

# **A Web-Based Programming Environment for Novice Programmers**

Nghi Truong

Bachelor of Information Technology  
(Queensland University of Technology, Australia)

A dissertation submitted in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

**School of Software Engineering and Data Communications  
Faculty of Information Technology  
Queensland University of Technology**

**July 2007**

## **Keywords**

Computer programming, flexible delivery, web, tutoring system, on-line learning, feedback, fill in the gap, static analysis, cyclomatic complexity, automated testing, dynamic analysis, black box, white box, feedback, XML, assessment



## **Abstract**

Learning to program is acknowledged to be difficult; programming is a complex intellectual activity and cannot be learnt without practice. Research has shown that first year IT students presently struggle with setting up compilers, learning how to use a programming editor and understanding abstract programming concepts. Large introductory class sizes pose a great challenge for instructors in providing timely, individualised feedback and guidance for students when they do their practice.

This research investigates the problems and identifies solutions. An interactive and constructive web-based programming environment is designed to help beginning students learn to program in high-level, object-oriented programming languages such as Java and C#. The environment eliminates common starting hurdles for novice programmers and gives them the opportunity to successfully produce working programs at the earliest stage of their study. The environment allows students to undertake programming exercises anytime, anywhere, by “filling in the gaps” of a partial computer program presented in a web page, and enables them to receive guidance in getting their programs to compile and run. Feedback on quality and correctness is provided through a program analysis framework. Students learn by doing, receiving feedback and reflecting - all through the web.

A key novel aspect of the environment is its capability in supporting small “fill in the gap” programming exercises. This type of exercise places a stronger emphasis on developing students’ reading and code comprehension skills than the traditional approach of writing a complete program from scratch. It allows students to concentrate on critical dimensions of the problem to be solved and reduces the complexity of writing programs.



# Contents

<b>KEYWORDS .....</b>	<b>I</b>
<b>ABSTRACT .....</b>	<b>III</b>
<b>CONTENTS.....</b>	<b>V</b>
<b>LIST OF FIGURES .....</b>	<b>IX</b>
<b>LIST OF TABLES .....</b>	<b>XI</b>
<b>LIST OF ABBREVIATIONS.....</b>	<b>XIII</b>
<b>STATEMENT OF ORIGINAL AUTHORSHIP .....</b>	<b>XV</b>
<b>PREVIOUSLY PUBLISHED MATERIAL.....</b>	<b>XVII</b>
<b>ACKNOWLEDGEMENTS.....</b>	<b>XIX</b>
<b>CHAPTER 1 - INTRODUCTION.....</b>	<b>1</b>
1.1 AIMS AND OBJECTIVES .....	2
1.2 THESIS CONTRIBUTIONS .....	4
1.3 OUTLINE OF THESIS .....	9
<b>CHAPTER 2 - LITERATURE REVIEW.....</b>	<b>13</b>
2.1 INTRODUCTION.....	13
2.2 EDUCATION BACKGROUND .....	14
2.2.1 <i>Scaffolding</i> .....	14
2.2.2 <i>Bloom's Taxonomy</i> .....	16
2.2.3 <i>Constructivism</i> .....	17
2.3 OVERVIEW OF APPLICATIONS IN COMPUTER SCIENCE EDUCATION.....	20
2.3.1 <i>Computer Assisted Instruction</i> .....	20
2.3.1.1 Intelligent Applications.....	23
2.3.1.2 Non-Artificial Intelligent Applications .....	25
2.3.2 <i>Computer Assisted Assessment</i> .....	29
2.4 FUTURE DIRECTION OF APPLICATIONS IN COMPUTER SCIENCE EDUCATION .....	34
2.5 WEB-BASED PROGRAMMING ENVIRONMENTS .....	36
2.5.1 <i>CodeSaw</i> .....	36
2.5.2 <i>CourseMaster</i> .....	37
2.5.3 <i>WebToTeach</i> .....	38
2.5.4 <i>CodeLab</i> .....	39
2.5.5 <i>InSTEP</i> .....	39
2.5.6 <i>WWW for Learning C++</i> .....	40
2.5.7 <i>W4AP</i> .....	41
2.5.8 <i>JERPA</i> .....	41
2.5.9 <i>VECR</i> .....	42
2.5.10 <i>Ludwig</i> .....	42
2.5.11 <i>ALECS</i> .....	43
2.6 PROGRAM ANALYSIS AND AUTOMATED MARKING SYSTEMS .....	43
2.6.1 <i>ITPAD</i> .....	43
2.6.2 <i>Expresso</i> .....	44
2.6.3 <i>TRY</i> .....	45
2.6.4 <i>BOSS</i> .....	46
2.6.5 <i>Datlab</i> .....	46
2.6.6 <i>ASSYST</i> .....	47
2.6.7 <i>Web-CAT</i> .....	47
2.6.8 <i>GAME: A Generic Automated Marking Environment</i> .....	48
2.6.9 <i>HomeWork Generation and Grading Project</i> .....	49
2.6.10 <i>PASS</i> .....	49
2.6.11 <i>IRONCODE</i> .....	50
2.6.12 <i>Online Judge</i> .....	50

