The University of Nottingham
Department of Computer Science and Information Technology

An Automated Marking System for Graphical User Interfaces

Geoffrey Richard Gray, BSc.

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy

February 2008

Abstract

This research investigates the feasibility and effectiveness of assessing students programming solutions to Graphical User Interface exercises in an automated fashion. Automated marking systems ease the burden on the staff involved in running a course and allow students to get results and feedback in a timely fashion. Several automated marking systems exist but are currently unable to mark GUIs. The inherent complexity of GUIs and the need for aesthetic analysis has rendered GUIs beyond the scope of most marking systems.

The marking approach described in this thesis implements a number of novel concepts. By exploiting language design properties such as the hierarchical relationship between components, it was possible to develop a framework capable of testing and marking students' GUI programs. Introspectively analysing the interface enables the marking system to obtain access to the intrinsic elements contained within the GUI. Once access has been obtained, the tests can be performed on the actual interface components themselves rather than a mere representation. GUI assessment is more than functional testing, aesthetics play a major role in the creation of an interface. Existing aesthetic metrics do not provide the analytical capabilities required due to their failure to include colour. The distractive effects that colours have were quantified and incorporated into the metrics.

The results of the dynamic and aesthetic testing show that through the implementation of the novel components detailed, the creation of a GUI marking system is feasible and its marking both consistent and effective. The design enables the system to return results in a timely fashion and the effects that colour has can be seen in the results of basic aesthetic testing.

Acknowledgements

"Remember, we all stumble, every one of us. That's why it's a comfort to go hand in hand." - Emily Kimbrough

There are those who deserve to be thanked for all their inspiration, advice, love and support. Without such friends, the completion of this thesis would have been an infinitely more difficult task. However, there is no need for a list of names as those to whom I am indebted already know who they are and that I owe them everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"The unexamined life is not worth living" - Socrates

Computer programming courses are changing. The days of command line driven programs are almost over. The use of Graphical User Interfaces (GUIs) is fast becoming universal [Mye95]. Previously the term Human Computer Interaction (HCI) was almost unknown, with the subject being minimally taught to students. HCI is about more than just technology, it involves understanding users' goals, capabilities and what humans find aesthetically pleasing [PA06]. With the increasing ubiquity of the GUI, the old course model is having to be altered to include HCI in the curriculum. The principal concepts emerging from HCI that need to be taught to students are generally agreed upon [Gre96] these include usability principles and heuristics. The ability to correctly test and assess interfaces is essential to any interface programmer and stems from adequate teaching. Without these skills and their implementation, the results, although dramatic, can be fatal [MHC96].

With changes to the course structure comes additional marking responsibilities. Widely regarded as the least interesting part of academia [FHST01], marking is a "necessary evil". Several automated assessment solutions have been created to ease the problem, but none of these have any GUI marking or testing capabilities. There are commercial programs which enable the testing of GUIs normally through record and playback functionality which itself is error prone. When marking systems are created, the focus is often put on the areas of assessment that are easy to perform [Jac00]. GUI marking is one area so far avoided due to their complexity. This complexity stems from the fact that their input / output is not linear as with command line interfaces. Graphical User Interfaces allow for information to be disseminated in a variety of ways, simultaneously. They also utilise advanced Object-Oriented programming techniques to enable complex structures to be created.

The automation of the marking process significantly reduces the time needed to return a mark for each submission [JL98b] and increases the accuracy and consistency of the mark. The additional time it takes to analyse a GUI by hand due to their inherent complexity makes GUIs an ideal candidate for automated marking. However, it is also this complexity that has kept people from tackling the problem. This thesis shows that the automation of the marking process for a Graphical User Interface is not only feasible, but also that it is both efficient and effective.

This chapter introduces the concepts behind the teaching of computer programming and the automation of marking. It also presents an overview of the contents and

structure of the remainder of the thesis.

1.1 Computers in Education

Computers have become such an intrinsic part of assessment that they are now the subject of several areas of research. Known broadly as Learning Technology, it is described as the application of technology for the enhancement of teaching, learning and assessment. Contained within learning technology are two areas of research, Computer Assisted Assessment and Computer Based Assessment. Computer Assisted Assessment (CAA) is a term used to describe the use of computers to support assessment. Computer Based Assessment (CBA) refers to the use of computers to deliver, mark, score and analyse assessments [CE98a].

The approaches known as Computer Aided Assessment and Computer Based Assessment may not be new concepts but are more significant today than they were when the ideology was conceived. Through the use of CAA/CBA techniques, the automation of tasks previously performed manually not only ensures their consistency but also saves lecturers time [FHH+01]. This time can then be spent more productively elsewhere, for example improving the quality of the course. The use of automation to guarantee consistent marking, without influence through bias or tiredness, has long been a topic of research [Hol60]. Whilst considerable work has since been done along this vein, the marking of Graphical User Interfaces is still very much novel. Graphical Interfaces are more complex than standard command line based programs due to the infinite number of possible combinations of input and output [Bel01]. By allowing the components of a GUI to appear anywhere on screen it is impossible to predict where a component will be and what its functionality is, posing additional questions to any markers. With the commercial demands for "easy to use interfaces" they are also becoming increasingly common [Mye93]. All this places new demands on course conveners and the marking that needs to be done. The added complexity of the GUI increases the time costs of marking students' solutions.

1.1.1 Why Automate?

It is no revelation to say the larger the class size, the greater the resources required

to communicate the course effectively. Irrelevant to how many extra staff are employed to assist the course, there comes a limit when alterations need to be made to keep the course feasible [CS98, PS98]. When this limit is breached, students are no longer able to obtain a desired level of feedback [PP97]. Although face to face feedback may no longer be a feasible option, feedback can still be provided in an alternative fashion, namely electronically. This hinges upon changes being made to provide more efficient methods of course delivery and assessment [DH95]. Due to their programmable nature, computers are the obvious choice to automate certain parts of the assessment process. Numerous reasons [MMM04] have been suggested as to why academics want to use computers to assist the running of their course. These reasons include:

- To increase the frequency of assessment.
- To encourage students to practice their skills.
- To increase feedback to students and lecturers.
- To increase consistency.
- To reduce the marking burden.

Automation and the use of computers, in addition to the stated reasons, places more emphasis on students to take their learning into their own hands. This does not always go unopposed. Students are often reluctant to make alterations to their accustomed learning methods [AT95], even though computer based assessment has been shown to improve performance in summative assessments [CE98b]. If computer automation increases student ability, then the motivation exists to bring about complete automation should the opportunity arise. Computer programming is an ideal candidate for the automation of assessment. Whilst such systems do currently exist, they all have failings in that they are unable to test and mark Graphical User Interfaces.

The need for automation is clear, it also provides us with the following question: "Is it possible to automate the marking of Graphical User Interfaces without compromising the benefits afforded by the hand marking approach?". This question is to be answered through investigation into the makeup of Graphical User Interfaces and analysis of each of the problems that will be faced by a GUI marking system such as object recognition and test execution.

1.1.2 Teaching Programming

The teaching of programming is an important field in many disciplines throughout the academic spectrum. The learning of any language, programming or spoken, is a complex hands-on process requiring significant practice [Jen02]. Over the years, the methods used to teach programming to students have undergone several revisions [FBC+01, Jen98]. For example, changing students from passive to active recipients of the teaching by making the students participate more in teaching sessions. The demands being placed upon current programmers are but one cause. The inception of the Graphical User Interface and the advancement of programming languages have altered the skill sets employers are now looking for [Emi01]. No longer is it enough to just create a working program. The program must now be efficiently designed and be both ergonomic and aesthetically pleasing. However, it is not merely the methods that have been changed, but also the languages taught. Today's object oriented programming languages such as C++ [Str00] and Java [Mic05] provide both power and flexibility in a high level environment, thus making them obvious candidates for teaching to students [KKR95]. Adequate support and feedback along with a well structured course can reduce any problems that may arise from language complexities, as will be shown by the students' results in a later chapter.

General Teaching Practices

In order for students to become all round competent programmers, there are several distinct parts to the teaching of programming that need to be covered. Each topic is equally important, lack of knowledge in any one domain will significantly hinder a student's programming ability. For explanation purposes, the sphere of programming has been broken down into four distinct sections.

Syntax and Fundamentals

There are four schools of thought about how Object Oriented (OO) languages should be taught [YJ04]. The first is to teach it as a procedural language at the beginning, covering all procedural aspects e.g. data types, conditionals, functions / methods

before then explaining the concepts of objects. By the time the students have a grasp of the basic concepts they will have need to create functional objects. The second is to start off teaching Object Orientation, covering all other topics as they are required in the explanation of the OO programs. The third is GUI-first. This involves teaching students how classes and objects operate through the use of Graphical User Interfaces, before tackling the fundamentals of the language and Object Oriented programming. It has up to now not been proven that one approach yields better results and all approaches can be used. Finally there is the school of thought that the "objects first" approach should be taken even further and that student should be taught design patterns first instead [PPP06].

Before being able to write a working program, it is self-evident that a programmer must first learn the syntax and the fundamentals of the language in question. Returning to our foreign language metaphor, you cannot write a sentence without knowing how it is to be structured and how the words are spelt.

Whichever approach is being used, the syntax needs to be tackled early on in a programming course. It is an intrinsic part of learning to program, without it students will not be able to create a program that compiles. The syntax explains how to create legal sentences or in this instance, lines of code. Once completed, the semantics can be covered.

Graphical User Interfaces

After examining course descriptions for several university computer science departments[1], it became clear that there is a general pattern to the way in which GUI programming is taught. All courses offer a basic programming module in which an overview of the programming language is presented. As there is not sufficient time to cover every aspect of a language in the detail required, certain areas are bypassed. The overlooked topics often include distributed computing, graphics and security. Graphical User Interfaces are normally included as part of the basic programming course, with the instruction focused on the use of different components, event handling and basic layout concepts. It is evident that Graphical User Interface programming is a complex topic as it warrants an advanced course of its own in most institutions. The

[1] Including such institutions as Nottingham, Southampton and Stanford

advanced courses are occasionally of a general programming nature and tend to include topics such as layout techniques, java beans etc.

Testing and Debugging

The benefits of regular testing during the development of a computer program include allowing programmers to ensure they are meeting the specification and to locate errors early. However, the number of students[2] who think it reasonable to submit a solution without a thought for testing is significant. To encourage regular testing, it is a practice best nurtured at an early stage of programmer development in order for it to be ingrained in the programmer's actions. It is not the case that experienced programmers are unable to be taught to test and debug, they just need the desire to change [dJ05]. To have become an experienced programmer, it surely follows that high quality, working programs must have been produced. Without testing, such programs are unobtainable.

Testing is a two fold process and must be taught in conjunction with debugging [AEH05]. Debugging is the process of analysing faulty code in an attempt to locate the errors contained within. Debugging is a skill which programmers develop by making mistakes in the first instance and being able to recognise when they have recreated them. Novice programmers have not yet gained sufficient experience and therefore need to be taught basic methods for error identification and correction e.g. binary chop which involves commenting out sections of source code until the block containing the error is located. These techniques, however simplistic, form the basis of the techniques they will use later in their programming careers.

Design and Layout

There are several typographical conventions as far as programming is concerned [Gib95]. These recommendations include the length of identifiers, the amount of white space and the indentation of the source code. The conventions have but one aim, improved readability. This in turn leads to several other advantages, one of which is that tidy source code is an order of magnitude easier to debug than source code put together in a haphazard fashion. Making source code more readable will allow

[2]Based on experience as a demonstrator in a first year Java programming module

programmers new to a project to understand its makeup more readily, thus reducing the time needed for acclimatisation.

Whilst important, the layout only helps us read each line independently. The design of the program should provide a better indication of the program's aim. The hardest "programming" skill to master is that of design. Countless books have been written on this and other closely related topics, one of the most famous and instrumental of which being Design Patterns by the 'gang of four' [GHJV95]. Students must first be taught the basics of design before attempting to comprehend such topics as frameworks and patterns. To give students good foundations from which to proceed, all areas of Object Oriented design need to be covered, from procedural methods, through classes, to inheritance and polymorphism.

Design and layout does not only relate to the source code. When Graphical User Interfaces are concerned there is a new dimension that must be considered, namely the way in which the input and output is relayed to the user.

1.1.3 Assessing Students

Programming is a hands-on discipline and requires considerable practice. It is more valuable to students if insightful comments can be made on the programs they have written [PP97]. Therefore, a form of assessment is required, whether it be summative or formative. The structural nature of programming allows lecturers to split the teaching of programming into concepts, which can then be used as a basis for assessment e.g. conditionals, methods, inheritance.

Before continuing, a distinction that needs to be made between testing and assessment. Testing is a binary operation where the result is either right or wrong depending on whether the program passes or fails the tests which it is subjected to. Assessment should be a graded scale where marks can be awarded for partial correctness. Feedback is another important concept that assessment depends on, it is this that enables students to learn from any mistakes they have made.

Exams are the stalwart of the educational assessment arsenal. Programming gives you two main options when it comes to exams, a standard written exam or an online programming exam. The online exam is more suited to programming allowing students to both compile and test their solutions. It also allows students to write a

program in an environment they have experience with. Another assessment method is via the use of Multiple Choice Questions (MCQs). MCQs enable lecturers to cover several, if not all, details within a set conceptual area. A fine line exists, which when breached stop MCQs from testing knowledge and instead see how well students read the question or can learn obscure facts about the concepts being tested. Both exams and MCQs are expensive with regards to time needed to organise and run, however, there is another option, coursework. By no means is coursework a substitute for either MCQs or exams, they are important in their own right. MCQs and exams enable lecturers to assess individual rather than communal knowledge under restrictive exam conditions.

Regularly assessed coursework is a convenient way to measure student progress. Not only can questions be written relatively quickly compared to exams, students do not need to complete the exercises in exam conditions. Course conveners can also benefit from the regularity of the exercises. Through analysis of the results, lecturers are able to see where mistakes are made and whether the difficulties uncovered are general or concept related. The requirement of having to mark each solution written is where problems are encountered.

Marking Exercises

Traditional hand marking methods have their place in academia, unfortunately the marking of programming solutions is not one of them [AB99a]. Visually checking a program to determine whether it is syntactically and dynamically correct is a difficult and time consuming task. This is especially true when large numbers of scripts are involved. In order to set regular programming exercises, there needs to be a fast turn around with regards to the marking of solutions. This routinely involves the lecturers enlisting help, usually from postgraduates. Additional manpower reduces the time needed to process all scripts, but introduces consistency issues it also relies on the postgraduates hired to complete their marking competently and within a reasonable time frame. A form of automation is required to remove inconsistencies and improve the marking turn-around. Lecturers often took it upon themselves to write tools to speed the marking process along, automating such tasks as script collection. Over the years more tools were written, one such collection at the University of Nottingham eventually became Ceilidh [BBF+95, FHG96].

Marking exercises does not solely consist of analysing it functionally, although this must be a priority as a program that does not meet its specification is of no use. There are other concepts that must also be considered, these include source code typographical testing and more importantly aesthetic testing. Comprehensive aesthetic testing would include analysing layout choices and consistency across the interface.

1.2 Marking System Requirements

To determine whether a marking system is of value to an institution, a list of marking system requirements should first be collated. This can be aided through examination of an example of what is too be marked (see Figure [1.1]) and the desired output. The interface on the left was created as a model solution to a basic calculator exercise, where as the interface on the right has two flaws. The '7' button has been mislabeled as 'A' and the multiply button has the incorrect functionality and will subtract rather than multiply. The calculator examples shall be used throughout the thesis in an attempt to better explain certain concepts.

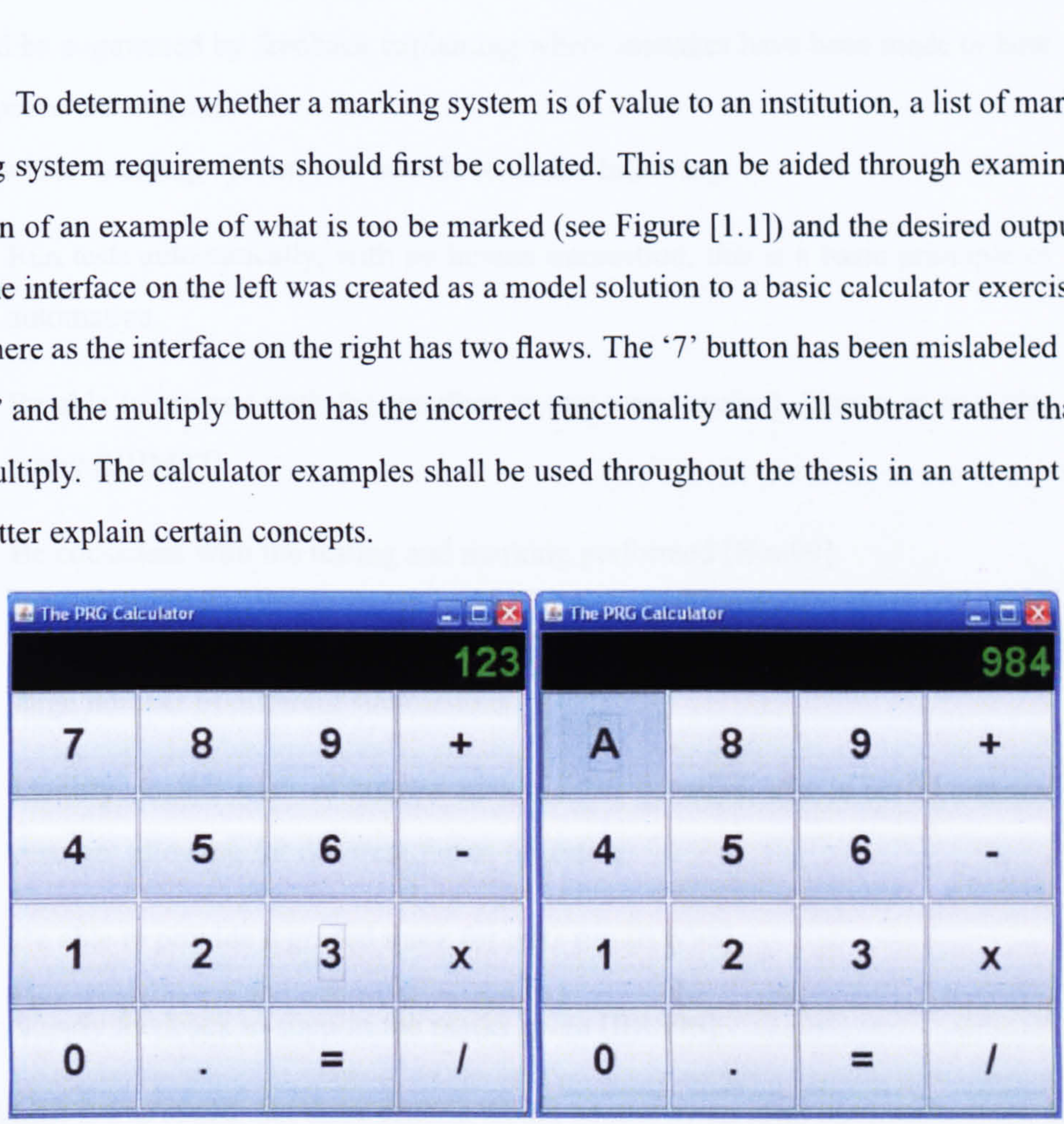

Figure 1.1: Example of interfaces that could require marking - Calculators

The interfaces shown in Figure [1.1] highlight some of the problems that need to be addressed by a marking system. There are often several ways in which the user can interact with the interface, be it through text entry, buttons or through the menus. Any marking system would need to be able to replicate this. There are also design issues to be considered. If a number of students are creating similar interfaces, it is unlikely that they will all be identical. A GUI marking system should be able to

identify components e.g. buttons in a variety of ways such as through the icon used or text displayed. The system needs to be able to take into account different layouts. For example, whilst it may be conventional to have buttons in the order 'new document', 'open document' and 'save' this need not be the case and any system must be able to cope with variations.

The tests need to be complimented by marks which correspond to the success or failure of the tests run. These should work on a graded scale so that the final mark obtained is not either 0% or 100%. In addition, the marks for each individual test should be augmented by feedback explaining where mistakes have been made or how to improve the solution.

An ideal marking system will be able to do the following:

1. Run tests automatically, with no human interaction, this is a basic principle of automation.

2. Be able to interact with the interface in any way required, known as test adequacy [ZHM97].

3. Be consistent with the testing and marking performed [Bon99].

4. Handle varying layouts for the same problem, this is a by product of having a large number of different submissions.

5. Identify objects such as buttons in a number of ways, this is an educational restraint allowing for different forms of testing.

6. Be able to provide a graded mark for tests run [BW98].

7. Supply feedback to support the marks given [BRS96].

Whilst it was stated that these criteria would be met by an ideal marking system, it is not unfeasible for a system to exist that would exhibit all of the required traits.

1.3 Motivation

Education cannot exist without assessment, students must be asked to demonstrate what they have learnt for it to be of value. Whilst the benefits of assessment far outweigh the demands marking places upon lecturers, marking still remains a consider-

able problem. In an attempt to remedy this, several systems have been created to automate marking [AB99b, BBF+94, JL98a]. Whilst these systems all provide assistance to those running courses, they all have one particular failing. None of the systems currently available provide any mechanism for the automated testing or marking of Graphical User Interfaces.

The marking of command line driven programs, is a relatively simple task involving the use of a collection of unchanging test data that is to be entered. GUIs do not have that advantage, it is possible to have a large number of solutions to the same problem. Even if you only consider the location of the components concerned, the test data for a GUI is still changeable. Therefore, the time required to mark a GUI is significantly longer than for command line driven programs. The longer and more complex test data is, the more likely a mistake is to be made when marking by hand. The savings that would be provided by the automation of the marking process would be considerable. The removal of erroneous testing, on-demand feedback and instant responses are but a few of the benefits that it would yield. There is also the extra available time freed up by the use of such a system that can then be put back into the course.

The difficulties of marking Graphical User Interfaces are apparent and the solutions that currently exist merely divide the marking responsibilities. The creation of automated systems to mark programming coursework highlights the desire that exists to reap the rewards that such systems provide. The only area not yet tackled is that of Graphical User Interfaces. With the increasing commonality of GUIs, the need for the automation of GUI marking is clear. Therefore, it is the aim of this thesis to propose and design a system capable of marking student's basic GUI programming exercises from a dynamic, typographic and aesthetic perspective. The benefits of this system will also be presented through implementation and testing against actual programs created by students undertaking a programming course.

1.4 Scope

At the University of Nottingham, there is already a CBA assessment system in place, CourseMarker formerly CourseMaster and Ceilidh. Whilst the theory behind the automated Graphical User Interface marking system allows for implementation in

any scenario, it was implemented for insertion into the CourseMarker system. It was used in the second semester of the first year programming course for two years running. This allowed for the theory to be tested and any holes in the design to be highlighted and corrected. This real world testing provided results that could be analysed to see how effective the system was. The integration into CourseMarker also meant that the students were able to utilise CM's abilities without the need to learn how to use a new marking system.

The marking system was designed and implemented with Java in mind. The decision of which programming language to teach students is a controversial one with several factors influencing the decision [How95], including topics as language design and the user-friendliness of the language. The decision was made in 1999 to change the curriculum at Nottingham, now the first year programming course is solely Java based. The GUIs the students shall be creating will be written in Java using SWING. The tests that were created for the system only includes those that were needed in the marking of the assignments set. However, the system was designed to allow for the inclusion of additional tests without significant knowledge of how the GUI marking system works. The final restriction that was placed upon the implementation of the system referred to the aesthetic and ergonomic tests. At the current time, design considerations of Graphical User Interfaces are not explicitly covered in the programming course, but are left for a later module. This meant that the design measures discussed were not tested alongside the main student marking but were implemented independently and a number of tests were performed.

1.5 Synopsis

This thesis uncovers and explains the complications involved with marking Graphical User Interfaces. It also describes solutions to these complications and explains how they can be implemented to create a system capable of automating the marking of Graphical User Interfaces. Outlined below is the contents of the following chapters, it is also shown in the form of a mind map (see Figure [1.2]). The mind map highlights important sections of each chapter and their relative position in the thesis.

Chapter two will provide an overview of how learning technology can be utilised in the assessment of students undertaking programming courses. There will also be an

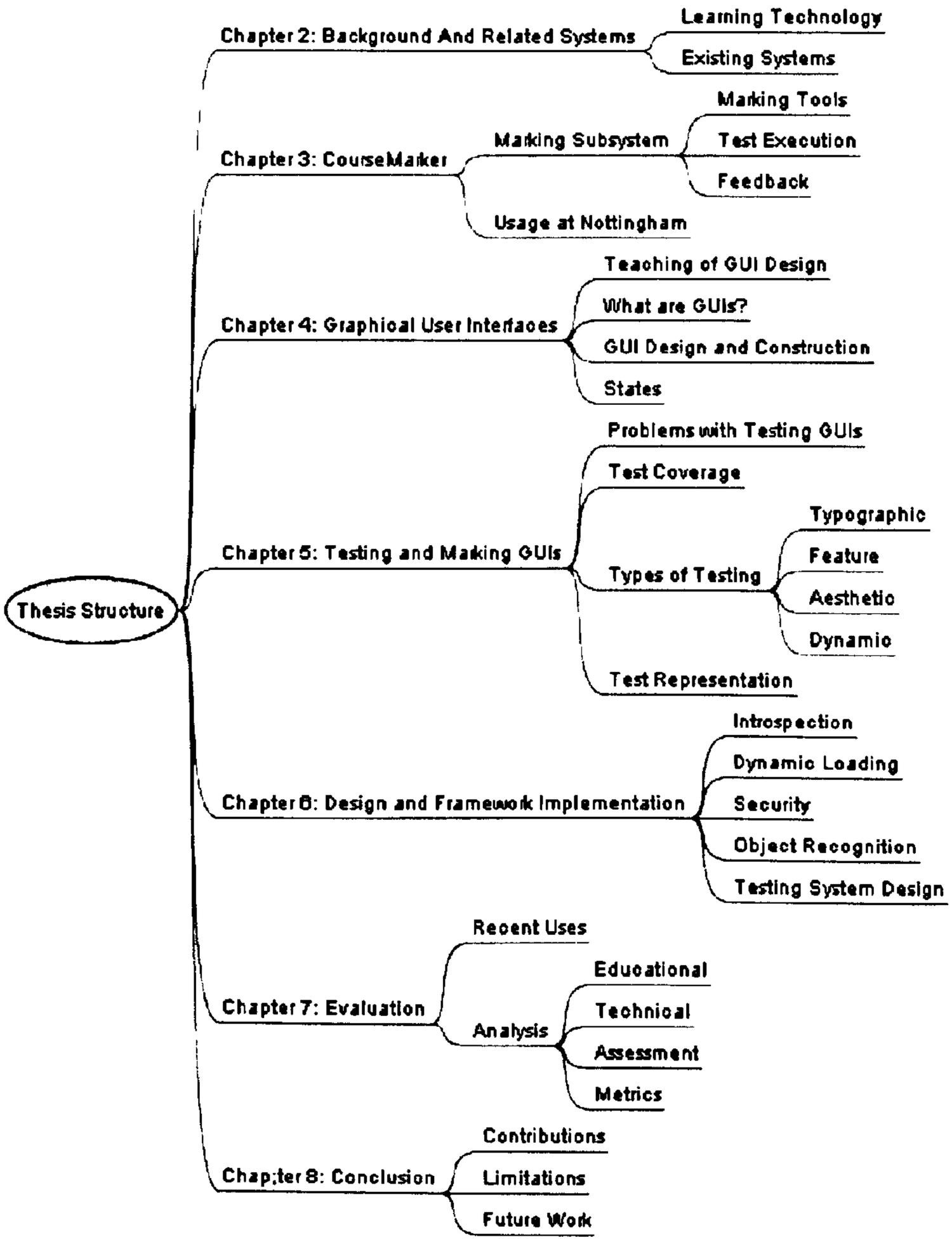

Figure 1.2: A Mind Map of the Structure of the Thesis

analysis of systems currently available for the automated testing and marking of both command line and GUI based programs.

Chapter three will concentrate solely on the CourseMarker CBA system. It will give a brief explanation of the system as a whole before going into detail on the sections of the marking system relevant to the implementation of all new marking systems. This chapter has been based upon the journal paper "Automated Assessment and Experiences of Teaching Programming" [HGST06].

Chapter four is where Graphical User Interfaces are considered. There shall be a description of what they are and their purpose, before outlining how to create a GUI and important considerations to take into account during their design.

Chapter five shall take the concept of the Graphical User Interface further and

tackle the problems of testing such a program. It will highlight the problems involved in the testing of GUIs whilst mentioning ways in which the problems could be overcome. It will then describe the two important aspects of marking GUIs, functional testing and the aesthetic analysis of the program. The section on aesthetics shall describe a number of pre-existing metrics before explaining, through the implementation of colour theory, how colour can be used to enhance the results obtained.

In chapter six, the design of the marking system is explained. It will show how the problems faced by marking GUIs can be solved using techniques available to any Java programmer. The introspective approach has been peer reviewed in the paper "An Introspective Approach to Marking Graphical User Interfaces" [GH06]. Each phase of the marking process will be outlined from object recognition to the actual execution of the tests themselves. This will be combined with a description of how the theory was implemented and integrated with the CourseMarker system.

Chapter seven will provide an evaluation of the results obtained through the use of the GUI marking system over two semesters at the University of Nottingham. These results will demonstrate that the automated marking of GUIs is indeed possible. The results analysis shall be performed from a number of perspectives to show how such a marking system be have an impact from an educational, assessment and metric point of view. It shall also highlight the effectiveness of the system and how all the advantages of automation are received by those running and taking the course. This shall be followed by the conclusion and a selection of possible extensions that could be made to the marking system in the future.

Chapter 2

Background and Related Systems

"We recommend grading programs to all who teach programming and numerical analysis to masses of students, but the prospective user should first carefully investigate the systems available to him" - Forsythe + Wirth

This chapter will present an overview of the research important to the creation of an automated interface marking system. The fields covered are those of Learning Technology and the existing approaches to automated marking.

The first section contains an overview of learning technology and all its facets, assessment, Computer Assisted Learning, Computer Aided Assessment and Virtual Learning Environments. It will explain why assessment is important to the educational process, its different types and what it aims to achieve through its existence. The rationale behind the move to automated assessment from more traditional marking approaches is also discussed.

Finally there will be a discussion of existing systems. Whilst no systems with the capability to mark GUIs exist, a large number of related systems have been created since the early 1960s when the first automated marker was suggested [FW65]. This section will detail systems from all areas of the automated marking and testing spectrum.

2.1 Learning Technology

Learning Technology is a term used to describe a wide range of information and communication technologies that are used to support and enhance learning, teaching and assessment [Pap04]. These technologies include the use of multimedia, Computer Aided Assessment, Computer Assisted Learning and Virtual Learning Environments. If implemented correctly, the use of learning technology can greatly improve the effectiveness of teaching through its ability to provide students with extra support in the form of online notes, links to external resources etc.

One use of Learning Technology is in distance education, which as the name suggests, involves taking courses at a remote location. Therefore, as attending classes is impossible, a new method of delivering the course to the students was needed. Learning technology provided the solution. One problem with distance learning is that the majority of the research done in this area is nationally localised. For useful work to be done the research needs to become more global [Coo01]. Whilst research is not being done directly in distance learning, people are researching related topics. It is the direct implementation of these related topics that is more relevant than distance learning.

2.1.1 Assessment

Before covering the different uses that learning technology has within education, it would be prudent to first tackle the problem of assessment and why it is important.

Assessment in education can take many forms e.g. exams, multiple choice questions, essays and programming exercises. Each form of assessment does have its uses, but it is important to pick the correct form for the situation in question. When deciding how to assess students there are guidelines that can be followed. Assessment should play a positive role in student learning. Assessment should also provide students with feedback and the methods used should be valid [BRS96]. This is particularly relevant to this research. Valid methods for assessing programming would be the marking of students' programs and the use of an automated system enables feedback to be delivered to the student moments after a solution is submitted. Each educational institution has its own preferred method of assessing the students with regards to programming in Computer Science courses, the University of Nottingham is no exception [HGST06].

The complications do not disappear once the decision over how the students are to be assessed has been made. Creating assignments that enable students to demonstrate their knowledge is a complicated task, although it is relatively simple to write assignments for the sole purpose of generating grades. Grades, however, do not show what a student knows but represent the extent to which a student has successfully met the faculty member's requirements and expectations for a course [Rog03]. The last statement may be a generality about assessment, but can be avoided by following existing defined criteria for the creation and grading of programming assignments.

Creating Programming Assignments

The creation of programming assignments is a complicated matter. It is a fine balance between not supplying a strict enough specification and having to interpret the output results, or restricting the students to such an extent that there is no opportunity for them to exhibit creativity in their solution. There are criteria that good programming assignments should meet in order to get the most out of the students taking the course [SW06]. Assignments should:

- Be based on "real-world" problems.
- Allow students to create realistic solutions.
- Allow students to focus on current topics from the class within the context of a larger solution.
- Be challenging.
- Be interesting and achievable by basing the question around practical and / or current issues.
- Have multiple levels of achievement, this allows for the awarding of partial credit.

Whilst the above is by no means a comprehensive list, it will ensure a certain degree of confidence in the assignments created. Unfortunately for course lecturers, creating well written assignments is not enough, they also need to be graded in the appropriate fashion.

Grading Programs

The biggest problem when marking is that of consistency. It is something that can be affected by bias, laziness or fatigue etc. Consistency becomes a bigger problem when there is more than one marker and has lead to universities creating their own scales for marking programming assignments [HKHRT83]. However, these scales are rarely universally applicable having been created for specific use at one educational institution. More user friendly program marking criteria have been separately devised. One such set of criteria [How94] consists as follows:

- Program Execution - are the results correct?
- Specification - has it been met explicitly?
- Program Design - analysis of the structure, modularity etc. of the source code.
- Coding Style - are standard conventions met?
- Comments - are programmer comments useful and occur regularly?

- Creativity - has the student gone beyond the call of duty and created an interesting solution?

All of the criteria, with the exception of creativity, lend themselves to automation. Most of the research done into grading solutions exists in the form of automatic assessment tools where the above criteria are often the basis.

What Assessment Achieves

One question so far unanswered is "Why are students assessed?". Assessment is used to allow students to demonstrate the knowledge they have acquired throughout the duration of a course. It also enables lecturers to rank the students in order of how well they have performed. However, if incorrectly applied these are the only advantages of using assessment. Much more can be achieved if the assessment is implemented correctly, it can be used to increase knowledge. Analysis of studies has shown that the practice of formative assessment produces significant and often substantial learning gains [BW98]. The studies this information was extracted from were wide ranging encompassing students from the age of 5 years to university undergraduates.

2.1.2 Branches of Learning Technology

Within Learning Technology, there are certain areas of research that are of importance to the automated marking of Graphical User Interfaces. The research can be divided into two sections, learning through the use of computers and using computers for assessment. They are very closely connected, both striving toward the same goal, enhanced student ability. It is also fairly common to see the two concepts combined within the same system, normally a Virtual Learning Environment, which shall also be discussed.

Learning with Computers

The use of computers to aide learning is a contentious topic. Surveys have shown that students do receive benefit from their usage compared to more traditional teaching methods [Leu99]. However, counter-claims have been made that many of the surveys

are flawed through the use of inadequate statistical measures and claims that variables such as Computer Assisted Learning are precise and controllable [Mit97]. Not all surveys are flawed in that way. The aforementioned survey [Leu99] attempts to prove that students' underlying approach to learning influences how well they interact with Computer Assisted Learning systems. Whilst still relevant, the target of the survey was economics students rather than computer science students who would naturally be more used to computer interaction and would be more likely to embrace the transition to Computer Assisted Learning methods.

There are two significant areas of research with regards the use of computers in learning, Computer Assisted Learning and Computer Managed Learning. Computer Assisted Learning (CAL) is the implementation of technology to present educational experiences electronically. This is often accomplished in a variety of means. Computer based tutorials where the material presented depends on students answers, dialogue systems where the student and computer conduct a learning dialogue or more simply the inclusion of multimedia such as sound or video etc. [KN81]. In comparison, Computer Managed Learning (CML) uses technology to better administer educational experiences through electronic systems [Zel93]. It is an extension of CAL which ensures that students are working at the correct pace and are receiving the information required. It can also involve the inclusion of testing systems, grade books and computer mediated communication.

The use of CAL methods has both advantages and disadvantages. The disadvantages include the perception and reaction by the users of systems involved, poorly designed and implemented systems and other factors such as cost and initial set up time. Within a Computer Science department, there should be few issues with the use of computers in learning due to the extensive exposure to computers the students already have. The advantages of using courseware far outweigh the disadvantages. The ability to track students' progress through the software enables lecturers to see where problems are arising and tackle them early on. There is also the ability to archive notes, exercises and past papers from previous years for later use. One of the more common implementations of CAL techniques can be seen in Virtual Learning Environments, these shall be discussed shortly.

Using Computers for Assessment

Whilst it is not just programming courses where computers are used to assist the assessment process, they are a large contributor with regards the automated systems created.

Before computers were used to automate various parts of the exercise delivery, collection and assessment process, it had to be done manually. This involved collecting and assembling the submissions in an appropriate location, then compiling and testing the programs by hand. Besides being a laborious task, there are several areas where problems could arise [GG06], examples include files that do not compile, non compatible compilers could have been used and minor errors would sometimes be fixed by marking assistants of a generous nature. If the output did not meet the specification, assistants would need to determine the significance of the discrepancies. In order for marks to hold weight, the way in which solutions are marked needs to be consistent. Computers, through their ability to be programmed and strict adherence to the rules make perfect candidates to assist the process.

Computer Aided Assessment (CAA) refers to the use of computers in assessment. The term encompasses the use of computers to deliver, mark and analyse assignments or examinations. CAA also includes the collation and analysis of data gathered from optical mark readers [Bul99b]. There exists a specialisation of CAA, known as Computer Based Assessment (CBA). Computer Based Assessment refers to the delivery of materials, the input of solutions by the students, an automated assessment process and the delivery of feedback, all achieved through an integrated, online system [HB06].

The use of CAA and CBA also have their advantages and disadvantages. Disadvantages include the perceived validity of CAA, the fear associated with using technology and changes needed to invest time in designing and creating new assessments rather than marking traditional assessments [Bul99a]. However, the use of CAA / CBA technologies offers many pedagogic benefits, the most obvious is that of time saved by those running the course. This time saved could be used to improve the student learning experience by devoting more time to student contact [SC04]. More valuable from a knowledge development point of view is the delivery of instant feedback. It has not been shown that feedback provided by a CBA system improves overall marks, but that it does reduce the length of time students spend discussing problems with course tutors

[OHZ01]. This finding can be compared to another study performed. Students were split into two groups. The students in one of the groups had to regularly meet with the lecturer and their work was graded there and then in the student's presence, whilst the other group's work was graded alone. The results showed neither group performed significantly better at the end of the course, despite the extra time that had to be spent by the course lecturer [ES05].

A large number of Computer Based Assessment systems have been created, some of which include multimedia such as graphics, sound and video. There is also a current trend for CBA systems to be web based [O'L99]. CBA systems can be used for everything from marking multiple choice questions, through diagrams to programming solutions. The use of such systems can cause a change in the way students are asked to answer questions, but despite any additional demands that can be placed upon students by these systems, they are generally happy and rarely are complaints made [LTH+99].

Virtual Learning Environments

Creating an exact definition of a Virtual Learning Environment (VLE) can be a difficult task. This is due to the fact that VLEs are flexible and can be used to perform a plethora of tasks in a variety of ways [Dil00]. This does not mean that a definition cannot be created. A Virtual Learning Environment is an online computer environment where various tools are provided to facilitate learning [Chi05]. They were created to allow for the easy communication of information, whether it be for distance learning or locally. VLEs encourage collaboration and can be used to share information, either between lecturers and students or just between students [JIS05]. The advantage of VLEs are that lecturers can keep tabs on their students through the system by seeing which notes they have accessed, exercises undertaken etc. This is significantly easier than tracking students turning up to lectures where there is often limited direct contact with individual students.

There are a number of Virtual Learning Environments available, some have been created by educational establishments and others commercially. Some of the more common are WebCT [Web], Blackboard [Bla] and Virtual Campus [Cam]. VLEs are now used by the majority of UK universities and higher education establishments [KF05]. They do not just have uses with regards to the delivery of courses but can be adapted to provide automated assessment. This is a simple extension, whereby a CAA

/ CBA system is incorporated into the VLE. The use of VLEs in education have been shown to save staff time, and improve the experiences of the learners, although there are times when the systems can be a little unstable [Sma02]. For the use of VLEs and also CBA systems to become universal, stability will need to be guaranteed, until then paper-based backup systems shall always be required.

2.2 Existing Systems

Writing a program is not just a case of programming and then releasing the result, it is a cyclic process involving regular testing and refinement. The process can be assisted by useful feedback, this is especially helpful for novice programmers [VH03]. However, the feedback needs to be provided on-demand to be of use. In order to analyse a program and provide on-demand feedback, the testing needs to be automated. There are two distinct sets of programs that need to be considered. Software designed solely for the testing of programs and education based software created with assessment in mind. It is a relatively simple task to classify these systems either by the way they operate or by their overall goals. The systems described below do not by any means constitute a comprehensive list, they are merely examples of the different types of system.

2.2.1 Assistive Systems

Not all tools designed to assist the teaching of programming courses were created to grade solutions. Of the two tools described below, the first was created to automate the delivery of the course and assessments and the second is a system to simplify the creation of GUIs which has been incorporated into several marking systems.

JERPA

The Environment for Remote Programming Assignments in Java (JERPA) is a distance learning educational tool [ET02]. JERPA provides students with access to the programming assignments, class libraries, test data etc. through it's interface. It connects to the server information using standard HTTP connections. This allows the

client to be used on any machine with both Java installed and web access. However, the functionality exhibited by JERPA is rather lacking. It may succeed as a course delivery agent offering opportunities to download, compile, run and submit the program written. It unfortunately does not offer any marking abilities to the users.

