much more flexibility.

The first action carried out on the text from the imported files is the removal of all comments. The text is imported line by line and so line comments which are indicated by the characters ‘//’, are removed at this stage. Once an escape character ‘//’ is found, the rest of the line is discarded. Other comments indicated by the escape characters ‘*/’ and ‘*/’ are removed after the text has been split into words and put in an array. All words between the two escape characters are removed from the array inclusively.

The source code parser and the textual summary parser are implemented in the same class but with different constructors. This is because they both use the same objects but the textual summary parser uses a simplified method.

The source code parser consists of two loops. The first loop is used to extract information from the class declaration. It stores the class name, whether or not the class extends another class and if so, the name of the extended class. Next it stores whether not the class implements any interfaces and if so, the names of the interfaces are stored. When a left curly brace is found, this indicates the end of the class declaration.

The second loop is used to extract data from the class body. The data needed consists of any variables or methods declared in the class and also any aggregate classes declared within the class. Methods and variables are found by searching for primitive data type keywords such as
‘int’ and ‘String’. These are differentiated between by checking for brackets which are needed in a method declaration but not in a variable declaration. Constructors are particular types of methods which do not have return values and so do not have data types in their declarations. They are found by searching for a modifier such as ‘public’ or ‘private’ and then looking for brackets after the method name.

 Whilst parsing the class body, a ‘brace counter’ is used to detect the end of a class. The brace counter is incremented whenever a left curly brace is found and decremented when a right curly brace is found. It is first incremented at the left curly brace after the class declaration when the class body begins. This means that when the brace counter reaches zero again, it is the end of the class body.

6.2.3 Textual Summary Display
When all the source code files have been parsed the data from the array is used to print a summary of each class in the JInternalFrame named ‘CLASS VIEW’.

The summary is displayed in the form of a JTree. This makes the data more manageable than if it was displayed in a long list of text. As mentioned in the design chapter, a textual representation can be thought of as one-dimensional. By displaying the information in a tree structure, secondary notation is added to the representation. This involves indentation, vertical and horizontal patterns and white space which all add value by creating a meaningful layout and presentation (Petre et al, 1998).

The number of clicks required to expand and contract the tree was changed from two clicks, which was the default, to one click since this is the convention used in a windows operating system. To edit the text in the tree, the user must either triple click on the text or press F2. This means it is unlikely that the user will edit the data unintentionally as it is fairly difficult to enter the editing state by mistake. Also the change is only saved if the user presses enter after editing, rather than simply clicking away. This complies with the usability principle by Nielson (2001) of error prevention stating that the editing state must not be too easily accessed or data could be changed without the user realising. Tests were done with different ways of allowing the user to edit the data by overriding the DefaultTreeCellEditor methods. If editing was allowed with one click then problems arise when trying to expand or contract the tree. The default method of a triple click or by pressing F2 was found to be the best way however short instructions on this were required in the help text and also in the frame containing the tree itself.

6.2.4 Textual Summary Parser
Once the data has been edited, it must be parsed again. The constructor used for this parser uses a simpler method than the source code parser as the input is taken from the tree in the ‘CLASS VIEW’. The layout is pre-defined and the position of the words that need to be extracted is known.

6.2.5 Layout Manager
The layout manager module is used to create diagrams of the source code. The diagrams are drawn by overriding the paint( ) method of the JPanel class. Figures 6.1, 6.2 and 6.3 describe the methods used to create an inheritance diagram.
The overall method to create an inheritance diagram is split into three methods:

- **Inheritance Diagram method** – The main diagram method. For each class, this method asks the questions: “Does this class extend another class?” if not it asks “Do any other classes extend this class?”. This groups together all classes that are connected by an inheritance relationship.

- **Extend Class Diagram method** – this is called if the answer to “Does this class extend another class?” is yes.

- **Extend Class Diagram2 method** – this is called if the answer to “Do any other classes extend this class?” is yes.

---

**Figure 6.1 Process Model for an Inheritance Diagram**
A similar method is adopted for the creation aggregation diagrams. However the process is slightly less complex as to some extent the classes will always be stored in the array in a logical order. When the source code is parsed, the parent class declaration will always be found before any aggregate component class declarations contained within it. Therefore the parent class data will always be stored in the array before the component class data. This means it is not necessary to search the array to check if the class is an aggregate component of any classes at later positions in the array. See Appendix II for aggregation diagram process models.

The process for finding the inheritance relationships between classes was outlined in figures 6.1, 6.2 and 6.3 but these flow charts do not explain how the layout of the diagram is drawn. To enable the correct spacing, the branches of the diagram need to be drawn from left to right.
and the nodes need to be drawn from the bottom up. This enables each box to be drawn with knowledge about the boxes below it and the boxes to the left of it. In Figure 6.4, the boxes are numbered in the order that the system would draw them.

![Figure 6.3 Diagram Layout](image)

When the diagrams are drawn there is a minimum size for each box. A minimum sized box can hold the class name, three variable names and one method name as discussed in section 4.3.3. If a class has more data than this, then the box is extended which affects the diagram layout. In figure 6.4, box 7 is drawn at a higher level because box 3 is bigger than a normal sized box. Before a box is drawn the location must be decided, given the size of the boxes in the level below. The box must be moved to a higher position according to the height of the biggest box on the level below. For this layout to work correctly, if a box is bigger than the normal size, it must extend upwards rather than downwards. Also box 6 is drawn further to the right because of the presence of box 2. This layout enables any possible diagram to be drawn.