2.7 DISCUSSION AND SUMMARY .....	51
<b>CHAPTER 3 - THE ENVIRONMENT FOR LEARNING TO PROGRAM (ELP).....</b>	<b>57</b>
3.1 INTRODUCTION.....	57
3.2 THE DIFFICULTIES ENCOUNTERED BY NOVICE PROGRAMMERS .....	59
3.2.1 <i>The Programming Task</i> .....	59
3.2.2 <i>The Difficulties of Learning Programming</i> .....	60
3.3 THE KEY CHARACTERISTICS OF THE ELP .....	66
3.3.1 “ <i>Fill in the Gap</i> ” Programming Exercises.....	66
3.3.2 <i>Web-Based</i> .....	70
3.3.3 <i>Provide Timely and Frequent Feedback</i> .....	71
3.3.4 <i>Exercise-Centric</i> .....	72
3.4 THE ARCHITECTURE OF THE ELP .....	73
3.4.1 <i>System Overview</i> .....	73
3.4.2 <i>Student View</i> .....	75
3.4.3 <i>Lecturer View</i> .....	78
3.4.4 <i>The Exercise Representation</i> .....	78
3.4.5 <i>Functionalities of ELP</i> .....	79
3.5 THE IMPLEMENTATION .....	80
3.5.1 <i>The Exercise XML Format</i> .....	82
3.5.2 <i>Database Structure</i> .....	85
3.6 SUMMARY .....	87
<b>CHAPTER 4 - STATIC ANALYSIS.....</b>	<b>89</b>
4.1 INTRODUCTION.....	89
4.2 BACKGROUND .....	92
4.2.1 <i>Overview of Static Analysis Approaches</i> .....	92
4.2.2 <i>Software Metrics</i> .....	94
4.2.2.1 Software Complexity Metrics .....	95
4.3 NOVICE PROGRAMMERS’ COMMON MISTAKES .....	98
4.3.1 <i>Semantic and Conceptual Mistakes</i> .....	99
4.3.2 <i>Programming Practice</i> .....	99
4.3.3. <i>QUT Students’ Common Mistakes</i> .....	100
4.4 GAP CATEGORIES .....	101
4.5 THE DESIGN .....	104
4.5.1 <i>Overview</i> .....	104
4.5.2 <i>Software Engineering Metrics Analyses</i> .....	108
4.5.3 <i>Structural Similarity Analysis</i> .....	115
4.5.3.1 Challenges in Identifying Differences between Two Programs .....	115
4.5.3.2 The analysis design.....	117
4.6 AUTHOR CONFIGURATION.....	124
4.7 THE IMPLEMENTATION .....	126
4.7.1 <i>Software Engineering Metrics Analysis</i> .....	131
4.7.2 <i>Structural Similarity Analysis</i> .....	135
4.7.3 <i>Database Structure</i> .....	136
4.8 SUMMARY .....	137
<b>CHAPTER 5 - DYNAMIC ANALYSIS .....</b>	<b>139</b>
5.1 INTRODUCTION.....	139
5.2 OVERVIEW OF SOFTWARE TESTING .....	142
5.2.1 <i>Black Box Testing</i> .....	142
5.2.2 <i>White Box Testing</i> .....	143
5.3 NOVICE PROGRAMMER EXERCISE CATEGORIES .....	145
5.4 THE DESIGN .....	146
5.4.1 <i>Overview</i> .....	146
5.4.2 <i>Black Box Testing</i> .....	149
5.4.3 <i>White Box Testing</i> .....	157
5.5 AUTHOR CONFIGURATION.....	164
5.6 THE IMPLEMENTATION .....	165
5.6.1 <i>Test Driver Generation Process</i> .....	167
5.6.2 <i>Black Box Testing</i> .....	170