There are future plans to expand JERPA to allow it to also work with programming languages other than Java and to allow roaming-profile support. At the current time JERPA works best if students use it on one machine only.

TCL and TK toolkit

TCL and TK are two toolkits, which combined provide a system for developing and using Graphical User Interfaces [Ous94]. They are both implemented as a library of C procedures which allows them to be used in a variety of other systems. The C / C++ complexity with regards to GUIs is hidden through the use of TCL and TK, this allows interfaces to be created after a few hours learning the system.

The simplicity of the TK toolkit for creating GUIs, along with the knowledge that they are created using C procedures has meant that they have been integrated with automated assessment systems. Ganesh is one example of a system that has integrated TCL into its own program development and grading process.

2.2.2 Testing Systems

The following systems discussed provide users with the ability to interact with their interface directly. There is no automation to allow users to create tests quickly or with very little effort. A high degree of programming competency is required to create the test initially. Once created, however, the test can be reused ad infinitum often with minimal changes required to reuse the test in different situations.

GUITAR

GUITAR is a framework solution which integrates various tools and techniques to be used in the various phases of GUI testing [Mem01]. Of GUITAR's constituent components, the most interesting is that of the regression tester. Once the tests have

been automatically generated and run, the regression tester checks the results supplied by the system for any invalid test responses. On subsequent testing iterations, any test that previously completed successfully is bypassed. The tests that failed to complete returning an invalid result, most probably due to modifications to the GUI, undergo a repairing process. They theorise that by altering tests that previously did not complete, the major failings of the system will eventually be highlighted.

The power GUITAR exhibits when testing Graphical User Interfaces are also its hindrances when marking is considered. For marking to be consistent, every solution needs to be run against the same tests. By modifying the tests using the regression tester, the solutions will be subjected to different test sets, thus negating any consistency rendering it worthless for marking purposes.

Java Robot Class

The Robot Class [Mic04] was created by Sun Microsystems for their 1.3 release of the Java programming language. It was designed for the purpose of automating tasks Graphical User Interfaces could perform, with a view to demonstrating the software created. The class allows the programmer to take control of all native input to the operating system. This capability can be used to simulate a user attempting to operate the application under test and could feasibly be adapted for testing purposes. Although the Robot has the potential to be extremely powerful, it's inflexibility is a major hindrance.

The robot needs to be told explicitly where each object in the interface is located. It does not handle changes made to the design of the interface without requiring a major rewrite to the robot.

In the hands of an experienced programmer, most dangers should be avoidable. However, because use of the robot involves control over the native input, it is possible to access the underlying operating system, be it accidentally or with malicious intent, and cause great damage to the machine being used.

Jemmy

Jemmy is one part of the SUN sponsored netbeans project [Pro]. It is a relatively new project, which allows users to create demonstrations or tests for Java Graphical

User Interfaces. A lot of the tests are performed directly upon the event queue. However, they have recently added functionality which can perform tests using the Java Robot class. Whilst a valuable library, it had recently been updated for the first time in three years. Such regularity of support may cause issues as the language develops.

Unfortunately, the documentation states that the GUI tests are unstable. To be of any use for marking purposes, any instability issues would need to first be addressed. The other significant issue with Jemmy, is that all tests would need to first be programmed before they could be used to test a students solution. This requires an in-depth knowledge of both Java and the way the Jemmy library works. Ideally a system for testing students' work should be usable by those with only a limited knowledge of the testing system.

JUnit / JFCUnit

JUnit [JU] is a Java framework designed for programmers to create test suites. These suites consist of testing sequences to be carried out to determine whether a program is working. JUnit was created for testing command line driven programs, but has been extended and superseded by JFCUnit which works with the Java Foundation Classes enabling the testing of Graphical User Interfaces.

The main advantage of the JFCUnit software is that with a few parameters, the test suites will attempt to locate the object required. With the test suites themselves being Java programs, the opportunity for errors being present in the test suite exists. Also unless the tests have been previously written and can be reused, it can take a considerable length of time to create them initially.

2.2.3 Capture and Replay Tools

Of all the variations of testing software, this is by far the most commonplace. This is down to the fact that a large proportion of all commercial testing software falls into this category. Each piece of testing software may work in a slightly different way, but they are all based around the concept of the video recorder. They allow you to load a Graphical User Interface and use the system to record the user interactions. When required, users are able to replay the actions previously performed.

The trouble with Capture / Replay tools are that the tests contain a lot of information but little meaning about the intent of the interactions [Sil03]. Performing the capture at a logic level would allow for changes in the interface without having to rerecord the tests.

jRapture

jRapture is a Capture / Replay tool designed for use with Java programs [SCFP00]. It works by capturing the executions created during testing. The capturing is done with the use of modified Java API classes. When a program is started, these modified libraries are loaded instead of the standard ones. These classes store the process of events that have occurred along with how they can be recreated. Captured executions can then be replayed by the system to recreate the same testing process as before, possibly for analysis by testing experts. It does provide a level of GUI support. The majority of the interactions with GUIs are done by posting actions to the event queues.

This approach works well when just recreating actions that were recognised by additional layers of the JVM. It is less efficient for marking purposes. The authors state that they cannot guarantee a completely faithful recreation of tests performed, especially with GUIs, which when marking is of the utmost importance. However, jRapture has been used as a platform for creating other similar pieces of software, one example is SCARPE which is currently at the prototype phase [OK05].

qftestJUI

Of the many programs on the market for testing Graphical User Interfaces, one of the more advanced is qftestJUI [QFS], created by Quality First Software. qftestJUI works by allowing the users to record their test sets within the testing system. The software then splits the tests up into the separate actions performed, thus allowing users to add and remove tests from specific places in the testing sequence.

The ability to alter the test sequence without having to rebuild it from scratch makes qftestJUI a very powerful piece of software. However, it does not have the flexibility to deal with solutions from large numbers of users where the exact details of objects are not known. Precise information is required for the software to find the

GUI objects.

2.2.4 Automated Submission and Assessment Tools

All the systems hereto mentioned were designed with testing in mind. There are systems which have been created with assessment in mind. The following systems have all been created to assist with the marking and teaching of programming.

Ceilidh

Ceilidh [FTHS99] was created at the University of Nottingham from a collection of unix shell scripts that had been developed to automate certain tasks. As the number of scripts increased it was decided they should be packaged up to give not only the lecturer but also the students the advantages they could provide.

Ceilidh's marking system was developed around the fact that it was able to mark programs from several perspectives. It had the capability to not only mark programs dynamically, i.e. does the program work providing responses consistent with the question specification. Ceilidh was also able to perform marking directly upon the students submitted source code. These tests were concerned with the programming style and layout of the code.

In addition to the automated marking, Ceilidh was also the distribution system for not only the coursework exercises, but also the course notes. It was initially created to deliver a C course but its extensible design meant that marking tools were rapidly developed for several other programming languages. This extensibility contributed to Ceilidh's downfall, over the years it had effectively become a collection of extensions based around user suggestions received since it's inception.

Ceilidh was not without its limitations. There was no network support, users had to log on to the server hosting the system to be able to use it. It also did not have a full X-Windows or PC-Windows graphical interface. Ceilidh was updated in 1998 when it became the more functional CourseMarker [FHTS00, FHST01] (formerly known as CourseMaster).

Agar

Agar was designed to take the mantel of "Killer App" [WP06]. The authors claim that current systems are too restricting on the user and have remedied this with Agar. It is to work alongside a human grader as an assistant [WP05], allowing the human grader to overrule any mark awarded by the system. The fact that it only aides assessment in this way makes it different from all other CAA/CBA systems currently in use.

There are obvious disadvantages to a system of this nature. If the tests created are not comprehensive, it relies on the human grader to complete all other marking. Whilst the system does assist by providing different views for the grader to use in this process, it is still time consuming. Another worry is the fact that the students' solutions are not run in a sandbox (a secure environment for running potentially unsafe programs), certain restrictions are placed on the running programs but nothing to protect it from malicious code. The lack of GUI marking is a significant failing of the system.

ASAP

The Automated System for the Assessment of Programming (ASAP) [DLO+05], assesses programs through the use of other systems such as Virtual Learning Environments. The students submit solutions through the VLE, this passes their source code to the Automatic Java Marker and tests the solution against a set of test data. The marks are returned to the student again with the use of the VLE.

With the exception of using an already existing Virtual Learning Environment to act as an interface to the marking system, ASAP does not appear to do anything novel. The marking is basic as it seems to be restricted solely to the Java programming language. No GUI support was mentioned in their literature.

Boss2

Boss2 [JCL00], an update of the BOSS system [LJ99, JL98a] created at the University of Warwick, is an assistive rather than a marking tool. It provides students with the information required to complete the task and also a means to submit their solution. BOSS's marking capabilities are somewhat limited, it is able to run com-

mand line driven programs against a set of test data but returns just a mark of either 100% or 0% with no feedback. Once all student submissions have been received, it randomly allocates all submissions to a list of designated markers for hand marking. Thus reducing any bias from marking the scripts of known students.

The Boss system was updated in an attempt to remove some of the limitations of the previous version. These limitations included the lack of network support for the system and the operation of some of the result comparison utilities such as `diff`. The new design implementation, which uses a standard Java client / server architecture using RMI, does provide extra functionality. One improvement of the new system is that instead of supplying the expected result as text, the results can be expressed as an object. This allows the course designer to incorporate different levels of strictness in the mark scheme.

CodeLab

CodeLab, formerly WebToTeach [AB99b], is a web-based interactive assessment environment. Its designers claim it ideal for introductory programming courses. It provides students with large numbers of questions of increasing difficulty on all topics covered. The answers which can range from single word answers to fragments of code are marked automatically by the system.

CodeLab appears to only mark either single line or short answer questions. Students using the system may understand the theory about the concepts the system examines, however, they will have no experience of writing complete programs themselves.

DATSYS

DATSYS is an extension to the Ceilidh - CourseMarker system [Tsi02]. It provides the capability for CourseMarker to assess a wide variety of technical diagrams e.g. circuit diagrams, flowcharts and entity relationship diagrams. It works by converting the diagrams into programming code which can then be tested using the standard CourseMarker procedures.

Ganesh

Ganesh is a web based learning environment for computer science courses [LM98, LM00]. Its goal is to assist students' learning of programming. It does this not only through the delivery of coursework, but by offering help via the use of tools for debugging, editing and other help systems.

It is a client-server system using the Internet as its transmission layer, enabling it to be used as a distance learning system. The client provides the students with access to the exercises and relevant tools, where as the server along with performing any marking required, allows lecturers to keep track of the students' performance.

The programming course offered by Ganesh uses an assembley language and the Apoo virtual machine [RM98]. The Apoo virtual machine was developed to ease the learning of assembley languages in conjunction with the Apoo interface. This interface provides users with a look at the contents of any registers being used along with other properties such as the system stack.

Apoo and Ganesh provide a great deal of assistance to those learning assembler. There are naturally a few drawbacks too. Firstly is the fact that it seems to only be able to test and grade programs written in assembler using the Apoo interface. The general move toward Object Oriented programming languages means that Ganesh is becoming dated unless new courses and virtual machines can be written for it. There is also reference in the documentation to being able to create GUIs by interfacing the TCL and TK toolkit although no information on how this is achieved could be found.

Kassandra

Kassandra [uvM94] like the majority of automated assessment systems was designed to relieve pressure from teaching assistants. It was predominantly used to grade both Maple and MathLab programs, but can be adapted to mark programs written in more conventional languages such as C. Kassandra, through its design, allows additional assignments to be added to the system with ease. This provides it with a certain flexibility over the assignments the students are set.

There are limitations to the system, it must be run on a UNIX operating system to function. As with other current software there is also no GUI marking capabilities.

RoboProf

RoboProf [Dal99] is another web-based system. It is different from CodeLab in that it simulates an electronic course book. The students are provided with the course notes in HTML form and RoboProf will not let them advance until they have mastered the chapter currently being studied. If the student answers a question incorrectly they are given another similar question to complete. This does promote learning, but cannot be used for assessment as the opportunity for inconsistencies is too great.

TRAKLA2

TRAKLA2 is a system developed for the assessment of visual algorithm simulation exercises [SMK01, KMS03, LSKM04]. The exercises are individually tailored and delivered to the students. The students then use a GUI to manipulate the underlying data structures with the solution to the exercise being a sequence of discrete states of data structures.

The TRAKLA system and methodologies were also incorporated into a web-based version of the system WWW-TRAKLA. Unfortunately, the TRAKLA system is limited to assessing algorithm simulations only.

2.2.5 GUI Marking

This is the smallest section concerning existing systems. There are systems available that can test GUIs in a variety of ways as have previously been described, but there is only one that claims to be a GUI marking system.

JEWL

The John English Window Library (JEWL) was designed to provide novice programmers with a simplified windowing tool kit [Eng04]. It was suggested that Java's SWING API was too complicated for students as a first GUI building system. JEWL was created as just that, not a replacement for, but a simplified version of SWING with some of the complexities hidden away.

Whilst it is essentially just a windowing API, it does allow for automated marking. An alternative implementation of JEWL exists. This implementation is used only by the marking process, it suppresses the display of any windows and provides the marking system with several methods that can be exploited to interact with the GUI. The test harnesses within the JEWL system are able to monitor what the program is attempting to do and provide marks accordingly.

The JEWL system does attempt to mark Graphical User Interfaces, however, the way in which it does so is not beneficial to the students using it. The major problem is that although students will be able to create GUIs using simplified APIs, when they enter the software industry they will be faced by the full version of SWING with which they will have little to no experience. Also by suppressing the loading of the GUI, there is less opportunity to analyse the layout and aesthetics of the Interface created. JEWL may provide a solution to the problem of marking GUI, but it has done so by altering the questions asked.

2.2.6 Interface Quality Measures

The final section concerns ways of analysing Graphical User Interfaces and making decisions based upon the results of the metrics. The methods below attempt to discern several aspects about the interface without the need to actually create the interface itself. They are theoretical approaches to the creation of GUIs.

As well as the methods mentioned below, there are also metrics created that attempt to analyse the aesthetics of an interface. These metrics are discussed in detail in Chapter Five.

VEG Toolkit

Visual Event Grammars (VEG) is a specification language [BRRP05]. It is an extension of BNF grammars. VEG is used to describe the communicating components of a GUI. It attempts to describe sequences of user events as sentences in a formal language. By creating automata in this way it drastically reduces the number of states (or components) involved, these automata can then be used to verify whether the GUI created is correct. VEG is less concerned with actually testing the system more proving

that there are no deadlocks or unreachable states etc.

VEG is primarily a specification language and therefore cannot perform the aforementioned verification checks itself. These are done using the model checker Spin [Hol97]. Before it can be used, the VEG notation must be translated into the input language for Spin, Promlea.

VEG is still only in its early stages of development and so far has limited functionality with regards to system verification. However, VEG is also able to generate Java code to create the GUIs described. The authors claim there to be no difference in performance between VEG generated and Java code, although these claims were not substantiated.

2.3 Summary

This chapter has shown why CBA systems are needed to reduce the burden placed upon lecturers and how they can be used to solve the problem. It has also described the different ways in which technology has been incorporated into education to improve the students' learning experience. There are a large number of systems in existence that can be used to test and / or mark programs. However, each system has it's own failings and has been shown to be unable to perform marking adequately, especially when Graphical User Interfaces are concerned.

In the next chapter an overview of the CourseMarker, formerly Ceilidh, system will be given. The main area of focus will be the marking subsystem with a view to seeing what important attributes it has and how it could be extended.

Chapter 3

CourseMarker

"Let's just say I was testing the bounds of reality. I was curious to see what would happen. That's all it was: curiosity." - Jim Morrison

CourseMarker (CM) is a CBA system developed at the University of Nottingham to completely automate the assessment process for a first year undergraduate Computer Science programming module. It not only delivers the question to the student but also performs all marking and provides on-demand feedback to the students. It has successfully been in use since 1999.

CourseMarker is a reincarnation of the Ceilidh system. Due to failings present in Ceilidh, CourseMarker was designed from scratch [FHH+01]. This allowed the designers to include the requests for additional functionality that were not implementable under the old Ceilidh system. The redesign enabled the designers to incorporate knowledge about object-oriented methods, frameworks and design patterns. This provided major benefits in the areas of extensibility and maintainability.

This chapter is to outline the whole of the CourseMarker system before examining the marking subsystem in greater detail.

3.1 Overview

CourseMarker was initially designed with platform independence in mind. This was achieved via the use of the Java programming language. Java, being an interpreted language, runs under the Java interpreter's Java Virtual Machine allowing for the portability of programs between platforms. This removes the need for platform specific classes, reducing development time and simplifying technical support. Java offers the programmer an advanced exception handling mechanism and more importantly a way to dynamically link and unlink programs. Java also supports the development of distributed systems via the use of their Remote Method Invocation (RMI) mechanism.

3.1.1 Remote Method Invocation

Remote Method Invocation has allowed the logical processes of CourseMarker to be split into several separate subsystems. The advantage RMI has over the use of network sockets is that it effectively makes the network transparent to the programmer. CM's remote objects communicate with each other using a set of common interfaces, which if required would enable the subsystems to be placed on separate servers.

3.1.2 Subsystems

After detailed analysis, Ceilidh's processes were extracted and seven subsystems were created (see Figure [3.1]). The responsibilities of each subsystem with regards to the operation of CourseMarker are both separate and distinct. These responsibilities are described below.

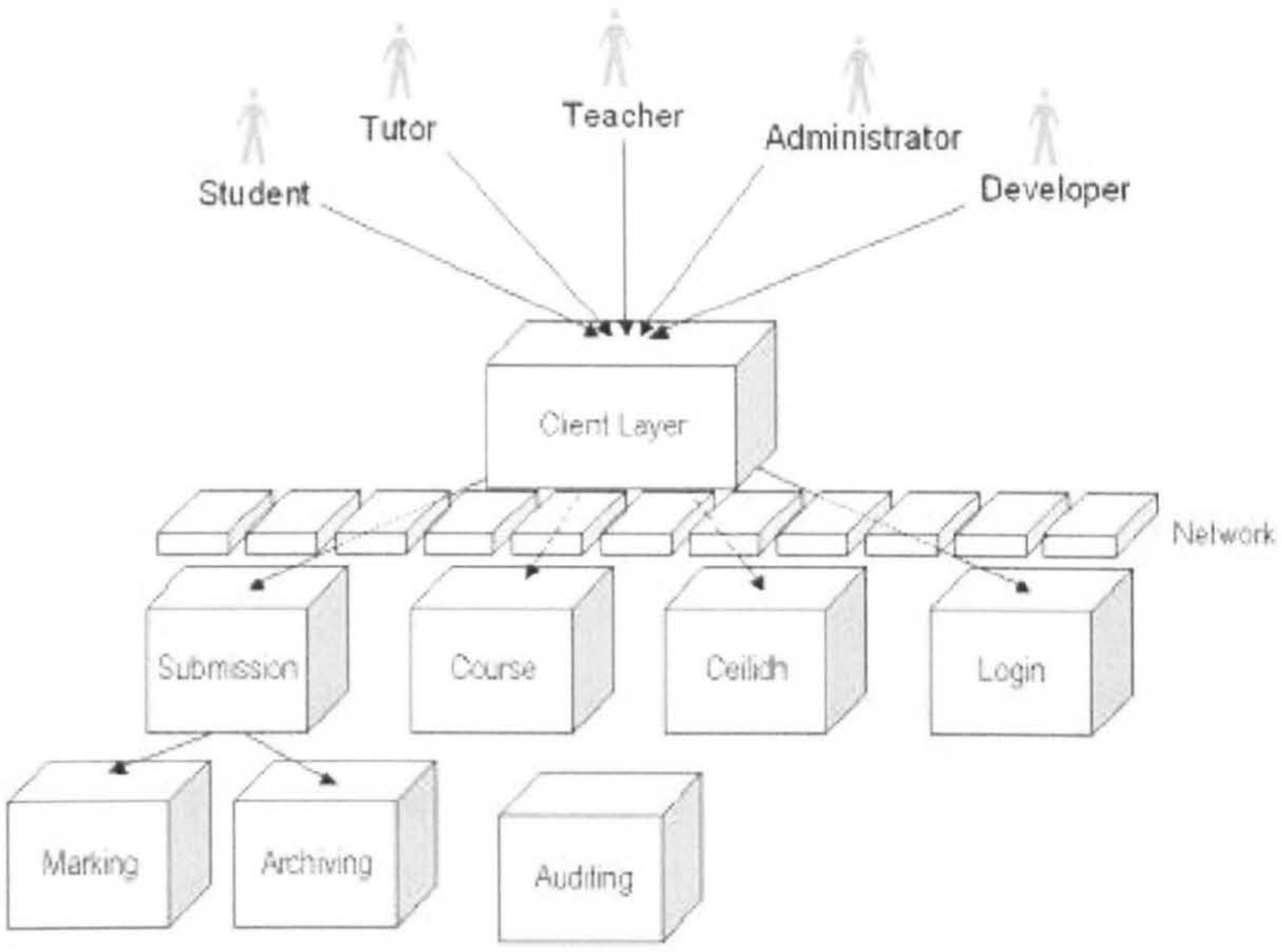

Figure 3.1: A high level view of CourseMarker's subsystems

Archiving Subsystem

The archiving subsystem is responsible for, once the submission and marking has been completed, collecting and storing a complete back up of the students solution. Not only are the student's files collected, but also a copy of the marking receipt file which stores the student's mark and shows where mistakes were made.

Audit Subsystem

The audit subsystem is solely for the use of the system administrators. By providing the other subsystems with the ability to maintain logs, it keeps track of what the system is doing at any point. This information is output to the server console so

were something to go awry, the administrators would have a fair indication of what was happening prior to the malfunction.

Ceilidh Subsystem

The Ceilidh subsystem, named after the old Ceilidh system, handles the communication between the subsystems. It is responsible for establishing all the RMI connections between the subsystems. When clients try to connect, it is the Ceilidh subsystem that is queried to discern whether the other subsystems are up and running.

Course Subsystem

Each CourseMarker exercise consists of a collection of several different files. It is the responsibility of the course subsystem to provide access to these files when they are requested by the marking and submission subsystems. It also provides an object representation of all types of file that are encountered by the system, e.g. Java, C++ or text files.

Login Subsystem

The login subsystem attempts to recognise and authorise every attempted connection to the system. A list of users is created to assign students to modules. Passwords can either be individually assigned or CourseMarker can use their current system password. Only if the login process is successful is the client GUI loaded.

Marking Subsystem

The marking subsystem is responsible for the execution of the students' solutions against the sets of test data. The subsystem also calls the marking tools, collects the results and returns a mark along with feedback to the user.

Submission Subsystem

The submission subsystem essentially controls the calls to the other subsystems. It is this subsystem that decides whether marking is to be performed or not. The submission subsystem not only checks the exercise parameters to see whether archiving or marking should occur, but it also checks the number of submissions already made to see if the maximum has been reached or not. If marking is allowed then the subsystem makes the call and accepts the results which are then passed to the archiving subsystem for storage.

3.2 Marking Subsystem

Several desirable features were kept in mind during the creation of CourseMarker's Marking Subsystem. Such features included ensuring the security of CourseMarker, the ability to paramaterise the exercise settings, improved reliability and the most important from an assessment viewpoint, expressive feedback.

The view of the marking system differs depending on whether you are a student using CourseMarker or the course administrator. Whilst it is always important to consider the students viewpoint when creating courses and exercises, it is the internal details which are of interest to this research.

During the creation of an exercise, several files are used that can alter the way in which the system is to mark a solution. This enables the course administrator to have complete control over what is to be marked right down to the case of output text. The usage of the marking files will be described below, separate documentation also exists on how to create an exercise [Sym98].

3.2.1 Exercise Properties

It is possible to parameterise the exercise settings. The "`properties.txt`" file gives the course administrator the ability to alter the number of submissions open to students. In addition to this, it also allows them to turn off the automatic marking process, restrict the amount of output the student's program should have and also change

the status of the exercise from open to late or even closed. If an exercise is set as late only students who have been given permission are allowed to submit.

3.2.2 Marking Tools

The actual marking processes performed by CourseMarker are controlled by the "`mark.java`" file. Within this file, the types of marking to be undertaken are defined. It contains calls to CM's marking tools and sets the mark distribution between the tools. Each call is passed not just the solution file, but also another mark file which contains the tests themselves. These tests, depending on the tool concerned, can contain anything from a search string to typographical test parameters.

CourseMarker currently has at its disposal, a collection of six different marking tools. Each tool runs a completely different set of metric tests against the students' solutions. These are essentially quality checks for the programs which then provide users with feedback on how their submission fared. These tools comprise of two Java files. A control file, which collects all the files needed to perform the marking and sets up the system in preparation for the results. The second file is the actual tool itself, this file determines how the marking is to be performed, augmented by parameters provided in the marking files. The code contained within the Java files is procedural enabling system administrators to write new tools with very little knowledge of CM's workings. The collection of tools currently available consists of the following:

Typographical Tool

Several metrics have been created to perform tests against the style exhibited by students' source code [Mic96, AMUJ04], it is considered by many to be an important part of the assessment of programs including the original CourseMarker designers. The typographical tool performs tests against the students' source code, which analyse certain layout aspects of the program. There are several metrics available [Gib95], the most commonly used of which checks for correct program indentation, appropriate use of comments, amount of white space and line and identifier length. The exact parameters used by the tests can be altered when creating the exercise.

Features Tool

The features test provides the capability to search through the submitted source code. This is of use, if for example the students have been asked to use certain methods or concepts in their solution to a problem e.g. it is possible to check whether a student as used an array instead of a vector if required by the question specification. The functionality of the features tool is not just restricted to use on submitted source code, but can be used on any file. This allows for the analysis of files created during runtime. It also has uses regarding the security of the system as a whole, this again involves analysing the source code for commands that should not be run such as UNIX commands 'rm'.

Dynamic Tool

With the use of threads, this tool starts the student's submission and supplies it with test data. Then, utilising the power of the features tool, it checks the program output for the required responses. Unlike the other tools mentioned so far, it uses multiple files to assist in the marking, one containing solely test data and another containing the test oracles. The oracles are comprised of sets of individual tests, each with their own mark value, search string and possible feedback options.

Flowchart Tool

In order to mark flowcharts, the tool converts the diagram into a BASIC program. Once the conversion is complete, it can be treated as a standard program submission and marked using the dynamic tool returning both marks and feedback to the student.

Object-Oriented Tool

The Object-Oriented tool checks program designs created by the students. It is able to test for completeness, correctness and accuracy. The relationships between components are also examined. Students can be penalised for the inclusion of classes that are not required.

CircuitSim Tool

The CircuitSim tool allows the students to model electrical circuits. The marking is performed by simulating the running of the circuit with students losing marks for incorrect wiring.

3.2.3 Dynamic Test Execution

All of CourseMarker's marking tools run in a comparable fashion. During the creation of the exercise question, a `mark.java` file is written. Contained within this file is a collection of calls to the various marking tools required, including the compilation tool. The tool of most interest to the research is the dynamic tool so that the difference in complexity between marking CLIs and GUIs can be seen.

There are three stages to the marking process, loading the solution, running the test data and analysing the results. The way this occurs is as follows. The marking tool loads the students now compiled solution as a separate process, regardless of the operating system being used on the server. This separate process can then be monitored for security and reliability reasons. Once loaded, the marking tool is able to send the test data to the program through a data channel connected to the program. Care needs to be taken by the students when receiving the data as it is all sent in one batch not as required. Therefore, depending on how the data is received, it is very possible for the students to lose data. To avoid this problem, at the University of Nottingham a class has been created to handle the data input streams enabling the students to concentrate on the workings of the system until they have sufficient knowledge to capture the input themselves. In a similar way, the information output by the students' solutions are retrieved by the marking tool. The dynamic tool then enlists the assistance of the features tool as a string comparator to determine whether the information output meets the answers created by a model solution. Once complete a mark is then returned to the marking system.

3.2.4 Feedback

Within the realm of assessment, whether formative or summative, feedback is

equally as important as a well designed question. In his inaugural professorial lecture Hattie said "...the most powerful single moderator that enhances achievement is feedback" [Hat99]. Feedback enables lecturers to impart further information to the students. Correctly worded feedback can cause students to think critically about mistakes they have made or reinforce a student's knowledge with motivational comments. However, to be effective, feedback must be: constructive, timely and meaningful [Bon99].

The timing of feedback is important. If the delay between submission and receiving the feedback is too great, the work will no longer be fresh in the students mind and the comments will no longer have the same relevance. CourseMarker has no problems on this count. Due to CM's automatic marking mechanism, it is able to provide on-demand feedback. On-demand feedback is supplied as soon as the submission and marking process completes, which using CM is almost instantaneously.

Meaningless feedback has no use whatsoever. All comments returned in the feedback should be relevant to the tests being undertaken. CourseMarker has strict marking criteria meaning that at any point what the test is attempting to ascertain is known explicitly. This allows the question designers to create feedback, which is closely related to the tests performed.

The final requirement of feedback is for it to be constructive. Whilst feedback needs to point out mistakes students have made, a comment of "Wrong!" is of no benefit. The majority of students would prefer the system to pinpoint the exact problem in the exact location that their work exhibits, but experience has shown that finely detailed feedback can be detrimental to the students learning experience. Therefore, the feedback should indicate to the students the direction the errors are taking their program. CourseMarker's typographical tool is a good example of this sort of constructive feedback. The tool allows the question writer to supply five different feedback comments which are returned depending on how close to ideal a solution is.

CourseMarker delivers the feedback to students with the use of a "tree" GUI widget, see Figure [3.2]. The tree structure contained within the widget represents the breakdown of the marking into the test sets and individual tests that they comprise of. Each level of the tree has a description of the specific test and the mark awarded. The exact mark the student obtained is not shown, merely a coloured grade representation of the mark along with relevant feedback. To complement the mark, the marking tool also returns corresponding feedback which is displayed within a panel of the GUI.

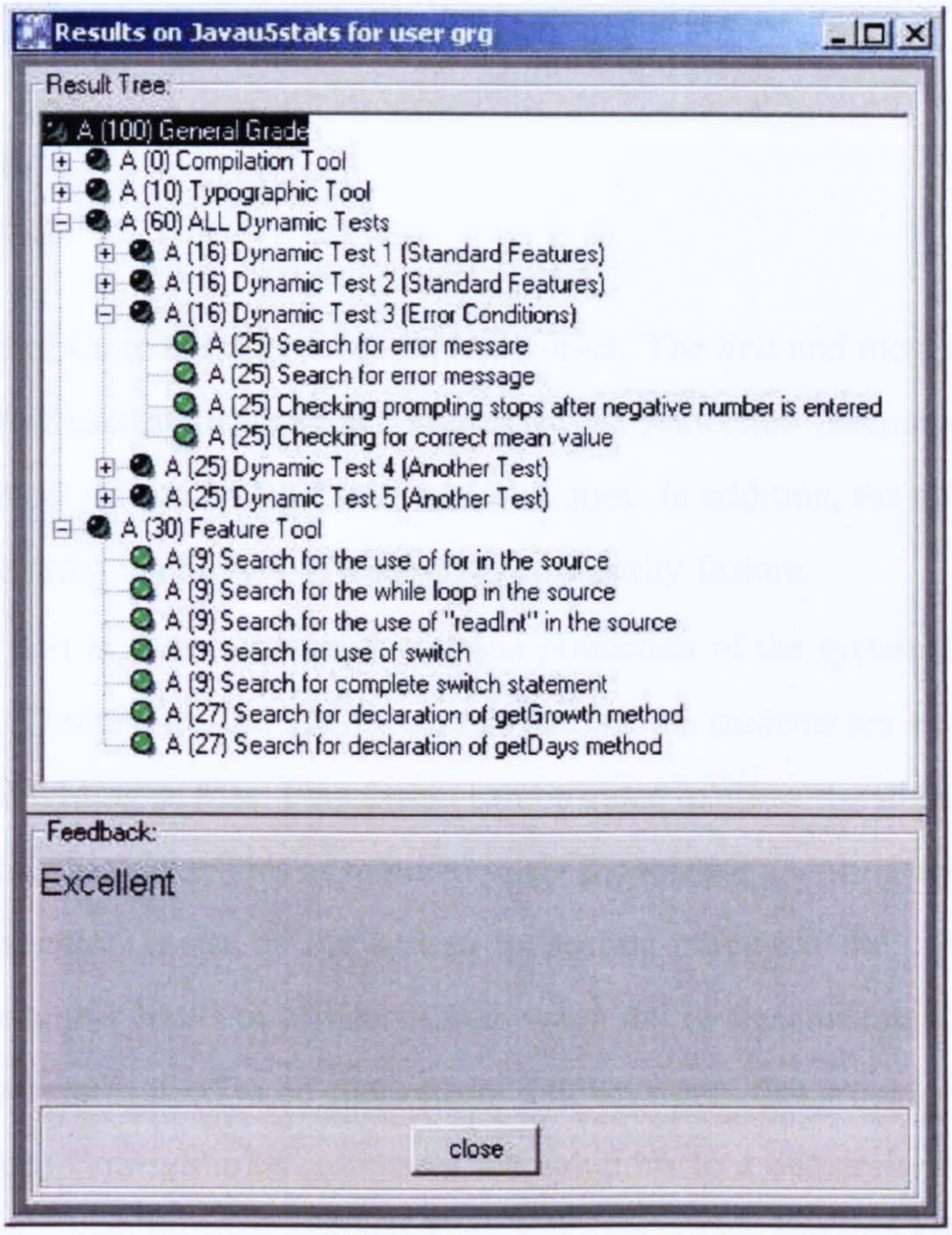

Figure 3.2: CourseMarker's Results Tree

3.2.5 Reliability

A major factor concerning systems for marking coursework is reliability. Students will rapidly lose faith and patience with a system that needs to be regularly restarted, especially when their marks are dependent upon it. CourseMarker's reliability comes from its capability to separate itself from each marking call. The extraction of each marking process means that were something to go wrong with the marking system or more likely with an incorrectly formed solution, it would not affect the system as a whole. The student would be awarded zero marks for that set of test data and the subsequent set of test data would be run against the program as if nothing happened.

Other controls are in place to stop damage occurring to the system. These include timing the runtime of the students' solutions in addition to counting and limiting the number of lines of output the program being tested can produce. This all but removes the chance of programs containing infinite loops from interrupting CourseMarker's

assessment process.

3.2.6 Security

The security of CourseMarker takes on two guises. The first and most obvious to the users is that of user authentication. The password protection mechanism is also encrypted to protect it from TCP/IP sniffing techniques. In addition, the client is also checked for tampering via the use of a session key security feature.

The second part is more concerned with the protection of the system from malicious programs. Through the use of CM's own sandbox, the students are stopped from both reading and writing to files. Permission can be given to allow the students access to files with certain file extensions as required to set file reading / writing assignments. As a student's solution is run by the system for testing purposes, the program will most likely have higher levels of privileges than when run by the students themselves. Therefore, the programs need to be checked for any keywords that would indicate the code is malicious. CourseMarker compares the solutions to a comprehensive list of commands, the list includes such commands as `delete`, `link` and `unlink` to name but a few. If any of the list occur in the source, the marking process is instantly halted before the program is run and a security exception is thrown. This not only gives the student zero for their solution but also prevents any damage occurring to the system whilst taking a copy of the program to analyse what the student was attempting to do.

3.2.7 Extensibility

Careful design of the marking server has meant that CourseMarker is easily extensible. New marking tools can be created, or existing ones altered easily without significant in-depth knowledge of how CourseMarker as a whole works. Due to the tools being written procedurally, the only controls that CM places upon the tools are how the results are stored and returned to the system.

CourseMarker's internal state can be queried from within the marking system, therefore, it is possible to author novel marking strategies that perform a variety of new tasks. For example, a marking strategy may be devised that compares the current student's submission with all the work that the student has ever submitted during the

course in order to establish whether the student is improving or not. In the latter case, CourseMarker would suggest to the student to seek advice and help from his / her tutor in order to solve any problems. Full details on CourseMarker's customisability can be found in a technical report [Sym01].

3.3 CourseMarker's Usage at Nottingham

CourseMarker has now been used in the University of Nottingham's Computer Science Department without significant problems since September 1999. The only major change to have affected the way in which programming was taught happened alongside the changeover to the CourseMarker system. The curriculum was changed so that instead of C, a procedural language, Java, an object oriented language, was to be taught.

3.3.1 Assessment

The students are subjected to continual assessment which takes a variety of forms. The majority of their assessment comes in the form of weekly exercises. The exercises are based on the topics they have been taught most recently and increase in difficulty each week. Not only are the students given exercises they must write from scratch they are also set exercises which involve debugging faulty programs. The bugs written into these programs are based upon common compilation errors the students have experienced in past years [AEH05].

The students are also subjected to more formal assessment. Over the course of the year, they must sit several multiple choice tests, which cover everything from programming concepts to following through a few lines of code to discern whether they can recognise what the correct output will be. In addition, there are several formal assessments, which come in the form of an online programming exam. They are undertaken in the same fashion as a normal weekly exercise, but the test is performed using a separate deployment of CourseMarker hosted on a secure network. The students must complete the exercise in the time available without the use of their notes or any assistance they could gain from the Internet or lab demonstrators etc.

3.3.2 Course Coverage

The first year programming course at Nottingham is spread over the entire academic year. It is also split into halves, each being taught by different lecturers. The first half focuses on procedural programming and the fundamentals of Java, finishing off by touching on objects, but this is essentially the domain of the second semester's course. Alongside objects, they cover other advanced topics such as Graphical User Interfaces and concepts such as the Model View Controller architecture.

While the weekly exercises still encompass the topics covered in semester two, the way they are marked changes. CourseMarker was previously only able to mark command line driven programs. This meant for the majority of the second semester the lab assistants were required to undertake the marking duties. However, for a student to be able to get real-time feedback on their solution, they needed to get it marked during the lab session. Therefore, the marking scheme was significantly cut down compared to the exercises marked by CM.

3.4 Summary

This chapter has given readers an insight into the CourseMarker system. The main topic of focus was CM's marking subsystem and how the marking itself is performed. Attention was paid to several areas of the marking system, such as its reliability, the security measures enforced and its extensibility. These are areas which are of importance to any marking system and will naturally be of significance when considering how to mark Graphical User Interfaces. The chapter concluded with a brief overview of the teaching and assessment practices at the University of Nottingham.

The rest of this thesis will attempt to break down the problem faced with regards to the automated marking of Graphical User Interfaces. In outlining an approach that can be used to solve the problem, this research will examine the tests required for sufficient test coverage along with the properties of Graphical User Interfaces that can assist the marking process.

In order to be able to test and mark Graphical User Interfaces, firstly an understanding of what they are and how they should be created is required.

Chapter 4

Graphical User Interfaces

"Making the simple complicated is commonplace; making the complicated simple, awesomely simple, that's creativity." - Charles Mingus

A User-System Interface is broadly defined to include all aspects of system design that affect system use [Smi82], the remainder of the program is known as the application. Therefore, a Graphical User Interface is nothing more than a graphical representation of the previous statement. Application usability has become critical for commercial software, users now expect to be able to sit down and use an application without having to first read a manual [Mye94]. These demands placed upon software companies have reiterated the importance of testing applications before release.

Graphical User Interfaces were first introduced in 1973 by the Xerox Corporation, but did not become ubiquitous until after such developments as the creation of the mouse pointing device. GUIs were designed to provide users with a simplistic way of interacting with the underlying computer system. To encourage faster acceptance of GUIs, the on screen objects were given metaphorical names e.g. window, menu and document. Such was the uptake of GUIs that now a large percentage of software released has its own Graphical Interface. Not only have GUIs become more common place but they also account for a large proportion of work required to create a system. Surveys have suggested that GUIs account for over half the project time and also half the source code [Mye93]. Their increasing commonality has required university courses to now incorporate GUIs into the core of their computer science degrees.

This chapter will discuss what Graphical User Interfaces are and their constituent parts. It will also distinguish between poorly and well made interfaces and how you can tell the difference. At the University of Nottingham, the students are required to take a first year programming module. The language taught in this module is Java, whilst the theory behind this research is language independent, it is Java for which it has been designed. Therefore the research from here on is so inclined.

4.1 The Evolution of Interfaces

Interfaces have always been used as a façade to a program, they are used to mask all of the unnecessary input / output tasks performed by a program, which are only of relevance to the programmers. Users are not concerned with the mechanics of an operation, just where to input data and retrieve their results. The first generation of interfaces were CLIs or Command Line Interfaces. They were entirely text based and

often involved users remembering specific commands or key strings to perform certain tasks.

It was not until the 1960s when this started to change. The movement from first generation interfaces, or command line driven interfaces, to second generation interfaces, provided programmers with yet another level of abstraction and allowed for direct manipulations to be visualised [Shn80]. It was the inception of the Windows, Icons, Menus, Pointing device (WIMP) model, that caused a revolution in the way interfaces were designed. Graphical User Interfaces, through the use of icons and metaphors, along with rapid, incremental and reversible actions, allow even those with a minimal level of computer literacy to use an application. The biggest change was the ability to visually represent actions using buttons e.g. submit buttons. This enabled users to interact with the systems more efficiently as they were no longer required to remember commands to perform tasks. Efficiency can be further improved through the use of keyboard shortcuts. Keyboard shortcuts involve combinations of key presses that will instantly perform the task requested. One of the first GUI applications created was the Star operating system, created at Xerox park [BRSV83, SCKH86]. It was one of the first instances of the use of metaphors to describe the components seen on screen. There was then a lull until the mid 1980s when several new GUI operating systems started to appear on the market, these included Apple's Lisa, the Amiga Workbench and Microsoft Windows.

Since the mid 1980s significant work has been done to improve the way in which Graphical User Interfaces operate in an attempt to improve efficiency and other operational parameters. The second generation interfaces have been superseded by a new generation of interface which have taken on a "doc-centric" design. This places the emphasis on documents with the applications themselves becoming less relevant. It has been suggested that the next logical evolution would be to create "role-centred" designs, which emphasise the tasks being performed [SP94]. However, this research is only concerned with the construction of the interfaces themselves.