The size of the JPanel that holds the diagram needs to increase if the diagram is bigger than the original size of the panel. The paint method can not return any values as the return value is void by definition. This means the panel must be resized within the paint method when the size of the diagram is known. This has been implemented using two variables, canvas height and canvas width. Whenever a class box is drawn, if the location of it goes over the canvas height or canvas width then they are increased. At the end of the paint method the size of the JPanel is set to the final canvas height and canvas width.

**6.2.6 Comments**

A simple icon is drawn in the corner of each class box in the diagrams. The position of these icons are stored in an array when they are drawn. This enables a mouse listener to detect when they have been clicked by checking whether the mouse event occurred on co-ordinates within one of the comments icons. The icons are used to bring up a new JInternalFrame where comments can be written and stored about the class that was clicked on.

An extra string variable was added to the data structure which stores all the class data described in section 4.2.2, to store any comments about the class. If a comments frame is open when the comments icon on a different class box is clicked, the new frame replaces the last one and displays the comments about the new class. Every time a comments frame is
closed, the text contained in it is stored. This means that the comments are still available if a different type of diagram is created of the same classes.

To enable comments about the entire project, an extendable JTextArea with the heading ‘Project Comments’ was added to the top of the diagram frame.

A description of each class can be found in Appendix III and screen shots of the implemented system can be found in Appendix IV. The source code and compiled files for the implemented system are available on the attached CD with instructions on how to execute the system on the readme.txt file.
7. System Testing

This chapter describes the testing strategy used to test and evaluate the system.

Somerville (2001) describes two techniques to be used in verification and validation of a system.

1. **Software Inspections** – ‘analyse and check system representations such as the requirements document, design document, design diagrams and the program source code’.

Software inspections have been continually applied at all stages of the project. They are static techniques, meaning they do not require the system to be executed. Validation of the requirements and design was supplemented with user feedback as described in chapter 5.

2. **Software Testing** – ‘executing an implementation of the software with test data and examining outputs of the software with its operational behavior to check it is performing as required’. Software testing uses executable representations of the system which means it is a dynamic technique.

The methods of verification and validation described in this chapter fall under the heading of software testing. The tests have been split into various sections depending on the type of test being carried out. Each test has mainly been linked to a requirement from the requirements document. This helps to ensure the system does exactly what is required. However, further tests have been designed to ensure the system itself is designed well and that all components fit together.

Functional testing or black-box testing was carried out to check how well the system conforms to the final requirements specification.

Structural testing or white-box testing was used to test the output of the ‘class view’ and the diagrams. Test data was derived using knowledge of the structure of the program to ensure that every statement in the sections of code that contributed to this output was executed at least once.

The scenarios described in chapter 4 were carried out to evaluate the overall performance of the system.

User tests were carried out to evaluate the design of the system and how easy it is to complete the desired tasks. Current final year students were asked to participate in the tests using both specially designed test data and also source code files from their own projects. As well as evaluating the design of the system, this also provided further test data.

The tests revealed several problems in the system, many of which were solved at the time. This chapter highlights any problems that remain.
7.1 Functional Tests

A table of the functional tests and their results can be found in the Appendix V. Where appropriate both functional and non-functional requirements were tested in this way.

The implemented system meets the critical and essential requirements defined for the system given the constraints put on the prototype. Several of the requirements were not satisfied due to the implementation being a prototype and not a final product. These constraints were mainly due to the time restrictions.

The following requirements were not met because the features they describe were not implemented at all:

- Requirements: 3.2.3, 3.2.4, 10, 10.1, 10.2, 10.3, 12, 12.1, 12.2, 12.3, 12.4, 13.1, 14. These requirements are all classified as desirable features and not essential.

- Requirements 7, 8 and 9 describe the open existing file function, print function and save function. These features are classified as essential to the final product but were not implemented in the prototype as they are not necessary in demonstrating the concept of the system.

Requirement 11 was failed due to problems with the diagram layout. The layout does not always work when some boxes are bigger than the standard size due to many variables or methods within one class. This will be explained in detail in the following section describing structural testing using specific test cases.

7.2 Structural Tests

Test data was designed to test all paths of the system which contribute to the output functions of the textual summary and the diagrams. The test data consists of sample source code files which demonstrate the different paths of the system involved in producing the source code summaries. The test data used can be found in Appendix VI.

The system’s response to the following aspects were tested using the source code test data:

- line comments indicated by //
- comments indicated by /*....*/
- extra long class, method and variable names that would not fit in diagram boxes
- inheritance relationships with classes that are not declared in the source code e.g. extending JPanel
- simple inheritance / aggregation diagrams
- inheritance / aggregation diagrams with many relationships
- simple inheritance / aggregation diagrams with classes with many methods or variables and therefore extra long diagram boxes
- inheritance / aggregation diagrams with many relationships and extra long diagram boxes
- importing more than one source code file
- closing files and then opening new files
**Results:**
The system produced correct outputs for most tests. The following test data produced problematic results:

Test8 – This source code produced an inheritance diagram with many relationships and also diagram boxes that were extended in length. Problems occurred with the layout of the diagram as shown in Figure 7.1.

![Figure 7.1 Diagram produced using test data 8](image1)

The problems occurred because of the extended box, not because of the many relationships. Test9 was the same source code except that a different class contained the extra variables. This data produced the correct results as shown in figure 7.2.

![Figure 7.2 Diagram produced using test data 9](image2)

Comparisons of figure 7.1 and 7.2 show where the problem occurs. In figure 7.1, the extended box holding the class named ‘extra’ should mean that the boxes on the two levels above it are drawn at a higher position. The top level does move higher but the level directly
above the extended box does not. This is a problem in the implementation of the class extendDiagram. Test 9 does not show the same problems as the order that the classes appear in the source code means that the class extendDiagram is not called but only extendDiagram2 is used.