4.2 The Teaching of Graphical User Interface Design

The importance of programming Graphical User Interfaces has already been emphasised with regards to simplifying system usage. Therefore, it is of paramount im-

portance that today's computer science students leave university with a good grounding in that skillset. As a skill that can only be learnt through considerable practice, the teaching must cover two areas. Firstly, students must be able to comprehend GUI components and how they can be combined to create a working interface. Secondly, and possibly more importantly, students need to learn the principles behind creating an interface. The students need to be taught how to tailor an interface to a user's needs without overly complicating the system, making it unusable.

As a skill that requires regular practice, as with command line programming, the students should be assessed as often as possible. However, to fully test a GUI program will take longer than a CLI program due to the fact that in addition to all the previous tests performed, the interface should now be checked for aesthetic and functional correctness. It is unrealistic to expect a lab assistant to be able to comprehensively check an interface and return it to the student within a reasonable amount of time. Therefore, this task requires automation. In order to be able to automate it, a certain amount of knowledge on both the make-up of an interface and the way in which they should be created is necessary.

4.3 What are Graphical User Interfaces?

Without input or output, computer programs would be of no value. However, it is only the actual programmers of the system who are concerned with how the input and output occurs. The every day users of a system are solely interested in the results I/O provides. Programmers are able to mask the ways in which their programs perform all I/O processes. It can be accomplished using Graphical User Interfaces (See Figure [4.1]). GUIs are essentially no more than a façade that can be used to hide program workings whilst still providing the same functionality in a user friendly way. The removal of the command prompt and the limitations it imposed has allowed for extra levels of complexity previously unobtainable such as concurrent multiple data input.

Graphical User Interfaces have numerous advantages over text based programs, the most obvious are the novel ways in which data can now be displayed. Gone are the days of ASCII tables etc. exact graphs can now be created and graphically rendered at run time, thus allowing for swifter interpretation of the information being displayed. Another advantage is the capability of inputting multiple data simultaneously. It is

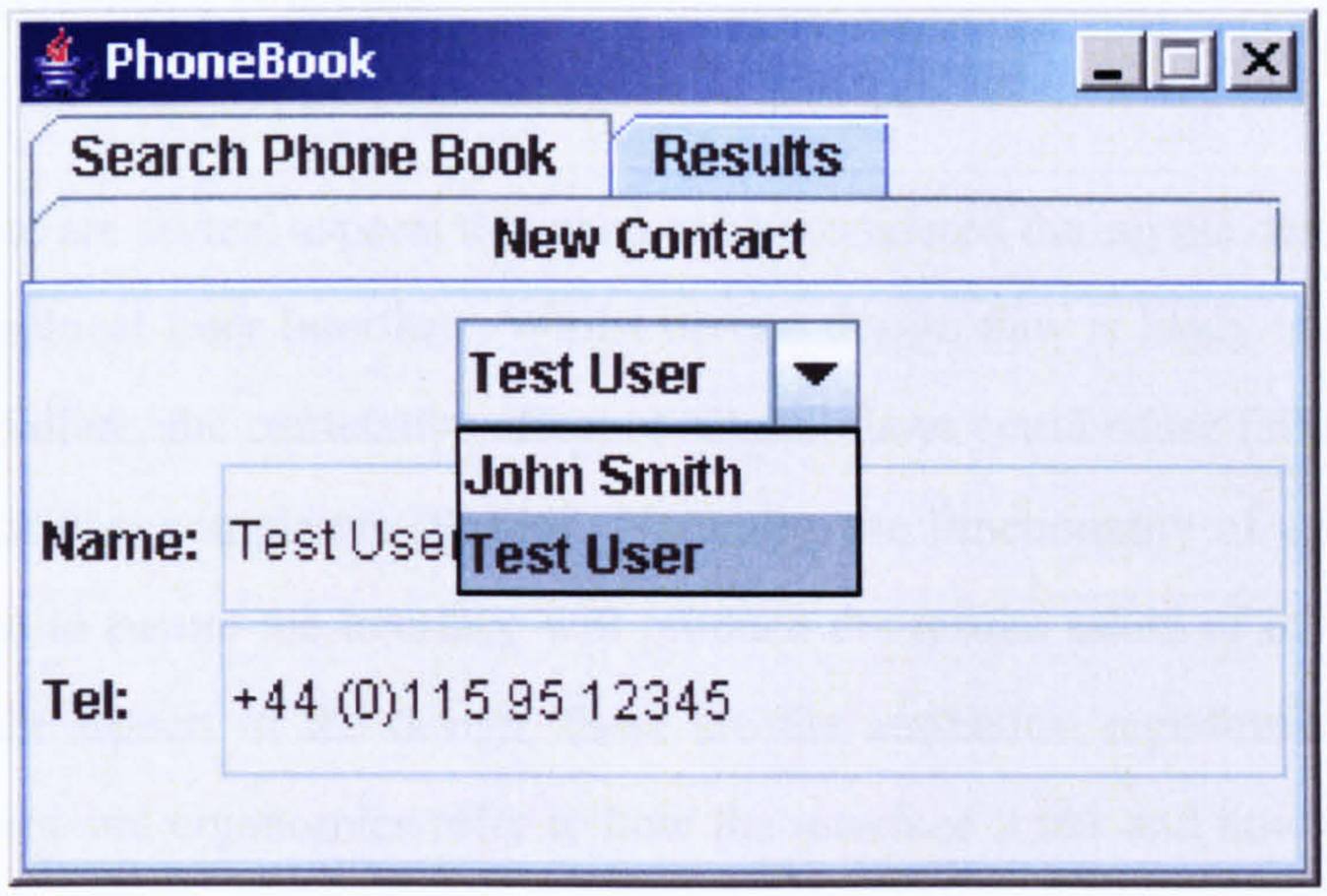

Figure 4.1: An Example Interface

highly likely that such data will be analysed consequently rather than concurrently. However, the majority of users will be unaware of this ordering.

Advancements never come without a cost and GUIs are no exception to this rule. Their inclusion in a system adds to the list of considerations that need to be taken into account during the program design. This list includes the real time requirements of both the interface and program, and the need for added robustness due to the wider range of data that can now be entered by users.

The User Interfaces themselves are constructed from a large number of objects. All the objects are customisable and allow the programmers to define where they appear on screen and what operational properties they and the system have.

4.4 Design and Construction

As previously mentioned, the design and creation of Graphical User Interfaces constitutes a significant proportion of a project's resources. This is in part down to their inherent complexity but also rests upon the fact that being an expert programmer does not make you an expert on interfaces or vice versa.

4.4.1 Design Problems

There are several aspects that need to be considered during the design and creation of a Graphical User Interface. Whilst no one design flaw is likely to cause complete system failure, the cumulative effect of several flaws could cause failure, poor performance or user complaints [SM86]. Naturally, the functionality of a system needs to be tested to ensure the interface will produce the results asked of it. There are other important aspects of the design, these are the aesthetics, ergonomics and usability. Aesthetics and ergonomics refer to how the interface looks and how easily users can interact with it, where as the usability refers to how easy a system is to use.

Software is not just arbitrarily created, it is designed specifically for a group of users. The users and what they hope to achieve with the software is already known before the programming starts. This should not be where the users' participation in the creation process ends, they should also be enlisted in the testing phases. However, care should be taken when having users test. According to Jakob Neilsen, the first rule of usability is to watch what users do and not listen to what they say [Nie01]. The same approach should be used when redesigning existing software, but user consultation can be important to discuss observations previously noticed [RSP95].

Suggestions have been made on how to reduce the amount of testing that users need to undertake [NP04]. The ability of computers to automate tasks, such as testing, enables programmers to minimize the need for user contact. This lessens the possibility of design flaws through erroneous information from users. Whilst the automated testing of a system can never replace the way in which an actual user would undertake a task, it can be used to ensure that the tasks that can be performed are working correctly. This is also the premise for automated marking. The degree of realism demonstrated by automated marking will be higher than for testing a commercial system solely due to the comparable complexity of the systems. Programs written by novice programmers will be more simplistic due to the basic level of their skills. It will therefore be likely that there is only one way in which tasks can be performed, thus meaning an automated marking system will be replicating tasks performed by human users.

4.4.2 What Constitutes a Good Graphical User Interface?

Every competent programmer has the ability to create a Graphical User Interface. Unfortunately, it is far simpler to create a poor interface than an effective one. Attaining perfection with any interface you create is a Herculean task. The goal of creating a functional GUI, although obtainable, is often way out of reach for most programmers who undertake the task. It takes many years experience to combine the users' requirements with the conventions and capabilities of the Graphical User Interface. It is often the case that the people with experience are no longer programmers but work as specialist interface consultants. It is a common occurrence for an interface consultant to be brought in to assist programmers in the creation of new software.

Whilst it may take many years to fully master the art of the Graphical User Interface, there are however, several guidelines [NM90] that if followed will significantly improve the quality of the interfaces produced. As outlined by Johnson [Joh00], the basic principles all designers / programmers should follow when creating a Graphical User Interface are:

1. Focus on the users.
2. Conform to the user's view of the task.
3. Try it out on users and then fix any problems.
4. Function first, presentation later.
5. Do not complicate the user's task.
6. Promote learning.
7. Deliver information not just data.
8. Design for responsiveness.

As the list highlights, the interface should be focused around the user's requirements. A programmer's creativity and flair for design should always come second. The principles can be separated into two different sections. Principles 1 through 3, whilst always important, have more relevance from a commercial software perspective. They encourage interaction with users to enhance the system being created, not a plausible operation with student coursework. The remainder of the principles do have

importance with regards to student assessment. They focus on keeping the interface clean and tidy to allow users to quickly gain an understanding of how the program works (5, 6) whilst efficiently delivering as much useful information as possible (7). The other two principles (4, 8) focus on the functionality of the program and how programmers should ensure that the system works efficiently before beautifying the design.

Whilst following the guidelines will help to create an interface that the users are happy with, it is not all that needs to be considered. The complexity of Graphical User Interfaces means that there are a large number of areas on which significant focus could be placed, however, the focus shall be on the 3 concepts that have the most relevance. Therefore, when designing and creating a GUI, programmers should have three things at the forefront of their thoughts, consistency, usability and simplicity. Although such statements may seem sweepingly obvious, quite what is meant by each one shall now be explained.

Simplicity

It stands to reason that the more overly complicated a system is, the harder it will be to use. While something may seem simple to the programmer who created the program, users have not had the benefit of seeing the program in all stages of design. Therefore, in order for users to be able to quickly comprehend how a system works, it needs to have been designed with simplicity in mind. This does not mean the programmer should patronise the users and treat them like children (unless of course they are). It would be easy for the programmer to just create one big button for the users to press setting the program running and just assume what all the tasks required will be. However, it would be of little to no use to those using the system.

One way simplicity could be achieved is through minimalism. While it may seem like a good idea to display all the possible options at any one time, such cluttering of the screen makes it more difficult to distinguish between them. Moving certain levels of functionality will make the screen far more readable and allow users to select the settings they are striving to find.

Consistency

There are numerous conventions that are adhered to by most, if not all (well written) Graphical User Interfaces. One such convention follows that any option in a menu, which when selected will load a separate options menu should be suffixed by ". . .". Holding consistency with common interface stylistics means that new users to the software will automatically feel a certain familiarity with the system. Naturally this is an advantage as users will feel less daunted by the new software and, if they have not already done so, more willing to change over to the new system.

Consistency does not only refer to systems already created, but also to the rest of the system under test. The importance of creating a familiar rapport with users has already been mentioned. There are always programmers who want to be different and create a system using a new set of rules. Whilst this is not necessarily the best idea from a business point of view, there is nothing, besides poor programming, to stop it being usable. Consistency in this situation is as important. If the system performs the same task in different ways depending on which screen it is on, the learning curve for the program may be too steep for the users to get to grips with.

Unfortunately, it is very possible to create a Graphical User Interface that is fully consistent but is still unusable.

Usability

Saying that an interface should be usable, may seem like an obvious statement to make. However, there are many subtle ways in which a well designed interface can be ruined through not giving the required respect to the smaller elements of the GUI.

An example of a major usability issue is the failure by a programmer to attach the necessary function to a button, thus rendering the system worthless. This is an unlikely example of a usability problem, more common problems are generally less glaring mistakes. Textual errors are a good example of this sort of problem. Poorly written menus can turn a well designed system into a maze with no obvious way out. Through not structuring the menus in a clear way or changing the location of menu options depending on the state of the program will result in confusion for the users. Also if the menu options are not labeled clearly and concisely it will again make the

system harder to use.

The errors mentioned above are relatively trivial problems when you consider all the work that goes into creating a fully working system. It is these small inconsistencies that can turn a good program into one that will never be released into the public domain. Alongside the numerous books that have been written on creating the ideal GUI, taxonomies on how to assess the usability of GUIs have also been created [IH01].

4.4.3 Hierarchies

Hierarchies are used to provide both structure and order to whatever situation they are applied to. They are recognisable in many everyday situations whether it be government, the food chain, or programming languages. Exemplified by the development of object orientation in the late 70s and early 80s. This was a new programming methodology that involved the encapsulation of functions inside packages of data known as objects. These developments went on to introduce us to the concept of inheritance. Similar to genetic inheritance it allowed objects to acquire the traits of other objects. It is this hierarchy of objects that gives us the Object Oriented programming languages that are currently in use.

With any modern programming language, it would be expected that the object orientation has propagated throughout the entirety of the language. This implies that one would also expect a stringently hierarchical structure to the building blocks of Graphical User Interfaces. With a language such as Java, there is no disappointment. The hierarchical structure exhibited by GUIs is a testament to how inheritance and other OO techniques can be utilised whilst providing us with intrinsic information about what we can expect to find at any point.

The most common way to represent a hierarchy pictorially is as a tree structure, programming languages are no exception. Due to their OO nature we are able to think of GUIs as tree structures. Located at the head of the tree is the window construct, this is required by all GUIs into which the remainder of the interface is built up. Working down the levels of the tree, the containers are encountered. Finally at the roots of the tree the objects located there can be one of two things, either the functional components of the interface or empty containers, as they do not have any internal objects and therefore no children in the hierarchy.

Components

Graphical User Interfaces themselves are created from of a large collection of objects (See Figure [4.2]). It is these objects that are used to both define the operational properties of the system and also set how the interface is displayed to the user. In order to achieve this, the objects can be classified into one of two different types: containers or components.

The containers as their name suggests, are predominantly responsible for storing the objects that comprise the interface. Included in the list of objects stored by the containers are layout managers, which enable layout criteria to be predefined. Containers are not only able to store just components but also other containers. This allows for a degree of nesting, providing the GUI programmers with finer control over the location of the objects.

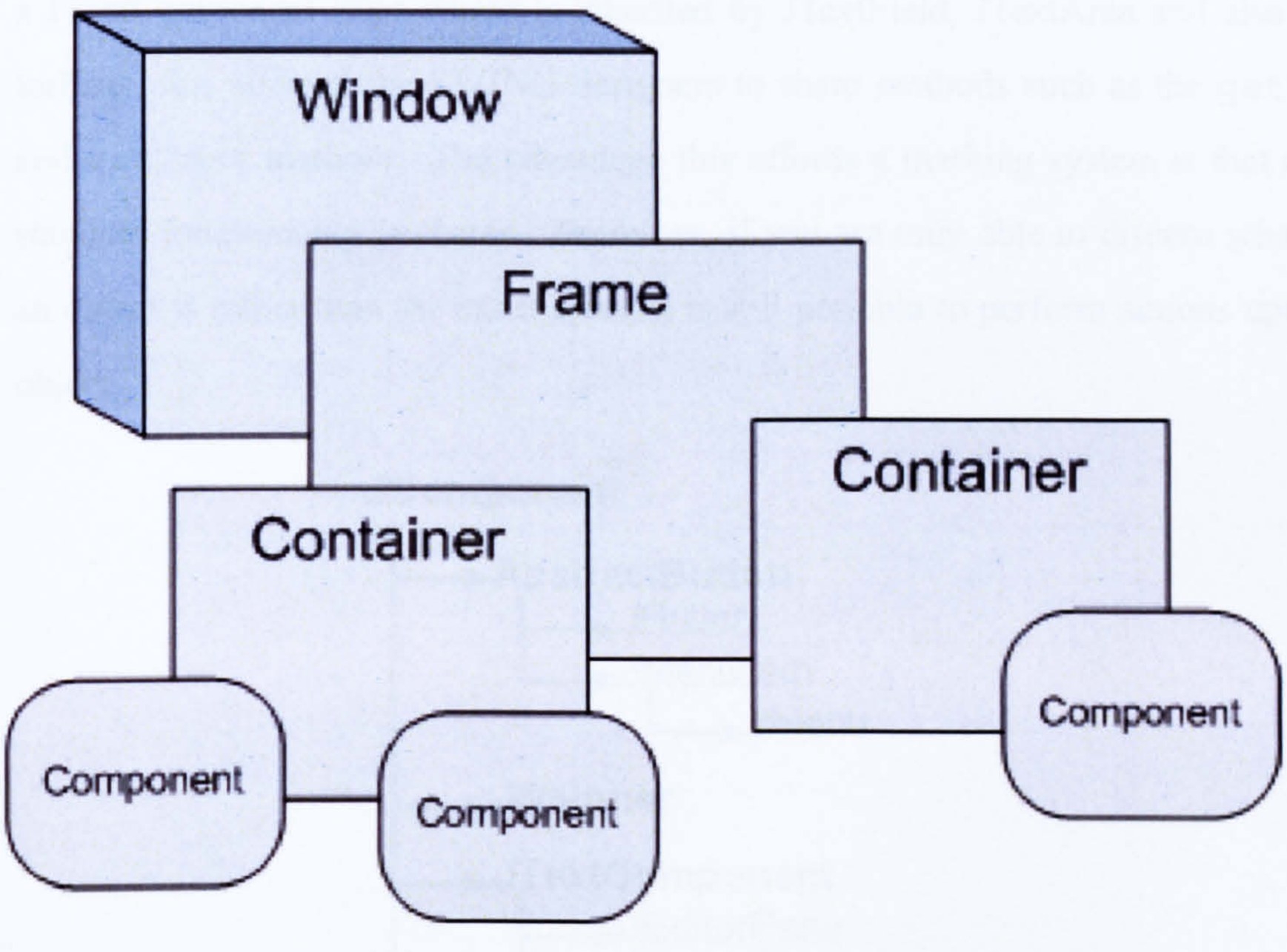

Figure 4.2: A Simplistic Breakdown of a GUI

The components are the actual input / output tools of the GUI, such as the text fields and buttons etc. They provide the program with its character and functionality. Each component is fully customisable, this gives the programmer the ability to explicitly define the function it is to perform. The look and feel of each component can also be altered, for example setting the text that appears on a button. Naturally, this allows for distinction between the components in the interface. This ability to differentiate

between objects is not only of help for the users of the system, but provides a handle that can be used when attempting to locate components in the GUI.

Having mentioned how components can be altered to make them dissimilar, it is imperative to talk about their similarities as well. There is a significant amount of commonality between interface objects. This commonality allows for a degree of standardisation as far as testing the interface is concerned.

Component Similarity

The use of Object Orientation and Inheritance means that similar objects are able to share functions and attributes by extending particular classes. Graphical User Interface frameworks also take advantage of this, for example, Java SWING collectively groups components whose inherent functionality is similar (See Figure [4.3]). There exists a JTextComponent class which is inherited by JTextField, JTextArea and also JEditorPane, this allowed the SWING designers to share methods such as the `getText` and `setText` methods. The advantage this affords a marking system is that all the standard functionality is shared. Therefore, if you are only able to discern what type an object is rather than the exact class, it is still possible to perform actions upon the object.

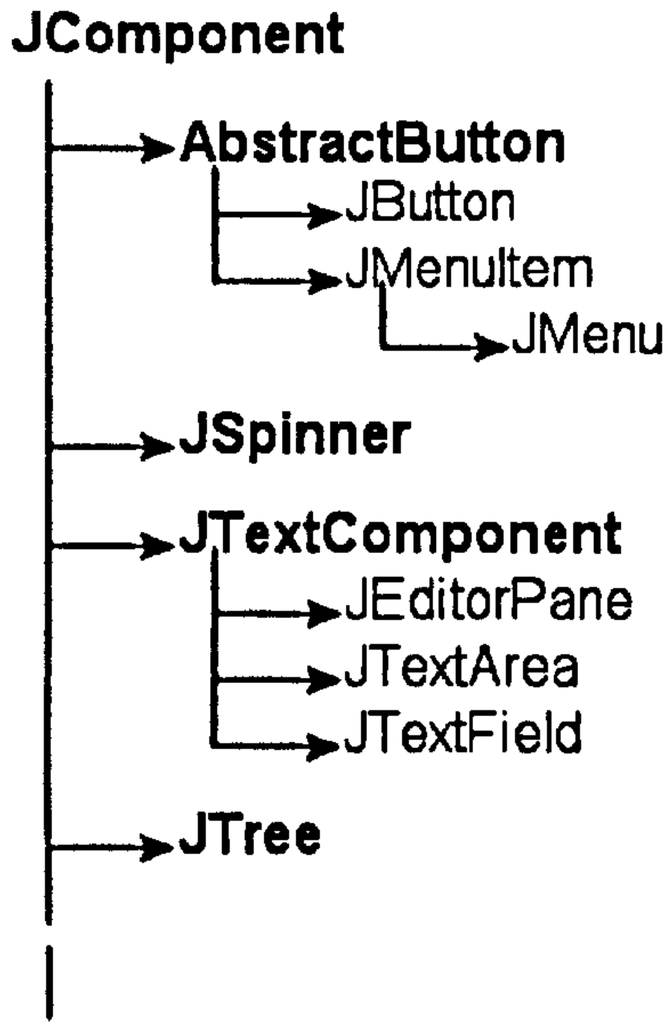

Figure 4.3: GUI Component Inheritance

The structure the SWING designers have used means that when tests are being

constructed to mark GUIs, the same groupings can be used to save on repetition. By being able to bypass what type of object is interacted with e.g. which type of text component the system is entering text into. It also means that it will still function whether a standard Java object, or a customised version of a GUI component that has been created specifically for the application under test, is being used.

4.4.4 Interface Layout

The layout of an interface is as important to its usability as the construction of the individual components. There are several factors that affect the way in which users view an interface. These are mainly based around how the eye interprets what it sees. This includes such things as the size of a component, the colour contrast with its surroundings and the order in which the objects are placed. The order of the objects, refers to the way in which users are most likely to read what is on the screen, whether it be right to left, or left to right, depends on the country in which it is being developed.

The visual appearance of a GUI is determined by means of a combination of techniques. Layout managers are used to arrange the objects stored in a container. There are several default layout managers in a language such as Java. They vary from ordering the objects horizontally or vertically to creating a grid in which the objects can be placed. It is also possible to create a unique layout manager if so required. However, it is known that it is not just components that can be stored in containers, they themselves can be nested, allowing for finer control over where the objects are located.

4.4.5 Summary

In this section various concepts concerning the design and construction of a Graphical User Interface have been covered. The main principles behind the creation of a well built interface have been broken down into three obvious headings, consistency, simplicity and usability. While a long way from being comprehensive, they provide a starting block for novice programmers to work from. Also discussed was the make up of the GUI components themselves and how inheritance allows for easier creation of objects. This along with the hierarchical structure of a GUI is of great assistance to programmers.

4.5 States

In order to both test and mark a Graphical User Interface, the state of the interface at any point in time is required knowledge. Without this information, a user whether real or automated, would be unable to cause the program to function correctly. It is imperative that the user knows what the system is attempting to do and what functionality is currently open to them. The question is "How can the state of the system be discerned?". A way to describe an interface is needed.

It has already been stated that a GUI is essentially a collection of components with each constituent component being fully customisable, for example, buttons can be disabled and components can be set visible or not as the situation requires. Each component has a set of properties, these describe the objects appearance along with other operational settings. As the interface is used, the properties of each object will change according to the current situation. If a snapshot were taken of all the properties of all objects at any point in the execution cycle of a program, it would offer up an overview of the system, or the programs state at that moment in time. As previously mentioned, the objects' properties change over time, consequently so will the state.

It is feasible to describe a Graphical User Interface as a collection of states, each state consisting of a collection of objects and their properties [MPS01]. A valid state is one that is reachable from another state via a series of legal moves or in this case, defined operations. These operations are the events triggered by the objects within the interface. The advantage marking has over testing, is that the operations the interface can perform were predefined in the question specification. By knowing what operations the system will perform, it allows the expected state to be determined.

The marking of a Graphical User Interface can therefore be broken down to the comparison of the current state and expected state along with all the transitions between them. This is achievable through the analysis of all objects and their properties contained within each state.

4.6 Summary

This chapter relates what students should know about Graphical User Interfaces

and their actual make-up to how that information can be used for automated marking. In order to create a viable GUI, programmers should ensure their systems are consistent, simplistic and usable. If students keep these theories in mind, they would be on the way to creating well designed GUIs.

To be able to create GUIs, students also need a comprehensive knowledge of the components which are used to give GUIs their functionality. The hierarchical structure of interfaces automatically imparts certain information about the objects depending on where in the hierarchy they appear. However, it is the use of customisable components that allows us to determine what state the interface is in. By analysing the properties of all objects present, and with knowledge of the functions the interface can perform, the final properties of the aforementioned objects can be determined. It is this transition between states that allows the interface to be marked.

In the next chapter the tests that are required by a Graphical User Interface marking system will be discussed.

Chapter 5

Testing And Marking Graphical User Interfaces

"Interpretation is the revenge of the intellect upon art." - Susan Sontag

Having highlighted the absence of a system capable of truly marking a student's Graphical User Interface, the GUI structure was analysed. With the knowledge gained from this analysis, it was possible to identify certain properties of the GUI components that could be used to assist the marking process. This identification provided only a theoretical basis for the recognition and execution of operations. Before the theory could be realised, a further in-depth look at the problems faced when considering the marking of GUIs was required.

This chapter will not only look at the challenges that will need to be overcome, such as location of the objects and suggest what needs to occur to solve them. There will also be a look at the tests required, what they need to achieve the results and the theory behind them. In order to fully assess a GUI, the program will need to be analysed from several perspectives. The types of testing required are as follows: aesthetic, dynamic, feature and typographical testing. In addition, the way the tests will need to be represented shall be explained.

5.1 Problems Faced by Testing GUIs

Graphical User Interfaces, as already shown, are complex structures. The ability to customise all available components may give the final interface a great deal of flexibility, but it creates problems when attempting to test a system. By allowing the components to appear anywhere on screen and perform any action, it makes it impossible to predict where the objects are and what they are capable of.

The problems discussed in this section have been set down logically in the order in which they would occur in the actual process of testing and marking a GUI. They are followed by a commentary on setting questions for interface creation exercises. It is not enough to have a good system to mark the programs, the question specifications also need to be well written.

5.1.1 Locating Components

Of the problems faced when attempting to automatically test a Graphical User Interface, the most critical is being able to identify and locate the required component.

It is of relevance, not only when trying to perform the test, but also when the marking is being performed. The problem is exacerbated when large groups of students are involved. It is unlikely that all the students will create comparable GUIs with the components in the same place, even if the question specification were to state the desired location of all the objects. Therefore, a method for locating the components is required that is flexible enough to work on altering layouts.

To be able to identify the component being looked for, it is imperative to know what type of object the component is e.g. a button or a text field. Each active component[1] in an interface is unique, were two objects not unique then one would be redundant as they would perform the same task. This is something that should be checked when marking. It is significantly easier to locate objects known to be different, all that is needed is a handle with which to do so. There are several properties possessed by objects that could be used for identification. These include such properties as the component's text, either the text contained within a text field or a button's label, or possibly even whether an object is currently in focus or not, this would more commonly be used with Frames or dialog boxes.

It shall for now be assumed that the objects have been located and that the operations are to be performed.

5.1.2 Performing Operations

Once the components have been located, it is time to concentrate on performing the operations required. In order to perform an operation certain information is needed:

- The operation.
- Any test data.
- Knowledge of the object on which to perform the operation.

This information needs to be supplied to the marking system in a way it can readily interpret. After the interpretation of the testing instructions, the testing may proceed. However, before this can be attempted, the best way of running the tests needs to be ascertained.

[1]Not counting objects such as spacers used for screen layout.

One way in which the operations could be performed is by taking control of the native input of the machine on which it is being run, this can be achieved using such tools as the aforementioned Java Robot tool. However, problems can arise if the exact location of the objects is not known or if someone is attempting to use the underlying system at the same time. The robot tool could cause unexpected errors to occur not just with regards to the testing but also with the machine on which it is being executed. A more effective way would be to perform the operations directly onto the components themselves. This would involve having access to the objects stored within the GUI, as therefore, the system would have access to all the same methods the programmer had when the program was being written. Being able to access the internal components of the GUI would remove the dangers to the underlying system, e.g. being affected by a rogue mouse click.

5.1.3 State Comparison

Once a system is in place to locate the objects, the state comparison becomes a mere formality. As access to the objects is ensured, all that is needed to be able to return a mark to the students is the analysis of the properties of all the components involved. This could be as simple a task as checking the text output in a label or text field.

Whilst the most obvious use for state comparison is to mark the interface, it can also be used to ascertain the program is still working according to the specification. By checking the state of the system as the testing progresses, any potential problems can be identified early. Such problems could include the absence of buttons required to trigger functionality. If identified early, the marking system could adapt, bypassing certain sections of the testing. The student would then be returned a failing grade for a particular test, but it would allow for the next part of the marking to proceed without any complications.

These two points can be demonstrated with the use of a couple of examples. Firstly, the use of state comparison for marking purposes. It is known what buttons make up the initial state of a calculator and what the value of the answer panel should be once the test has completed. Therefore, it is possible to search for each button in turn to establish if it exists and if necessary whether it has been enabled, and also to check the

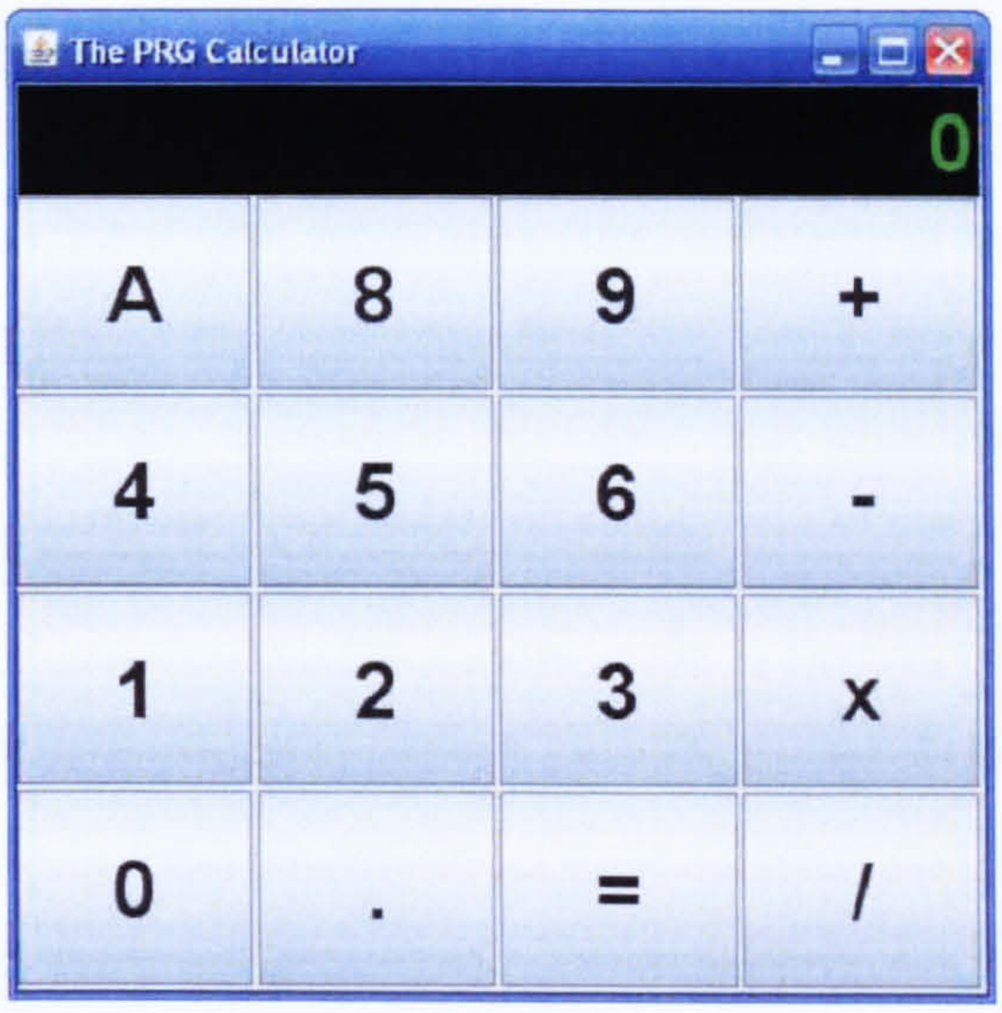

Figure 5.1: A Faulty Calculator

value contained within the text panel. If these tests were to be run against the faulty calculator example (See Figure [5.1]), it is clear to see that the majority of the tests would be successful with the exception of the test which looks for the button labelled '7'.

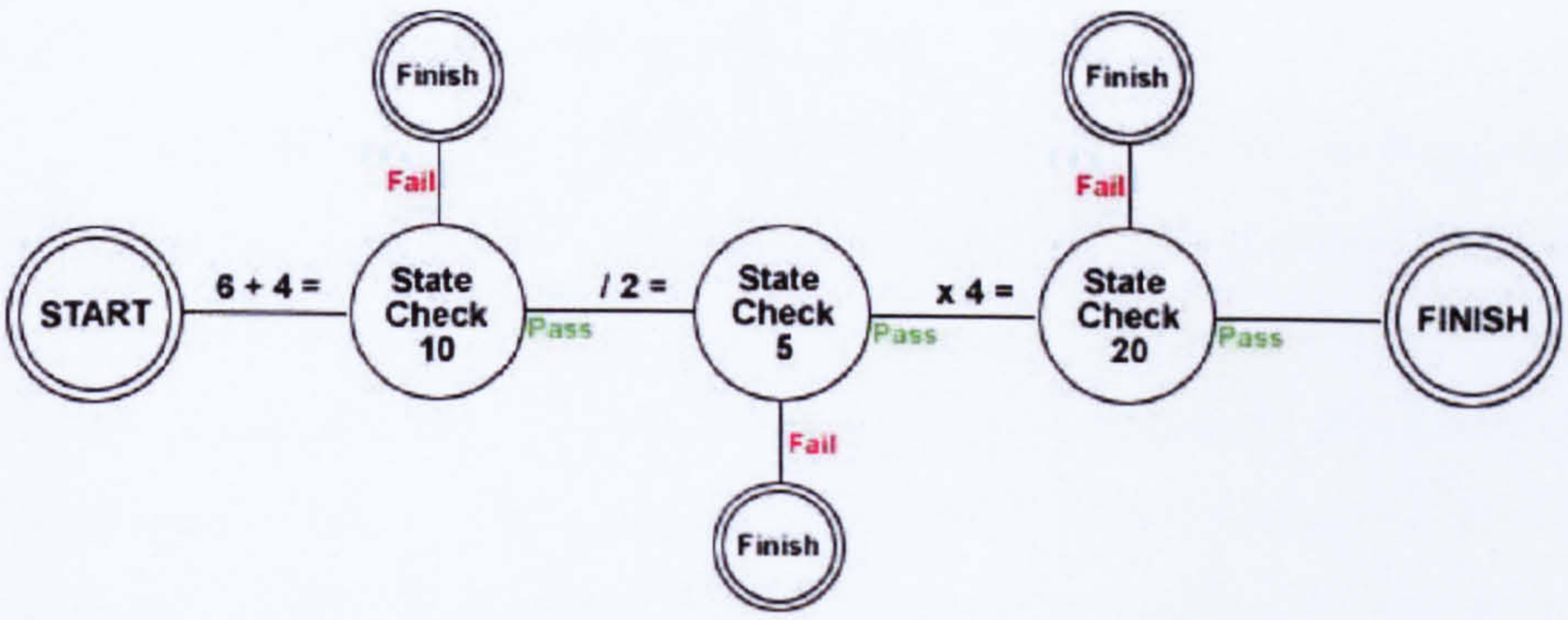

Figure 5.2: State Transistion Checks

The use of state comparison to assess the functionality of a program can be demonstrated with a state transition diagram (See Figure [5.2]) and the faulty calculator. The state transition diagram describes the tests and checks that need to be run. As can be seen it consists of a number of sequences of button presses e.g. 'x 4' and state checks which in this example only check the current running total. Given that the multiplication operator does not multiple but subtract, it is clear that the state checks will fail on

the third attempt as the system will find '1' when it is looking for '20'. The benefits that this affords is that the system is then able to return a fail result knowing that the final test will also fail or pass incorrectly. This will save on both time and also server resources. More importantly, knowing the exact point at which a system fails enables it to return more specific feedback to the users. This feedback is invaluable for users to locate and debug problems with their program.

The only significant issue with regards to the comparison of the states, is the way in which the state / comparative tests are described. The description needs to include information on how to identify the object under scrutiny along with the property being compared, not to mention the feedback for the student. For this system to operate, a new way of storing the test oracles was devised.

5.1.4 Setting the Assignment

Although it is the marking aspects that are the predominant concerns, the marking cannot be performed if a question has not been set. Unfortunately, this is not as simple as coming up with an idea for the students to tackle, the question needs to be tailored for both the students and the marking system.

When writing assignments for students, the question needs to be specific enough for them to understand the task at hand. Care must also be taken not to write fully comprehensive instructions as otherwise there will be no opportunity for the students to use any creative flair they may have in solving the problem. These are educational restraints required for the purposes of assessment and include such restrictions as demanding students use a dropdown box instead of radio buttons for an options select.

The opposite is true for the marking system, the more free range the students have, the harder it will be for a marking system to analyse. In order to locate objects, a marking system needs to have a handle with which to identify the correct component. For effective marking to occur, the question specification needs to define additional constraints to restrict certain aspects of the task in order to ease the problem of the object location. Marking restraints would consist of ways to identify objects, such as the button labels e.g. the multiply button on a calculator could be specified to be either 'x' or '*'.

With the exception of needing to locate the objects there are no other constraints,

other than the actual question, that need to be placed upon the student. By balancing the requirements of student and the marking system in addition to following the standard guidelines from Section [2.1.1], comprehensive assignments can be written.

5.1.5 Summary

The marking of Graphical User Interfaces brings up many issues of importance that need to be addressed. However, when it comes down to the actual process of marking, there is one issue that stands head and shoulders above the others, this is the method of locating the components within the interface. Its importance stems from the fact that all tasks a marking system needs to perform are reliant on this one operation whether it be to actuate a test upon an object, or just to check the object's attributes.

5.2 Test Coverage

An important aspect of testing a system is that of coverage, this refers to the amount of the program that actually gets tested. It would be relatively simple to test only the major functionality of a GUI, but that would not ensure its correctness. In order to guarantee a program works, it needs to be tested comprehensively. Complete coverage, the testing of every possible situation, is the only method of ensuring the interface created is without error[2]. It is evident that to perform testing of this nature would require a significant length of time, especially if a whole year group is to be tested. This is not unachievable when the testing is automated.

Whether the testing is done by hand or automatically, the tests still needed to be designed at some stage of the process. If the tests are converted into a form the marking system can comprehend, it will be possible for each solution to be fully tested. This is assuming that all programs under test have the same or less functionality than the question specification demands[3]. Another reason for the comparative ease of obtaining complete coverage is the simplicity of the students' solutions. Unlike commercial systems which need to be able to perform a myriad of tasks, students are rarely asked to create a system capable of executing but a couple of operations.

---

[2] Although this does not account for load testing.

[3] Additional functionality would not be marked automatically unless expected and included in the test descriptors.

In order to accomplish the goal of complete test coverage, especially from a marking viewpoint, a Graphical User Interface needs to be marked from several perspectives. Not only does the source code need to be checked for typographical and feature based correctness, it also needs a new approach to the dynamic testing along with an analysis of the actual design and layout of the interface. These testing mechanisms and what they aim to determine will be discussed over the next few sections of this chapter.

5.3 Typographic Testing

The only difference when testing GUIs over CLIs is the way in which the information is entered and displayed by the program being marked. Therefore, this will have no bearing on the way in which the source code should be written. All typographic standards e.g. the way in which the code is indented, or ensuring the source code is sufficiently commented, still holds true even when GUIs are involved.

When the existing typographical tests were created [Gib95], they were designed to allow for variations in programming stylistics. Each test is parameterizable, with adjustable boundaries for each marking metric enabling the question writer to make adjustments if for example the average line length is going to be longer. Therefore, these tests are able to be used for marking solutions that use Graphical Interfaces as well as those that are based around the command prompt.

5.4 Source Code Feature Testing

Feature testing is not just of use with Command Line Interfaces, it is also of use when Graphical User Interfaces are considered.

Obviously, the tool still has the use for which it was designed, ensuring that the students have incorporated certain ideas within their program. By examining their source code, it can be determined which methods and constructs the students have used and whether they meet the specification set. It can also be adapted to enhance the reliability of the marking system. A feature tool can check the source code for regular expressions and then return a mark depending on whether they were found or not. This power can also be harnessed for other means. Students' solutions could be searched

for problems that could arise during testing. These problems include the creation of windows which take the focus of control away from the marking system, e.g. modal windows. If unexpected, a modal window could cause the system to halt and wait for the marking system to timeout. A feature tool could also be used to ascertain whether the program contains the buttons expected, if not this would cause the tests to fail. By checking beforehand, and depending on the problem, it could be possible to perform the operations in an alternative way e.g. by using menu options rather than on screen buttons. This would not only improve the reliability and stability of the system, but also allow for slightly more creative flair to be shown by the students.

5.5 Analysing the Aesthetics of an Interface

A lot of emphasis is placed on creating an interface that not only meets the users' needs but is also simple for them to learn to use. It was earlier shown that this simplicity came from a variety of sources and having a working program was just one of them. The Graphical User Interface needs to be well designed and have a clear, well organised layout. A disorganised interface is only going to cause problems for the users of the system. It is not just the functionality of the students' solutions that should be marked but also the aesthetic and ergonomic aspects.

Before continuing, what is meant by aesthetics and ergonomics should be clarified. The Merriam-Webster Dictionary defines them in the following ways:

> **Aesthetic** - a particular theory or conception of beauty or art; a particular taste for or approach to what is pleasing to the senses and especially sight.

> **Ergonomics** - an applied science concerned with designing and arranging things people use so that the people and things interact most efficiently and safely – called also human engineering.

Although the definition of ergonomics still holds true when talking about the design of Graphical User Interfaces, the definition of aesthetics needs a slight modification. The aesthetics of an interface are still concerned with what is pleasing to the senses, but are also broader than that suggests. They incorporate any way in which the senses interpret the interface, for example, the way in which the eye would read the screen.

It is the optical effects the design of the system has on users that need to be measured and evaluated.

Essentially what is required is a collection of testing metrics, similar to those contained within a typographical tool. The metrics and the way in which they function shall now be discussed.

5.5.1 Interface Metrics

Due to the massive flexibility of Graphical User Interfaces, there are numerous different styles of metric that can be created to assess whether the interface has been well designed. However, this theory is not going to be implemented by companies wanting to assess their latest piece of software for correctness, it will be marking students' programs. Therefore, an implementation of a comprehensive set of metrics is not required, the students will still be learning the basics, so it should only be the basics that are analysed.

Work has already been done in this area and fourteen interface metrics have been identified [NSA00, Bal06]. Some metrics are beyond the scope of what is required for marking, and there are also several other aesthetic tests in existence that are important for a basic programming course. A separate collection of metrics has been implemented as the SHERLOCK system [MS97]. However, as will be shown with the other collections of metrics they are also missing certain important variables, namely colour. Other work has also been completed on layout appropriateness which has implications not just in GUIs but could be employed in other situations [Sea93]. The most significant of the tests are covered below.

Meeting the Specification

When answering a coursework question, it is essential, as with any real world solution, to meet the specification provided. Unlike when working in industry, the students can have their solution checked and are often given a second chance. Therefore, as with standard feature testing, an automated marking tool would be indispensable to assist the learning of the aesthetic conventions. In this scenario, a feature-esque tool would check components' properties within the interfaces such as their colour, size

and images they are displaying. This tool could then be extended to analyse multiple objects to ensure that objects appear in the correct order. For example, if the students were modelling the sequence of a traffic light, it would be unsatisfactory for the lights to be in the wrong order.

There are several other basic layout measures that could be implemented [Joh00], these include assessing the alignment of grouped objects e.g. form text fields, or text labels. Another important test would be to examine the style of the primary window, the way in which the main system window appears should be different from that of a dialog box. Dialog boxes have no menu and all the controlling buttons appear at the bottom of the screen, this should not be the case with the main window.

Once the system has been checked against the basic layout measures, it should be tested against the more complex design standards. These are tried and tested standards that should be adhered to when creating an interface for any use.

Balance

The balance is the optical equivalent of calculating the centre of mass of an object. For a GUI to be perfectly balanced, the centre of optical mass should be in the centre of the interface. The centre of mass is calculated using certain properties of all objects within the interface. Every component in an interface has an effectual weight, this weight is calculated using the object's size and its moment (See Figure [5.3]). The moment it applies is an off shoot of the location of the object, the further away from the centre a component happens to be, the greater the effect it has on the centre of balance.

The centre of balance can be calculated using the following formula. Whilst the formula returns a result split into horizontal and vertical sections, the result is not an actual pixel location due to the inclusion of area in the equation. It is merely an indication of which quadrant of the screen is 'overweight'. It assumes that the centre of the interface has the co-ordinates (0,0).

$$balance = (W_L - W_R, W_T - W_B)$$

where

$$W = \sum_i a_i d_i$$

a - area of the object

d - distance from axis

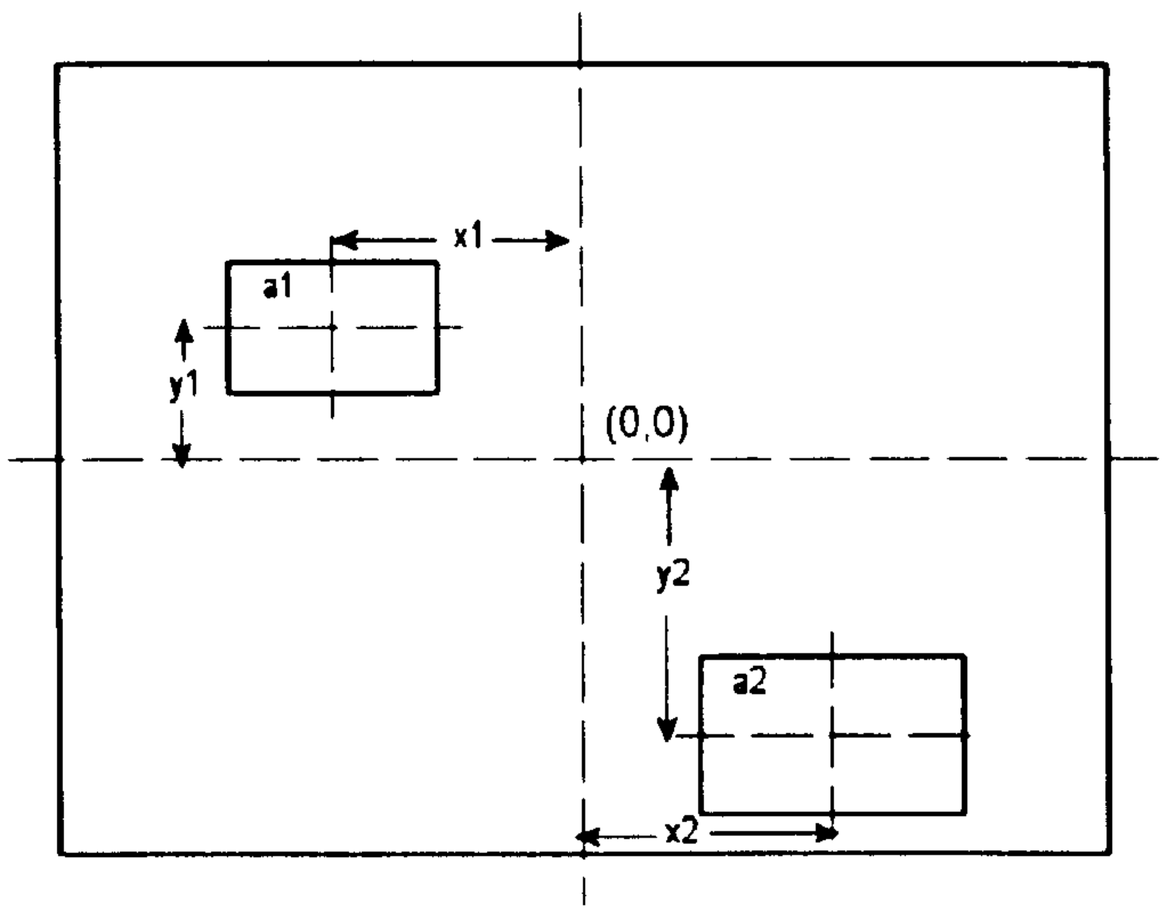

Figure 5.3: Calculating the Balance

The metric can be easily demonstrated with the aide of two simplistic examples. In the first example, there are two objects placed diametrically opposite from each other. The Figure [5.4] below shows a representation of the screen, it also attempts to highlight which areas of the screen are denoted by Left, Right etc. Both rectangles are the same size, 10x5 pixels, and the co-ordinates of the central points of the rectangles are shown.

Using this example we can calculate the centre of balance as follows:

$$W_L = \sum a_L d_L = 50 * 40 = 2000$$

$$W_R = \sum a_R d_R = 50 * 40 = 2000$$

$$W_T = \sum a_T d_T = 50 * 20 = 1000$$

$$W_B = \sum a_B d_B = 50 * 20 = 1000$$

therefore

$$balance = (W_L - W_R, W_T - W_B) = (2000 - 2000, 1000 - 1000) = (0, 0)$$

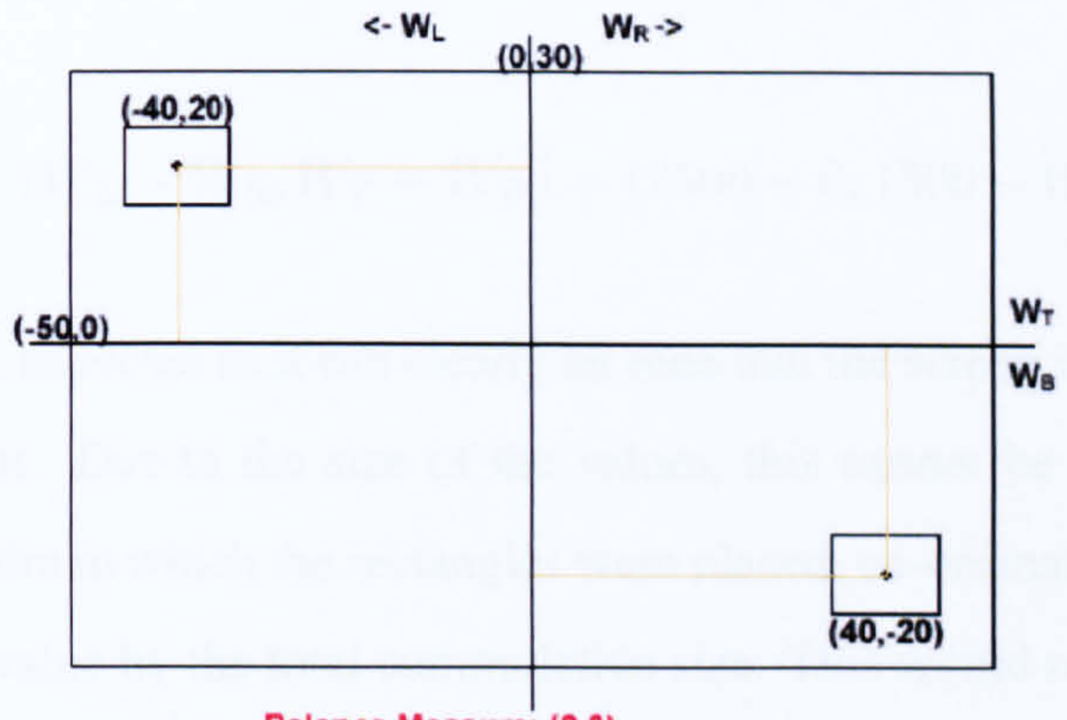

Figure 5.4: Balance Example 1: Diametrically Opposite

This proves that the screen being analysed is in perfect balance. Naturally this is not always the case as shown in the next example (See Figure [5.5]). As with the previous example, both rectangles are the same size (10x5 pixels) and the co-ordinates of their centre is shown. By performing the balance calculations it can be seen that the screen is not in balance.

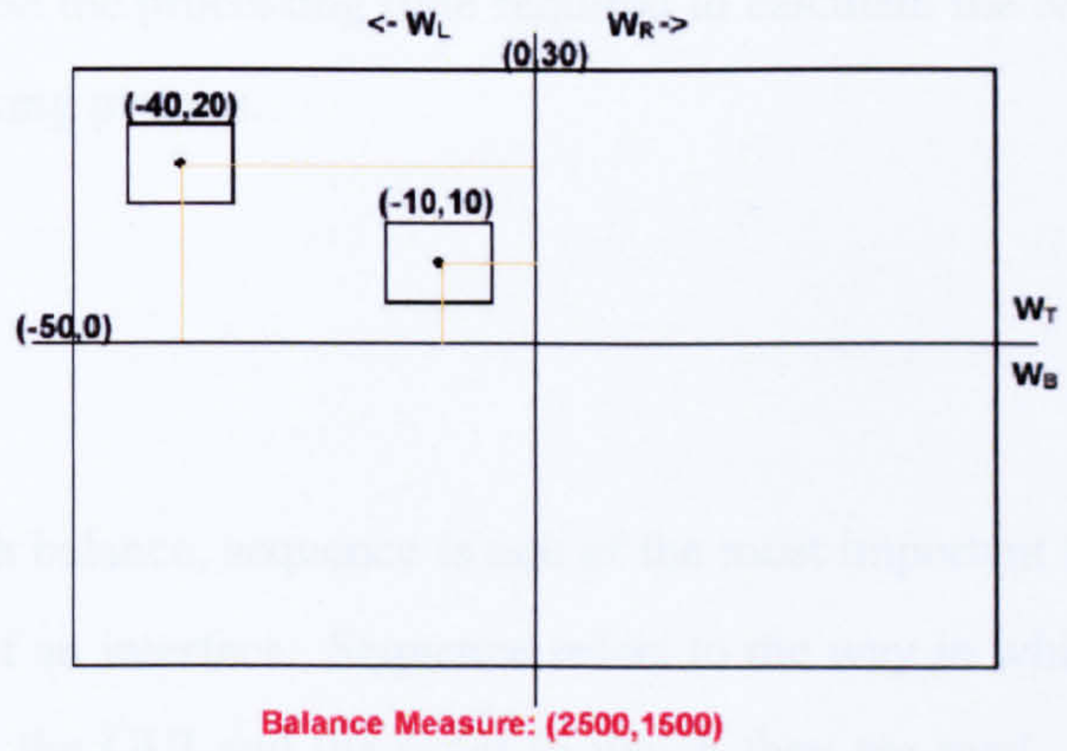

Figure 5.5: Balance Example 2: Unbalanced

$$W_L = \sum ia_i d_i = 50 * 40 + 50 * 10 = 2500$$

$$W_R = \sum a_R d_R = 0 = 0$$

$$W_T = \sum ia_i d_i = 50 * 20 + 50 * 10 = 1500$$

$$W_B = \sum a_B d_B = 0 = 0$$

therefore

$$balance = (W_L - W_R, W_T - W_B) = (2500 - 0, 1500 - 0) = (2500, 1500)$$

This result was expected as it can clearly be seen that the screen is not balanced around the centre point. Due to the size of the values, this cannot be an actual co-ordinate value in the realm in which the rectangles were placed, co-ordinates can be obtained by dividing each value by the total cummulative size. This would return the co-ordinates for the centre of balance as (25, 15).

There is also another method which can be used to calculate the visual balance. It involves using the crystallographic visual balance measure [LFN04]. This process involves creating what are known as weightmaps. Weightmaps are graphical representations of the interface within which, the heavier objects are represented as brighter colours. This representation is then used to create a weightmap pyramid. The processing required to create the pyramid shrinks each level in turn, calculating pixel colour averages as it progresses. This eventually enables a value for the balance of the image to be calculated. Although this is also a proven technique, it is significantly more complicated and the processing time required to calculate the balance would severely delay the marking process.

Sequence

Along with balance, sequence is one of the most important factors that will affect the usability of an interface. Sequence refers to the way in which the eye recognises objects within the GUI and the order in which they are read. Although it could be different elsewhere in the world, it shall be assumed that the users are most commonly used to reading from left to right. This changes the area of the screen that users are likely to notice first, see Figure [5.6].

The sequence can be described as a path from the most attractive to the least attractive object within a GUI. Naturally an interface whose sequence is out of order would be difficult to use. However, there is another worse situation. If none of the components located within the interface are more attractive than any other, no flow can be detected to create a path between them. This leaves the interface in a state of chaos where it is impossible to predict what the user is likely to notice next. This is

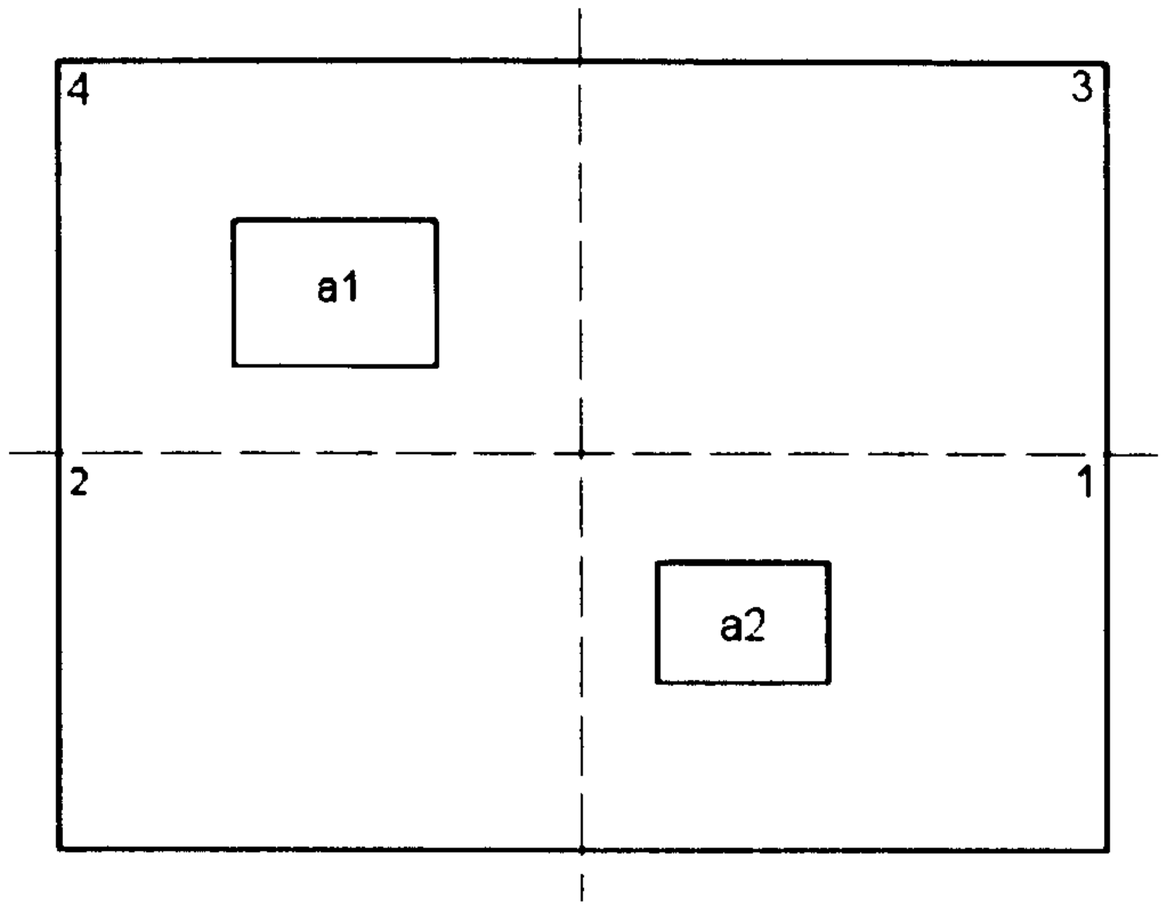

Figure 5.6: Calculation the Sequence

something that should never occur if a GUI is to be usable. A real world example of the importance of sequence is photography, it is common for photographers to flip images before developing the still. They do this to ensure the object of interest is on the left hand side, thus adhering to the sequence metric outlined below.

The sequence is a path that should start in the top left of the interface continuing until it reaches the bottom right without at any time returning on itself. It should incorporate all components otherwise the path is not complete. The attractiveness of each component used to decide upon the path is based around the formula:

$$attractiveness = area \times location$$

The location value is the value of the quadrant in which the component is placed. There is a higher value for the top left of the screen compared to the bottom right, see Figure [5.6].

Proportion

In the realm of Graphical User Interfaces, proportion could be defined as the comparative relationship between dimensions of screen components and proportional shapes [NTB03]. It is a case of ratios. Throughout the history of man, civilizations have always referred to proportional relationships. A good example of this is the an-

cient Greeks, they were especially interested in the Golden Ratio, an irrational number approximately 1.618, because of its recurrence in geometry. They attributed its discovery to the Pythagoreans, a group of mathematicians working under the tuition of Pythagoras. According to western culture, shapes proportioned using this ratio have been considered aesthetically pleasing. The shape created by this ratio is known as the Golden Rectangle. In fact, the Parthenon has been shown to exhibit such properties.

It is not just the Golden Rectangle that is considered aesthetically pleasing there are also several others. The Square, ratio (1:1), the Double Square (1:2), the Square Root of Two (1:1.414) and the Square Root of Three (1:1.732). Being simply a comparison between the dimensional proportions of the objects and the layout, this can be measured using the formula outlined by Ngo, Teo and Byrne.

Density

Density is the final existing metric that has been noted as worthwhile measuring. As the name suggests, it is a measure of how cluttered the screen is with objects. An interface littered with buttons, whilst it may seem as though the user has access to everything they could possibly require, will only hamper their efforts to use the system. Density could be measured by comparing the total area covered by the objects compared to the total area of the screen. This measure has the potential to be parameterized using an optimum screen density level to allow question designers to adjust their specifications accordingly.

Summary

In addition to the metrics mentioned above, there are several additional metrics that could be used to test the design and layout of a Graphical User Interface. However, most of them are superfluous for an introductory programming course. The most significant useful metrics are balance, sequence, proportion and density. Whilst only basic tests, it could be counted as a success if the students had learnt to create interfaces which met these criteria.

The balance and sequence measures, whilst being correct in their own right have one significant failing, they do not take colour into consideration. Colour can be used

to completely change the way in which an interface is interpreted by the senses. Therefore, new colour tests have been theorised and the metrics described above have been adapted to incorporate the effect colour has on the interface.

5.5.2 Colour Theory

Colour has a power of extraordinary magnitude. It has the ability to evoke feeling and trigger responses though its use in branding e.g. the colouring of a political party. It could be argued that the first work done on colour, was that performed by Sir Isaac Newton, who used the existence of colour as part of his mathematical analysis of light rays. In the 18th century, Goethe in his "Theory of Colours" [vG10] decided that Newton was wrong and rewrote the reasoning behind colours, their existence and effect. His ideas were slated by the scientists of the day despite Goethe believing it to be his most significant work. One of his concerns was that of colour harmony, which he described by splitting colours into two groups. The plus side, red through orange to yellow, which produce excitement and happiness in the viewer and the minus side, green through violet to blue, which are associated with weakness and other unsettled feelings. Goethe's work was not completely ignored, it intrigued several notable philosophers and scientists including Schopenhauer and Wittgenstein who went on to write "Remarks on Colour".

Whilst work on colour continued, it was not until artists and academics such as Josef Albers, Johannes Itten, Faber Birren, Albert Munsell and other artists from the German Bauhaus art movement, that major advances in colour knowledge and influence were demonstrated. The main focus of their work that is of relevance to this research is their work with colour systems, colour concord and the effect that colours have on each other.

In this section, the relevant work on colour concord will be discussed along with the way in which colours are described. This information will then be used to adapt the aesthetic measures already mentioned in this chapter.

Colour Spaces

Colour spaces are abstract mathematical models that can be used to describe the

way in which colours can be represented as three (or more in the case of CMYK) values. These values or colour components are what make up each colour. There are several different models that can be used, each of which has its own idiosyncrasies.

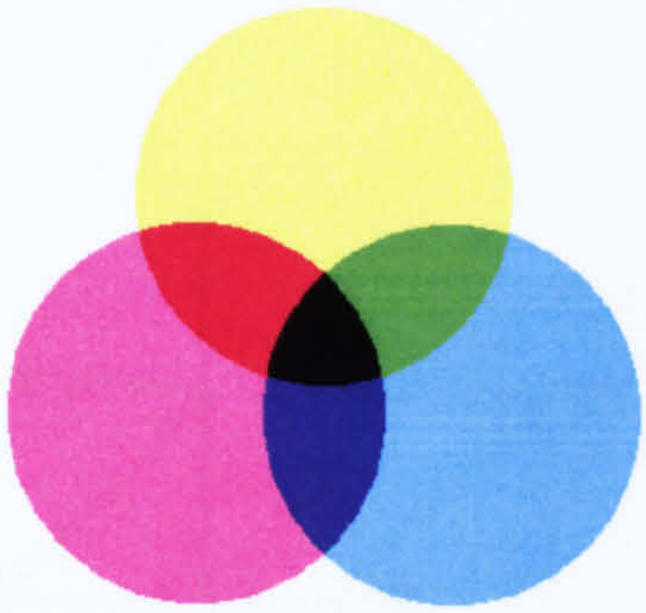

Figure 5.7: Subtractive Colour

As you will be aware, the primary colours are Red, Yellow and Blue. However, this colour combination does not make for a good colour space. It is a subtractive colour system, it works in the same way as if you were painting. As more colour is added to the mix, the closer it gets to black, the way this operates means that not all shades of magenta and cyan are reachable. This is shown using the more common CMY combination, see Figure [5.7]. By using Red, Green and Blue an additive colour space is created, see Figure [5.8]. This allows all colours to be created and works in the same fashion as the human eye and also cathode ray tubes.

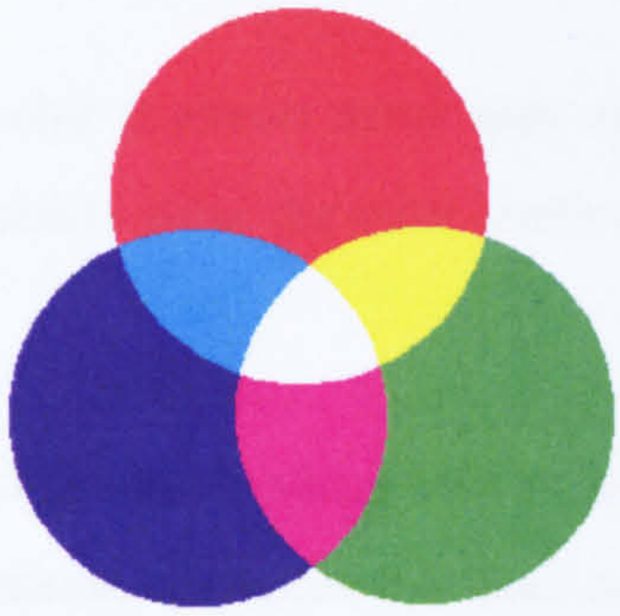

Figure 5.8: Additive Colour

Another model that is commonly used is HLS space, Hue, Saturation and Luminance (or Lightness). This is a model regularly used by artists due to the more intuitive values of saturation and lightness. It is essentially an RGB space (see Figure [5.9]) that has been rotated so the greyscales form a vertical line, with the hue as an angle (0-360) converting the space into a hexagonal shape. Although there are many more colour

spaces that have been created over the years for different reasons, it is not necessary to look any further than these two for the purpose for which they are required. Colours can easily be converted between RGB and HLS values through the use of a basic formula. However, this is also excessive, all that is required are the relative positions of the colours in the model.

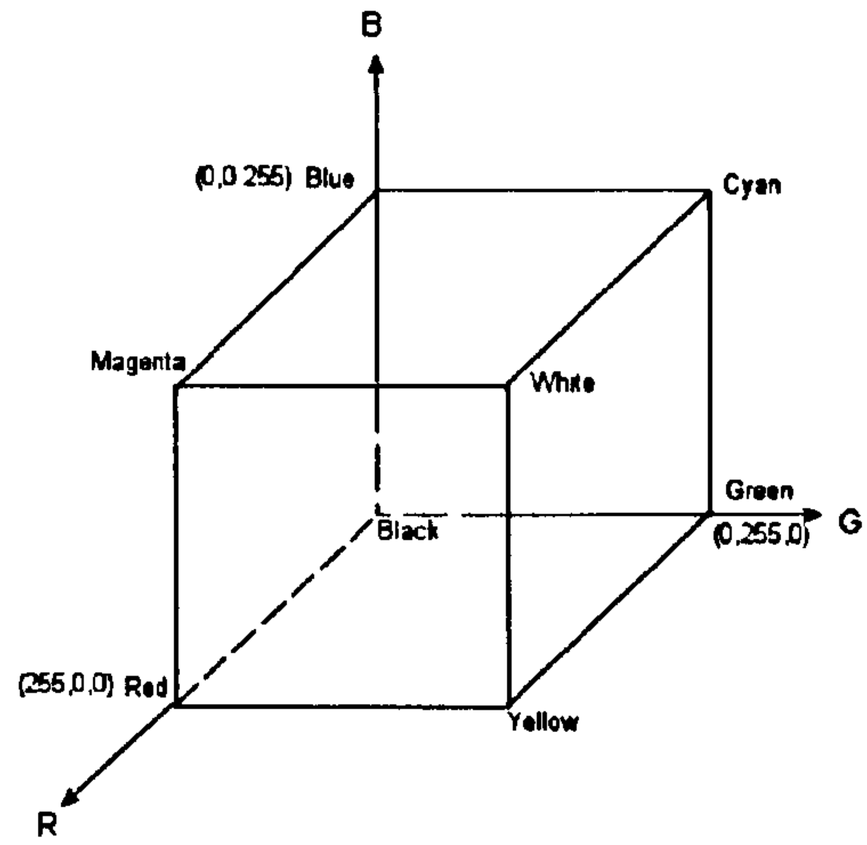

Figure 5.9: RGB Space

Colour Harmony and Distractions

> "In visual perception a color is almost never seen as it really is – as it physically is. This fact makes color the most relative medium in art." - Josef Albers [Alb75]

What Albers was describing is the way in which colours interact with each other. It is not possible to have a combination of colours that does not affect the way in which they appear to the naked eye. A blue on a red background will look different to a blue on a green background even though the pigments used are identical, this is known as colour interaction.

Figure [5.10] shows a colour wheel, variations exist aplenty, from Newton's which had seven colours (the colours of the spectrum), to Goethe's which only had six, and Schopenhauer's whose wheel had segments of different sizes. The creation of the different sizes depended on the ratio in which Schopenhauer thought they should appear

as to increase the harmonious effect that one colour would have on another. Whilst the wheels developed were drawn with only a few colours, it is possible to use an order of magnitude more. Therefore, the wheel is no longer of use. A new structure needs to be used, it is time to reinvent the wheel. If instead of the wheel you consider the RGB colour space you will notice that the same colours are diametrically opposite from each other, red and cyan, magenta and green, blue and yellow and finally black and white. These are known as colour opposites and provide the eye with the greatest possible distraction.

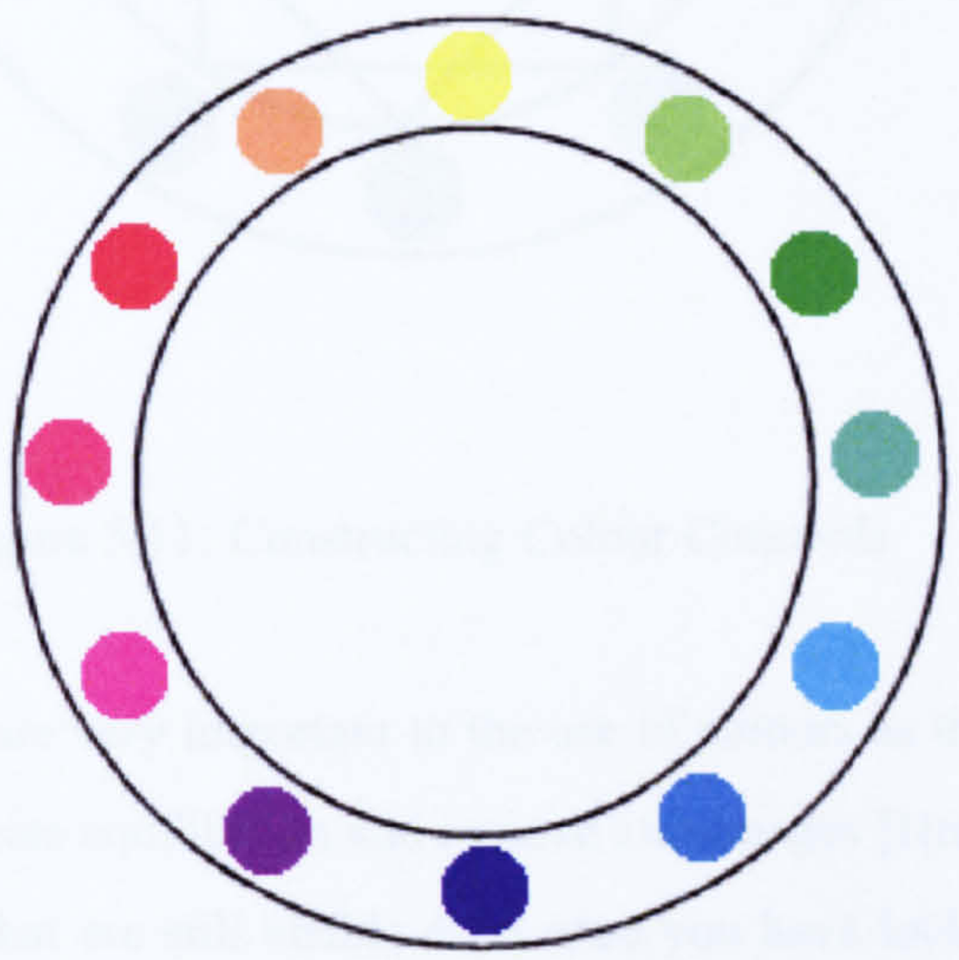

Figure 5.10: The Colour Wheel

Colour harmony is concerned with the balance of forces created by the colours involved [Itt03]. For each colour there is another that compliments it. When teaching colour use, there is a simple experiment that can be performed to illustrate this. If a grey square is inserted into a block of pure colour, the grey square will become tinted with the background's complimentary colour. When only two colours are involved, the complimentary is that which is opposite in the colour system. However, when more colours are used, concords can be constructed using methods as shown in Figure [5.11]. Obviously, this is illustrated using only two dimensions, but could be converted for use in a 3D colour space if required.

The wheel shows a two dimensional approach to creating concords. There is another thing to consider when choosing a colour scheme and that is the importance of greys, running through the centre of the RGB colour space is a range of greys from

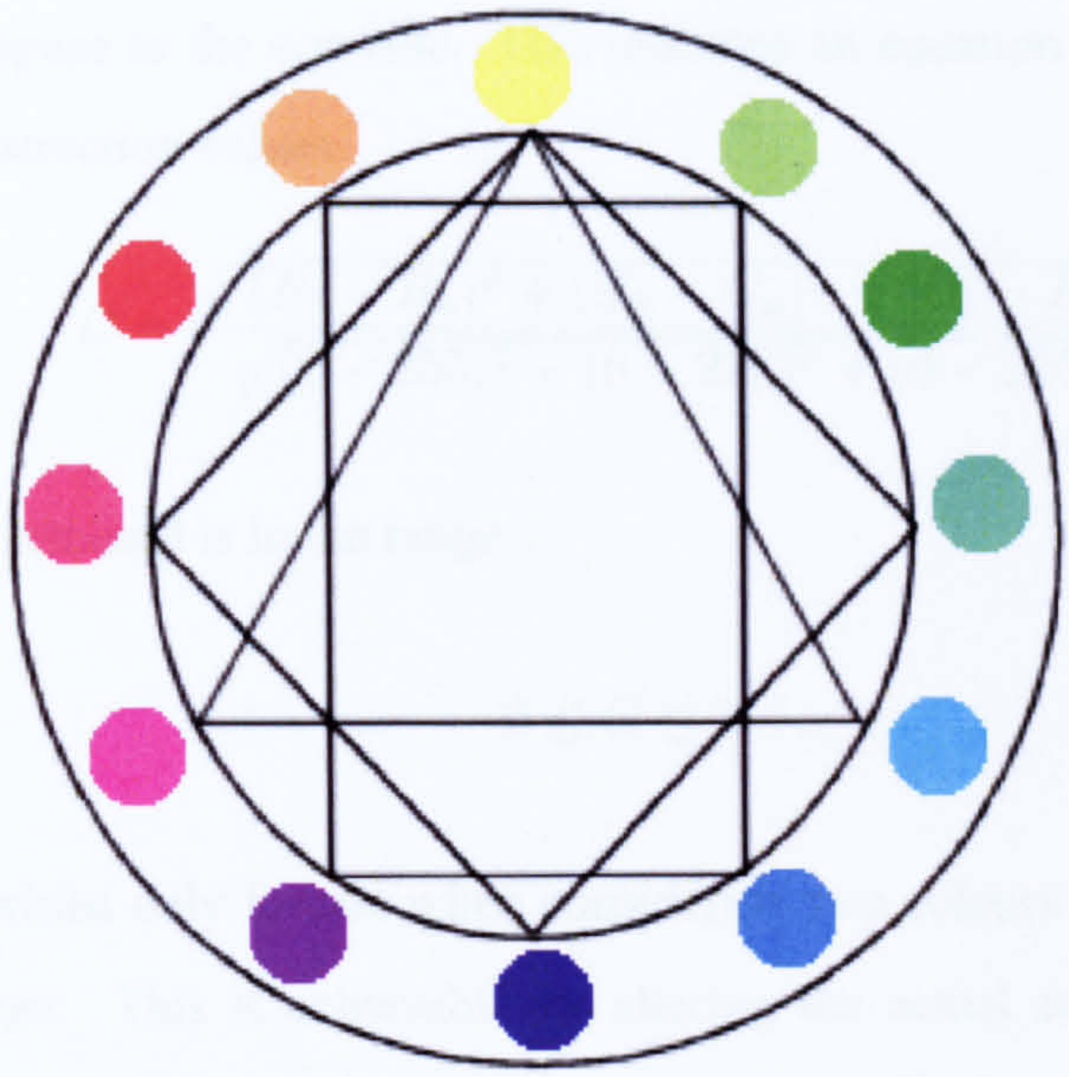

Figure 5.11: Constructing Colour Concords

white to black. Greys are very important to the use of colours as they are needed by the eye and brain to create equilibrium and remove afterimages [Her78]. Afterimages are patches of colour that are still visible even once you have looked away from an object, they commonly occur when looking at bright materials. Greys have the ability to bring several distinct colours closer to harmony due to its calming nature.

Colours should be selected for an interface carefully to ensure that a degree of harmony is achieved. Whilst assessing the harmonious nature of an interface is a valuable measure, it is a metric in its own right and does not aide the adaptation of the aforementioned metrics. A measure of difference between colours is required. While two colours, if they are opposites, are described as in harmony, they are also colour distractions. It is this concept of distraction that allows for the analysis of the colours.

By looking at an RGB space as a co-ordinate system, it is possible to use the individual colour components to generate the Euclidean distance values between colours. As already mentioned, opposites, such as black and white are one of the most distinct pairs and located in opposite corners of the colour space. The distance between colours is at its greatest when opposites are concerned. The greater the distance between colours in a colour space, the greater the difference between the colours and therefore the distraction. It is possible to determine relative distraction values by di-

viding the Euclidean distance between the two colours by the distance from one corner of the colour space to the opposite. This provides an equation capable of producing comparable distraction values.

$$C = \frac{\sqrt{(R_b - R_a)^2 + (G_b - G_a)^2 + (B_b - B_a)^2}}{\sqrt{(0 - 255)^2 + (0 - 255)^2 + (0 - 255)^2}}$$

The value returned is in the range

$$0 \leq C \leq 1$$

The formula whilst only for use when considering two colours could be extended for multiple colours. This is achievable by altering the actual and maximum distance between colours within the colour space used, e.g. for 3 colours maximum, we would use an equilateral triangle as shown in the colour wheel. See Figure[5.11].

The creation of a metric to determine the relative distractive effect of two juxtaposed colours whilst important to the design of an interface, does not yield its full potential unless it can be incorporated into other metrics. With the colour metric producing a relativity value, it can be used to adjust the effect that objects have on other interface values. Balance and sequence are both measures of how objects within an interface are visually regarded. Ignoring colour brings the effectiveness of these metrics into disrepute. With colour being the powerful medium it is, there is no doubt that it should be taken into account when calculating the balance and sequence.

Balance

It is now known that colour has a pronounced effect when it comes to object recognition, for example on a red background a green button will be more visible than a maroon button of the same size. So while our previous formula did not take colour into account, it can now be extended given the new formula for colour.