Test10 – This source code demonstrates more classes with several variables that cause the boxes to increase in size. The test results in the same problems as shown in figure 7.1 above.

Test14 – This source code produces an aggregation diagram with many relationships and also extended diagram boxes. The layout problems occurring with this test data were similar to those with Test8 and Test10 however in this case the errors only occur with the connecting arrows as shown in figure 7.3.

![Figure 7.3 Diagram produced using test data 14](image)

### 7.3 Scenario Tests

The scenarios described in chapter 4 were carried out using source code from a past student project.

#### 7.3.1 Scenario1 and Scenario3

– The main task of both of these scenarios was to produce a suitable code representation that would refresh the user’s memory of the source code contents and structure.

The following results were produced when this scenario was carried out.

The errors that occurred in the structural tests were reproduced. Also some of the very large classes produced boxes in the diagrams that were so big that they went off the top of the screen. A further interesting result was noticed in that the system took a long time to open the files due to the large volume. This is due to a bug in the imported JFileChooser class. This has been fixed in Java 1.4.2 and Java 1.5 but machines with the older versions will experience delays.
Due to the errors in the diagram layout, the ‘class view’ was the most useful function in producing a clear summary of complicated code.

### 7.3.2 Scenario2

The tasks of this scenario were:

1. to produce a code summary which could be used to explain the overall structure of a large piece of source code.
2. to remove any unwanted parts of the textual code summary produced and use this data to produce a diagram.
3. to add comments to the diagram to aid explanations and discussions of the source code.
4. to save a file produced by the system.

**Results:**

1. Producing a code summary resulted in the same errors that have been mentioned previously. The textual summary was more useful than the diagrammatic summary due to the layout problems with the diagrams.
2. The system allowed the user to remove parts of the textual summary and successfully used the new data to produce diagrams.
3. Comments could be added about each class in the diagrams. The comments were stored when the comment boxes were closed and could be retrieved easily.
4. The save function of the system has not been implemented.

### 7.4 User tests

The goal of the user tests was to evaluate the level of usability of the system by analysing what users do and how long they spend on each task. Five student users participated in the tests. Users were asked to think aloud whilst using the system. Any major problems were recorded as critical incidents. A (negative) critical incident is an event or occurrence observed whilst performing a task that is likely to be an indicator of one or more usability problems.

Users were first asked to carry out a task based on the scenarios described in chapter 4. Samples of the test data from the structural tests were used. The users were given a set of instructions based on the scenarios. For example:

- *Import the file named Textfile4 in the folder named Project.*
- *In the class view, delete one variable from the class named “extra” and the method from the class named “Nicola”.*
- *Create an inheritance diagram.*
- *Add comments to the classes named “Louzado” and “more”.*

Users were then asked to use the system with their own Java code.

**Critical Incidents:**

One user clicked off the system whilst the ‘Import File’ FileChooser dialog box was open. This caused the FileChooser box to close and the system to crash. Later tests revealed that this did not happen every time and that clicking off the system at any other time did not cause a problem.
When the ‘class view’ textual summary was displayed, the tree was not expanded and some users first thought that this meant that there was no data. However, all users tried clicking on the class folder anyway which expanded the tree and revealed the classes.

When deleting data from the textual summary, some users tried to edit the data before reading the instructions at the top of the frame. They tried pressing delete which had no effect. After reading the instructions the data was edited successfully. All users tried to remove the actual tree node as well as the text. This function has not been implemented as it allows users to replace data if it has been removed by mistake. Being able to remove nodes would result in a tidier tree after the modifications.

One user used the help menu when trying to create a diagram. This explained how to complete the task but then caused further issues. The user clicked the diagram button while the help frame was still open. This meant that the diagram was created but could not be seen as it was behind the help frame. There was no indication that a diagram had been created and so the button was clicked again. Eventually the help frame was closed to reveal the diagram. This issue has been resolved by changing the size of the ‘help’ frame that is created when the ‘help’ button is pressed. The frame is now smaller and less intrusive so that the background frames can still be seen.

When using their own source code data, expected errors occurred that have been described earlier in this chapter. No extra issues were found.

Requirement 17 stated that users shall require a maximum of 15 minutes training before becoming a competent user. The user tests revealed that no training was required for the system, and users were very quick to understand both how to carry out tasks and how to work with the representations produced by the system.
8. Evaluation and Conclusions

The implemented system meets the critical and essential requirements defined for the system given the constraints put on the prototype that was implemented. It also meets some requirements that were classified as desirable features. There are a number of additions and improvements that could be made to the system which will form future work for the project.

8.1 Evaluation of the implemented system

The original aim of this project was to design and implement a tool for student projects which can produce simple documentation by the reverse engineering of source code. The main purpose of the system was two-fold:

1. To aid the communication between computer science students and their project supervisors. The summaries produced by the system could be used to aid explanations and discussions in project meetings.
2. To help students understand the structure of their own source code.

The system satisfies its purpose with the functionality to produce two types of software visualisations; a textual summary displayed in a tree structure, and diagrammatic representations.

During the project the original purposes of the system were build upon iteratively until a final requirements specification was produced in section 5.4 where the following were defined as critical requirements and therefore imperative to satisfying the main goals of the system.

1. The system shall aid program understanding by abstracting information from the source files.
2. The system shall take source code as input.
3. The system shall produce multiple representations of source code as output.
13. The system shall allow integrated annotation the diagrams.
17. Usability: The system shall be designed around a clear user-centred conception of the process and thereby minimize the need for formal training. End-user validated task-based scenarios shall be employed as the mechanism to secure this clear conception. Users shall require a maximum of 15 minutes training before becoming a competent user.