The following is still true:

$$balance = (W_L - W_R, W_T - W_B)$$

only the way in which W is calculated has now changed

$$W = \sum_i C_i a_i d_i$$

C - colour value

a - area of the object

d - distance from axis

This can again be shown with the assistance of an example. In Figure [5.12], as previously the two rectangles are the same size (10x5 pixels) only this time they are coloured differently. It shall be assumed that the screen background is black for simplification purposes. As can be seen, were colour not a factor it would be known

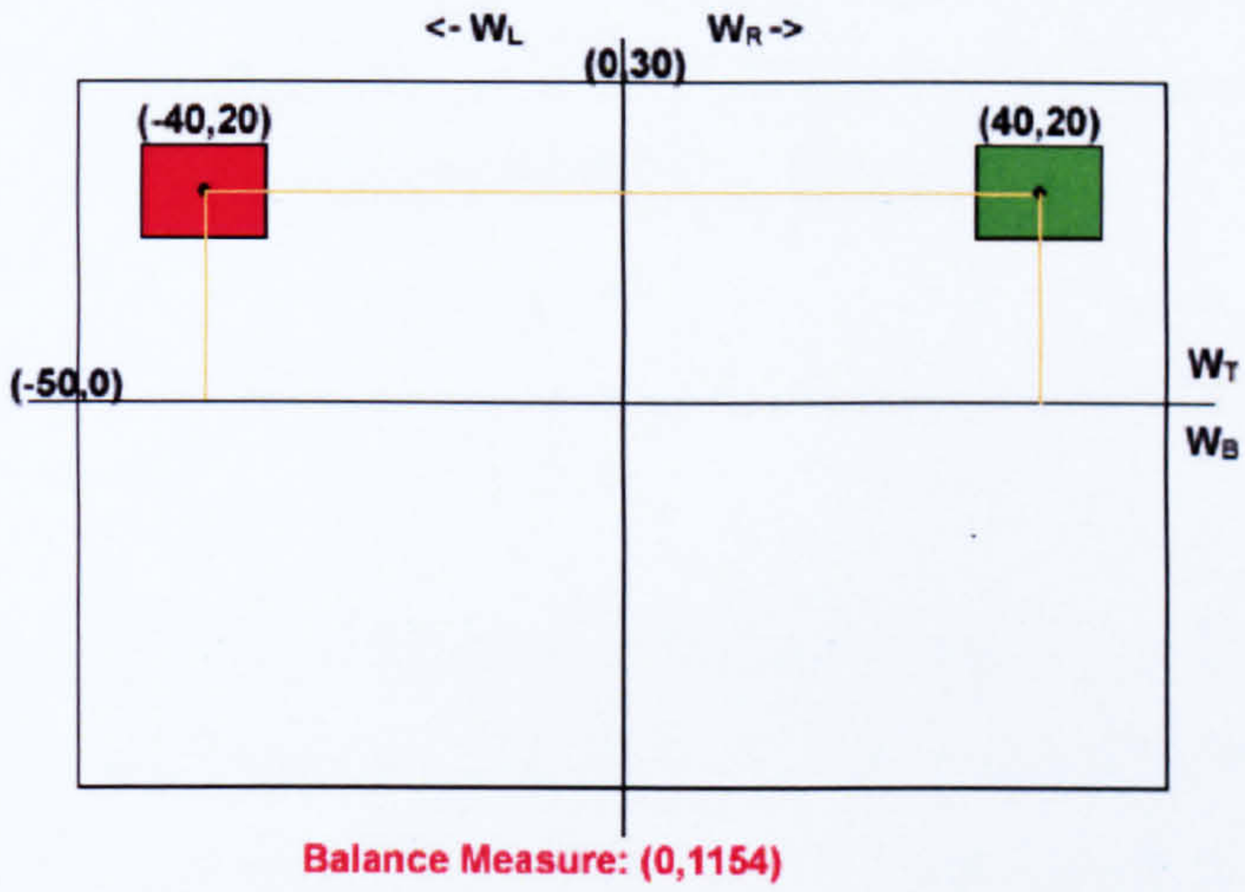

Figure 5.12: Balance Example 3: Colour

that the screen would balance along the horizontal plane. However, the colours of the rectangles are comparable and so the screen does still balance horizontally.

Calculating C:

$$red = \frac{\sqrt{(255-0)^2 + (0-0)^2 + (0-0)^2}}{\sqrt{(0-255)^2 + (0-255)^2 + (0-255)^2}} = 0.577$$

$$green = \frac{\sqrt{(0-0)^2 + (255-0)^2 + (0-0)^2}}{\sqrt{(0-255)^2 + (0-255)^2 + (0-255)^2}} = 0.577$$

$$W_L = \sum a_L d_L = 50 * 0.577 * 40 = 1154$$

$$W_R = \sum a_R d_R = 50 * 0.577 * 40 = 1154$$

$$W_T = \sum a_T d_T = 50 * 0.577 * 20 + 50 * 0.577 * 20 = 1154$$

$$W_B = \sum a_B d_B = 0 = 0$$

therefore

$$balance = (W_L - W_R, W_T - W_B) = (1154 - 1154, 1154 - 0) = (0, 1154)$$

This example shows how colour alters the calculated weight values, whilst maintaining a consistency between colours of the same relative value such as pure red and pure green.

Whilst it could be argued that the colours of the surrounding objects should be taken into account, the interaction colours have with each other is much stronger when they are juxtaposed. Therefore, it is more significant to only consider the colour of the object in question and those upon which it has been placed.

Sequence

The other aesthetic metric that can be easily extended to take colour into account is that of sequence. It is not merely the area and location that cause the eye to notice an object but colour distraction plays a role also. Again, for the extension to occur, none of the original variables need to be altered. The value for the colour comparison can be added to the existing attractiveness equation.

$$attractiveness = C \times area \times location$$

Summary

When considering the aesthetics of anything be it a painting or a Graphical User Interface, colour is never far from the thoughts of those regarding it. Whether it is a conscious appreciation of the effect they create or something deeper and more ingrained, it is a force that cannot be ignored. Therefore, for any system to fully test the aesthetics of an interface, colours must be involved.

This section has discussed how colour can be incorporated into already existing metrics. It was also mentioned how it could be discerned which colours work best in harmony, although no method was described to convert this for use in a colour space or into a test metric.

5.5.3 Summary

Being able to measure the aesthetic properties of an interface is a valuable tool. Although an interface may appear to be suitable to the programmer's eye, they are rarely experts in this matter and such a tool would allow programmers to correct any mistakes made before deployment. In an educational environment, the feedback such tools could provide would not only assist students in getting better marks in their work, but would hopefully teach them in an interactive way why certain design choices are wrong.

Whilst the metrics were implemented and tested, it was done completely independently of the dynamic tests and at a later date.

5.6 Dynamic Testing

So far this chapter, several testing ideals have been discussed. Although each test is important in its own right, they are all immaterial if the program itself does not work. This not only includes a complete functional failure, but also the incorrect execution of tasks. Of all the testing mechanisms, the dynamic test has the most importance, both educationally and generally.

When considering dynamic testing, there are certain aspects that need to be addressed. Such as what the tests hope to achieve and how they must operate in order to reach that goal. It can not be assumed that the testing will execute as expected or that the solution being assessed is an innocent piece of code, security and reliability should be utmost concerns. These shall both be covered in this section along with what tests are required and what is needed for them to function.

5.6.1 Test Types

The answer to the question "What kind of tests are required?", is a simple one. In order to test an interface effectively, the system needs to be able to simulate a human using the system. This infers that the system must be able to recreate any task the user would have access to at any point in time. This also includes actions that should not be performed but could be, albeit mistakenly or maliciously. Suffice to say, any action it is possible to program into the system should be testable, whether it is expected or not. Following this statement it is possible to limit the tests to anything the programmer could use to edit the system's state. Essentially, the tests should consist of all methods that can be called on an object in the interface which would cause a change in state.

Ideally, during the dynamic testing of an interface, all possibilities would be tested providing the system with complete coverage. To be able to guarantee complete coverage, the testing system needs to offer the exercise designer the choice of any action a real user could perform. Realistically is it unlikely that all exercise designers will create tests that provide the system with complete coverage, however, the opportunity is there and the system needs to be able to respond to such a request.

The arguments for incorporating all possible testing actions into the system are compelling and need to be addressed. Before the tests can be written, there needs to be some more thought on the actual execution of the tests and other related issues.

5.6.2 Performing Tests

It has been acknowledged that the tests should comprise of any method that can cause a change in the state of the interface. It has also been stated that the most effective way of executing operations on a GUI would be directly onto the object concerned. Therefore, to be able to perform the tests, a method of locating and extracting the correct object along with the test description is required.

5.6.3 Result States

It is not just the execution of the tests that is relevant. After the tests have been performed, the results need to be collected but nothing has as yet been suggested.

Without being able to analyse the results, no conclusion can be reached regarding the success of the system.

When any system is created, be it as a coursework assignment or not, the desired functionality of the system is always known. It is this knowledge that allows for testing to be performed. Testing is merely a case of comparing the expected state of objects within the interface to their actual state. This state comparison involves checking whether all of the relevant components of the GUI have the correct properties. Using the example of a calculator (See Figure [5.13]) the state comparison would be minimal, only involving testing whether the text set on the results bar is correct given the calculation that was just computed. Fortunately, the ideas already expressed in this chapter allow this to occur with relative ease.

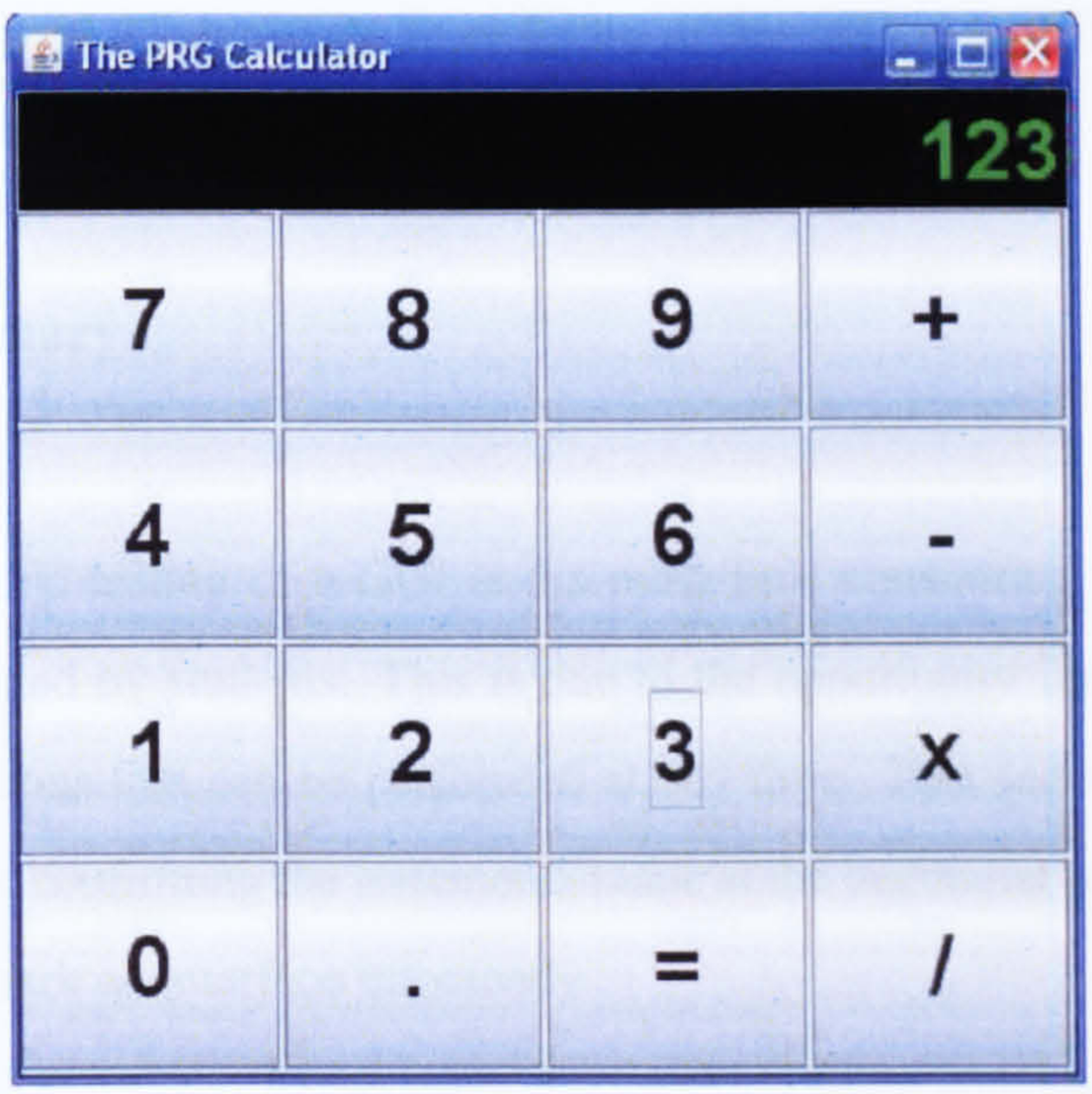

Figure 5.13: A Calculator ready for State Comparison

If during the testing phase, the operations are performed directly upon the objects in the interface, then it stands to reason that similar methods can be used to extract results. After locating the object to be analysed, instead of performing an action upon it, methods that extract information can be used. The properties and state of each object can be determined and checked against the expected values. Not only does this solution provide the system with all the information required, it also allows for the reuse of the object location and data extraction functionality.

5.6.4 Security Considerations

By loading the program as a separate process within a sandbox, the program can be marked with a degree of confidence that nothing untoward will occur to the marking system. Whilst this works satisfactorily for command line driven programs, it does not offer a GUI marking system the capabilities such a system would require. To obtain access to the objects within the interface, a new methodology for loading and testing the solutions is required. This in turn demands new safeguards and security policies.

The main aim of any security policy must be to protect the system as a whole. It is better to have an excessively restrictive policy rather than to inadvertently allow students access to sensitive or advantageous material contained within the system. After security, all other considerations are secondary. Although it would be preferable, rather than dismiss a submission in violation of this policy, to still return comments to the student in case it was a genuine mistake.

5.6.5 Summary

The dynamic testing of a GUI is the most time consuming part of marking any solution provided by students. This is due to the opportunity for error and the larger number of actions that can be performed at any time. This section has briefly talked about the tests, confirming the statements made at the beginning of the chapter on what is needed to mark an interface effectively.

5.7 Test Oracles

Now all the different types of tests have been examined, all that remains is the way in which the tests are described to the marking system. A common way of doing this is through the use of test oracles. These oracles contain all the information the system requires, the mark to be awarded for the test, the regular expression to be searched for and the different feedback to be supplied depending on the result.

The oracles in their current form, such as those used by the CourseMarker system, are exactly what is needed for the available testing mechanisms. However, in the

past couple of chapters, it has been shown how marking a Graphical User Interface is significantly more complicated. There are more factors that need to be taken into account at each stage. A redesign of the oracle blueprint is needed for the GUI marking system to be effective.

5.7.1 GUI Oracle Representation

With the previous test oracles, all that was contained within them were the instructions on how to examine the program and return a mark. It has been suggested that the testing and marking sections of the GUI marker could share the same operational functionality. It would therefore be sensible to combine both the test data and the marking data in a single oracle due to their similarities.

A combined oracle would have to distinguish between what is a test and what is a marking command. There are often tasks that need to be performed repeatedly when using a GUI based system, this is another useful inclusion that should be allowed within the oracle. As well as the extra functionality needed for both mark schemes and tests to share the same files, certain common information needs to be expressed. The information needed to locate the objects is required, this includes the object identification handle and also the type of handle it is. Once the object has been found, the operation to be performed upon it needs to be known along with any test data required by the operation e.g. text to insert. Naturally the mark for each test and marking request are also important.

With all this information needed in a single file, the standard single line plain text oracle does not have the capability to be able to store the information. Not if it is required for the oracle to be readable by the administrator also. A new format is needed to contain the tests, one standard that meets all the demands placed upon it is XML. The tests can be complex yet easily readable and can be fully defined during the design of the system.

5.8 Summary

In this chapter the different types of test that a comprehensive marking system should contain have been discussed. However, the marking of GUI driven programs

are more complicated than command line programs and need extra information to perform the tests. The combination of the information needed and the variety of tests to perform requires a different approach not only to the way they are performed but also the way the tests are described.

Through a system that involves extracting the objects stored within the GUI, tasks could be performed easily and with less chance of error. It also offers the marking system the opportunity to extract information about all properties of the object thus determining its state, an ability that would be invaluable to any marking system.

Alongside the usual marking mechanisms, the inclusion of Graphical User Interfaces brings about new marking requirements. Namely those of the aesthetics and ergonomics of the interface itself. In this chapter, several pre-existing metrics were discussed amongst standard layout conventions. The existing metrics did not include any thought for colour differences in the system, which is an important concept when analysing interface design. The distractive effects that colours have upon each other have now been quantified. This allows for the metrics described to be extended to include colour as a contributing factor to their results.

Following the discussion on the tests and what the marking system needs to be able to achieve, the next chapter will focus on the design of the marking system and the concepts involved, which make the existence of such an interface marking system possible.

Chapter 6

Design and Framework Implementation

"You miss 100 percent of the shots you never take." - Wayne Gretzky

In the previous chapters, what constitutes a Graphical User Interface has been determined along with several conventions that should be adhered to during their creation. These conventions were then taken a stage further and converted into tests. It is the tests that now have to be used as a starting point. What the tests should determine is known, it is only the exact method of how this is to be achieved that requires clarification.

This chapter will explain how it is feasible, by combining standard programming techniques to create a novel approach to interface analysis, to create a system capable of taking an interface and running predefined tests upon it. This chapter shall cover the ideology behind marking GUIs, the methods that can be used to realise it and the problems that they pose. It shall finish with a comprehensive assessment of the interface tests and the oracles needed. It shall also include how these concepts were implemented and later integrated into CourseMarker for real time testing purposes.

6.1 Introspection

For any given assignment on a course of approximately 200 students, it is highly unlikely that any two submitted interfaces will be exactly the same, either due to layout differences or the way in which they operate. The layout changes are not only a problem with regards to aesthetic criticism but in order to perform any kind of dynamic testing, a degree of knowledge of the components' positions are needed. Without knowledge of these positions, damage could inadvertently be done to the underlying system. Even though Capture / Replay tools are the most reliant on static object locations, knowledge of an object's location or the ability to locate objects is always a requirement to perform marking tests. The chosen method should not only be simplistic in nature but also be able to provide a degree of security either for itself or to any underlying systems. All tasks should also be able to be performed at run time.

The approach that best meets all of the requirements of a Graphical User Interface marking system is introspection.

6.1.1 What is Introspection?

The word introspection has been used for a number of purposes, each with a slightly different meaning. At its core it has the following meaning: Introspection comes from the Latin *introspicere* "to look into, look at," from intro- "inward" and specere "to look at". A more relevant definition for the purpose to which it is going to be used is to look inwards for the purpose of analysis. This is not just a technique that would seem more at home in a therapist's office, but also one that can be utilised in the current situation. Object introspection, in the context of Object-Oriented programming languages, is the ability to observe and change the state of an object using reflection. It provides a window to the object states of the current execution of a program and allows changes to them by means of a general gateway to existing legitimate interfaces [CKW98].

With regards to the marking of programs, we can take it to mean a glance inward for an inspection of the constituent parts of a GUI. "For example, an introspective Object-Oriented language allows querying (of) an object for its attributes and methods, their types and values. Introspection is enabled by a reflection property, meaning that the program structure is reflected to the program itself" [DSG03]. It is the reflection of the program structure that will benefit the GUI marking system. However, a distinction needs to be made between reflection as a concept and the reflective properties just mentioned.

6.1.2 Reflection

Reflection is a less restrictive approach to the introspective concept, it allows systems to observe and modify properties of their own behaviour. In Java there is a package that allows systems to represent, or reflect, classes, objects and methods stored in the Java Virtual Machine (JVM). This package allows the programmer to leave making choices, such as what object is to be created or what method is to be called, until runtime. At first glance, it would seem that this package is ideal. It cannot be argued that the package does not perform several necessary functions. It is able to locate objects and run methods. It is also able to undertake the location of the objects at run time. However, there are a number of drawbacks with this method.

Whilst useful, reflection goes well beyond the call of duty. It can procure access to the classes and objects within any program, but it also offers the opportunity to manipulate the internal properties of the program. This could be advantageous depending on the system in which it is being used. With regards to marking systems, it creates a possible opportunity for either inconsistent marking or malicious alteration of a running program. This is not the only problem with reflection as a solution. It is possible to take a Java class and display the names of all members. Unfortunately for the approach it is required for, reflection reflects the program structure to itself. It creates a mirror image of its inner self. Naturally, this means that the code which is to perform all the analysis needs to be within the class being run. This is not feasible for a marking system as it would either involve manipulation of the students' solutions after submission, or providing the students with code they need to insert into their solution. Both approaches are flawed.

There are many uses for which reflection is perfectly suited. GUI marking systems is not one. When marking students' solutions, it will be necessary to in turn load each solution and run the tests upon it. The reflect package, is able to provide the user with a great deal of information on the fields contained within classes, it is actual instances of these classes that are of importance. What is required is a system that can access class member fields and perform tasks upon them in a rigid environment restricting what can be performed or what properties within the system can be altered.

The reflective properties which allow for introspection are normally used by programs such as debuggers and interpreters. It makes use of meta-information which describes the structure of the program. It is this meta-information that relates reflection and introspection and potentially provides a marking system with access to the information it requires. If this information can be extracted, it can be harnessed to locate objects within the interface.

6.1.3 Using Introspection

Having ruled out reflection as a marking solution, it is not necessary to ignore all the concepts based upon it. As it is an automated marking system that is being designed, it can be summarised that the assignment question has been written prior to the marking stage. Therefore, certain structural information is known about the pro-

gram being marked. This information can be used in collaboration with the program structure and meta-information to obtain results.

Through the use of the more restrictive concept, introspection. A system can be designed that will allow access to the GUI components yet restricting the writer of the mark scheme to predefined tasks. It is also possible to achieve this at runtime as the program is loaded for marking. Advances in Object Oriented programming languages have granted this privilege, namely dynamic class loading [LB98].

Dynamic class loading does several things to assist the introspective marking process. Not only will it load the interface ready for testing, but it will also provide access to the interface's object structure [GH06]. When a class is dynamically loaded it creates an object instance of the class that is run. This object is now a variable inside the system that created the instance, in this case the marking system. As it is a variable, it is possible to use it in any way the system is instructed. For example, were a calculator interface run it would provide the marking system with a calculator object. The creation of this object not only causes the interface to load, but it also contains all the constituent parts of the interface. The adoption of Object-Oriented programming means that once access to the main interface object e.g. a JFrame is achieved, it is possible to navigate down through the nested collection of children until every component has been found and analysed. This access as is provided through introspections enables an automated marking system to meet three of the marking system requirements outline in Chapter 1. It allows for a system to be able to interact with an interface in any way required, handle varying layouts and identify the components in a number of ways.

Dynamic loading avoids the problems faced by reflection. The code that accesses all the internal information is in the class that loads the students solution, rather than in the students' own programs. If all components within the Interface are available for use, then it is also possible to create an marking interface so that the marking scheme is not directly interacting with the students code. All marking must then go through specific, authorised marking tools. This will enable restrictions to be put in place, which will limit or even remove the availability of some functionality.

Having found solutions to the problems exhibited by reflection, we have in introspection found the ideal approach on which to base an automated Graphical User Interface marking system.

6.2 Dynamic Program Loading

It has just been said that dynamic class loading will provide access to the object structure of the interface. All that is required to load a class dynamically is information about the relative package location. Defining where the students' solutions are to be located, allows the system to determine where the files are. Therefore, when the source code is compiled, the package location is known. When loading classes dynamically there are certain issues that need to be addressed. How will the class be loaded? What security implications does this create?

6.2.1 Class Loaders

In Java, classes are loaded into the Java Virtual Machine via the use of a class loader. It is possible to load separate classes with different class loaders, this means that programmers are able to define their own class loaders. However, as class loaders are also classes themselves there is indeed a hierarchy of class loaders as well.

Primordial Class Loader

The primordial class loader is the base class loader, it is responsible for "boot-strapping" the class loading process. All Java classes responsible for the running of the JVM are loaded by the primordial loader whether the user knows it or not. The problem with using the primordial class loader for the students' solutions is that they will be loaded in an unrestricted fashion, for this reason a new class loader will be required.

Designing Class Loaders

Class loaders are written for several reasons, one of the most important is that it allows for the inclusion of security policies. If a class is loaded dynamically by the primordial class loader, it will automatically have the same permissions as the program that invoked it. With regards to a system such as CourseMarker which has the ability to write to and delete files, this would allow any program loaded significant access

to the file store. As a new class loader is being written, it is possible to customise the loader in such a fashion that restrictions are placed upon its operation. It is with this knowledge that a secure policy capable of stopping students performing malicious tasks was designed.

6.3 Security and Reliability

When any system is to be created, security should be of paramount importance. Without the enforcement of a security policy, any claims about the reliability of a system can be disregarded. Not to mention the damage that could be done or information gleamed from other systems on the machine or network. The need for good security is heightened when the system contains sensitive information, in this case students' marks, that only administrators should have access to.

When marking interfaces, it has just been said that for the marking system to gain access to the internal variables of the interface, the class needs to be dynamically loaded. This means that it is not possible to load the class as a standalone process and alternative security measures need to be created solely for the marking of GUIs.

There are several stages at which security restrictions can be imposed upon a student's solution. These are when the marking is initially called, during the compilation and during the class loading. The enforcement available at each stage shall now be considered.

6.3.1 Sandboxes

Sandboxes [Oak98] are used to trap unexpected behaviour. Most marking systems already employs a sandbox in some form or another. CourseMarker, for example, uses a sandbox to refuse students' solutions access to files on the file store. This can be turned on and off and the type of files students' are granted access to can also be controlled. When permission is granted to access certain files, a new level of security is invoked, this involves searching the source code for anything that may be malicious. It is looking for any method that could be used for a purpose the system administrators do not feel appropriate, this involves searching for commands that could remove files from the file store etc.

These security measures can be extended to include other interface related restrictions. The extent to which a student's program can be analysed at this stage of the process is limited. However, there are various inclusions that can be made to the security policy to stop problems being caused to the marking system. One such inclusion is that of modal windows. Modal windows are windows from which the focus of control cannot be removed until the window closes. This can be abused through the removal of all functionality of buttons and window close events forcing the system to keep attempting to close the window. When creating Graphical User Interfaces, there are often situations in which modal windows are appropriate, for example when software is guiding a user through a process. For the purposes of a first year University programming module, it may be considered that modal windows are excessive. Therefore, they can be added to the security controls at this stage of the process, if left until the program is running, they will only be recognised when loaded and the program is stuck in a state it cannot escape from.

6.3.2 Compilation

To load a class dynamically, the class loader needs to be provided with the package location for the files. However, when the students are creating their solutions, they do not know where their solutions are going to be stored or marked. The less information the students have about the file store in which their programs are going to be run, the better. Without setting the package line in the classes, it is not possible to load them dynamically, this is where the compilation security becomes involved.

When the question is written, the marking scheme contains a list of files that are to be collected from each student and marked. When a student requests marking to be performed, a compilation tool is called. This tool goes through all of the files listed and adds a package line to each one. If any rogue files found themselves in the marking directory for the student, when the program is compiled it would be unable to find them or use them in the same fashion as the package description would be incorrect.

Adding the same package line to all of the files expected will not alter the way in which the programs run. When the files are stored for archiving purposes, it will be the unaltered version that is stored enabling the administrators to demonstrate that a student's code does not work either in the altered or unaltered state.

6.3.3 Class Loading Security

It is during the class loading where the most of the security protection is exacted. If a particular class loader is used, it is not only used to load the class in question, but all classes that are called from within it. So if a new class loader is written, it can be used to govern which classes, if any, it is allowed to load throwing a security exception each time it attempts to break the policy.

The reasons why there is a need to restrict the students' solutions include the removal of any opportunity to cause malicious damage through inappropriate altering of files or running of operating system commands. This is very implementable by extending the existing SecureClassLoader. Each class loader has its own `loadClass` method, it is in here where the classes are located and calls are made to load it into the Java Virtual Machine. However, it is possible to check each class before it is loaded into the JVM, these checks can include looking for specific classes or whole packages.

Loading Restrictions

There are a few obvious checks that can be made on the classes that are requesting loading. The first is to check their package information. From this it is possible to discern whether a class is contained within the local marking directory, rather if it is part of the package that all official files were assigned to. Another important check is to see whether the package information starts with "`java`" or "`org`", if not then the program is attempting to load an external class and should be stopped.

The basic package check instantly enables the refusal of classes that the marking system has no control over. The more complex problem is restricting the students from accessing certain parts of the standard Java functionality. To enable the students to still be able to program freely, a certain amount of investigation was required to determine which classes would not be needed by students to create working solutions to the majority of problems. There are currently 59 items in the banned list created for testing purposes, this comprises of both whole packages and in some instances where the package is required by students, individual classes. It was simple to identify certain packages that should be added to the list, namely anything that would allow the students access to the underlying file store. Other banned items include such packages

as threads, security, RMI and class loaders. The list has been extracted to allow for easy modification by the system administrator. It has not been made public to reduce any chances of the students being able to hack the system.

Protection Domains

The most common method for controlling the permissions of classes loaded though self-written class loaders is with the use of protection domains [HCC+98, GS]. They provide the programmer with the ability to take different permissions e.g. socket permissions or runtime permissions and set the attributes which allow or deny functionality. Although this is a very powerful security ally if in the right hands, it does not quite meet the needs of the system. By extracting the security policy in the way stated above, it allows for modifications to be made with great ease. Were protection domains to be used, then each time a change was needed, the administrator would have to make changes to the permission set. This requires a greater level of knowledge than just knowing which classes or packages should be restricted.

6.3.4 Reliability

The reliability of a marking system can guaranteed by running each student's solution as a separate thread. Therefore, were something to go wrong during the running of the solution and the program were to stop, then it would be noticed by a thread running on the server and the process would be killed. It has just been shown that in order to test a GUI based program, it is essential to dynamically load the program rather than loading it as a separate thread. However, to continue to be able to ensure the reliability of the system, the way in which the interface tests occur has been altered from the current methodology.

To maintain the integrity whilst marking GUIs, the marking process still uses a separate process. Instead of just loading the students' solutions, it is a section of the testing and marking system that is loaded as the process. This process takes the details about the tests to be performed, loads the solution and executes the tests upon it. The results are returned by the marking tool via messages output and read by the marking server. It would be reasonable for you to think of this as a security lapse,

as students, were they to find out the format of the messages, would be able to print similar messages to the console thus giving themselves a different mark. This is not the case. In the previous sections, it was mentioned how the system denies the programs being marked access to certain classes and packages. Streams are included in this list due to their use in creating and altering files. This also means that access to any methods that would enable the student to output to the console is also denied, thus keeping the system safe from abuse.

6.3.5 Summary

This section has discussed the way in which the security measures need to be implemented to ensure the security when it comes to marking interfaces. It mentioned the three ways in which controls can be placed upon the solutions being marked.

For the purposes of this section is has been assumed that all security features related to user authentication are still in place and unchanged. Even though the marking system may have been rewritten, the way in which the students are to submit and have the work marked is untouched.

6.4 Object Recognition

Having the ability to access the structure and components of a GUI makes the task of analysing its properties and functionality much easier. It is possible to explore the collection of objects and retrieve the information directly, rather than from interpretations. The previous few sections of this chapter have explained how the marking system can create an instance of a program submitted by a student, it is from this object that the component information can be extracted.

As with any object created in an OO language, incorporated in the object are all its constituent variables. However in the case of the program solutions, all the variables are actually the components of the GUI. It is therefore a case of taking the object and systematically working through each component in turn.

6.4.1 Descent Parser

When considering how the component information should be accessed, it became clear that assistance can be gained from the inherently hierarchical architecture of GUIs. This architecture means it is possible to predict where certain components should be found in relation to others. Using this knowledge, it has previously been shown that all leaf elements of the structure are the components and all other nodes the containers. Therefore, with the aide of a simple descent parser it is possible to extract information about all or specific objects.

A parser is able to descend through the branches of the hierarchy and extract each object in turn. The parser implemented, descends through the object structure using a depth first approach, this allowed the parser to be written recursively. Again due to the hierarchical nature of the Java SWING package, certain methods have been inherited that can assist the parser. At each stage of the descent, checks can be made on the type of object being viewed. If it is a container of some variety, then the `getComponents` method inherited from the Container class can be used to retrieve all other components stored within the object in question.

Now that access to the interface components has been gained, the information can be stored for later use. However, there are certain measures that can be made at this point that will make future tasks such as writing the tests significantly easier, namely object abstraction.

6.4.2 Object Abstraction

By abstraction, what is meant is the extraction of the essential details of an object while ignoring the inessential details. In the case in which it is to be used here, what matters is the type of object it is rather than the exact class. As the parser works through the GUI's structure, it supplies each object found with a token relating to the type of object it is, this is for faster recognition when the tests are being performed. There are similarities between objects that can save a significant amount of time and unnecessary work.

If you consider the hierarchical structure of the Java SWING package, the use of inheritance is clear. It is this inheritance that allows for the simplification of the testing

strategy. The use of inheritance infers similarities in the way in which certain interface components operate, a good example of this is the AbstractButton class and its subclasses, the subclasses include JButton, JMenuItem and JToggleButton. During testing, if a button is to be used then there are a limited number of operations or analysis options that can be performed upon it. The button can be clicked, its text examined or it could be checked to see if the button is currently operational. The methods that perform each of the three functions mentioned are actually methods from the superclass. Therefore, the testing process can be refined so that actions on menu items or buttons are handled by the same operation without having to worry about what type the object actually is. If the AbstractButton methods have been superseded by methods in the relevant subclasses, this is of no concern as due to the polymorphic properties of Java, it will locate and run the overwriting method.

The other main instance of this occurs when text components are considered. JEditorPanes, JTextAreas and JTextFields can also be combined for basic methods. However, to be able to use the extracted information, more than just the component type is required, a method of identifying the requested object is necessary.

6.4.3 Object Handles

Tests will, on the whole, have to be object specific but access to the actual objects means that their internal variables are available for use. These variables provide an opportune way of locating particular objects for testing. In order to locate objects effectively we need to know certain parameters about what is being searched for.

While any question that is written for students should have a fairly loose specification to allow creativity in their solutions. There is naturally a need for certain restrictions even if it is only in the requirements of the program itself. For the purposes of object recognition added restraints need to be imposed. It has just been shown how the object type can be recognised. However, it is rarely the case that there will only be one instance of each type of component, especially not when we are grouping components using the aforementioned abstractions. Handles need to be actively set in the question. Handles are properties unique to each object that can be used to distinguish an object from others in a set.

Being able to identify components in a number of ways is an important require-

ment of any GUI marking system, this can be achieved by implementing multiple handles. The handles themselves can consist of anything that can be altered by the programmer, during the creation of the testing methods, nine different handle types were implemented. These were as follows:

- Name - Programmers are able to assign names to objects for later identification.
- Text - This would depend upon the object type but could be either a text label or text contained within a component.
- Icon - An image icon can be used to differentiate between similar objects.
- Action - The action command given to an object.
- Number - If the quantity of a certain type of object is known then the one required can be selected.
- Title - Title of a window etc.
- Focus - Whether an object has focus, i.e. is it currently visible on the window in the foreground.
- Type - Can be used for anything that sets types, e.g. the set of option buttons that appear at the bottom of a dialog box, this is normally an integer value.
- Colour - The colour of a component, can be either the foreground or background depending on the test designer.

Extra handle recognition can be added into the object identification with ease. More details will be given when the design of the testing system is discussed momentarily.

6.4.4 The Parser Package

The class diagram, Figure [6.1], illustrates the way in which the descent parser can be implemented. The ParseAction class as previously mentioned was written recursively so therefore contains all the methods required. As the parser locates an object, it is stored as a GUIComponent along with the token used for swift identification. The GUIComponent is an implementation of the composite design pattern, this was chosen because it allows for the representation of a hierarchy.

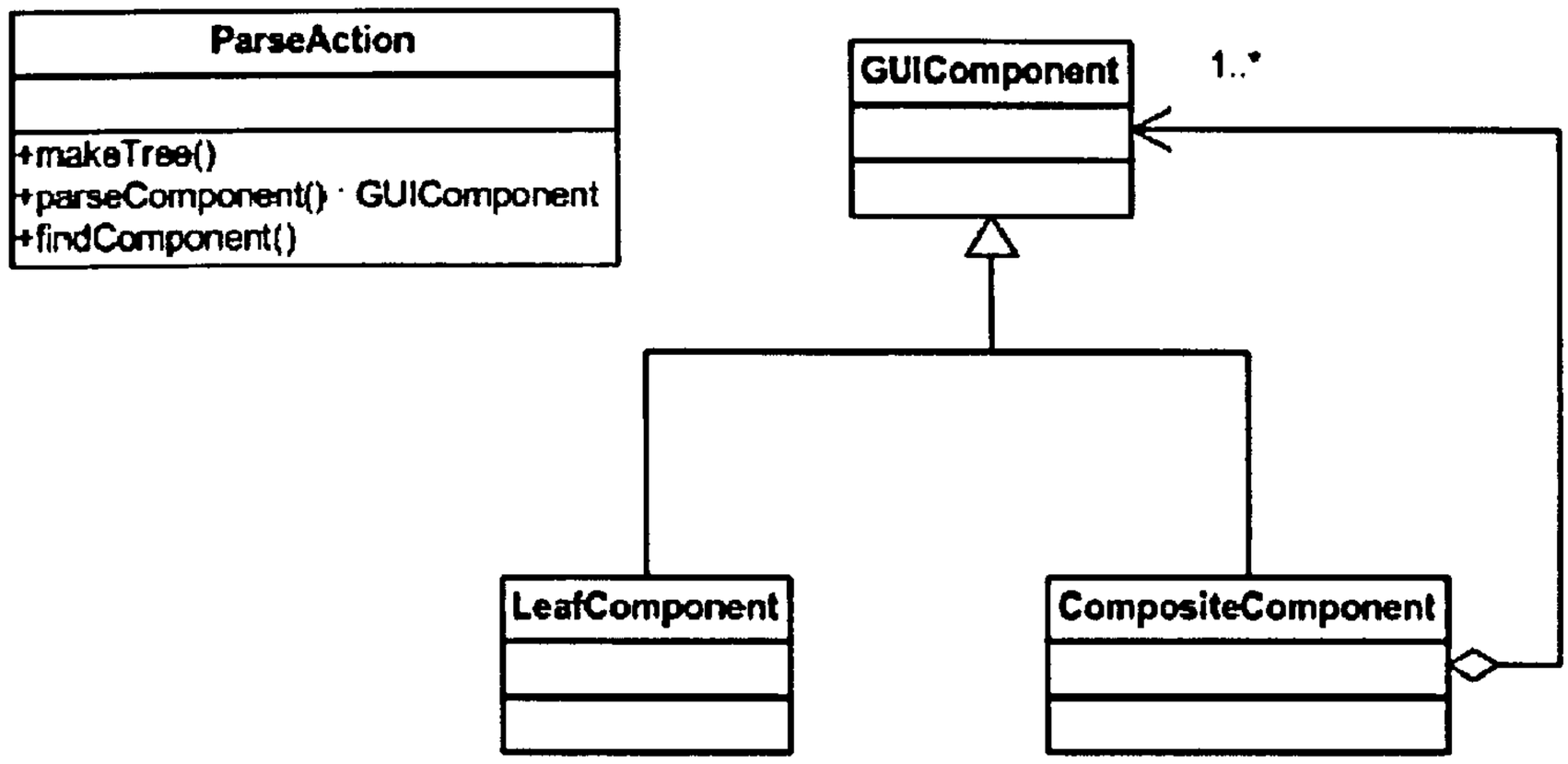

Figure 6.1: Descent Parser Class Diagram

6.4.5 Summary

In this section, the methodology behind the extraction of the internal variables of a Graphical User Interface has been explained. This was then followed by a description of how the hierarchical structure of GUI components can be used to reduce the number of different test classes that need to be written. In the next section the focus will be moved to the tests themselves and how they shall be designed and written.

6.5 Testing System

This chapter has so far provided the reader with a collection of concepts, which if implemented would create a framework, with all the capabilities necessary to test Java Graphical User Interfaces, all that is required is the creation of a suite for the interpretation and execution of tests.

The testing system comprises three main sections, the way in which the testing requirements are represented and interpreted, the tests and their execution and finally the state comparisons. Having already explained the concept behind the dynamic class loading and interface component retrieval, it will be assumed that this has already been implemented and that the results can be readily obtained.

The first stage of testing relates to the oracles. It has previously been said that the standard oracles do not provide the system with adequate opportunity to store information. Therefore, a new system was devised.

6.5.1 XML Test Oracles

The third requirement of an ideal marking system was to be consistent with the testing and marking performed. One way for this to occur is through strict adherence to a rigid rule / test set as can be described in a test oracle. During the redesign of the test oracle to take into account the additional requirements, it seemed appropriate to change their format. The obvious choice for representing the new oracle was XML. Not only is it very flexible, but it allows for the prior definition of what is to be expected in the file. This caused the decision on the format of the oracle file to be made at an early stage, it is the structure of this file that shall now be examined.

Each file has the opening and closing tags:

```
<GUI_TESTS>
</GUI_TESTS>
```

These allow the marking system to ensure that the file has been correctly formed and also acts as a marker for the end of the file. Within each file, there could be a need for multiple test sets rather than just the one, therefore another descriptor is needed.

```
<TEST_SET DESCRIPTION="Simple Test" MARK="50">
</TEST_SET>
```

Within the `TEST_SET` tag are two attributes that can be set by the question designer. The `DESCRIPTION` attribute allows the students to be informed of what the test is trying to achieve. This test description can be used to appear as a test heading in a results tree. The `MARK` is a relative value for how the marks in the current oracle should be allocated, this will require the marking system to assign the correct proportion of the actual mark to each test.

With the file basics now covered, it is time to look at the test and the results mechanism.

```
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[Ss]ubmit</TARGET_TEXT>
</TEST>
```

Above is shown an example of a test. Inside the TYPE tag is stored the information on the exact test that is to be performed, this consists of two parts, firstly the component on which the test is to be executed. BUTTON indicates it will be on an AbstractButton component, other examples include TEXT for text components, LABEL or even FRAME. The component type is then followed by an underscore and finally the actual test that will be performed upon it, in this case a button click. The test types can be anything that it is possible to perform on an object of that type, e.g. text entry into a text component. The tests available can easily be extended or complimented, the system administrator just needs to change the code within the system's GUITests package. Following the TYPE tag is the TARGET tag. This explains to the system how to recognise the component under test. The component type is already known from the TEST type earlier, however, it has been said many times that there could be multiple components of the same type and it is this tag which provides the information used for differentiation. In the example, it is the text of the button that is being used as the handle. This need not be the case, it could be any of the handles mentioned in the previous section, naturally this would change the tag e.g. <TARGET_ICON>. The data held within the tag is that which is needed by the identification method. This could be a regular expression for the text that is to be searched for, or the file name of the icon etc.