The tests carried out in chapter 7 proved that all critical requirements were successfully met and therefore the resulting system is a user-centered system that allows the simultaneous review of source code using multiple abstractions and integrated annotation of the diagrams.

The multiple representations create more opportunity to develop a good understanding of the problem. As stated by Petre et al (1998) ‘metal transference between representations forces reflection beyond the boundaries and details of the first representation and an anticipation of the correspondences in the second’. This deeper level of analysis can reveal gaps in understanding that may have otherwise been missed. Also the availability of textual and
diagrammatical representations offer the user a choice, depending on personal preference and suitability to the problem.

The user is able to modify the textual summary before a diagram is created from it. This was classified as a desirable requirement but was implemented as an extension of the basic system. This feature gives the user the ability to control their focus. Sections of the problem can be set aside and analysed separately as well as allowing the removal of irrelevant data.

The functionality that allows annotation about both the whole project or about each class allows further insight into their use, significance and limitations. It offers the chance to explain the architecture, design trade-offs, engineering constraints and the application domain. An added benefit occurs in the scenario of a student project meeting, as it allows the opportunity to note down any important comments made in the discussion about the project or particular classes.

The system was produced with relatively few problems and does satisfy the purposes listed above. The biggest problem that remains in the system is with the layout problem when the diagram includes boxes of an extended length. If this issue was corrected then the resulting system would be successful.

### 8.2 Future Work

The process of testing and evaluation has identified several possible improvements and extensions on the implemented system.

The requirements that describe system features that are not met due to the project constraints could all be added with extra time. This would mainly involve adding features that are categorised as desirable requirements. These features include the functionality to:

- produce a diagram showing inheritance and aggregation relationships together.
- produce a method diagram.
- save files produced by the system.
- open files produced by the system.
- print the code representations produced by the system.
- modify the diagrams – drag ‘n’ drop, add and remove arrows, delete blocks and resize the diagram.
- view the original source code within the system.

The only requirement that was not met due to problems with the implementation was requirement 11, stating that the diagrams produced must be of a suitable quality to be included in project documentation. This was due to problems with the diagram layout when class boxes contained extra information and so the box lengths were extended.

Currently the system allows multiple ‘class view’ frames and multiple diagram frames to be viewed simultaneously. This does allow flexibility with the files that the user can open and view but can become confusing. Another problem can occur if two ‘class views’ have been created using different sets of source code files. If the first is being viewed whilst the second is minimised, a diagram produced will be of the data from the last ‘class view’ that was created, which is not the one that is currently open. This is not the normal expected behaviour. An improvement would be to always produce a diagram of the ‘class view’ that is
currently open. Alternatively, the ‘class view’ frame and the diagram frame could be linked or could be in the same frame so that they would have to be opened, minimised and closed at the same time.

A limitation of the system is that it only recognises variables of primitive types. It was stated that the system would only consider newly written code and not code from imported libraries covering variables such as JPanel. However the system does not acknowledge new objects that are defined within the files, for example, when a class object is instantiated from another class. Further work could include extending the parser to search for and store any objects that are defined in the source code files. This would involve parsing the text more than once.

A major improvement on the system would be the functionality to link the different representations. Problems occur with multiple representations involving finding the relevant place in each one and understanding the correspondence between them. One way to solve this would be to use colour to link aspects in each representation. For example, if a class is selected and highlighted in the textual summary, the same class could be highlighted in the diagrammatic view. Similarly when comments are being written about a certain class using the comments boxes, that class could be highlighted in both the textual and diagrammatic representations to focus the users attention.

Currently the system allows the user to open a number of files at once and creates a textual summary of them in the class view. If the user wants to add another file to the class view, they must close the existing summary and import all the files again plus the extra ones that had been missed. An improvement to the system would be to allow the user to add files to the existing textual summary.

Sometimes very tall boxes extend past the top of the screen when a diagram is produced. This could be overcome if the system recognised when this happened and responded by repositioning and repainting the diagram.

The JTree display of the textual summary could be improved. When the tree is displayed, an enhancement would be to show an expanded view of the tree so that the list of class names is visible immediately. This would remove any confusion over whether there are class details. Further improvements could be made to the editing function of the tree. Ideally, the entire tree node would be removed if the text on that node has been deleted. This would make the resulting tree neater and more aesthetically pleasing. It would remove the possibility for recovery if data was deleted by mistake, however this could be overcome by including the functionality to add new nodes. This would also mean that the user could add details that have not yet been implemented in the source code. The textual summary could also benefit from the use of colour to reinforce the differences between the nodes containing class, method and variable names.

The comments boxes have been designed so that only one box can appear on the screen at a time. This is to maintain some control of the layout of the frames on screen and to avoid cluttering. It does however prevent the comments of different classes being viewed simultaneously and compared.

Another option for the comments boxes would be to add some default headings to prompt the user to add comments about certain aspects of the class. The headings could include: author,
date written, the name of the project that it was originally written for, design trade-offs and known issues with the code.

An extension of the system would be to provide the functionality to open, edit and compile source code files. This would extend the system to an integrated development environment which was not the original aim of the project but could be useful. This feature would mean that if the textual summary or the diagram revealed that the source code was not correct, then the it could be amended from within the system.

An alternative method that could have been used in the implementation is the use of the Java.lang.reflect package instead of implementing a source code parser. The main use of reflection is to allow 'an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program' (McCluskey, 1998). This would normally involve adding code within the actual classes to enable them to introspect upon themselves, which is not desirable in an analysis tool. However the package could be manipulated to retrieve information about the fields, methods, and constructors of the classes or perhaps copies of the classes. If this was successful, it could make the implementation of the source code parser module tidier and easier to understand.
9. Bibliography


10. Appendices
Appendix I: Paper Prototypes

The following designs were used for the paper prototypes for the user studies described in chapter 5. The menus at the bottom on the page correspond to the menu bar at the top of the screen.

Project Aid Technologies

Welcome to Project Aid!

Project Aid is a system to support the process of code comprehension. It can be used to produce diagrams to summarize source code for the purpose of aiding explanations and understanding of the code.

To begin a new project, click on the FILE menu above.

NEW enables you to import a new source code file. The system will produce a simple summary of the code.

The DIAGRAM menu can then be used to produce relevant diagrams.
Although this screen uses colour, for the user studies it was printed in black and white to maintain an un-polished look.
Class: main
Method: main (String())

Class: myClass1
Extends: JPanel
Variable: int i
Variable: String text
Method: myMethod1(String)

Class: myClass2
Implements: ActionListener
Variable: int counter
Method: actionPerformed(ActionEvent)
Class: main
Method: main (String())

Class: myClass1
Extends: JPanel
Variable: int i
Variable: String text
Method: myMethod1(String)

Class: myClass2
Implements: ActionListener
Variable: int counter
Method: actionPerformed(ActionEvent)

<table>
<thead>
<tr>
<th>FILE</th>
<th>DIAGRAM</th>
<th>HELP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLASS DIAGRAM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>METHOD DIAGRAM</td>
<td></td>
</tr>
</tbody>
</table>

- **main**
  - main()

- **JPanel**

- **myClass2**
  - int counter
  - actionPerformed(ActionEvent)

- **myClass1**
  - int i
  - String text
  - MyMethod1(String)
Appendix II: Aggregation Diagrams

The following diagrams describe the methods used to create an aggregation diagram.

The overall method to create an aggregation diagram is split into two methods:

- Aggregation Diagram method – The main diagram method. For each class, this method asks the questions: “Does this class have any aggregate components?” This groups together all classes that are connected by an aggregation relationship.
- Aggregation Diagram2 method – this is called if the answer to “Does this class have any aggregate components?” is yes.

![Process Model for Aggregate Diagram Method](image1)

Figure A2.1 Process Model for Aggregate Diagram Method

![Process Model for Aggregate Diagram2 Method](image2)

Figure A2.2 Process Model for Aggregate Diagram2 Method
# Appendix III: Class Descriptions

This appendix contains a brief explanation of each class used in the implementation of the system.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Calls Interface()</td>
</tr>
<tr>
<td>Interface</td>
<td>The main JFrame for the system. It uses a JDesktop pane and JInternalFrames to create a multiple document interface.</td>
</tr>
<tr>
<td>ActionEventHandler</td>
<td>Inner class of Interface. This handles all events occurring on the interface and calls the appropriate methods in response.</td>
</tr>
<tr>
<td>HelpText</td>
<td>Adds the text to the JTextFrame that displays the help information.</td>
</tr>
<tr>
<td>Info</td>
<td>Initialises the class data array and the classNumber counter.</td>
</tr>
<tr>
<td>fileOpener</td>
<td>Uses a JFileChooser to allow the user to select files to import. Uses the file filter in class Filter so that only Java files are displayed. Opens the files and create a String of all text from the files.</td>
</tr>
<tr>
<td>Filter</td>
<td>A file filter to only allow Java files.</td>
</tr>
<tr>
<td>Tokeniser</td>
<td>Removes comments from the text. Searches for the word class and calls classData every time it is found, creating a new object in the classData array time.</td>
</tr>
<tr>
<td>Comments</td>
<td>Contains method to detect the Java comment escape characters.</td>
</tr>
<tr>
<td>classData</td>
<td>Stores all the data about each class in objects in an array. Contains two constructors. The first is used to parser / extract data from the source code files. The second is used to store the data from the JTree which could have been modified by the user.</td>
</tr>
<tr>
<td>printTree</td>
<td>Puts the data from the classData array into a JTree.</td>
</tr>
<tr>
<td>inherDiagram</td>
<td>Creates an inheritance diagram using paint(Graphics g). Contain a method to return whether or not a mouse click is on the comments icon on a class box. Also has a method to return the class name of the box that was clicked on.</td>
</tr>
<tr>
<td>extendDiagram</td>
<td>Also used to create an inheritance diagram.</td>
</tr>
<tr>
<td>extendDiagram2</td>
<td>Also used to create an inheritance diagram.</td>
</tr>
<tr>
<td>AggDiagram</td>
<td>Creates an aggregation diagram using paint(Graphics g). Contain a method to return whether or not a mouse click is on the comments icon on a class box. Also has a method to return the class name of the box that was clicked on.</td>
</tr>
<tr>
<td>AggDiagram2</td>
<td>Also used to create an aggregation diagram</td>
</tr>
<tr>
<td>Positions</td>
<td>Used to hold details about the co-ordinates of the boxes in the diagram.</td>
</tr>
<tr>
<td>createBox</td>
<td>Draws the boxes on the diagrams including all the data about each class. Contains a method to return the height of the box.</td>
</tr>
<tr>
<td>connectLine</td>
<td>Draws the connecting lines in the diagrams. Draws either a triangle or diamond arrow head depending on the type of diagram being drawn.</td>
</tr>
<tr>
<td>commentsSquare</td>
<td>Stores details about the comments icon on the class boxes. Stores the class name and the co-ordinates of the icon. This is used to check if a mouse event occurred on an icon.</td>
</tr>
<tr>
<td>saveFile</td>
<td>Displays a JFileChooser dialog box but the save function is not implemented in the prototype.</td>
</tr>
</tbody>
</table>
Appendix IV: Screen Shots of the Implemented System

The opening screen displays the help instructions. At later stages in the system, the instructions can be accessed from the help button which displays the same text in a smaller frame.