One advantage of using an XML oracle for both the results and tests is that it means you can add state checks in the middle of the test set, you do not have to wait until the end. Therefore, results tags can appear at any point in the testing process, or even on their own to check the original state of the system. They also have uses within loops which shall be mentioned shortly.

```
<RESULT>
<TYPE>TEXT_TEXT</TYPE>
<TARGET_NUMBER>1</TARGET_NUMBER>
<ORACLE>9</ORACLE>
<FEEDBACK>
  <BAD>Answer calculated incorrectly</BAD>
  <GOOD>Correct Answer</GOOD>
</FEEDBACK>
<MARK>50</MARK>
```

```
    <DESCRIPTION>Checking answer field</DESCRIPTION>
</RESULT>
```

The results criterion is similar to the test in the fact that it also contains the same function regarding object recognition using the TYPE and TARGET tags. However, in this case the TYPE is actually the method that will be used to check the results rather than perform the tests. TEXT_TEXT in a test would involve setting the text, whereas in a result criterion it would involve checking the text. Inside the ORACLE tag is where the actual state check data is stored. In the example above a text component is being examined to see if it contains the value "9" anywhere within it. The use of regular expressions allows for complex phrases to be searched for. Thus enabling the tests to be as simplistic or complicated as the question designer wishes. The need for feedback has been emphasised on several occasions and every time a state check is made, feedback should be returned to the student. The feedback for the majority of tests will consist of either correct or incorrect or in this case the aptly named GOOD or BAD. Following the FEEDBACK tag is a MARK value for this particular result criterion, each test could consist of several state checks some of which may have a greater significance. To allow for this, the MARK value for each criterion can be set, these are again relative and the exact marks will need to be automatically calculated. The DESCRIPTION tag contains a description of the analysis being performed and again will appear in the results tree for the student to gain an understanding of the operations occurring in the marking process.

The existing typographical oracles are slightly different from standard test oracles. Instead of a single regular expression, they have a collection of values that represent boundaries for use by the typographic metrics. Were the aesthetic tests to be implemented, they would also require similar functionality. To account for this, similar properties were added to the XML oracles to enable easier integration of aesthetic measures at a later stage. The aesthetic result criterion would have the following additional lines:

```
<L>0</L>
<S>15</S>
<F>30</F>
<H>45</H>
<FEEDBACK>
```

```
  <BAD_MIN>Not enough</BAD_MIN>
  <TOO_LOW>Low trying increasing number</TOO_LOW>
  <GOOD>Spot on</GOOD>
  <TOO_HIGH>High trying decreasing number</TOO_HIGH>
  <BAD_MAX>Too Many</BAD_MAX>
</FEEDBACK>
```

The L, S, F and H values have been based upon the value CourseMarker employs in its current typographic tests [Sym98]. The different feedback values also work in a similar fashion but the order has been changed to make it visibly easier to comprehend. If the value is less than L then the BAD_MIN is returned, if it is between L and S then TOO_LOW is return etc.

The final addition to the oracle is the use of loops. When using Graphical User Interfaces it is often the case that certain tasks are repeated. Without the inclusion of loops in the oracle, the file could rapidly grow in size and be difficult to understand.

```
<LOOP>
  <TEST>...</TEST>
  ...
  <RESULT>...</RESULT>
  ...
  <UNTIL></UNTIL>
</LOOP>
```

The LOOP tag is used to indicate the presence of repetition. Within the tag, any number of TEST or RESULT tags can be added, these are the tasks that are to be repeated by the marking system. It is the UNTIL tag that is used to determine when the loop should be stopped. This can be done in one of two ways. Firstly a single integer can be stored within the tag, this integer would be the number of times the system should iterate through the loop. The other alternative is to store a RESULT tag inside the UNTIL tag. This would ensure that the loop continues to iterate until the condition holds true, at which point it would stop and move on to the next task.

All oracle options currently available have now been mentioned. An example of a test file has been included as Appendix [A]. While the oracle file is now readable by the question designer, it still needs to be interpreted by the marking system. The XML file needs to be parsed.

XML Parsing

It has briefly been explained how the actual marking is performed by a separate process. To be able to do this, it needs to first be passed the details of the tests to be performed. However, at any point the marking process only needs the details of a single test set rather than the entire file. The XML file needs to be parsed prior to the marking process being called.

This was achieved via the implementation of Java's default SAX parser. The content handler was written to incorporate all the new features that could occur in the oracle file. As the file is parsed, the data is collected in an object designed for the task, this ensures that when each `TEST` or `RESULT` is completed, the information is returned in the order the marking system is expecting. The XML file is transformed into strings, similar in form to the oracle files of old but less easily readable by eye. It is these strings that are passed to the marking system when the process is started.

Now the test data is available, it is the marking process and the way in which the tests are executed that are of interest.

6.5.2 Test Execution

Once the marking process has received the test information, it is ready to process it and start performing the tasks requested. This is an iterative process as each test set can contain several tasks. Once the student's solution is dynamically loaded and the security policy confirms everything is safe, the testing starts.

The testing method involves examining the test strings that were passed to the marking process, these strings contain the information about what is to occur. All tests classes that are written are a subclass of the ComponentTest class. This class contains the methods that are used to extract the operational data for each test that is about to be executed. The methods called are different depending on whether it is a test or state check that it has been passed. Therefore, the first task performed by the marking process is that of determining what type of component test is to be performed, an instance of the test class is created e.g. ButtonTest and the test string is passed to it along with information from the descent parser. The testing process then determines whether it is a test that is to occur or whether the state of a component is

being checked and calls either the `runTests` or `getResults` method accordingly. These are abstract methods inherited from an interface and implemented separately within each component test. It is at this stage where the calls are made to the descent parser to locate the component in question, once the component has been retrieved the operation can be performed upon it. Even though a copy is made, the component information has only been passed by reference and the method called is exacted upon the actual component within the interface.

This process continues until all test strings in a particular test set have been read and all actions performed. The instance of the class is then closed and the garbage collected before the next string is read. However, the way in which the results are passed back to the main marking system has not been mentioned.

6.5.3 Retrieving Marks

As the marking process is being run on a separate thread, the marking results tree cannot just be collated and returned to the students as has happened previously. Instead as the tests proceed, messages are sent to the marking system via the server console. As students are unable to output text to the command prompt due to the security restrictions imposed upon them, there is no fear of pollution of the messages.

There are certain pieces of information that are stored within the test strings that are not sent to the marking process. This includes the data stored on the test set itself, this is extracted before the strings are sent to the marking process ready for the results to be returned. When the testing is executing, problems can arise with the tests themselves such as buttons not being enabled. It is possible, via the use of an error message, to communicate this to the marking system and the process can then be stopped as it will be unable to complete successfully. The other messages returned are all concerned with the checks made upon the components properties. These all return similar information, the mark awarded, the description of the examination and the feedback returned. This information is then converted by the marking system and stored in a mark results tree. The use of XML oracles by the test execution and mark retrieval process enables a system to be created which meets the requirements to provide students with graded marks and relevant feedback.

6.5.4 The GUITest Package

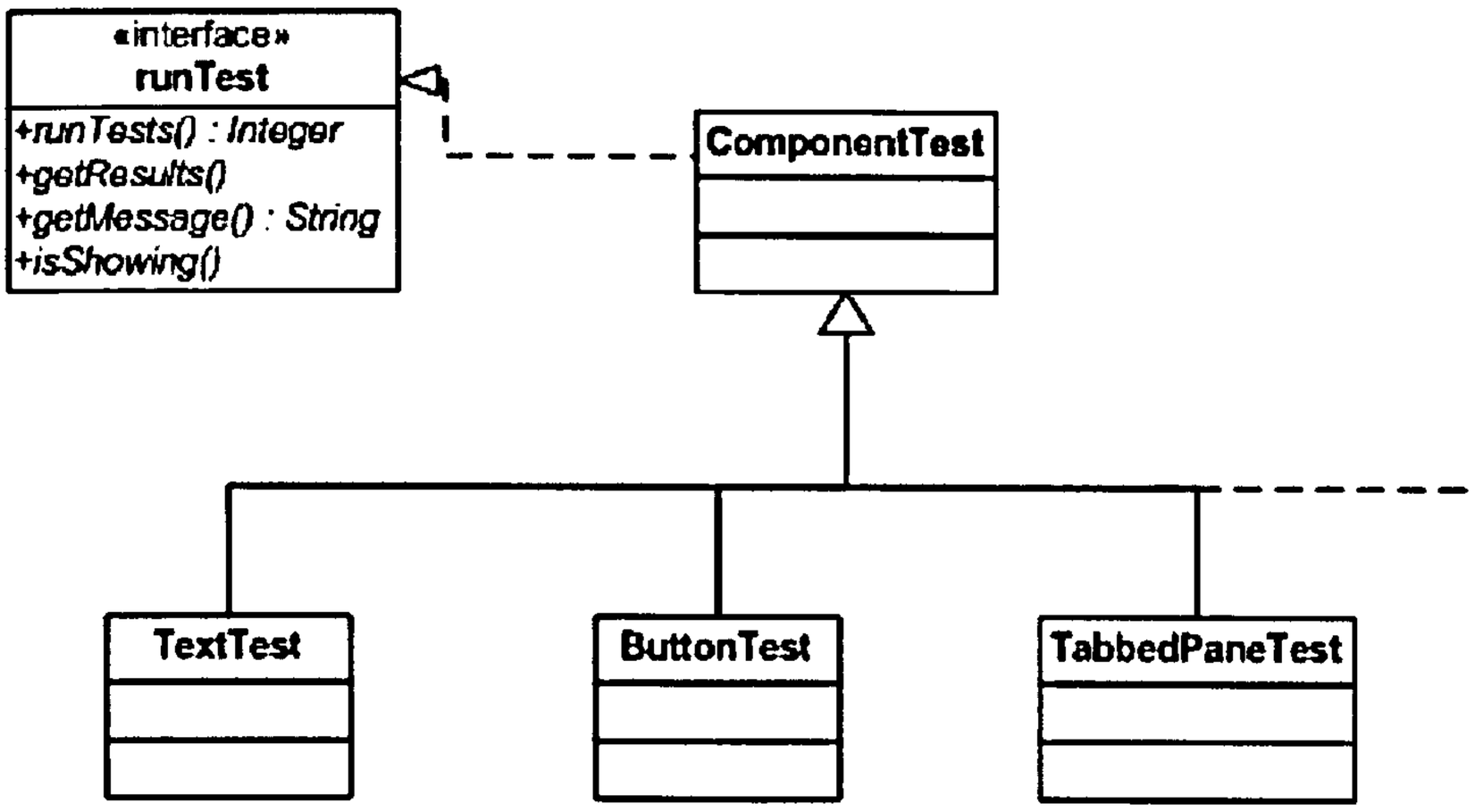

Figure 6.2: Interface Tests Class Diagram

The class diagram, Figure [6.2], shows how the test classes inherit from the ComponentTest class and consequently the runTest interface. It also highlights how a new test class can easily be added to the test suite. All of the child test classes have been created using a basic blueprint. This enables programmers to create new tests with solely their Java knowledge and a copy of the blueprint, intrinsic knowledge of the workings of the marking system is superfluous.

6.5.5 Summary

This section has gone into great detail on how the tests are represented in the XML oracle file. Examples of each tag used in the file have been shown enabling all readers of this section to now successfully be able to create a test for an exercise. It was followed by a description of how the testing system could be implemented.

6.6 CourseMarker Integration

The CourseMarker CBA system has been used to mark student's solutions for the past 7 years. It was also to be used for the foreseeable future. Whilst it was possible to create a GUI marking theory, without large scale testing it would be difficult to be

certain that the concepts created were plausible. With CM, there was a guaranteed set of student programs that would require marking either by hand or in an automated fashion. It was for this reason that the GUI marking was implemented and incorporated into CourseMarker.

The redesign that Ceilidh underwent to create CourseMarker was meant to have made the new system extensible for any additional tests that would be desired in the future. Whilst this was predominantly true, to allow for the marking of GUIs certain parts of the system needed to be rewritten to maintain the system's integrity. This section will discuss how the marking theory and CM's existing abilities were married to enable effective GUI marking.

6.6.1 Marking Command Programs

The actual marking performed by the marking subsystem is managed by two separate parts of the system. It is the marking command programs that request certain tests be performed. The test CMD files are explicitly called by the `mark.java` file. Two new marking CMD files were created for GUI marking.

The first was a new compilation tool. Naturally, a compilation tool must previously have existed to allow any program to be marked. For the student's program to be loaded dynamically by the system, the package location of the class file is required. Therefore, it was necessary to create a new compilation program that took the students solution and added a package line to the source code. If the program already contains a package statement, the relative location of the marking area is added as a prefix.

The other CMD file created was the tool that will initiate the marking process itself, whenever a new test is added to CourseMarker's testing framework, a new CMD file is required. It is this file that will call the corresponding tool once all precursory work has been done e.g. the extraction of all test and marking criteria. With all previous CM tests, the testing and marking oracles were stored as plain text, but the complex nature of GUI tests ensured that a more comprehensive format was used.

XML parser

Little needs to be said about the XML parser itself as it has just been described in

detail. Although it would have been possible to have parsed the oracle data during the running of the actual tests. It was decided that the functionality of CourseMarker was more important and a level of consistency should be maintained. Therefore, the oracles were extracted from the file and passed to the marking tool, in their new format, in a fashion similar to the previous dynamic tests.

6.6.2 Marking Tools

For every CMD file there is a corresponding test tool, it is these tools that execute the tests stated in the oracles. Again for each new test, a new tool needs to be written. However, there are a few significant differences between the interface tool created and the existing dynamic tool.

The functionality of both dynamic and interface tools is essentially the same. They assemble the data for each test set and pass this information along with any other relevant data and arguments to a class which sets a new process running. It is in the creation of the running test process where the differences occur. Previously, the class which ran the test data used to load the students solution as a new process and the test data was then piped through a standard stream and retrieved by the running program. To get the essential data from a GUI, the program needs to be loaded dynamically. Therefore, alternate versions of certain classes needed to be written.

The class that originally loaded the student's program and sent the test data was the first to be replaced. This was substituted by a class that performed similar tasks in that a new process was loaded. However, this process was not the student's solution. It was a class designed to dynamically load the solution and perform the tests upon it.

Dynamic Loading

The GUIRunner class was written not only for the purpose of loading the submitted program, but it also added certain reliability properties to the system. The reason the programs were run in a separate process was that it meant they could be tracked and killed if necessary. By doing the same with the Graphical Interfaces it is possible to again remove any fear of infinite loops etc. that may interrupt the system.

Security

When the GUIRunner decides it is time to load the interface, it needs to perform all the security checks that have been outlined. These are performed using an implementation of a Secure Class Loader. This implementation analyses every class that the program asks to be loaded. These are then checked against a config file which contains a list of packages and specific classes that should not be loaded, this consistency was maintained with the GUIRunner class. This class runs as a separate process returning marking results to the server console as required.

If the loading completes successfully, then an instance of the class is returned to the GUIRunner so that the marking process can continue.

Descent Parser

As soon as the class is returned, the tests can proceed. To be able to execute the tests, access to the internal components of the GUI is needed. This is gained thanks to the descent parser outlined above. The parser itself consists of five classes, the composite component classes are used to store the information about the components, the actual recursive parser and an exception class should the worst occur. The recursive depth first parser, once completed, passes the information extracted to the tests themselves.

GUI Tests

Previously, when solutions were marked, there was no need for actual tests. The testing was done by supplying a command line program with test data and the output was analysed. This can no longer be the case when GUIs are concerned, both tests and result analysis need to occur. Therefore, a new collection of marking tools needed to be created, the GUI Tests. They were written as an extra package to preserve the original structure of CM. Contained within the package are numerous classes created using the builder design pattern. There is a class for every component that would require testing, this is done at the highest hierarchical level possible e.g. all text components are combined inside a TextTest class. Each test inherits from a ComponentTest class,

this is responsible for breaking down the test information extracted from the XML file. This class is also abstract and implements an interface which contains methods needed by every test class. These are the `runTest` and `getResults` methods. Both methods need to work in a similar fashion, they need to locate the correct component using the handles provided in the XML file and then perform the required test upon it. The results are then output to the server console and picked up by the interface tool for conversion into the standard results tree.

6.6.3 Aesthetic Tests

Unfortunately, the importance of an aesthetically well designed interface has not been part of the Java programming course at the University of Nottingham for the past few years. Therefore, it was not possible to use the students solutions as official test subjects for the aesthetic marking part of the system. The aesthetic side of the marking system was implemented and tested so that the concepts presented could be shown to work. There were slight differences in the way in which this worked compared to the standard dynamic tests. Whilst the testing system was written in exactly the same method so that if requested it could be inserted into systems such CourseMarker with the minimum of effort, it was provided with a different interface. This was done so that different sections of the aesthetic testing could be accessed without having to create new programming exercises and a complete set of marking files.

Once the aesthetic marker was created to the specification defined, a number of test situations were created. The tests were designed to show how colour does affect aesthetic metrics. Therefore, it was not necessary for the tests to be run against actual graphical interfaces. The aesthetic marker was itself interfaced and the tests were predominantly based around conceptual images rather than actual GUIs.

6.6.4 Marking Criteria

A lot has been said about how the GUI marking system can be implemented, nothing has yet been said about what it will be used to mark. A number of example tests will now be described. This will explain what the system was used for and how it can be put to good use in performing tasks that would be laborious to undertake if marking

by hand. The majority of the results contained within the evaluation chapter relate to the standard dynamic testing of the interfaces as the aesthetic tests were run at a later date.

Dynamic Testing

The system can be written to be able to perform any function available to the programmer when they were creating the solution. To be able to test every possible action in this way would involve the students completing a comprehensive Java SWING course, a significant number of exercises being written and large number of test tools being created.

The students were restricted to a smaller subset of component tests. These included performing button operations on a scientific calculator exercise. The system had to locate the correct buttons and then iterate through a number of tests varying from basic subtraction, sin functions to much longer tests involving all operators. The system was also used to undertake a number of tests using JButtons and then performed the same tests this time using menu items instead. They were also required to load file contents into TextAreas, where the TextAreas were later assessed for their content.

A basic question specification for a Calculator exercise has been included in the appendix to highlight the type of program the system is more than capable of marking (See Appendix [B]).

Aesthetic Testing

The aesthetic tests were created to highlight the workings of the aesthetic metrics once colour has been included into the equation. Each test is explained in the following evaluation section along with a description of what the test is attempting to show.

6.6.5 Summary

This section has explained how the methodologies previously mentioned can be implemented and incorporated into an already existing marking system. A prototype of the system was used in 2004-05 in the second semester of the Java programming

course. Whilst not working at its full potential, it produced results and highlighted areas that needed to be improved for the system that was put into use in the course in 2005-06.

6.7 Summary

This chapter has discussed several important concepts that are instrumental to a successful implementation of a Graphical User Interface marking system. It was shown how the use of introspection and dynamic class loading can be used to access the internal state of a Graphical User Interface. This would expose any system using this method to a number of security frailties. Therefore, considerable time was taken to identify the problems associated with dynamic class loading and how the security loopholes can be closed to ensure the safety and reliability of the system.

Once it was discovered how the interface components could be accessed, it was possible to outline methods for the identification and retrieval of the internal components. This was completed with the use of a simplistic recursive descent parser. Finally, once an instance of the system was running and access had been obtained to all the internal variables it was then a case of explaining how the tests are represented in the oracles and how they are to be run and the results extracted. This completes the marking process and the mark results tree can be displayed on the students' client.

It has been mentioned how the various concept described enable a system to meet five of the seven ideal marking system requirements. The final requirements are for a marking system to be able to run tests automatically with no human interaction and consistency, which can be achieved by combining all of the aforementioned concepts. Whether the marking system meets all of the requirements shall be considered alongside the analysis in the next chapter. The next chapter will also describe how an implementation of the framework set out in this chapter has been incorporated into the CourseMarker system. It will also discuss how the new marking tools have been used by two classes in consecutive years and highlight the benefits of using an automated marking system when Graphical User Interfaces are being taught. Analysis was also performed on an implementation of the aesthetic tests once the effect of different colours had been accounted for in the equations.

Chapter 7

Evaluation

"When you blame others, you give up your power to change." -Douglas Noel Adams

As explained in the last chapter, an implementation of the Graphical User Interface Marking system was created as an extension to CourseMarker. This allowed for significant testing of the system to occur in a live, uncontrollable environment. Students in their solving of an assignment often created ingenious but complex and overworked solutions. This is very difficult to replicate during the standard laboratory testing process but is important for any marking system to be able to cope with.

This chapter will discuss the implementation of the marking system theory, its successes and failures along with changes that were made. Having included the system into CourseMarker, it has been used in the first year Java programming module at the University of Nottingham for two years. The results obtained from these years are also analysed.

The aesthetic marking tools were implemented independently of CourseMarker and were used to analyse the inclusion of colour into standard aesthetic metrics. Details of the tests and their results are are later explained.

7.1 Implementation

In January of 2005, a prototype of the Interface marker was completed. The tools were inserted into the then current version of CourseMarker so that it could be used in the upcoming semester. This enabled the system to be both load and real time user tested. As expected, it highlighted several aspects of the system that needed redesigning.

The system that has been presented here was predominantly based upon the initial design of the system. There was one major failing exhibited by the initial design, which required the system to be redesigned. All the concepts that make up the final design were used, introspection, dynamic loading, test abstraction etc. It was the way they were combined that caused the system to hang in certain circumstances. In the final design, when a student's solution is marked, a new process is started and it is this process that dynamically loads the solution and performs the marking. It then communicates the result back to the marking system. This design also allows the marking system to be easily incorporated into any existing marking program or to be run as a standalone process. However, this was not the case with the initial design.

At this time, the system was unprepared for some of the ways in which students with limited programming knowledge would attempt to solve the assignments set. This in turn caused ramifications on the stability of the system. No new processes were started and the marking system was more deeply embedded into the CourseMarker system. This had obvious problems with regards to the portability of the system. The main issue was that if there were a problem with the student's solution, it contained infinite loops or waited for input that it was never to receive etc. not only would the solution freeze but the entire marking system would also. Although the system could be recovered fairly easily and without the need for a restart. It meant the marking system was not as effective as was desired, nor did it have the level of robustness demanded by a high use system. Despite the failings within the system, it still managed to mark numerous solutions correctly and without incident.

The reliability issues were corrected the following summer by implementing the solution described herein. This allowed the marker to again be used in the second semester of the programming course in 2005-06. This second phase of testing occurred completely without incident. With the exception of power-outages and the host server being reset, CourseMarker ran continuously.

During this segment of the course, the students undertook five different GUI based exercises. CourseMarker and its logs have shown that all of the features built into the system have been tested. Students attempted to breach the security controls, most commonly by inadvertently leaving in print statements used during testing. These were all flagged by the system, showing uncategorically that the policy is working correctly. It has also managed to trap and destroy student processes that got lost either in infinite loops or through the creation of incomplete interfaces. One item of contention was that when CourseMarker was marking a solution, a fail would be returned if the program threw an exception and dumped a stack trace during the marking process. Several students experienced this over the duration of the year and often attempted to claim their solution perfect as the results displayed on screen were still technically correct. However, no student had ever noticed the exception thrown by the program and therefore had not attempted to remove the serious flaws from their code.

7.2 Recent Uses

The first year Java programming course has existed in its current form since 2003. Originally, there was no automated marking support for GUIs at all, that year all the GUI exercises were written by the course lecturer. The solutions the students created were hand marked by the lab demonstrators. The second year was when the prototype of the marking system was implemented. However, the exercises had already been written by the lecturer and included in his notes. This meant the tests had to be written around the questions, that is with the exception of the final exercise of that year which was written with the aim of testing both the marking system and the students.

Unfortunately the raw mark data for the 2003-04 year was unavailable. This was due to the data being entered into a web form and being stored on the lecturers own server. However, in that year the lab demonstrators hand marked all the exercises. There were limited instructions on how to assess the solutions and the demonstrators were to award a mark out of 6 for each exercise. This not only causes inconsistencies in the results, but also meant that only a very narrow band of marks were going to be awarded. In fact it is highly unlikely that the students received anything but zero or full marks. Especially as the demonstrators who marked it were also responsible for assisting the students with any problems they encountered, students were also allowed to have their work hand marked up to three times to maintain consistencies with CourseMarker.

7.2.1 The Current Course

The course (2005-06), featuring five graphically interfaced exercises, was undertaken by approximately 115 students. All the exercises were written to test both the students and ensure that the marker was able to perform all tasks that could be asked of it. Four of the interfaces the students have had to create since 2004 are shown as examples, See Figure [7.1].

Each new exercise attempts to assess a different concept from within the realm of Graphical User Interfaces. Whether this is by creating a system where the functionality is built solely into menus, or simply through the inclusion of different components each week. The inclusion of new concepts endeavours to stop the course from stagnating

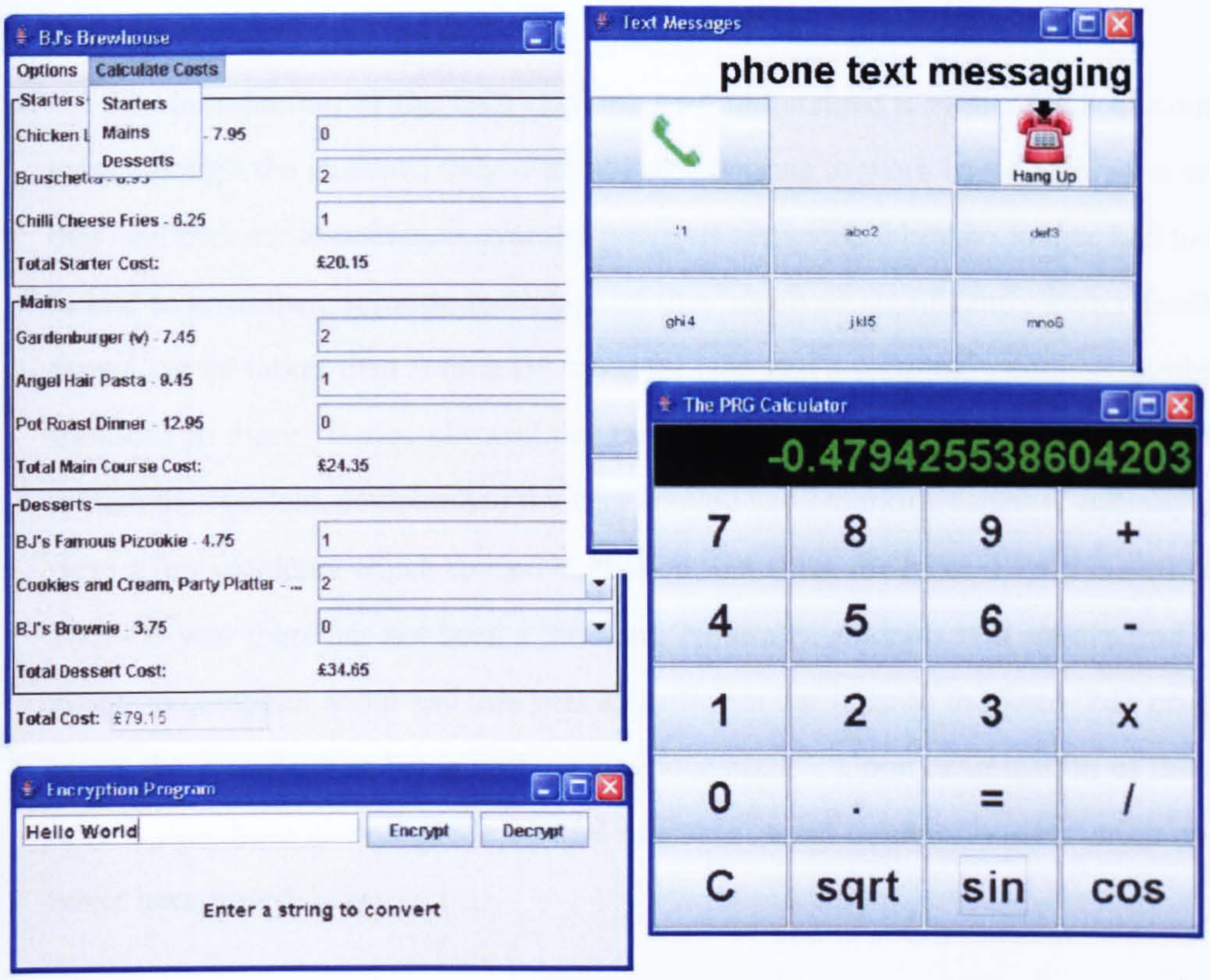

Figure 7.1: An Example of Interfaces Marked

and to keep the complexity of the exercises at a consistent level. The students were predominantly assessed on their ability to implement basic GUI components and get them to perform rudimentary tasks such as creating a calculator.

The changes that were enforced on the course simply by adding the marking functionality have all had an effect on the results and the way the course is run. It is the opinion of the author that these changes are all for the better and the results and statistics described will attempt to collaborate this.

7.2.2 Improvements

The improvements that have occurred over the course of the three years in question are plain to see. In the first year the solutions had to be hand marked by the demonstrators, this caused the students no end of grievances. It forced them to turn up to the lab sessions purely to have their solutions marked even if they were able to complete the exercises without assistance. When the students did turn up there was often a demand for the lab assistants time, whether for marking or talking students through problems

they had encountered.

The introduction of the GUI marking tool maintained a higher level of contentment amongst the students, they were able to continue to work in the fashion in which they had become accustomed over the previous semester. They no longer had to wait in line to have their solution marked, and they received significantly better feedback from CourseMarker than from a lab assistant who had a seemingly endless number of solutions to mark. It also allowed the students to get partial credit for areas of their system that worked, compared to the very limited hand marking scheme. Initially there were a few incidents which caused confusion, but since the rebuild for the start of the 2005-06 year there has not been a problem. Naturally, students will always find some aspect to complain about and this year all criticism has been of the form "My program meets the specification but is still given a poor mark". Upon examination of their solutions they have always made several mistakes, most commonly their program has never been tested fully.

7.3 Analysis

As the marking of Graphical User Interfaces is a multifaceted task, the analysis of such a system must also be. For this reason the analysis has been split to encompass each area in which the GUI marking tool has intrinsic value. The GUI marking tool has been analysed with regards to education, assessment, and from a technical and metric perspective.

7.3.1 Educational

This section concerns itself with whether the students are able to learn through the use of the system. This includes being able to improve their marks through feedback and the use of multiple submissions and also takes into account the students opinion on the methods of assessment.

In the two years of use, the GUI marking tool has been used to successfully mark 1508 Graphical User Interfaces. This in itself is a good indication that the automated marking of Graphical User Interfaces is feasible. Figure [7.2] shows the average marks

obtained by the 2004-05 year group for the five interface exercises, the 2005-06 equivalent can be seen in Figure [7.3].

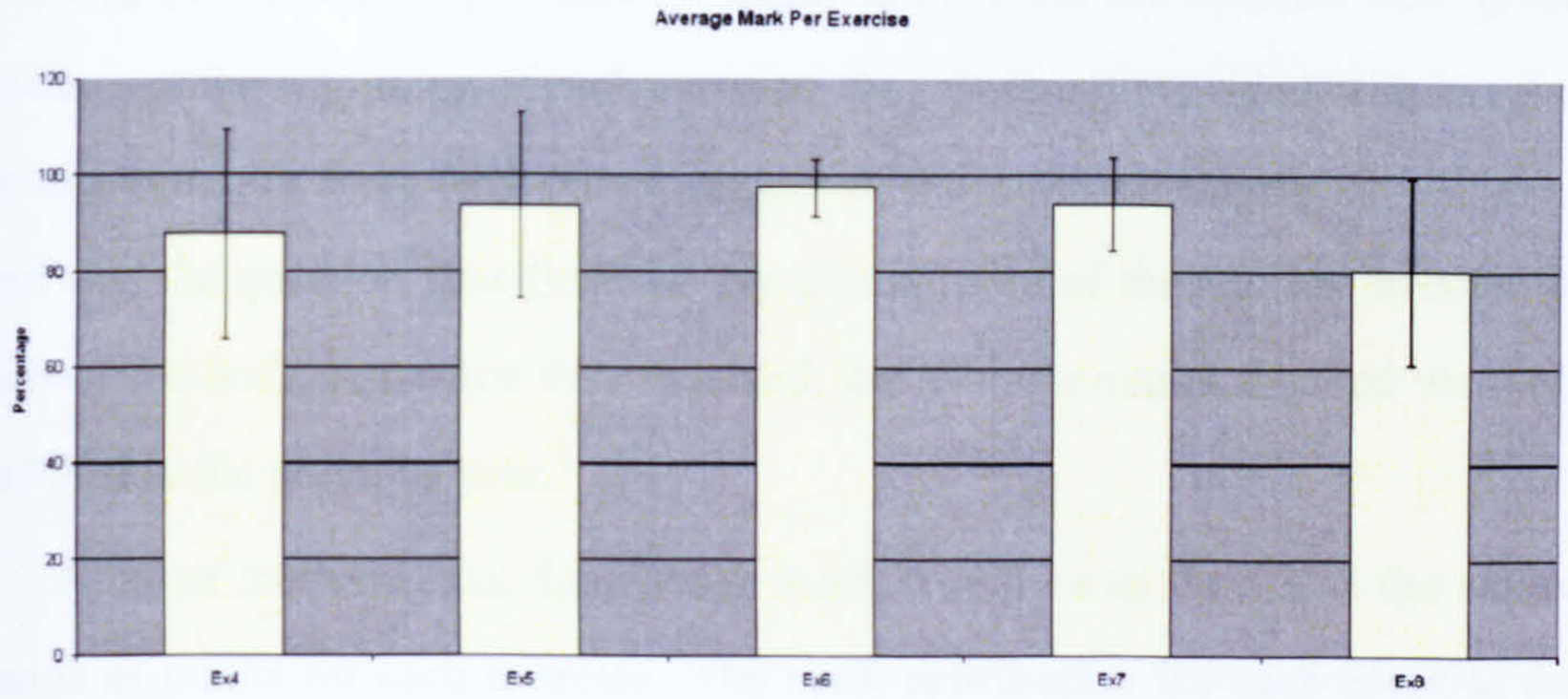

Figure 7.2: Average Marks (2004-05) with S.D. Bars

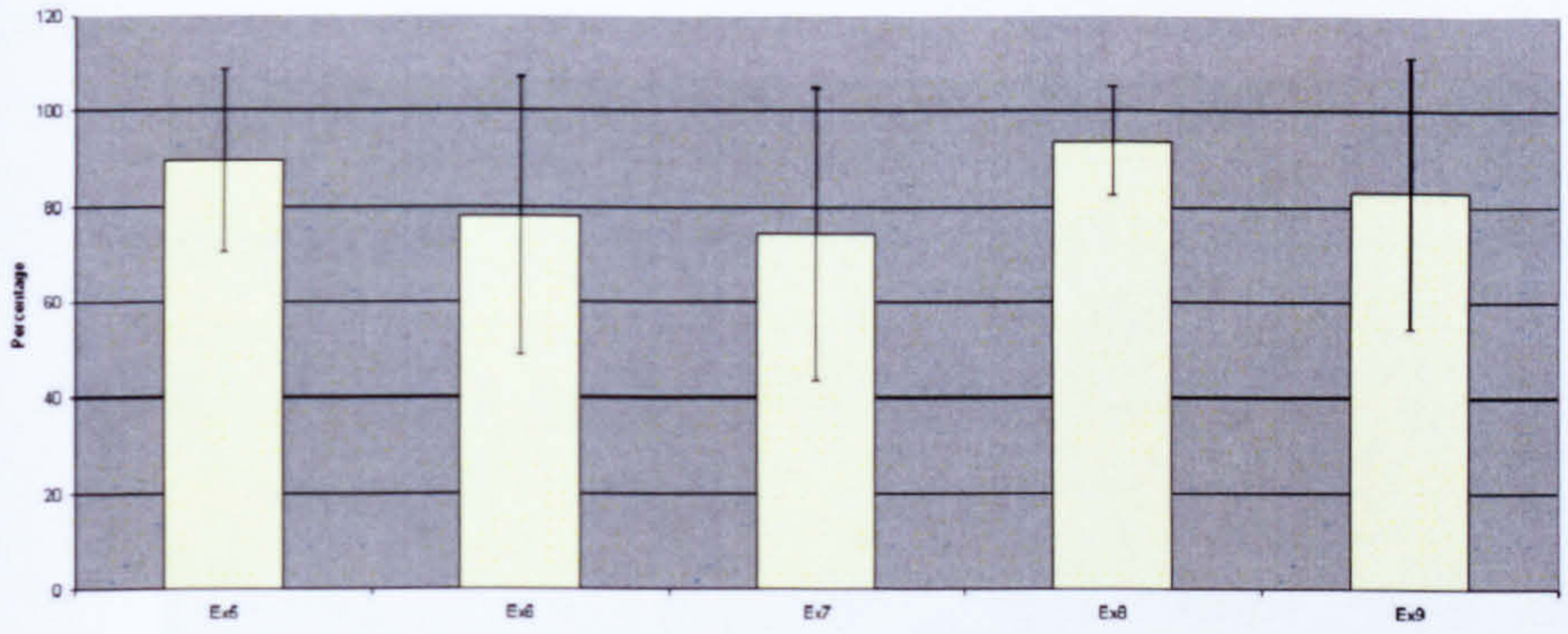

Figure 7.3: Average Marks (2005-06) with S.D. Bars

These graphs show that the average marks gained by the students for each exercise were higher in the 2004-05 year. There are several possible explanations for this, firstly the exercises themselves were written by the course lecturer before it was known that automated GUI marking was possible. These exercises were put into the course notes along with examples of the interface the students were to create. This instantly put preconceived ideas into the students' heads as for what they were to aim for. In addition, students were often given part written solutions from which to start work from. This meant that there was less scope for them to fail to meet the specification given. Regardless of this, the average mark is only 7% higher than the following year.

You will notice one exercise within the 2004-05 year has a significantly lower average than the others. This final exercise was not part of the original planned assessment but was added to the course after the notes were written, it was designed with the GUI

marking system in mind. Therefore, the students were given no template or graphical representation to work from. The exercise itself combined several parts of the previous exercises, no new concepts were included. In 2005-06, the students were given less assistance at the beginning of each exercise, they were not provided with any pre-written source code nor were they shown images of the expected interfaces. All the students had was the question specification. By asking more of the student through the reduction of the code assistance they received, the average marks attained were noticeably lower that the previous year.

A better measure than the average mark would be to show that the students get a range of marks for each exercise. The mark distribution for each exercise followed a similar pattern to that shown in Figure [7.4]. It shows that the majority of students (approximately 55%) received a very good mark. But there were also students getting marks across the spectrum.

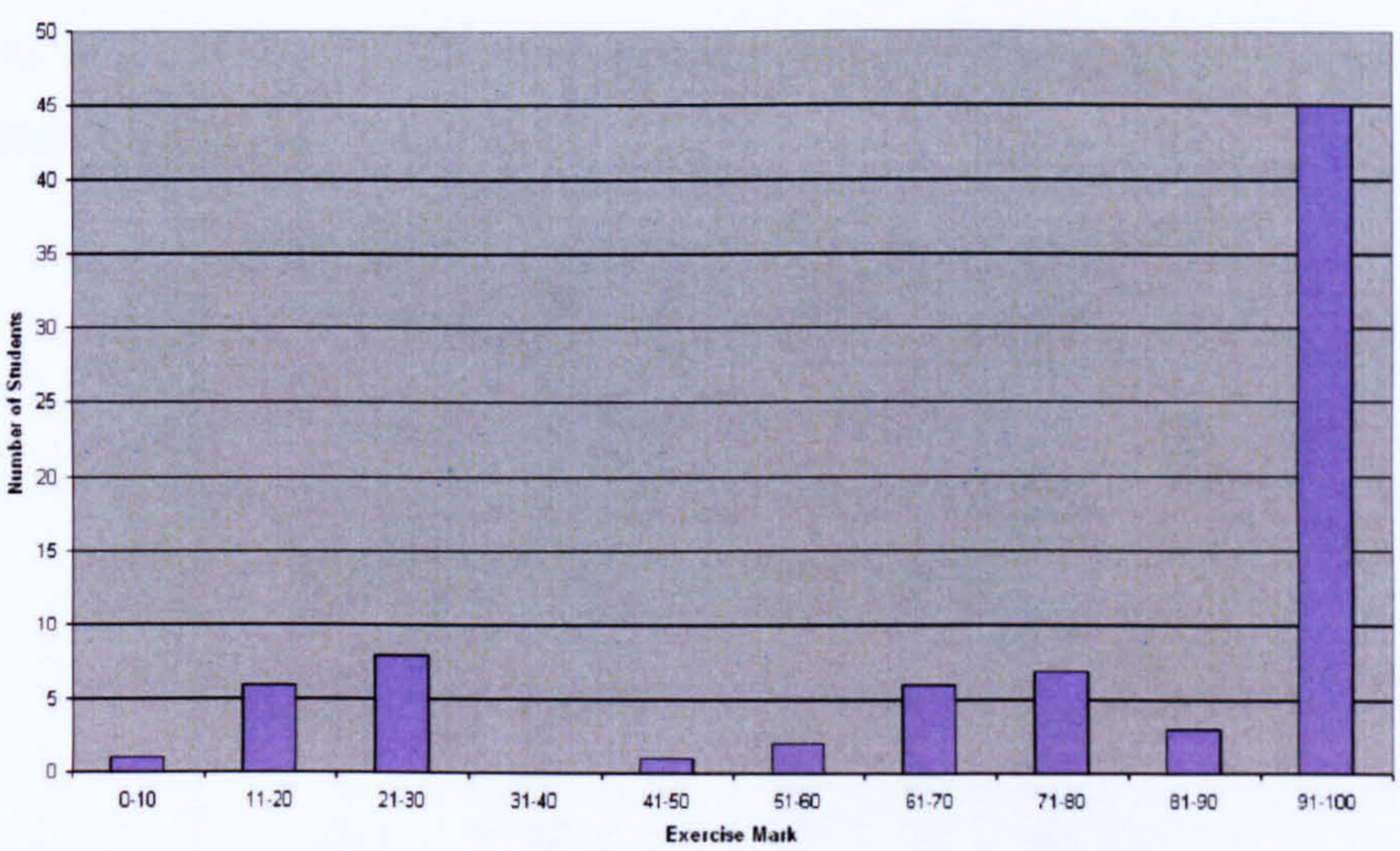

Figure 7.4: The distribution of marks for an exercise

The high marks are not unexpected because the students had the opportunity to correct their programs and resubmit several times. With one extra submission, the students were able to improve on their first marks by 8% average in 2004-05 and 10% average in 2005-06. Over all submissions, the students managed to improve their marks by on average 12.5% in 2004-05 and 14.4% in 2005-06. This would not be possible without the on-demand feedback delivered to the students. Were just a mark returned, students would have very little incentive to completely retest their solutions from scratch and determine where the mistakes lie. The feedback they are given speeds

up this process allowing the students to concentrate their testing in certain areas of the assignment.