![Figure A4.1 Screen shot 1](image)

Project Aid is a system to support the process of code comprehension. It can be used to produce diagrams to summarize source code for the purpose of aiding explanations and understanding of the code.

To begin a new project, click on the FILE menu above.

On the file menu, IMPORT FILE enables you to import all required source code files. The system will the produce a simple summary of the code.

The summarized code can be edited by either triple clicking on the text or by pressing F2. Pressing Enter confirms the change.

The DIAGRAM menu can then be used to produce relevant graphical representations.

Click on the square in the top right hand corner of each class box to add comments.

These instructions are available at any time by clicking the HELP button.

When ‘Import File’ is clicked on the File menu, the following box is displayed. Only Java files are displayed unless the file type is changed.

![Figure A4.2 Screen shot 2](image)
When a file is imported, a textual summary is displayed in a tree structure. The text contained in the tree can be edited.

Figure A4.3 Screen shot 3

A diagram can be created to display the inheritance relationships between the classes. The diagram frame includes space to add comments about the overall project.

Figure A4.4 Screen shot 4
By clicking on the icon in the corner of a class box, comments can be added about the chosen class.

A diagram can be created to display the aggregation relationships between the classes.
At any point the help instructions can be accessed.

Figure A4.7 Screen shot 7

To begin a new project, click on the FILE menu above.

On the file menu, IMPORT FILE enables you to import all required source code files. The system will then produce a simple summary of the code.

The summarized code can be edited by either triple-clicking on the text or by pressing F3. Pressing Enter confirms the change.

The DIAGRAM menu can then be used to produce relevant graphical representations.

Click on the square in the top right hand corner of each class box to add comments.

These instructions are available at any time by clicking the HELP button.
Appendix V: Functional Tests

The requirements have been put in table format to enable the tests to be carried out in a controlled order and to ensure that all tests are completed. The req. column shows which requirements are being tested and corresponds to the requirements numbers in the requirements document. Where appropriate both functional and non-functional requirements were tested in this way.

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Req.</th>
<th>Description</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Data Input</td>
<td>2</td>
<td>System accepts source code as input</td>
<td>Pass</td>
</tr>
<tr>
<td>2.</td>
<td>Data Input 2</td>
<td>2.1</td>
<td>User can specify input file location</td>
<td>Pass</td>
</tr>
<tr>
<td>3.</td>
<td>Data Input 3</td>
<td>2.2</td>
<td>Data in files is not modified</td>
<td>Pass</td>
</tr>
<tr>
<td>4.</td>
<td>Multiple representations</td>
<td>3</td>
<td>System can output multiple representations of the source code</td>
<td>Pass</td>
</tr>
<tr>
<td>5.</td>
<td>Textual summary</td>
<td>3.1</td>
<td>The system outputs a textual summary of the source code.</td>
<td>Pass</td>
</tr>
<tr>
<td>6.</td>
<td>Textual summary contents</td>
<td>3.1.1</td>
<td>The textual summary includes classes, methods and variables.</td>
<td>Pass</td>
</tr>
<tr>
<td>7.</td>
<td>Textual summary contents</td>
<td>3.1.2</td>
<td>The textual summary includes inheritance and aggregation details.</td>
<td>Pass</td>
</tr>
<tr>
<td>8.</td>
<td>Textual summary format</td>
<td>3.1.2</td>
<td>The textual summary maintains clarity with large amounts of data.</td>
<td>Pass</td>
</tr>
<tr>
<td>9.</td>
<td>Diagram representation</td>
<td>3.2</td>
<td>The system outputs a diagram representation of the source code.</td>
<td>Pass</td>
</tr>
<tr>
<td>10.</td>
<td>Inheritance diagram</td>
<td>3.2.1</td>
<td>The system can produce an inheritance diagram.</td>
<td>Pass</td>
</tr>
<tr>
<td>11.</td>
<td>Aggregation diagram</td>
<td>3.2.2</td>
<td>The system can produce an aggregation diagram</td>
<td>Pass</td>
</tr>
<tr>
<td>12.</td>
<td>Inheritance and aggregation diagram</td>
<td>3.2.3</td>
<td>The system can produce a diagram showing aggregation and inheritance.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>13.</td>
<td>Method diagram</td>
<td>3.2.4</td>
<td>The system can produce a diagram of the method calling sequence in the source code.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>14.</td>
<td>Diagram contents</td>
<td>3.2.5</td>
<td>Diagrams include classes, methods and variables.</td>
<td>Pass</td>
</tr>
<tr>
<td>15.</td>
<td>Simultaneous Display</td>
<td>4</td>
<td>The textual summary and diagrams can be viewed simultaneously.</td>
<td>Pass</td>
</tr>
<tr>
<td>16.</td>
<td>Modifying textual summary</td>
<td>5</td>
<td>User can remove unwanted parts of the textual summary before it is put into a diagram.</td>
<td>Pass</td>
</tr>
<tr>
<td>17.</td>
<td>Class blocks</td>
<td>6</td>
<td>Class diagrams consist of class blocks.</td>
<td>Pass</td>
</tr>
<tr>
<td>No.</td>
<td>Name</td>
<td>Req.</td>
<td>Description</td>
<td>Evaluation</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------------</td>
<td>------</td>
<td>----------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>18</td>
<td>Class diagram contents</td>
<td>6.1</td>
<td>Diagrams include all classes from the textual summary.</td>
<td>Pass</td>
</tr>
<tr>
<td>19</td>
<td>Class variables</td>
<td>6.2</td>
<td>Class blocks include the variables of that class.</td>
<td>Pass</td>
</tr>
<tr>
<td>20</td>
<td>Class methods</td>
<td>6.3</td>
<td>Class blocks include the methods of that class.</td>
<td>Pass</td>
</tr>
<tr>
<td>21</td>
<td>Class arrows</td>
<td>6.4</td>
<td>Arrows on the diagrams indicate relationships.</td>
<td>Pass</td>
</tr>
<tr>
<td>22</td>
<td>Print function</td>
<td>7</td>
<td>Users can print the summaries produced</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>23</td>
<td>Existing files</td>
<td>8</td>
<td>Users can open existing files produced by the system.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>24</td>
<td>Save function</td>
<td>9</td>
<td>Files produced by the system can be saved to the computer hard drive or to disk.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>25</td>
<td>Method blocks</td>
<td>10</td>
<td>Method diagrams consist of rectangular method blocks.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>26</td>
<td>Method diagram contents</td>
<td>10.1</td>
<td>Diagrams include all classes from the textual summary.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>27</td>
<td>Method variables</td>
<td>10.2</td>
<td>Method blocks include the variables of that method.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>28</td>
<td>Method arrows</td>
<td>10.3</td>
<td>Arrows on the diagram indicate the method calling sequence</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>29</td>
<td>Diagram quality</td>
<td>11</td>
<td>Diagrams are of a suitable quality to be included in project documentation.</td>
<td>Fail</td>
</tr>
<tr>
<td>30</td>
<td>Modifying diagrams</td>
<td>12</td>
<td>Users can modify diagrams.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>31</td>
<td>Drag ‘n’ drop</td>
<td>12.1</td>
<td>User can drag ‘n’ drop diagram blocks.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>32</td>
<td>Adding arrows</td>
<td>12.2</td>
<td>Users can add or remove connecting arrows to the diagrams.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>33</td>
<td>Deleting blocks</td>
<td>12.3</td>
<td>Users can delete diagram blocks.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>34</td>
<td>Resizing the diagram</td>
<td>12.4</td>
<td>Users can resize the diagrams.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>35</td>
<td>Comments</td>
<td>13</td>
<td>Users can annotate the diagram with comments.</td>
<td>Pass</td>
</tr>
<tr>
<td>36</td>
<td>System comments</td>
<td>13.1</td>
<td>Users can add comments regarding the system as a whole.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>37</td>
<td>Class comments</td>
<td>13.2</td>
<td>Users can add comments about individual classes.</td>
<td>Pass</td>
</tr>
<tr>
<td>38</td>
<td>Access to comments</td>
<td>13.3</td>
<td>Users can access comments about each class individually.</td>
<td>Pass</td>
</tr>
<tr>
<td>39</td>
<td>Viewing original source code</td>
<td>14</td>
<td>The system displays the original source code alongside the code summary.</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>40</td>
<td>Button positions</td>
<td>17.2.2.1</td>
<td>Buttons always have the same position, colours and font.</td>
<td>Pass</td>
</tr>
<tr>
<td>41</td>
<td>Screen layout</td>
<td>17.2.2.2</td>
<td>Screens and message boxes always have the same layout.</td>
<td>Pass</td>
</tr>
<tr>
<td>42</td>
<td>Button control and effect</td>
<td>17.3</td>
<td>Buttons should display a clear relationship between control and effect.</td>
<td>Pass</td>
</tr>
<tr>
<td>43</td>
<td>Recovery – input files</td>
<td>17.4.1</td>
<td>Users can reselect different source code files if needed.</td>
<td>Pass</td>
</tr>
<tr>
<td>No.</td>
<td>Name</td>
<td>Req.</td>
<td>Description</td>
<td>Evaluation</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>44.</td>
<td>Recovery – textual summary</td>
<td>17.4.2</td>
<td>Users can close a textual summary if it is not required.</td>
<td>Pass</td>
</tr>
<tr>
<td>45.</td>
<td>Recovery - diagram</td>
<td>17.4.3</td>
<td>Users can close a diagram if it is not required.</td>
<td>Pass</td>
</tr>
<tr>
<td>46.</td>
<td>Icons</td>
<td>17.7</td>
<td>Icons are used if appropriate</td>
<td>Pass</td>
</tr>
<tr>
<td>47.</td>
<td>Help system</td>
<td>17.8</td>
<td>System includes a simple help system</td>
<td>Pass</td>
</tr>
<tr>
<td>48.</td>
<td>Textual summary speed 1</td>
<td>18.1</td>
<td>Textual summaries are produced within 10 seconds of taking a source code file as input.</td>
<td>Pass</td>
</tr>
<tr>
<td>49.</td>
<td>Textual summary speed 2</td>
<td>18.2</td>
<td>Textual summaries are produced within 3 seconds of taking a source code file as input.</td>
<td>Pass</td>
</tr>
<tr>
<td>50.</td>
<td>Diagram speed 1</td>
<td>18.3</td>
<td>Diagrams are produced within 10 seconds of the request.</td>
<td>Pass</td>
</tr>
<tr>
<td>51.</td>
<td>Diagram speed 2</td>
<td>18.4</td>
<td>Diagrams are produced within 3 seconds of the request.</td>
<td>Pass</td>
</tr>
<tr>
<td>52.</td>
<td>Information accuracy</td>
<td>19.1</td>
<td>The source code representations produced are accurate</td>
<td>Pass</td>
</tr>
</tbody>
</table>
Appendix VI: Test Data

The test cases are not all compiled code which is the assumption made for the input data of the system, however this is not necessary to test the different paths of the system. Some files contain more than one class which would normally be separated into different files but have been put together for the simplicity of the test.