The graphs and consideration of the hand marked solutions have shown how to get the best possible results out of students. Through the use of well considered exercises and automation to perform more complicated marking tests more regularly, a large spread of marks can be attained by the year group being assessed. This instantly allows us to see that the difficulty of the exercises was correct for the current year group.

Multiple Submissions

In the current incarnation of the course, each assignment has at most three submissions available to the students, there has been one exception which is obvious from the upcoming data. Tables [7.1, 7.2] show the number of submissions that were taken by the students in each year.

Submissions	Ex4	Ex5	Ex6	Ex7	Ex8
1	94	99	102	98	84
2	49	34	30	45	47
3	30	12	4	12	14
4	13	n/a	n/a	n/a	n/a

Table 7.1: Submissions Used 2004-05

Submissions	Ex4	Ex5	Ex6	Ex7	Ex8
1	98	84	85	85	86
2	45	49	57	29	40
3	12	22	26	8	15

Table 7.2: Submissions Used 2005-06

The most worrying statistic that can be drawn from the data is the varying number of students who attempt each exercise. The number of students who only make one submission should theoretically never change and is something that should be investigated to discern the reason why some exercises just are not attempted. The relative simplicity of the 2004-05 course is also evident from the low number of students taking their second or third submissions. In 2004-05 on average 43% of students used their second submission and 15% used their third, whereas in 2005-06 the averages were 50% used their second and 18% used their third.

Student Module Evaluation

One advantage of implementing the marking system and incorporating it into an official Java module, is the student evaluation that occurs after the module has finished. The students are given a questionnaire to complete and are asked to rate certain aspects of the module, it also contains three open questions on their impressions of the module. It was possible to obtain copies of the Module Evaluation forms from 2004-05. The majority of the data collected is of little value to this thesis. However, there were questions that confirmed that the students believed they were being assessed in the correct fashion for the subject.

One of the sections of the course they had to rate was "The method(s) of assessment are appropriate to the content of this module". The results of this are shown in Table [7.3]. The results show that the majority of students agreed with the way they were assessed. This was verified by some of the answers to the open questions. The only comments referencing the exercises were positive and that the exercises were helpful and enjoyable. No negative comments on the weekly exercises or how they were marked were made.

n/a	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	9	22	10	6	1

Table 7.3: Student Evaluation - Are the methods of assessment appropriate?

Summary

The use of feedback and multiple submissions has in the past been shown to allow students to improve their grades for standard command line driven programs and the same has now been done for Graphical User Interfaces. By automating the marking and feedback process has meant that lab demonstrators had more time available to solve students' problems and that marking consistency was maintained across all solutions marked. The advantages afforded to the students through the automation of the marking process maintained their contentment with the way in which they were assessed as demonstrated by the SEM feedback.

7.3.2 Technical

Two technical requirements of any automated system are that it must be both efficient and consistent. Automation is of no use if the tasks cannot be performed in a fraction of the time it would take to complete by hand. Also if the system cannot reproduce the same result given the same input consistently, then the rules by which it operates are not strict enough. With this in mind a number of tests were carried out. The system was given a number of interfaces to mark, two of which had perfect solutions and the other two failed on a number of tasks. They were then marked 20 times each to determine the average time in which the marks were returned and to see if the system always returned the same mark given the same solution.

The testing was performed on an Intel Pentium T2500 DC (2GHz) with 1GB memory. The timings were done by hand, starting when the submit button was depressed and stopped when the user was able to view the results screen. This approach was taken to show the user perceived time for the duration of marking. In order to rule out the chance the system was caching either the solution or results, the exercise marked was varied and the system regularly restarted.

In all cases the marking system returned exactly the same mark each time the solution was submitted. The marks returned were also correct given the schema that was created for the exercise. The two complete solutions returned marks of 100% and the two incomplete solutions returned marks of 80% and 95.8%. The time each submission took was also consistant, there was not more than a 1 second difference between the fastest and slowest time. Human error could account for this variation. On average, the basic calculator exercise took 13.1 seconds, the automated restaurant menu 10.4 seconds and an exercise where the buttons were located using the images they were displaying took 13.2 seconds. When marking a solution, for each test, of which there were around 5 per exercise, the system had to load up a new version of the program into the Java Virtual Machine before starting to run the test. The times also account for code typographical tests and any source code feature analysis that may have been included in the mark scheme.

These two tests highlight the major advantages of an automated system. If hand-marked, the tester would take a greater time over the marking process to ensure they are performing the test correctly at each step. They would also require more time than

an automated system due to the fact that it will take time to move the mouse from button to button. If the solution submitted is correct, it would be possible for a hand marker to return the same mark on each occasion. If incorrect, it would be a more difficult task as the marker would need to calculate the mark based on the mark scheme. Which, in the case of the 95.8% mark would be a complicated task as it is unlikely the marker would have the confidence to differentiate between 95% and 96%. Automation also removes the chance of the wrong button being depressed and a test being falsely assessed, assuming the tests have been correctly defined in the first instance.

7.3.3 Assessment

The assessment section is predominantly concerned with whether the GUI marking system returns the correct result or not. Whilst it is important that a system is able to return results to the students in a reasonable time scale it is irrelevant if the results that are returned are incorrect.

The results returned by the marking system could be checked using human comparative analysis. For a given test set, this would involve having half of the solutions being marked by hand. No analysis of this sort was performed during the official testing. However, for this testing to be accurate, strict marking criteria should be created and followed. These criteria would contain the tests sets and corresponding results for the hand marker to iterate through. This is exactly the same process that is followed by the automated marking system. Test sets have already been written in XML format which are both machine and human readable (See Appendix A). These test oracles are interpreted and run by the automated system in an identical fashion to the method a human marker would use. Therefore, the results returned by the marking system or a human marker are comparable.

Example Interface Assessment

It is also possible to ensure the GUI marking tool is returning the expected marks by examining the marks returned from two different interfaces. The two example Calculator interfaces (See Figure [7.5]) were both submitted to the marking tool for analysis. As has previously been stated, the interface on the left is a model solution

where as the one on the right is faulty. The interface on the right has the '7' button mislabeled as 'A' and the multiply button will subtract rather than perform the expected function.

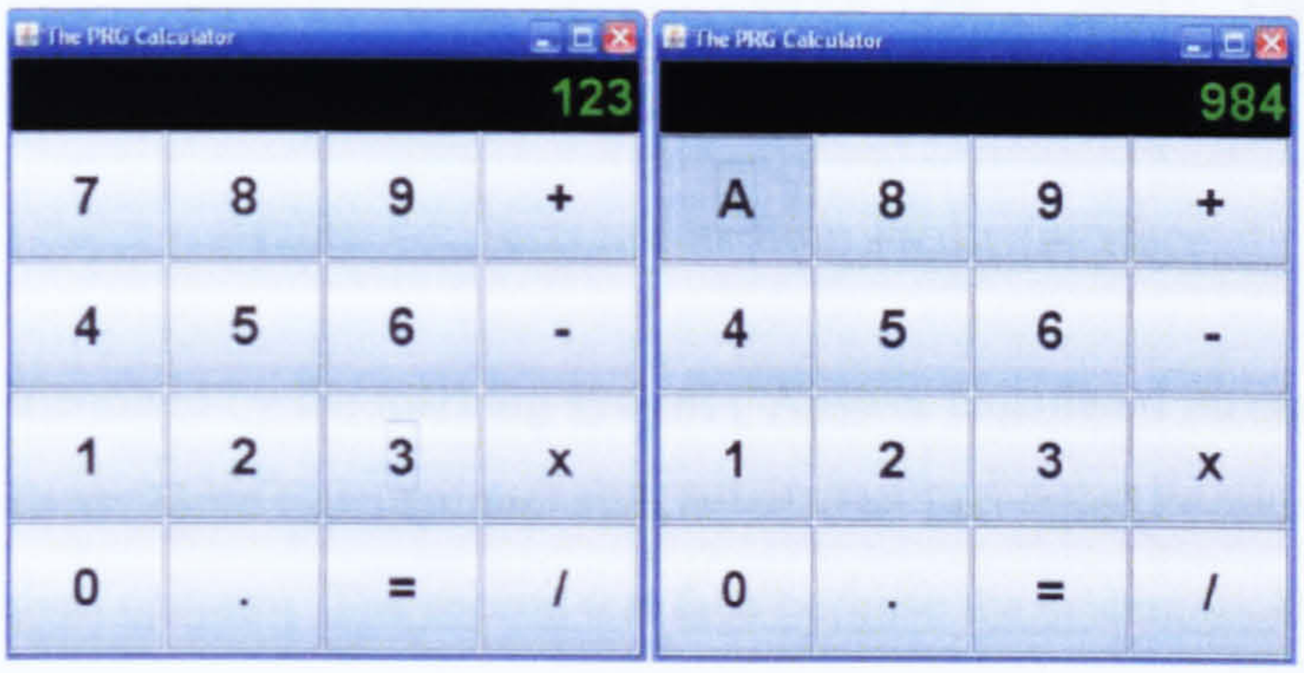

Figure 7.5: Example Calculator Interfaces

Naturally, it would be expected that the model solution would return a mark of 100%. However, the other interface had two different types of fault that would be noticed in different ways. Were the test sets comprehensive, then at every stage of the testing process the state of the interface would be checked to see that it was in a state ready to receive the next instruction. This would involve checking the state of all the buttons relevant to the next test, and would naturally return a fail each time it attempts to locate the seven button. It could also be the case that an initial state check is run as a standalone test before the others, this also would recognise the failing in the interface and could report the exact problem to the student in the form of feedback. Even if specific state tests are not run, they are implied when the system attempts to locate components. If a test requires the '7' button to be pressed, it will fail whilst locating the object in a similar fashion to the state checks mentioned, whereas the incorrect functionality should be noticed when the tests have been run and the results are analysed.

The interfaces were both run against basic tests which included using a variety of different numbers and the use of all four operators. The model solution returns a perfect score passing each test that is run against it and as expected the faulty solution does not. The solution received the following feedback show in Table [7.4]

As can be seen from the table, the interface failed three of the six tests and returned a result of 50%. To determine whether the marking system is assessing the interfaces correctly, the failed tests need to be investigated. It is known that the multiplication operator does not function as expected and should return an incorrect answer as was

Test Name	Test	Pass/Fail
Simple Addition	5 + 4 = **9**	Pass
Simple subtraction	12 - 21 = **-9**	Pass
Simple Multiplication	12 x 12 = **144**	Fail
Simple division	60 / 3 = **20**	Pass
More Calculations	6 x 4 = - 4 = **20**	Fail
More Calculations	2 + 2 = x 7 = / 3 = **9.333**	Fail

Table 7.4: Tests results from Faulty Interface

correctly identified by the marking system ("Answer calculated incorrecly"). The two tests known as 'More Calculations' also failed. The first failed because it also uses the multiplication operator. The second test fails because the system looks for a button labelled '7' which has been incorrectly named. This has been recognised by the marking system as demonstrated through the error message returned "No Button with required label found".

Summary

The assessment section of the analysis has shown the automated marking system is correctly recognising when the tests are failing and for the right reasons. It has also highlighted that the logical way in which a human would interpret the test sets written is the same method as is used by the marking system. It can therefore be inferred that a human marker, given sufficient concentration and consistency, would return the same results as the automated GUI marking system

7.3.4 Aesthetic Metrics

The final subsection of the analysis relates to the important role that colour plays when considering the aesthetic metrics. The first year programming course, which provided the dynamic testing results, did not covering the design or aesthetics of interfaces in any detail. Therefore, the aesthetic test tools were implemented and run independently at a later date. Incorporated into the balance test was the concept that the more dissimilar a colour is from its background, the more visible it is. Whilst the original metrics do provide value if black and white screens are assessed as soon as colour is introduced their limitations become clear. Therefore, the tests were run both with and without colour analysis to show that colour is required to deliver truly

valuable results.

The balance test was selected as the colour analysis causes a pronounced effect that can be easily recognised in the results. The effect that background colour has on objects can be seen in Figure [7.6] where seven greyscale objects were drawn on a background of white then grey. It is plain to see that on a white background, the white object is invisible and the black object is most visible. Whereas on the grey background the middle object vanishes and both the white and black objects are equally as visible but the contrast is not as great as on the white background. It is this concept of visibility that the enhanced metrics attempt to quantify.

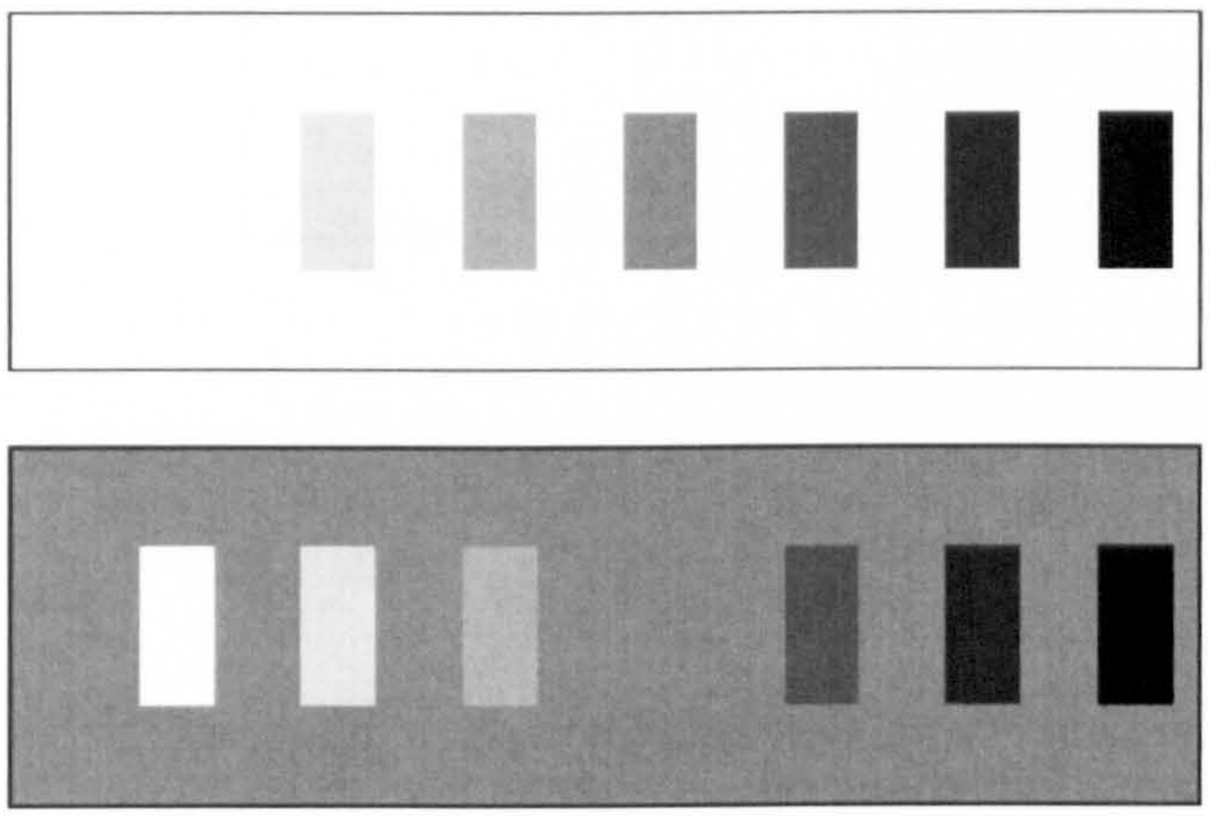

Figure 7.6: The effect background colour has on the visibility of objects

The testing was done on a 50x50 pixel screen with colour objects of size 4x4 pixels unless otherwise stated. The colour of each object and the background varied and will be stated as necessary. The concept of the balance test is similar to that of the centre of gravity of an object. This is calculated using the size and location of objects around the central point of the interface. It is suggested that interfaces should be centred around the middle of the interface so that no section is any more visible. Take the example of a text editor: When an empty document is loaded, the majority of the screen is empty with buttons generally at the top of the screen. The users attention will be drawn towards the buttons, which is a perfectly acceptable initial state for a system. However, once the document is being used, the colour of the letters on the blank document will draw the focus of attention back down towards the middle of the screen so that all parts of the software are equally visible. As our examples do not have initial states we want to assume the centre of balance should be the middle of the

interface.

Basic Tests

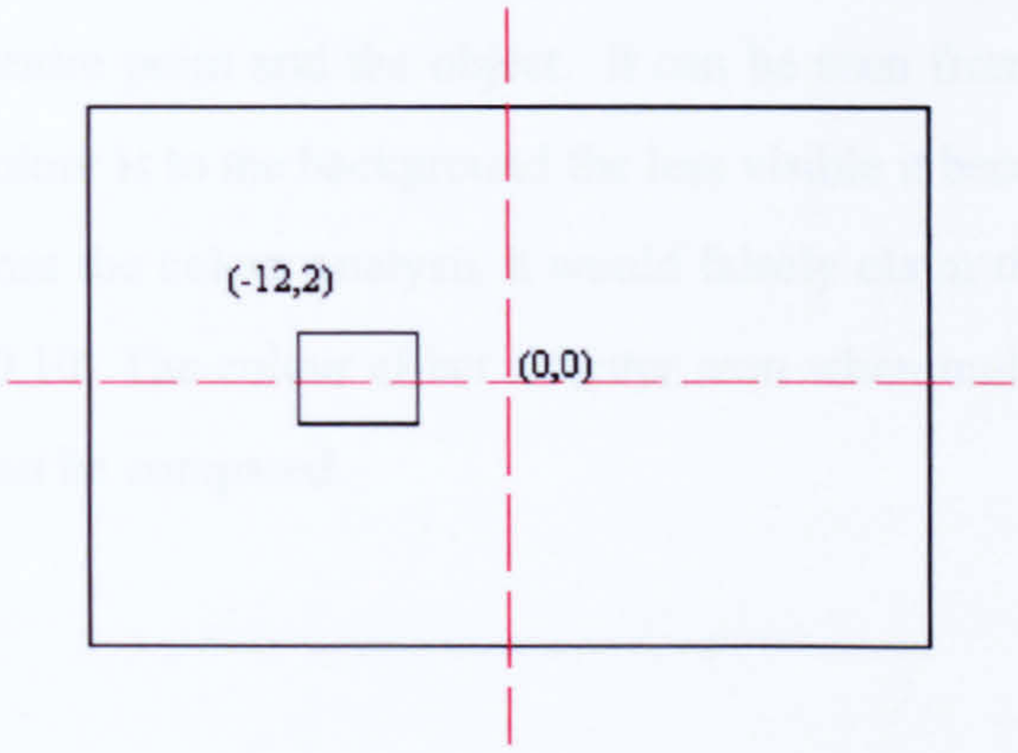

Figure 7.7: A Basic Aesthetic Test

The first tests performed were very basic tests involving one white object on a black background (See Figure [7.7]). With the two colours being opposites, there is a maximum difference (distraction value) between them and as such all of the results returned appear as if the colour was not a factor. In the tests, the object was moved around the screen to see how this altered the balance. Due to there being only one object, the centre of balance should be located at the centre of the object. The results e.g. Table [7.5], contain the x and y co-ordinates from centre of the screen of the top left corner of the object and the co-ordinates of the centre of balance as returned by the metric with or without the inclusion of colour.

Test Name	x	y	Metric Result	Including Colour
Central Test	-2	2	0,0	0,0
Left Test	-12	2	-10,0	-10,0
Right Test	8	2	10,0	10,0
Top Test	-2	12	0,10	0,10
Bottom Test	-2	-8	0,-10	0,-10
Top Left Test	-12	12	-10,10	-10,10
Top Right Test	8	12	10,10	10,10
Bottom Left Test	-12	-8	-10,-10	-10,-10
Bottom Right Test	8	-8	10,-10	10,-10

Table 7.5: Basic Aesthetic Tests - Moving an Object

Once the basic tests were completed, one test was repeated with a slight difference. The "Top Left Test" was altered in that the object was now grey in colour (RGB value 128 128 128). This change in colour halves the colour distraction and should influence the centre of balance because the object is not as visible as if it were white. The test was rerun and the metric result was as before (-10,10) but when colour was taken into account the centre of balance had moved to -5.02,5.02 which is approximately equidistant from the centre point and the object. It can be seen from this result that the nearer the object colour is to the background the less visible it becomes. However, were the test run without the colour analysis it would falsely claim that the centre of balance was still at -10,10. The colour effect is better seen when multiple objects are used as their colours can be compared.

Multiple Objects

The balance of an interface is less relevant if it contains only one object, it is the interaction between multiple objects that is of interest to interface designers. The use of one object is used as a control test from which other tests can stem. An example of the Horizontal Test can be seen in Figure [7.8] where the centre of balance has also been marked.

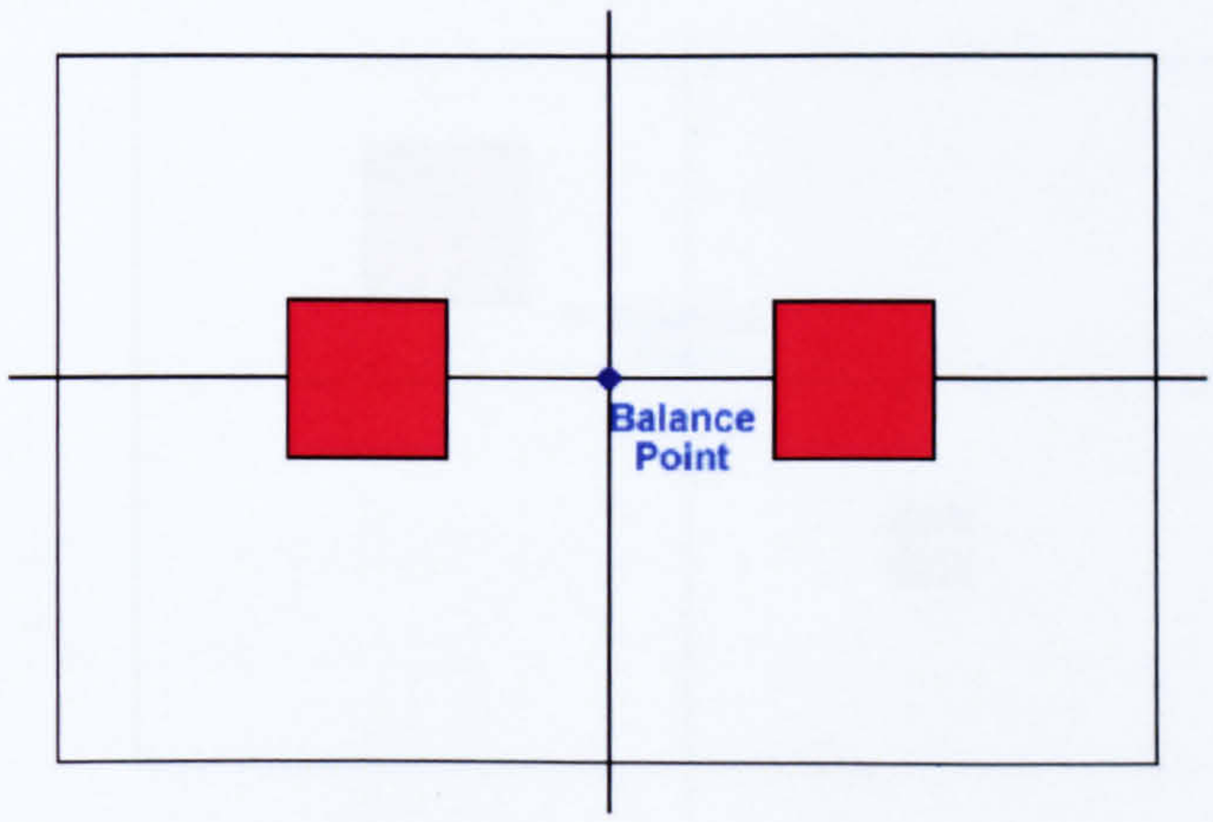

Figure 7.8: The Horizontal Test

Another object has now been introduced to the interface. It is of the same size and colour as the original object. The tests places the objects at opposite locations in the interface so that the centre of balance should be located at 0,0. The Table [7.6] contains

information in the same format as the basic tests, with the location of the second object having now been added.

Test Name	Object 1	Object 2	Metric Result	Including Colour
Horizontal Test	-12,2	8,2	0,0	0,0
Vertical Test	-2,12	-2,-8	0,0	0,0
Diagonal Test 1	-12,12	8,-8	0,0	0,0
Diagonal Test 2	8,12	-12,-8	0,0	0,0

Table 7.6: Multiple Objects - Basic Opposite Tests

The results returned are as expected and with the objects being the same colour, both metric results are the same. The results so far have shown how location affects the centre of balance. Another of the balance criteria is object size. In the next set of tests (See Table [7.7]) two different sized objects were used. Object 1 has a size of 8 by 8 pixels and Object 2 is still 4 by 4 pixels. An example of Diagonal Test 1 can be seen in Figure [7.9], again the centre of balance has been highlighted.

Test Name	Object 1	Object 2	Metric Result	Including Colour
Horizontal Test	6,4	-12,2	6,0	6,0
Vertical Test	-4,14	-2,-8	0,6	0,6
Diagonal Test 1	-14,14	8,-8	-6,6	-6,6
Diagonal Test 2	-14,-6	8,12	-6,-6	-6,-6

Table 7.7: Multiple Objects - Different Sized Objects

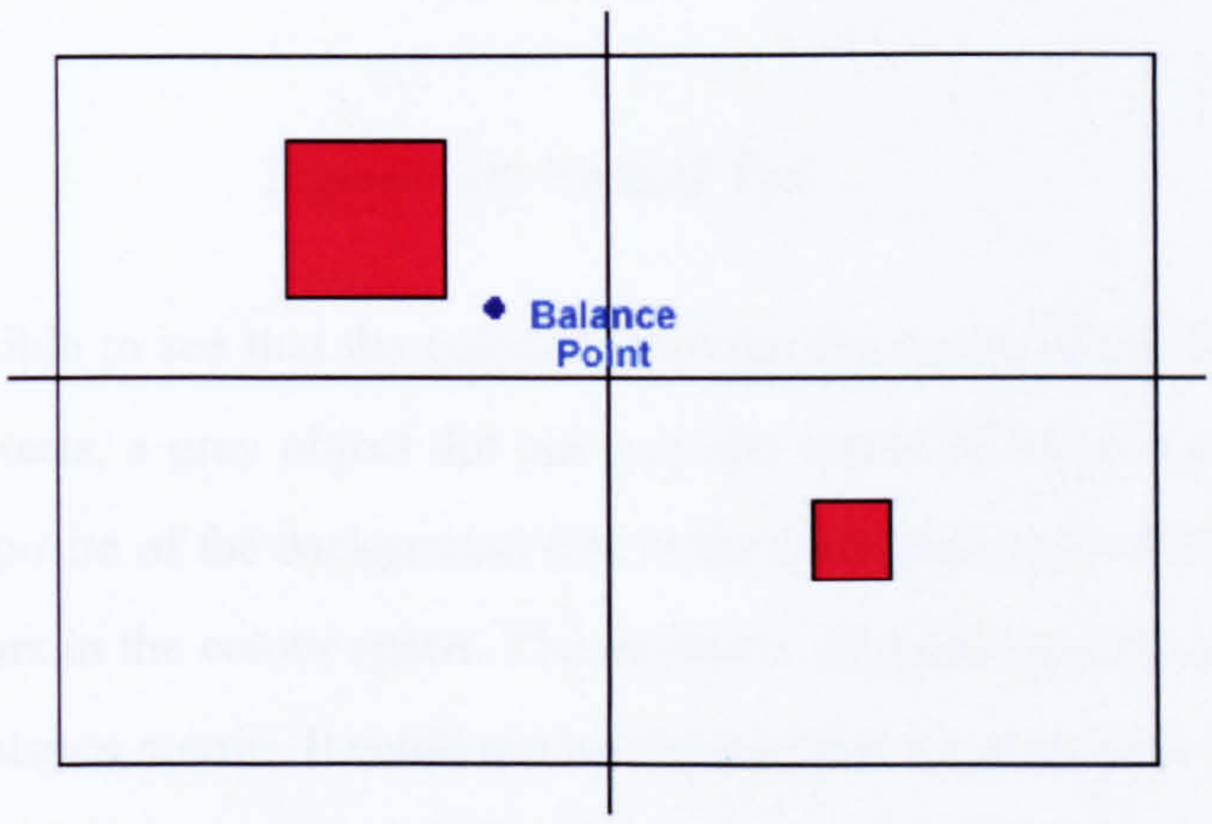

Figure 7.9: Diagonal Test 1

Whilst size is important to the metrics, it is the effect colour has on the results that is of particular interest. In the next set of tests (See Table [7.8]), two objects of the same size (4 by 4) were again used but they were of different colours. The screen

background was black (RGB value 0 0 0), Object 1 was always coloured white (RGB value 255 255 255) and Object 2 was always a shade of grey (RGB value 128 128 128). The Vertical Test is shown in Figure [7.10]. Because different coloured objects are used the balance point and co-ordinate returned by the standard metric are different both are displayed.

Test Name	Object 1	Object 2	Metric Result	Including Colour
Horizontal Test	-12,2	8,2	0,0	-2.49,0
Vertical Test	-2,-8	-2,12	0,0	0,-2.49
Diagonal Test 1	8,-8	-12,12	0,0	2.49,-2.49
Diagonal Test 2	8,12	-12,-8	0,0	2.49,2.49

Table 7.8: Multiple Objects - Different Colour Objects

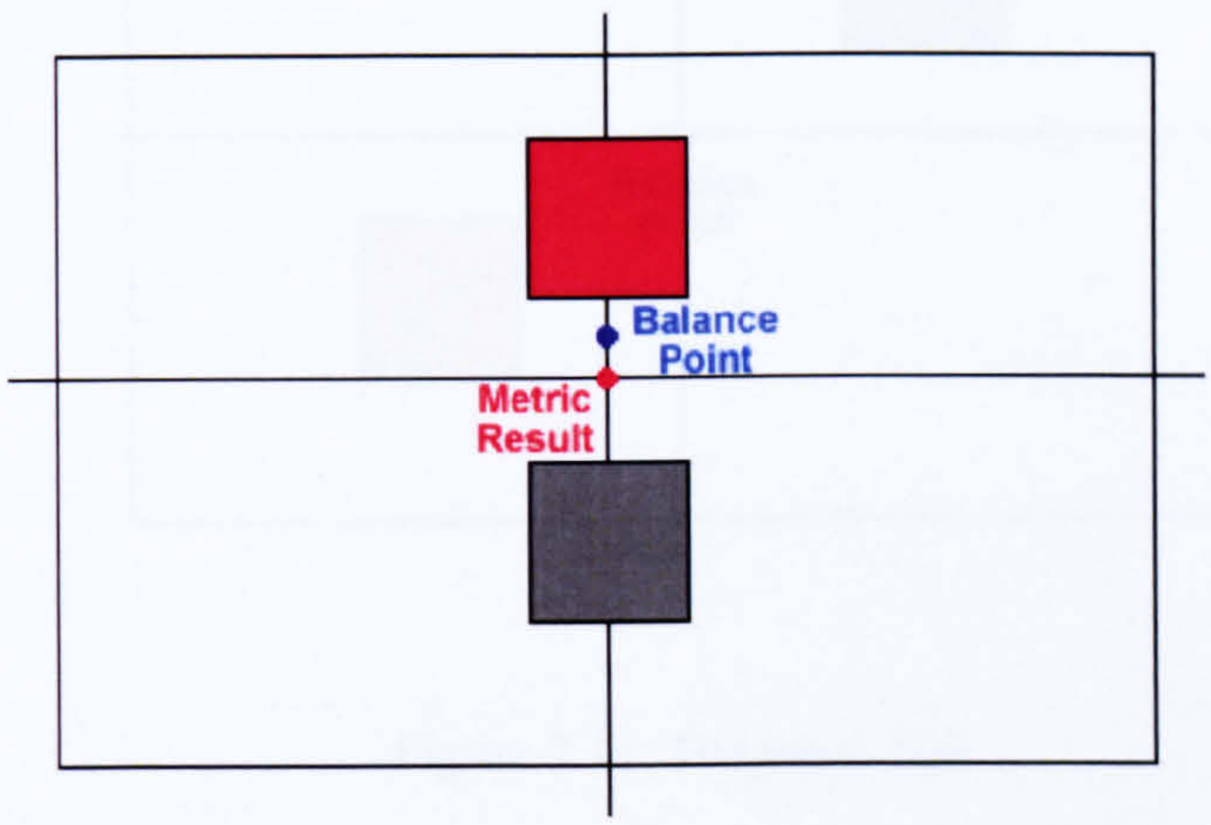

Figure 7.10: Vertical Test

It is now possible to see that the colour is moving the centre of balance. As was seen in the basic tests, a grey object did not vary the centre of balance as greatly as one the colour opposite of the background due to the difference in Euclidean distance between the colours in the colour space. The inclusion of colour greatly improves the accuracy of the balance metric. It could not be claimed that a colour with a distraction value of 10 from the background is just as visible as a colour with a distraction of e.g. 200. The closer a colour is to the background colour the harder it is to determine by eye that it exists.

In the next set of tests (See Table [7.9]), it will be shown that different colours with the same Euclidean distance have the same effect on the balance. In the tests, the background will again be black (RGB value 0 0 0). The colour of the objects will vary.

In the Horizontal Test Object 1 will be pure green (RGB value 0 255 0) and Object 2 will be pure blue (RGB value 0 0 255). In the Diagonal Test Object 1 will again be pure green and Object 2 will be pure red (RGB value 255 0 0). Figure [7.11] shows the Diagonal Test and how similar colours have equal effects on the balance point.

Test Name	Object 1	Object 2	Metric Result	Including Colour
Horizontal Test	-12,2	8,2	0,0	0,0
Diagonal Test	8,12	-12,-8	0,0	0,0

Table 7.9: Multiple Objects - Similar Euclidean Distances

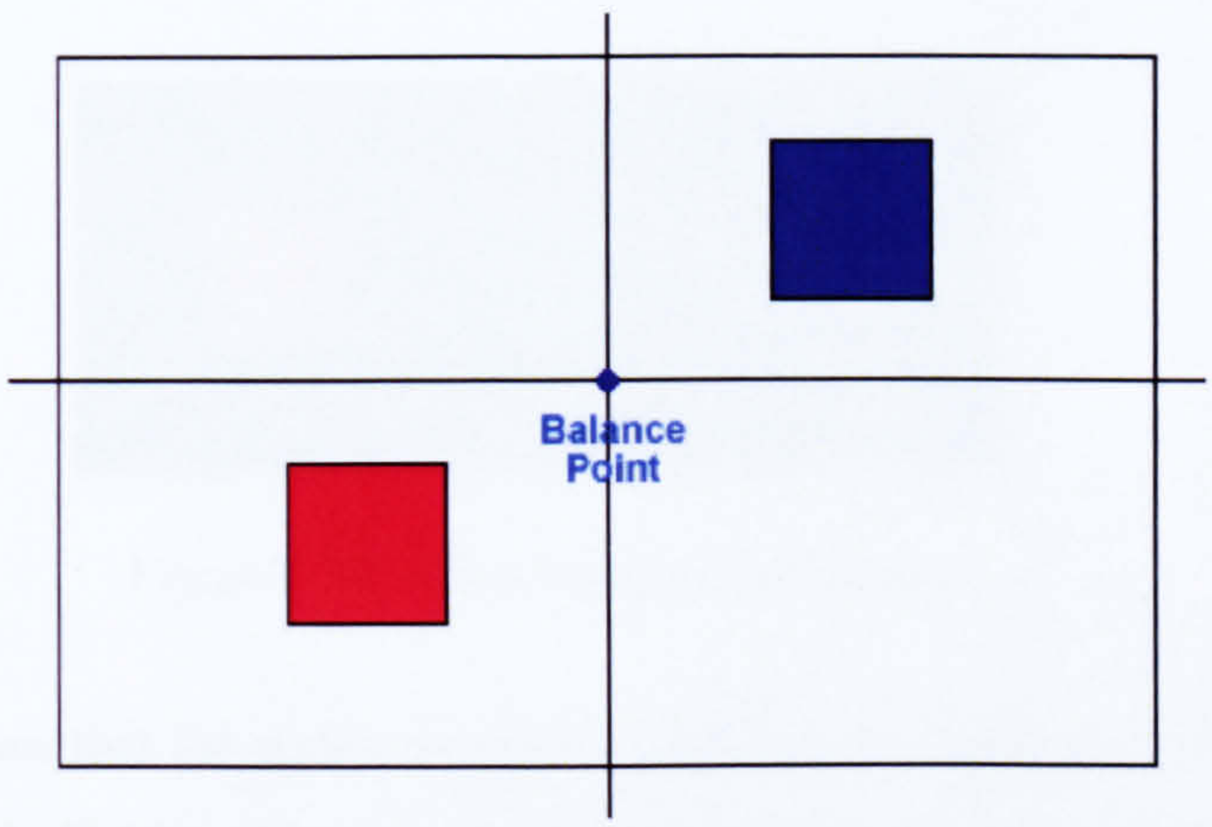

Figure 7.11: Diagonal Test

As expected, different colours with the same distraction values have equal effect on the centre of balance of an interface. While these objects cancel each other out, it must be noted that their distraction value is significantly less than the maximum value as the RGB values only vary along one axis.

Afterimages

In order to show that the application of colour to these metrics works as expected, an example is required. The example chosen is that of the reduction of afterimages.

Whether it be in the world of art or designing interfaces, it has been stated that grey is an important colour. The application of the colour grey reduces the chance of afterimages [Her78]. Afterimages are caused by having colours that are significantly brighter than their background and cause the viewer to still see the colours even after

they have looked away. Whilst interface designers want to use bright colours to make button icons visible, they are aware if the image is too bright it will cause afterimages. It can be noticed that a significant amount of grey is used in today's software.

The use of colour to reduce afterimages is effectively what Figure [7.6] shows, with the grey background reducing the contrast of the end objects. The way in which this works can be demonstrated through the application of the balance test. In this test, three objects are used: one pure red, one pure green and one pure blue. The objects and their location are essentially irrelevant, it is their colour that is of interest. For this test, the screen colour will be set to grey (RGB value 128 128 128) as shown in Figure [7.12].

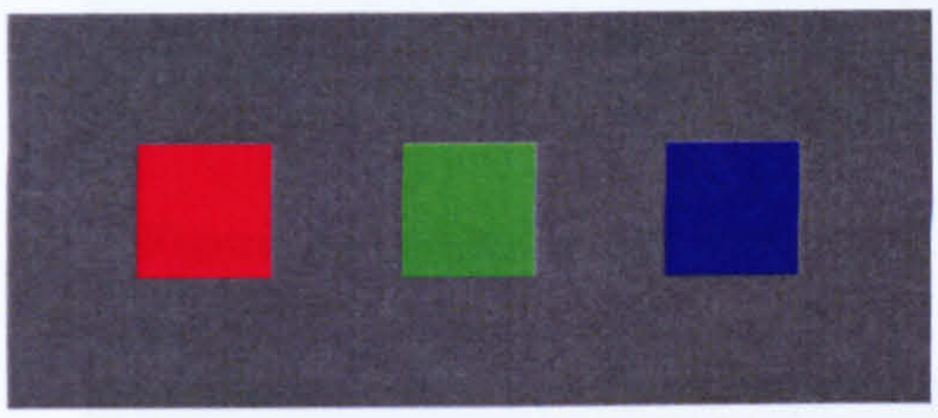

Figure 7.12: After Images Test Screen

It is clear to see that the metric results are significantly different when colour is included (See Table [7.10]). However, this is not the purpose of the test. In an attempt to see how grey alters the colour of the objects, the colour value generated needs to be analysed. In this test, the distraction values calculated were 0.5007. It should be possible to find another colour along the pure color plane that has the same distraction value. This can be done by taking the Euclidean distance from the grey background to any one of the three object colours, and then finding a colour the same distance away from the base colour. Black (RGB value 0 0 0) has been used as the base colour for ease of analysis of the RGB values. To obtain a distraction value of approximately 0.5007, a colour needs to be at a distance of 222 from the base colour. Therefore, if we run the tests with a black background and our three coloured objects with RGB value of 222 0 0, 0 222 0 and 0 0 222 the results (See Table [7.11]) should correspond to the original results.

Object 1	Object 2	Object 3	Metric Result	Including Colour
-12,22	-2,22	8,22	0,20	0,10.01

Table 7.10: After Image Test - Grey Background

As is visible in Table [7.11], the results are very similar. The variation can be put

Object 1	Object 2	Object 3	Metric Result	Including Colour
-12,22	-2,22	8,22	0,20	0,10.05

Table 7.11: After Image Test - White Background

down to errors in the rounding of the RGB values which need to be integers. The results can be seen in the following diagram (See Figure [7.13] and can be compared to an image which shows the pure colour values against the same background (See Figure [7.14]).

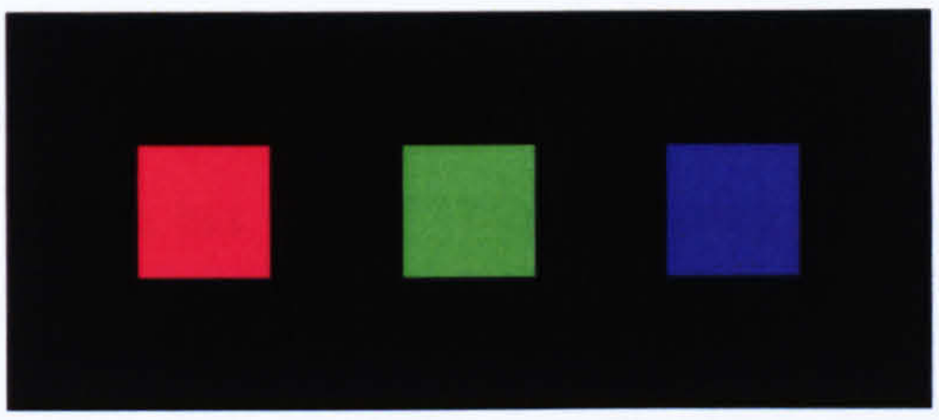

Figure 7.13: After Image Test Results

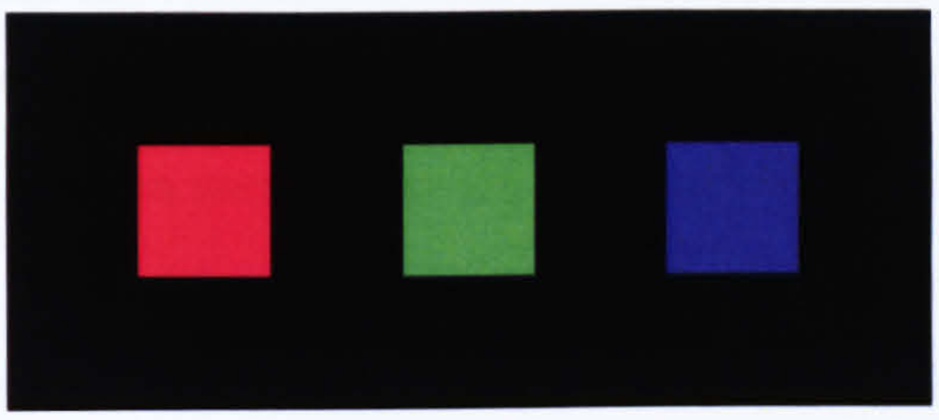

Figure 7.14: Pure Colours against a Black Background

The results show that by altering the background colour we have taken around 12.5% off the colour value, thus reducing the distraction values and the possibility of afterimages. The test results have confirmed that changing the background colour does alter the impact that colours have.

Summary

Whilst the results shown in this section do not cover all of the pre-existing interface metrics, it was decided that balance is the most obvious metric in terms of seeing the impact the introduction of colour has on the results. The first few results were used to show that the metric is able to calculate the centre of balance given a small number of objects on screen. The colour of the objects involved were then altered to show that different coloured objects have different effects on the centre of balance. The results showed that objects closer in colour to the background have less of a visual effect. The

effectiveness of the tests was highlighted by using the balance test to show how grey is used to reduce afterimages, a technique frequently used in the art and design fields.

With the colour theory shown to alter the way in which the balance is calculated, it would be plausible to incorporate this concept into all relevant aesthetic metrics such as sequence.

7.3.5 Summary

This section has shown how the development and use of an automated Graphical User Interface marking system has value. Through testing over a number of years, the GUI marking system has been shown to assist students in improving their marks by allowing student to submit their solutions multiple times and by supplying relevant feedback based on the quality of their solution. The technical capabilities were also analysed showing that the system is consistent and will return marks in a timely fashion. Another important aspect is that of assessment and whether the system can return the mark that any solution merits. Two example interfaces were marked and the analysis of these results showed that the GUI marking system correctly identified the issues with the interfaces and reduced the total mark accordingly. The final aspect that required analysis was the effectiveness of the aesthetic tests. These were run to show that whilst the original metrics do have value, the results returned were far more significant when colour is included. A test was also run to show how the balance metric also proved the idea that the use of grey in an image reduces the change of after images by softening the colour intensities.

The analysis of the aforementioned sections is relevant, however, more important is whether the automated GUI marking system meets the list of marking system requirements that were outlined in the introduction. By looking at the results the system has returned, it is clear that the system has met all of the requirements (See Section [1.2]). To confirm that all requirements were met, each one shall be analysed.

1. Run tests automatically, with no human interaction - None of the test run required any additional information to that supplied in the test oracle. Were additional input needed the tests as shown in the technical analysis would not have completed as promptly.

2. Be able to interact with the interface in any way required - The exercises the GUI

marking system was used to mark various exercises, each designed to test the students ability to use a number of components. Therefore, due to the number of different components implemented, the marking system needed to be able to perform a large number of different tests against the interfaces.

3. Be consistent with the testing and marking performed - The consistency was tested in the technical analysis by repeatedly running tests against the same interface.

4. Handle varying layouts for the same problem - As described in the educational assessment, each exercise was undertaken by approximately 100 students. The solutions they created varied but were all marked correctly.

5. Identify objects such as buttons in a number of ways - During the testing period, exercises were used that involved locating components in a variety of ways e.g. a mobile phone simulator which located buttons using both text and icons.

6. Be able to provide a graded mark for tests run - Over the two years in which the marking system was used, it provided marks to over 1500 solutions submitted for marking.

7. Supply feedback to support the marks given - Through the use of multiple submissions and the feedback supplied by the marking system, the students were able to improve their marks significantly.

By meeting all of the requirements, the system has proven its worth as a basic automated marking system. However, the rest of the analysis section has shown that such a marking system has the potential to provide significantly more value as shown by the implementation of aesthetic tests. The aesthetic tests showed how colour is invaluable when considering the layout of an interface.

7.4 Summary

In this chapter we have seen the accomplishments of the implementation of the GUI marking system. The fact that the Interface tool has successfully marked over 1500 GUIs indicates that the marking of GUIs is by no means impossible or impractical. The effectiveness of the system has also been demonstrated by both the average

marks and that a range of marks can be attained. This was also complemented by the fact that the students were able to improve their solutions through the use of the multiple submission and feedback mechanisms. The benefits of the security policy have also been shown, even if it was mostly triggered accidentally. This was followed by an analysis of the results collected by the GUI marking tool over the past two years. Independent tests were done to show that through automation a strong degree of consistency and efficiency can be maintained.

The aesthetic concepts were also implemented and shown how colour should be included when analysing interfaces. It is a powerful concept that should not be ignored: it can greatly influence factors when used correctly, or upset the usability of an interface if used incorrectly.

The results collected indicate that the Interface tool is not only able to mark a student's solution efficiently, but they are still able to gain the same benefits from the feedback provided and multiple submissions available that were noticeable with previous marking tools. The results also show that the exercises written with automated marking in mind provide a greater range of marks. The Interface marking tool, not only saves on the wasteful use of limited course resources, in the form of time and money needed to hire demonstrators to mark the work, but provides the students with more access to help as the demonstrators are no longer spending the majority of their time marking. Automated marking also removes the possibility of erroneous testing occurring through the use of strictly defined, automated rules.

Chapter 8

Conclusion

"Beware of the man who works hard to learn something, learns it, and finds himself no wiser than before." - Kurt Vonnegut

It is becoming increasingly rare to encounter computer software, which does not have a Graphical User Interface. This is most noticeable in the home and commercial software market. However, although GUIs are being written for applications on a daily basis, the amount of research into their analysis remains limited especially with regards to testing and marking.

This thesis has presented a solution to the problem of marking Graphical User Interfaces. It has described how a robust framework for implementing testing solutions can be created and then demonstrated its abilities by implementing it as an extension for the CourseMarker CBA system. The Interface tool created has been tested in a real time setting using two groups of first year computer science students. The results it has returned have shown the system to be both reliable and consistent in marking GUIs as well as reducing the time lecturers and demonstrators need to spend assisting the students with problems encountered.

This chapter shall now highlight how it has met all the objectives it strived to, describe what contributions it has made and finally explain how the system could be taken further.

8.1 Objectives

Before the research started, a number of objectives a potential solution had to meet were outlined. The objectives defined cover all important areas of the marking processes.

The main aim was to design a solution capable of automatedly testing and marking Graphical User Interfaces from both a functional and aesthetic perspective. The implementation of the solution presented has now been used for a number of years at the University of Nottingham for their programming assessment. In the first two years of use over 1500 interfaces were successfully marked. This conclusively shows that the creation of such a system is feasible. Not only feasible, but also flexible. The solutions marked were across a small number of exercises. With over 100 students participating in each exercise, a number of different solutions were presented each performing the same tasks. When GUIs are involved the variations between them are generally layout related. These variations were one of the deciding factors that caused the introspec-

tive approach to be designed. Over the duration of the system's use, no solution was ever rejected because it was unable to decipher the way in which the interface was implemented.

The adoption of the system by an official course also meets another of the objectives, which was to design a system capable of reducing the marking burden put upon staff and course assistants. With the marking no longer done by hand and no other issues raised by its adoption, this was another aim successfully completed.

The remainder of the objectives concerned the functionality of the marking solution. The automation of GUI marking demands that any solution be efficient, effective and increase the consistency of the marking. All three requirements were successfully demonstrated in the evaluation chapter. Each test set run involves repeatedly loading, testing and marking the program submitted. The results showed that the marking function could complete and return the results and feedback in under 15 seconds[1]. The consistency was shown through the repeated submission of the same solution. Each submission returned exactly the same score, even when defective solutions were marked. Effectiveness is a measure of whether a system performs the tasks required of it. By meeting all of the aforementioned objectives, the approach used to mark GUIs has demonstrated its effectiveness within the realm specified.

The final objective was to design a solution that is easily extensible for the inclusion of additional tests at a later date. Through the modular design of the test class hierarchy, new tests can be inserted into the system without the need to rewrite any of the existing code.

8.1.1 Summary

All of the objectives that were required by a useful automated marking system outlined previously were met by the theory presented. The introspective approach to marking has now proved that it is a suitable solution for the purpose that it was designed. This suitability is highlighted by the fact that the implementation of the solution is still in use at the University of Nottingham in their Java programming course.

[1]Given the tests runs, which included 5 different test sets.

8.2 Contributions

This thesis strived to make specific contributions through the meeting of the aforementioned objectives. The main objective of this research was to design an automated system with the capability to test and analyse Graphical User Interfaces in an educational environment. This goal was reached and emphasised by creating an implementation of the framework designed and testing it in a real world setting. This was achieved by incorporating it into the existing CourseMarker CBA system.

The concept created has made a number of contributions in different areas relating to Graphical User Interfaces and their creation and evaluation. These contributions are listed below along with a brief description of how they were achieved.

8.2.1 Education

The educational benefits from having a Graphical User Interface marking tool are clear. The marking of students programming coursework has always been a complicated task and time consuming task and although systems capable of marking CLIs have existed there was nothing to assist the marking of GUIs. With the added complexity of GUIs the time required for marking was considerable. However, the creation of an Interface testing framework has allowed for the creation of marking tools such as the one implemented for CourseMarker.

The implementation of the system has shown that the automated marking of GUIs is very much feasible and is able to efficiently and consistently mark numerous student submissions and return feedback within a reasonable time frame. It has also been shown that students are able to benefit from multiple submissions assisted by the feedback provided by the system, whilst still providing the course administrator with a good range of results. The quality of the results obtained have highlighted the fact that the course administrator and demonstrators are able to spend their time more profitably e.g. assisting students rather than marking.

8.2.2 Testing

The majority of testing solutions can be labelled as one of two types, Capture / Replay tools whose failings have been discussed or programmable test suites, which require the user to spend significant time creating the tests. This thesis has highlighted a new methodology for the testing of GUIs. Through the use of dynamic loading and introspective methods tests can be performed on an interface in a secure and stable fashion. Once the system has been initially created, it allows user to perform tests by describing them in the XML test oracles. Limited knowledge of both the marking system or the interface is required.

8.2.3 Aesthetic Analysis

The final contribution concerns the aesthetic analysis of an interface, several metrics have been created measuring a number of different criteria. However, these metrics have always assumed that the colour of the objects being considered is irrelevant, this is not the case. Colour has a pronounced effect on the viewer and can both help and hinder a user when attempting to locate a component in an interface. Differently coloured objects should not have the same influence when the aesthetics are analysed. It is especially true if the object is the same colour as the background as this would render the object invisible.

A way of quantifying the effect that colour has was created. An equation was constructed based around the difference in euclidean distances of the two colours from a specified point in RGB colour space namely that of black, RGB value (0,0,0). The equation was then used to embellish the preexisting metrics to allow them to take colour into account.

The successful inclusion of the influence that colour has into the basic metrics increases the validity of the metric tests. It is also a good base for a new breed of aesthetic testing metrics that are based around the colour of the objects involved.

8.2.4 Test Repository

In order for a marking system to be able to mark all components in all types of

GUI, test classes need to be created to mark each different type of object. This will be a complex task requiring comprehensive knowledge of all Graphical User Interfaces, not to mention the time required to create all relevant tests. Due to the modular design of the GUI Test package it is possible to plug in new test classes with great ease. This "plug and play" capability allows for the sharing of test classes. A central repository could be created to offer any test created to be shared between the users of the system meaning that only specific tests would need to be created.

8.3 Limitations

Whilst the GUI marking theory does provide many advantages when it comes to teaching programming, no system is ever perfect. The theory behind the marking system is solid, however, there are naturally a number of limitations.

The biggest limitation is that a test class needs to be written for every object that is to be tested. Even though the majority of the code could be copied between classes, as it will concern how the object is located, this is still a timely procedure. It was just mentioned that because of the modularity of the test classes, tests could be acquired from e.g. a central repository. This does not mean that all tests are instantly available, they will obviously need to be created by a user before they can become available. This would only be a problem in the initial stages. Once established, the system will only be hindered if new objects or interfacing packages such as SWT are created.

Another possible limitation concerns the programming languages the interfaces are written using. If the programming language does not support dynamic loading or the dynamic loader does not supply the system with access to the program internals then it will not be possible to mark GUIs in the method specified.

8.4 Future Work

Whilst the creation of a framework for the marking of Graphical User Interfaces as outlined in this thesis has shown to be both feasible and complete. It does not mean that the solution described is the limit of the functionality of the system. As the research progressed, further extensions to the system were suggested that would be beneficial

to whatever situation the solution was implemented for. These possible extensions are described below.

8.4.1 Aesthetics

The importance of creating an interface that is both ergonomic and aesthetically pleasing has been discussed in previous chapters. However, the main focus of the metrics suggested was the inclusion of colour into preexisting tests.

Therefore, the most natural extension would be the creation of all other interface tests, this not only includes the balance and proportion metrics that were described in detail, but other aesthetic measures. Other measures would include, analysis of the colour schemes used taking into account the different colours and the harmonious effect they create in the current proportions. More simplistic measures would just ensure that there is a level of consistency running throughout the interface, determining whether forms are correctly aligned or menu items follow standard conventions.

This extension would still be important in an educational setting, but the automation of alignment tests etc. would be better appreciated as a time saving device when used during commercial software development.

8.4.2 Assisted Test Creation

This research has focused on the automation of the process of marking Graphical User Interfaces. However, this need not be the only automation in a GUI marking system. The most laborious part of creating the students assignment is the writing of the tests to be performed along with the test data. As mentioned, this information is stored in an XML file for the system to parse at the relevant time. When large tests are to be executed upon the interfaces, it can involve the creation of a large file needing significant time to ensure it's correctness.

Using a concept similar to that of Capture / Replay software, it would indeed be possible to at least partially automate the test creation process. The GUI marker has described how it is possible to gain access to the internal components of a GUI. By harnessing this power whilst running the model solution, it would be possible to create a system that monitors the event queues and listeners registering whenever the user

performs an operation upon the model solution. These operations can then be augmented by the user, with additional information such as test data or object recognition preferences, allowing the system to write the XML testing sequence for the question designer. This would not only speed up the question writing process but also reduce the opportunity for errors to appear in the test oracles.

8.4.3 Test Expansion

The final part of the GUI marking system that lends itself to expansion involves the GUI tests. While the aesthetic tests could be counted as test expansion, they are but a small part of the possible extensions that could be included.

The deployment of the GUI marking system included only the tests that were required for that year's course assessment, therefore, there are not only several GUI components that currently have no marking tests written but the components included may not have a comprehensive list of tests available. However, these are just minor extensions, the suggested extension is rather more advanced but would dramatically increase the systems usefulness.

At the current time the GUI marker only has the capability to mark Java SWING or AWT interfaces due to the fact that this is what the students would be learning at the University of Nottingham. However, the marking system was designed with flexibility in mind. It is this flexibility that could be harnessed to expand the available test sets and allow for the marking of Eclipse based interfaces. SWT and JFaces are becoming more common in commercial interfaces due to their rendering of the interfaces at a native level again improving the portability of the Java language. Before long it will be necessary to include them in the curriculum meaning that new exercises will need to be marked. The descent parser attempts to identify each object it finds within the GUI, this can easily be extended by listing the new components that the parser should be looking for. In keeping with the abstraction previously mentioned, SWT components can readily be recognised by their inheritance from the Widget class etc.

The flexibility that would be utilised for this extension allows the GUI marking system to keep up to date with advances in the creation of Graphical User Interfaces.

8.5 Epilogue

This thesis has shown as originally stated that the creation of an automated system to test and mark Graphical User Interfaces is not only feasible but the resulting system is both efficient and effective. The lack of prior research in this subject area is down to the inherent complexities that are involved in the creation of GUIs. However, these are only minor problems and the results demonstrate that they can be overcome.

A novel and inventive solution has been presented to provide readers with the methodology needed to create a system capable of marking Graphical User Interfaces. The framework solution can either be used as a standalone system or be incorporated into currently existing software with equal success. The automated marking of Graphical User Interfaces is an advancement for automated assessment systems with uses within the software industry.

The problem was solved with the use of introspective approaches made possible through the dynamic loading of the student's solutions at run time. This along with the descent parsing of the interface provides the user with an extensible framework with which tests can be created for all Java interface components in any API of their choice.

The implementation of the outlined solution has been of great benefit to the University of Nottingham Computer Science department. It has reduced the number of demonstrators required to assist students and perform marking duties and has increased the time that lectures have available to put back into the course. Benefits which enable more to be returned to the students are always invaluable in education.

Appendix A

Example Test Oracle

The following in an example of an XML test oracle as would be used when marking a GUI solution.

```
<GUI_TESTS>
  <TEST_SET DESCRIPTION=''Simple Addition'' MARK=''100''>
    <TEST>
      <TYPE>BUTTON_CLICK</TYPE>
      <TARGET_TEXT>1</TARGET_TEXT>
    </TEST>
    <TEST>
      <TYPE>BUTTON_CLICK</TYPE>
      <TARGET_TEXT>[\+]</TARGET_TEXT>
    </TEST>
    <TEST>
      <TYPE>BUTTON_CLICK</TYPE>
      <TARGET_TEXT>2</TARGET_TEXT>
    </TEST>
    <TEST>
      <TYPE>BUTTON_CLICK</TYPE>
      <TARGET_TEXT>[\=]</TARGET_TEXT>
    </TEST>
    <TEST>
```

```
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\+]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>3</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\=]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\+]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>4</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\=]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\+]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>5</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\=]</TARGET_TEXT>
```

```
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\+]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON CLICK</TYPE>
    <TARGET_TEXT>6</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\=]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\+]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>7</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\=]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\+]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>8</TARGET_TEXT>
</TEST>
<TEST>
```

```
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\=]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\+]</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>9</TARGET_TEXT>
</TEST>
<TEST>
    <TYPE>BUTTON_CLICK</TYPE>
    <TARGET_TEXT>[\=]</TARGET_TEXT>
</TEST>
<RESULT>
    <TYPE>TEXT_TEXT</TYPE>
    <TARGET_NUMBER>1</TARGET_NUMBER>
    <ORACLE>45</ORACLE>
    <ORACLE_TYPE></ORACLE_TYPE>
    <FEEDBACK>
<BAD>WRONG WRONG WRONG</BAD>
<GOOD>Correct Answer</GOOD>
    </FEEDBACK>
    <MARK>50</MARK>
    <DESCRIPTION>Test for Addition</DESCRIPTION>
</RESULT>
  </TEST_SET>
  <TEST_SET DESCRIPTION="Feature Test" MARK="100">
<RESULT>
    <TYPE>BUTTON_OCCURS</TYPE>
    <TARGET_TEXT>4</TARGET_TEXT>
    <ORACLE>1</ORACLE>
    <ORACLE_TYPE></ORACLE_TYPE>
```

```
    <FEEDBACK>
<BAD>Wrong number of 4 buttons found</BAD>
<GOOD>Correct the 4 button only appears once</GOOD>
    </FEEDBACK>
    <MARK>50</MARK>
    <DESCRIPTION>Checking buttons labelled 4</DESCRIPTION>
</RESULT>
  </TEST_SET>
</GUI_TESTS>
```

Appendix B

Example Question Specification

A more basic exercise that the students would be given to complete and would be marked by the system.

CourseMarker Calculator: Calculator (weight 5)

==========

Problem Specification

==========

As in dge's notes. You are to fix the broken code provided in the skeleton solution, not that as shown in the notes, they are subtlely different.

Clarifications

==========

You must make sure you do the following inorder to get full marks.

Use a gridlayout (not gridbaglayout or any other) to layout the buttons.

The multiply button should be marked with the letter 'x' not the '*' symbol.

You are to use action commands for each button, the command should be the same as the location in the grid e.g. 7 would be "00", and 5 would be "11".

The JTextField should not be editable.

2 * 2 = * 2 = will equal eight where as 2 * 2 = 2 * 2 = will equal 4

Errors

======

If you try and divide by zero the answer field should say

overflow

Notes

=====

You will have three (3) Submissions

Marking Scheme

==========

20

20

60

Input / Output

==========

No typical input or output

Bibliography

[AB99a] D. Arnow and O. Barshay. On-line programming examinations using web to teach. *Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and technology in computer science education*, pages 21–24, June 27 - 30 1999.

[AB99b] D. Arnow and O. Barshay. Webtoteach: An interactive focused programming exercise system. *ASEE/IEEE Frontiers in Education Conference*, November 1999.

[AEH05] M. Ahmadzadeh, D. Elliman, and C. Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, New York, NY, USA, 2005. ACM Press.

[Alb75] J. Albers. *Interaction of Color*. Yale University, 1975.

[AMUJ04] K. Ala-Mutka, T. Uimonen, and H-M. Järvinen. Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 2004.

[AT95] G. Åkerlind and C. Trevitt. Enhancing learning through technology: When students resist the change. *ASCILITE 95 - Learning with Technology*, December 3-7 1995.

[Bal06] H. Balinsky. Evaluating interface aesthetics: a measure of symmetry. Technical report, Media Technology Laboratory, HP Laboratories Bristol, 2006.

[BBF+94] S. Benford, E. Burke, E. Foxley, N. Gutteridge, C. Higgins, and A. Mohd Zin. Software support for automated assessment and administration. *The Journal of Research on Computing in Education*, 1994.

[BBF+95] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. Mohd Zin. The design document for ceilidh version 2. Ltr report, Computer Science Department, The University of Nottingham, 1995.

[Bel01] F. Belli. Finite state testing and analysis of graphical user interfaces. *ISSRE 2001*, 2001.

[Bla] Blackboard. http://www.blackboard.com.

[Bon99] A. Bone. *Guidance Notes: Ensuring Successful Assessment*. Warwick Printing Company Limited, 1999.

[BRRP05] Jean Berstel, Stefano Crespi Reghizzi, Gilles Roussel, and Pierluigi San Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Trans. Softw. Eng. Methodol.*, 14(2):124–167, 2005.

[BRS96] S. Brown, P. Race, and Br Smith. *500 Tips on Assessment.* Kogan Page, 1996.

[BRSV83] W. Bewley, T. Roberts, D. Schroit, and W. Verplank. Human factors testing in the design of xerox's 8010 'star' office workstation. In *Proceedings of ACM CHI'83 Conference on Human Factors in Computing Systems*, 1983.

[Bul99a] J. Bull. Computer-assisted assessment: impact on higher education institutions. *Educational Technology and Society*, 2(3), 1999.

[Bul99b] J. Bull. Update on the national tltp3 project: The implementation and evaluation of computer-assisted assessment. In M. Danson and R. Sherrat, editors, *Proceedings of the 3rd Annual CAA Conference*, pages 11–17, Loughborough, UK, 1999. Loughborough University.

[BW98] P. Black and D. Wiliam. Inside the black box: Raising standards through classroom assessment. Technical report, Kings College, London, 1998.

[Cam] Virtual Campus. http://www.teknical.com/products/virtual_campus.htm.

[CE98a] D. Charman and A. Elmes. Computer based assessment: A guide to good practice. *Volume I*, 1998.

[CE98b] D. Charman and A. Elmes. Computer based assessment: A guide to good practice. *Volume 2*, 1998.

[Chi05] P. Chin. Virtual learning environments, advice on choosing a vle. Technical report, Physical Sciences Centre, The Higher Education Academy, 2005.

[CKW98] T.R. Chuang, Y.S. Kuo, and C.M. Wang. Non intrusive object introspection in c++: Architecture and application. *20th International Conference on Software Engineering*, April 1998.

[Coo01] P. Cookson. Editorial: Global diversity of distance education. *nternational Review of Research in Open and Distance Learning*, 2(1), July 2001.

[CS98] M. Canup and R. Shackelford. Using software to solve problems in large computing courses. *SIGSCE 98*, 1998.

[Dal99] C. Daly. Roboprof and an introductory computer programming course. *Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and Technology in Computer Science Education*, pages 155–158, June 27-30 1999.

[DH95] K. Dawson-Howe. Automatic submission and administration of programming assignments. *SIGCSE Bull.*, 27(4):51–53, 1995.

[Dil00] P. Dillenbourg. Virtual learning environments. In *Proceedings Virtual Learning Environments. EUN Conference 2000*, 2000.

[dJ05] P. de Jager. Teaching old dogs new tricks. *Computerworld Canada*, January 24 2005.

[DLO+05] C. Douce, D. Livingstone, J. Orwell, S. Grindle, and J. Cobb. A technical perspective on asap - automated system for assessment of programming. In *The Proceedings of 9th CAA conference*, July 2005.

[DSG03] F. Doucet, S. Shukla, and R. Gupta. Introspection in system-level language frameworks: Meta-level vs. integrated. *Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[Emi01] K. Emigh. The impact of new programming languages on university curriculum. In *The Proceedings of ISECON 2001*, volume 18, pages 193–198, 2001.

[Eng04] J. English. Teaching and assessing gui-based programming with jewl. In *The Proceedings of 5th LTSN-ICS conference*, August 2004.

[ES05] J. Philip East and J. Ben Schafer. In-person grading: an evaluative experiment. *SIGCSE Bull.*, 37(1):378–382, 2005.

[ET02] D. Emory and R. Tamassia. Jerpa: a distance-learning environment for introductory java programming courses. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 307–311, New York, NY, USA, 2002. ACM Press.

[FBC+01] A. J. Fernndez, M. Belmonte, C. Cotta, I. Gmez, J.A.Pedreira J.L. Pastrana, F.Rus, J. Snchez, and E. Soler. Foundations of programming: a teaching improvement. *Computers in Education: Towards an Interconnected Society*, pages 81–92, 2001.

[FHG96] E. Foxley, C. Higgins, and C. Gibbon. The ceilidh system : A general overview. Ltr report, Computer Science Department, The University of Nottingham, 1996.

[FHH+01] E. Foxley, C. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas. The coursemaster cba system: Improvements over ceilidh. *Fifth International Computer Assisted Assessment Conference*, pages 189–201, July 2-4 2001.

[FHST01] E. Foxley, C. Higgins, P. Symeonidis, and A. Tsintsifas. The coursemaster automated assessment system - a next generation ceilidh. *Workshop on Computer Assisted Assessment to support the ICS disciplines*, April 5-6 2001.

[FHTS00] E. Foxley, C. Higgins, A. Tsintsifas, and P. Symeonidis. The ceilidh-coursemaster system, an introduction. *4th Java in the Curriculum Conference*, Jan 2000.

[FTHS99] E. Foxley, A. Tsintsifas, C. Higgins, and P. Symeonidis. Ceilidh, a system for the automatic evaluation of students programming work. *CBLISS 99*, July 2-7 1999.

[FW65] G. E. Forsythe and N. Wirth. Automatic grading programs. *Commun. ACM*, 8(5):275–278, 1965.

[GG06] P. Guerrerio and K. Georgouli. Combating anonymousness in populous cs1 and cs2 courses. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, pages 8–12, New York, NY, USA, 2006. ACM Press.

[GH06] G. Gray and C. Higgins. An introspective approach to marking graphical user interfaces. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, pages 43–47, New York, NY, USA, 2006. ACM Press.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, first edition, 1995.

[Gib95] C. Gibbon. Writing typographically sound programs with ceilidh. Technical report, The University of Nottingham, 1995.

[Gre96] S. Greenberg. Teaching human computer interaction to programmers. *interactions*, 3(4):62–76, 1996.

[GS] L. Gong and R. Schemers. Implementing protection domains in the java™ development kit 1.2. Technical report, JavaSoft.

[Hat99] J. Hattie. Influences on student learning. Inaugural professorial lecture, University of Auckland, 1999.

[HB06] C. Higgins and B. Bligh. Formative computer based assessment in diagram based domains. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, pages 98–102, New York, NY, USA, 2006. ACM Press.

[HCC+98] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in java. *USENIX Annual Technical Conference*, June 1998.

[Her78] E. Hering. *Zur Lehre vom Lichtsinn*. Oxford U.P, 1878.

[HGST06] C. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas. Automated assessment and experiences of teaching programming. *JERIC, Journal on Educational Resources in Computing: Automated Assessment of Programming Assignments*, 2006.

[HKHRT83] R.W. Hamm, Jr. K.D. Henderson, M.L. Repsher, and K.M. Timmer. A tool for program grading: The jacksonville university scale. In *SIGCSE '83: Proceedings of the fourteenth SIGCSE technical symposium on Computer science education*, pages 248–252, New York, NY, USA, 1983. ACM Press.

[Hol60] J. Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 10(3):528529, 1960.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[How94] J.W. Howatt. On criteria for grading student programs. *SIGCSE Bull.*, 26(3):3–7, 1994.

[How95] J. Howatt. A project-based approach to programming language evaluation. *SIGPLAN Not.*, 30(7):37–40, 1995.

[IH01] M.Y. Ivory and M.A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4):470–516, December 2001.

[Itt03] J. Itten. *The Elements of Color*. John Wiley & Sons, inc., 2003.

[Jac00] D. Jackson. A semi-automated approach to online assessment. In *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education*, pages 164–167, New York, NY, USA, 2000. ACM Press.

[JCL00] M. Joy, P.S. Chan, and M. Luck. Networked submission and assessment. In *Proceedings of the 8th Annual Conference on the Teaching of Computing*, 2000.

[Jen98] T. Jenkins. A participative approach to teaching programming. In *ITiCSE '98: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, pages 125–129, New York, NY, USA, 1998. ACM Press.

[Jen02] T. Jenkins. On the difficulty of learning to progam. *3rd annual LTSN-ICE Conference*, August 27-29 2002.

[JIS05] JISC. Infonet http://www.jiscinfonet.ac.uk. Technical report, UK National Centre of Expertise in the Planning and Implementation of Information Systems, 2005.

[JL98a] M.S. Joy and M. Luck. The boss system for on-line submission and assessment. *Monitor: Journal of the CTI Centre for Computing*, pages 27–29, 1998.

[JL98b] M.S. Joy and M. Luck. Effective electronic marking for on-line assessment. In *ITiCSE '98: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, pages 134–138, New York, NY, USA, 1998. ACM Press.

[Joh00] J. Johnson. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann, 2000.

[JU] Junit testing framework documentation available at http://www.junit.org.

[KF05] J. Kelly and G. Ferrell. Effective management of virtual learning environments. In *Proceedings of EUNIS 2005 - Leadership and Strategy in a Cyber-Infrastructure World*, 2005.

[KKR95] M. Kølling, B. Koch, and J. Rosenberg. Requirements for a first year object-oriented teaching language. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 173–177, New York, NY, USA, 1995. ACM Press.

[KMS03] A. Korhonen, L. Malmi, and P. Silvasti. Trakla2: a framework for automatically assessed visual simulation exercises. In *Proceedings of the Third Finnish/Baltic Sea Conference on Computer Science Education*, pages 48–56, 2003.

[KN81] D.L. Kalmey and M.J. Niccolai. A model for a cai learning system. In *SIGCSE '81: Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, pages 74–77, New York, NY, USA, 1981. ACM Press.

[LB98] S. Liang and G. Bracha. Dynamic class loading in the java™virtual machine. *OOPSLA*, 1998.

[Leu99] J.H. Leuthold. Is computer-based learning right for everyone? In *HICSS '99: Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences-Volume 1*, page 1015, Washington, DC, USA, 1999. IEEE Computer Society.

[LFN04] S. Lok, S. Feiner, and G. Ngai. Evaluation of visual balance for automated layout. *IUI04*, January 13-16 2004.

[LJ99] M. Luck and M.S. Joy. A secure on-line submission system. *Software - Practice and Experience*, pages 721–740, 1999.

[LM98] J. Leal and N. Moreira. Automatic grading of programming exercises. Technical Report DCC-98-4, DCC-FC and LIACC, Universidade do Porto, June 1998.

[LM00] J. Leal and N. Moreira. Using matching for automatic assessment in computer science learning environments. In *Proceedings of Web-based Learning Environments Conference, 2000*, 2000.

[LSKM04] M.-J. Laakso, T. Salakoski, A. Korhonen, and L. Malmi. Automatic assessment of exercises for algorithms and data structures - a case study with trakla2. In *Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education*, pages 28–36, October 2004.

[LTH+99] R.G. Sutcliffeand E.M. Leonard, A. Tierney, C.W. Howe, I. Reid, and S.T. Goodwinand D.M. Mackenzie. ntroduction of a range of computer-based objective tests in the examination of genetics in first year biology. In *Proceedings of the Third Annual Computer-assisted Assessment Conference*, pages 193–206, 1999.

[Mem01] A.M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.d. thesis, Faculty of Arts and Sciences, University of Pittsburgh, 2001.

[MHC96] B. Myers, J. Hollan, and I. Cruz. Strategic directions in human-computer interaction. *ACM Comput. Surv.*, 28(4):794–809, 1996.

[Mic96] G. Michaelson. Automatic analysis of functional program style. In *ASWEC '96: Proceedings of the 1996 Australian Software Engineering Conference (ASWEC '96)*, page 38, Washington, DC, USA, 1996. IEEE Computer Society.

[Mic04] Sun Microsystems. Java robot class documentation is available at http://java.sun.com/j2se/1.5.0/docs/api/java/awt/robot.html, 2004.

[Mic05] Sun Microsystems. Java programming language specification. Technical report, 2005.

[Mit97] D. Mitchell. Making sense of computer aided learning research: A critique of the pseudo-scientific method. In *CAL '97: Proceedings of the Computer Assisted Learning conference*, 1997.

[MMM04] M. McCulloch, H. Macleod, and N. Mogey. Assessing online - an overview. *Reflections on Assessment*, 2, 2004.

[MPS01] A.M. Memon, M.E. Pollack, and M.L. Soffa. Coverage criteria for gui testing. *ESEC/FSE'01*, pages 256–267, 2001.

[MS97] R. Mahajan and B. Shneiderman. Visual and textual consistency checking tools for graphical user interfaces. *IEEE Trans. Softw. Eng.*, 23(11):722–735, 1997.

[Mye93] B.A. Myers. Why are human-computer interfaces difficult to implement? Technical Report CMU-CS-93-183, Carnegie Mellon University, July 1993.

[Mye94] B. Myers. Challenges of hci design and implementation. *interactions*, 1(1):73–83, 1994.

[Mye95] Brad A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, 1995.

[Nie01] J. Nielsen. First rule of usability? don't listen to users available at http://www.useit.com/alertbox/20010805.html, 2001.

[NM90] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. 1990.

[NP04] K.L. Norman and E. Panizzi. Levels of automation and user participation in usability testing. Technical Report HCIL-2004-17, CS-TR-4657, University of Maryland, USA and Universita di Roma, Italy, May 2004.

[NSA00] D.C.L. Ngo, A. Samsudin, and R. Abdullah. Aesthetic measures for assessing graphic screens. *Journal of Information Science and Engineering*, 16:97–116, 2000.

[NTB03] D.C.L. Ngo, L.S. Teo, and J.G. Byrne. Modelling interface aesthetics. *Journal of Information Sciences*, 152:25–46, 2003.

[Oak98] S. Oaks. *Java Security*. O'Reilly, 1998. Chapter 1.2.

[OHZ01] E. Odekirk-Hash and J.L. Zachary. Automated feedback on programs means students need less help from teachers. *SIGCSE Bull.*, 33(1):55–59, 2001.

[OK05] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[O'L99] R. O'Leary. An introduction to computer assisted assessment. *Computer Assisted Assessment*, 1999.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[PA06] B. Plimmer and R. Amor. Peer teaching extends hci learning. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, pages 53–57, New York, NY, USA, 2006. ACM Press.

[Pap04] M.-C. Papaefthimiou. Introduction to learning technology. Technical report, Centre for the Development of Teaching and Learning, 2004.

[PP97] B. Price and M. Petre. Teaching programming through paperless assignments: an empirical evaluation of instructor feedback. *ITiCSE 97*, 1997.

[PPP06] R. Pecinovský, J Pavlíčková, and L. Pavlíček. Let's modify the objects-first approach into design-patterns-first. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, pages 188–192, New York, NY, USA, 2006. ACM Press.

[Pro] NetBeans Project. Jemmy gui testing library, documentation is available at http://jemmy.netbeans.org.

[PS98] J. Preston and R. Shackelford. A system for improving distance and large-scale classes. In *ITiCSE '98: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, pages 193–198, New York, NY, USA, 1998. ACM Press.

[QFS] qftestjui, the java gui testtool.

[RM98] R. Reis and N. Moreira. **Apoo**: an environment for a firts course in assembly language programming. Technical Report DCC-98-9, DCC-FC and LIACC, Universidade do Porto, 1998.

[Rog03] G. Rogers. Do grades make the grade for program assessment? assessment tips with gloria rogers. *Assessment Tips is a quarterly column, exclusive to Communications Link, ABET, Inc.*, 2003.

[RSP95] A. Rose, B. Shneiderman, and C. Plaisant. An applied ethnographic method for redesigning user interfaces. In *DIS '95: Proceedings of the conference on Designing interactive systems*, pages 115–122, New York, NY, USA, 1995. ACM Press.

[SC04] D. Stephens and A. Curtis. Use of computer assisted assessment by staff in the teaching of information science and library studies subjects. *ITALICS, Innovations in Teaching And Learning in Information and Computer Sciences*, 1(1), 2004.

[SCFP00] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–167, New York, NY, USA, 2000. ACM Press.

[SCKH86] D. Smith, C.Irby, R. Kimball, and E. Harslem. The star user interface: an overview. In *on AFIPS Conference Proceedings; vol. 55 1986 National Computer Conference*, pages 383–396, Arlington, VA, USA, 1986. AFIPS Press.

[Sea93] A. Sears. Layout appropriateness: A metric for evaluating user interface widget layout. *IEEE Trans. Softw. Eng.*, 19(7):707–719, 1993.

[Shn80] B. Shneiderman. *Software psychology: human factors in computer and information systems*. Winthrop Publishers, 1980.

[Sil03] M. Silverstein. Logical capture/replay. *Software Testing and Quality Engineering Magazine*, November/December 2003.

[SM86] S.L. Smith and J.N. Mosier. Guidelines for designing user interface software. Technical Report ESD-TR-86-278, The MITRE Corporation, 1986.

[Sma02] J. Smailes. Experiences of using computer aided assessment within a virtual learning environment. In *BEST 02: Supporting the Teacher: Challenging the Learner*, 2002.

[Smi82] S.L. Smith. User-system interface design for computer-based information systems. Technical Report ESD-TR-82-132, USAF Electronic Systems Division, April 1982.

[SMK01] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 133–136, New York, NY, USA, 2001. ACM Press.

[SP94] Ben Shneiderman and Catherine Plaisant. The future of graphic user interfaces: personal role managers. In *HCI '94: Proceedings of the conference on People and computers IX*, pages 3–8, New York, NY, USA, 1994. Cambridge University Press.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, third edition, 2000.

[SW06] D.E. Stevenson and P.J. Wagner. Developing real-world programming assignments for cs1. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, pages 158–162, New York, NY, USA, 2006. ACM Press.

[Sym98] P. Symeonidis. Creating an exercise using coursemarker (formerly javaceilidh). Technical report, LTR Group, 1998.

[Sym01] P. Symeonidis. An in-depth review of coursemaster marking subsystem. Technical report, LTR Group, 2001.

[Tsi02] A. Tsintsifas. *A Framework for the Computer Based Assessment of Diagram Based Coursework*. Ph.d. thesis, Computer Science Department, University of Nottingham, 2002.

[uvM94] urs von Matt. Kassandra: the automatic grading system. *SIGCUE Outlook*, 22(1):26–40, 1994.

[vG10] J.W. von Goethe. *Zur Farbenlehre*. The MIT Press, 1810.

[VH03] A. Venables and L. Haywood. Programming students need instant feedback! In *CRPITS '20: Proceedings of the fifth Australasian conference on Computing education*, pages 267–272, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[Web] WebCT. http://www.webct.com.

[WP05] Titus Winters and Tom Payne. What do students know?: an outcomes-based assessment system. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 165–172, New York, NY, USA, 2005. ACM Press.

[WP06] T. Winters and T. Payne. Computer aided grading with agar. *Frontiers in Education CS*, 2006.

[YJ04] J. Yau and M. Joy. Introducing java: A case for fundamentals-first. In *EISTA 2004: Proceedings of the International Conference on Education and Information Systems, Technologies and Applications*, pages 229–234, 2004.

[Zel93] L. Zelmer. *The Impact of the Introduction of Computers into the Faculty of Health Science: A case study of organisational change*. Ph.d. thesis, Department of Education, University of Queensland, 1993.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy, 1997.