Test 1: One class with two variables and one method. This fits in a standard size class box when drawn.
```java
class test1
{
    int counter = 1;
    String letters = "";

    public void method(String s)
    {
    }
}
```

Test 2a: Contains comments indicated by the escape characters: /*.....*/
```java
/* this class myclass should not appear */
class/*extra class test */commentsTest/*these words should not appear */
/* this class myclass should not appear */
    int number
/* this class myclass should not appear */
}/* this class myclass should not appear */
```

Test 2b: Contains comments indicated by the escape characters: //
```java
//this class myclass should not appear
class commentsTest//this should not appear
{//this class myclass should not appear
//this class myclass should not appear
//this class myclass should not appear
//this class myclass should not appear
```

Test 3: Testing whether a class box increases to the correct size and also tests the result when names have too many characters to fit in the diagram box
```java
public class myClassNameThatWillNotFitInABox
{
    int var = 10;
    int myVeryLongVariableName = 0;
    String nicola = "nicola";
    String words ;
    char letter;

    public void myVeryLongMethodName()
    {
        int my;
    }

    public int main(String args)
    {
    }
}
```

Test 4: Demonstrates a very simple inheritance diagram
```java
public class Test4 extends JPanel
{
}
```

Test 5: Demonstrates an inheritance diagram
```java
public class Gabrielle extends JPanel
{
}
```
```java
public class Nicola extends JPanel
```
{ }
public class Louzado extends JPanel
{
}

**Test 6: Demonstrates a complicated inheritance diagram**

public class Nicola
{
}

public class Gabrielle extends Nicola
{
}

public class Louzado extends Nicola
{
}

public class test extends Nicola
{
}

public class extra extends Louzado
{
}

public class more extends Louzado
{
}

public class another extends Louzado
{
}

public class an extends another
{
}

**Test 7: This test demonstrates when classes appear in a non logical order. For example class extra extends class Louzado and is declared before class Louzado. This demonstrates the path through the method extendDiagram.**

public class extra extends Louzado
{
}

public class an extends another
{
}

public class Nicola
{
}

public class Gabrielle extends Nicola
{
}

public class Louzado extends Nicola
{
}

public class more extends Louzado
{
public class JP1 extends JPanel
{
}

public class JP2 extends JPanel
{
}

public class another extends Louzado
{
}

public class chaos extends JP1
{
}

class ultimate extends JP1
{
}

public class test extends Nicola
{
}

Test 8: This tests a complicated inheritance diagram with variables that cause some boxes to increase in size.

public class Nicola
{
}

class Gabrielle extends Nicola
{
}

public class Louzado extends Nicola
{
}

public class test extends Nicola
{
}

public class extra extends Louzado
{
    int 1
    int 2
    int 3
    int 4
    int 5
    int 6
    int 7
    int 8
}

public class more extends Louzado
{
}

public class another extends Louzado
{
public class an extends another
{
}

Test 9: This tests a complicated inheritance diagram with variables that cause some
test boxes to increase in size.
public class Nicola
{
}

public class Gabrielle extends Nicola
{
}

public class Louzado extends Nicola
{
    int 1
    int 2
    int 3
    int 4
    int 5
    int 6
    int 7
    int 8
}

public class test extends Nicola
{
}

public class extra extends Louzado
{
}

public class more extends Louzado
{
}

public class another extends Louzado
{
}

public class an extends another
{
}

Test 10: This tests a complicated inheritance diagram with variables that cause some
test boxes to increase in size.
public class Nicola
{
}

public class Gabrielle extends Nicola
{
    int 1
    int 2
    int 3
    int 4
    int 5
    int 6
    int 7
    int 8
}

public class Louzado extends Nicola
{
public class test extends Nicola
{
}
public class extra extends Louzado
{
}
public class more extends Louzado
{
}
public class another extends Louzado
{
}
public class an extends another
{
}

Test 11: Demonstrates a simple aggregation diagram

```java
class aggregateParent
{
    class component(
        method()
    )
}
```

Test 12: Demonstrates an aggregation diagram

```java
class Nicola
{
    class Nicky
    {
    }
    class Gabrielle
    {
    }
    class Louzado
    {
    }
}
```

Test 13: Demonstrates a complicated inheritance diagram

```java
class parent
{
    public class middle
    {
        int myVar = 0;
        class aggclass
        {
        }
        class aggclass2
        {
            class extension
            {
            }
        }
    }
}
String myString = "Nicola";

class aggclass3
{
    class trouble
    {
    }
    class trouble2
    {
    }
}
public class side
{
    class blurrr
    {
    }
}

class Nicola
{
    class Nicky
    {
    }
    class Gabrielle
    {
    }
    class Louzado
    {
    }
}

Test 14: Demonstrates a complicated inheritance diagram with boxes that have to increase in size.
class disaster
{
}
class parent
{
    public class middle
    {
        int myVar = 0;
        int 1
        int 2
        int 3
        int 4
        int 5
        int 6
        int 7
        int 8

        class aggclass
        {
        }
        class aggclass2
        {
            class extension
            {
            }
        }
    }
}

String myString = "Nicola";
class aggclass3
{
```java
public class side
{
    class inside
    {
    }
}

class Nicola
{
    class Nicky
    {
    }
    class Gabrielle
    {
    }
    class Louzado
    {
    }
}
```