<i>5.6.3 White Box Testing</i> .....	172
<i>5.6.4 Database Structure</i> .....	173
5.7 SUMMARY .....	175
<b>CHAPTER 6 - ELP FEEDBACK .....</b>	<b>177</b>
6.1 INTRODUCTION.....	177
6.2 FEEDBACK IN COMPUTER-BASED INSTRUCTION.....	179
6.3 FEEDBACK FOR STUDENTS AND STAFF.....	181
<i>6.3.1 Feedback for Students</i> .....	181
<i>6.3.2 Feedback for Teaching Staff</i> .....	191
6.4 THE IMPLEMENTATION FOR FEEDBACK GENERATION .....	192
<i>6.4.1 Feedback for Students</i> .....	193
<i>6.4.2 Providing Feedback for Teaching Staff</i> .....	197
6.5 SUMMARY .....	197
<b>CHAPTER 7 - ELP EVALUATIONS .....</b>	<b>199</b>
7.1 INTRODUCTION.....	199
7.2 OBJECTIVES OF THE EVALUATIONS .....	200
7.3 EVALUATIONS AND RESULTS .....	202
<i>7.3.1 Quantitative Evaluations among Students</i> .....	202
<i>7.3.1.1 Stage 1 – ELP Only</i> .....	203
<i>7.3.1.2 Stage 2 – ELP System and Static Analysis</i> .....	205
<i>7.3.1.3 Third Evaluation – ELP System, Static and Dynamic Analyses</i> .....	207
<i>7.3.2 Qualitative Evaluations among Students</i> .....	221
<i>7.3.3 Qualitative Evaluations among Staff</i> .....	224
7.3.4 DISCUSSION OF RESULTS AND SUMMARY .....	227
<b>CHAPTER 8 - CONCLUSIONS AND FURTHER WORK.....</b>	<b>231</b>
8.1 SUMMARY OF CONTRIBUTIONS .....	231
<i>8.1.1 The Environment for Learning to Program</i> .....	233
<i>8.1.2 The Static Analysis</i> .....	234
<i>8.1.3 The Dynamic Analysis</i> .....	235
<i>8.1.4 The Feedback Engine</i> .....	235
8.2 FURTHER WORK.....	236
<i>8.2.1 The ELP System</i> .....	236
<i>8.2.1.1 Collaborative Learning</i> .....	237
<i>8.2.1.2 Adaptive Exercises</i> .....	238
<i>8.2.2 The Static Analysis</i> .....	238
<i>8.2.3 The Dynamic Analysis</i> .....	239
<i>8.2.3.1 Testing File Input Output Programming Exercises</i> .....	239
<i>8.2.3.2 Testing Graphical User Interface Exercises</i> .....	240
<i>8.2.4 Improvement in Feedback</i> .....	241
<b>APPENDIX A .....</b>	<b>243</b>
A.1 ELP JAVA EXERCISES.....	243
<i>HelloMe</i> .....	243
<i>Results</i> .....	244
<i>Root</i> .....	245
<i>Coin</i> .....	246
A.2 ELP C# EXERCISES .....	249
<i>SumTwo</i> .....	249
<i>RomanToArabic</i> .....	250
<i>CircleArea2</i> .....	251
<i>Creating and Using Money Class</i> .....	252
<b>APPENDIX B .....</b>	<b>255</b>
B.1 XML SERIALIZED FORMAT OF PRIMITIVE, ARRAY AND USER DEFINED TYPES .....	255
<b>REFERENCES .....</b>	<b>257</b>



# List of Figures

FIGURE 1: NEW TEACHING AND LEARNING STYLE WITH ELP .....	6
FIGURE 2: THE EVOLUTION OF THE ELP SYSTEM AND THE PROGRAM ANALYSIS FRAMEWORK.....	9
FIGURE 3: BLOOM'S TAXONOMY LEVELS .....	16
FIGURE 4: DYNAMIC LEARNING ENVIRONMENT .....	20
FIGURE 5 NOVICE PROGRAMMING SYSTEMS AND LANGUAGE TAXONOMY .....	22
FIGURE 6: A PROGRAMMING FRAMEWORK.....	60
FIGURE 7: AN OVERVIEW OF THE ELP SYSTEM .....	74
FIGURE 8: EXAMPLE OF HTML SOURCE TO INCLUDE ELP APPLET .....	75
FIGURE 9: THE INTEGRATION OF ELP INTO OLT .....	76
FIGURE 10: STUDENT VIEW OF THE ELP SYSTEM .....	77
FIGURE 11: THE CONSTRUCTION OF VARIOUS EXERCISE VIEWS .....	79
FIGURE 12: HIGH-LEVEL IMPLEMENTATION OF THE ELP SYSTEM.....	81
FIGURE 13: FILES THAT MAKE UP AN ELP EXERCISE .....	83
FIGURE 14: DETAILED PROCESS OF GENERATING EXERCISE SOLUTION .....	84
FIGURE 15: THE DATABASE RELATIONSHIP AMONG TABLES IN THE ELP SYSTEM .....	86
FIGURE 16: EXAMPLE OF ILL-FORMED GAP .....	102
FIGURE 17: AN EXAMPLE OF WELL-FORMED GAPS.....	103
FIGURE 18: EXAMPLES OF GAP TYPES .....	103
FIGURE 19: VARIABLE SCOPES IN A PROGRAM .....	104
FIGURE 20: THE ARCHITECTURE OF STATIC ANALYSIS .....	107
FIGURE 21: STATIC ANALYSIS FEEDBACK FOR A CORRECT ATTEMPT .....	107
FIGURE 22: STATIC ANALYSIS FEEDBACK FOR AN INCORRECT ATTEMPT .....	108
FIGURE 23: AN OVERVIEW OF SOFTWARE ENGINEERING METRICS ANALYSIS .....	109
FIGURE 24: AN EXAMPLE OF PROGRAM STATISTICS OUTPUT.....	112
FIGURE 25: OVERVIEW OF STRUCTURAL SIMILARITY ANALYSIS .....	118
FIGURE 26: GAPS AND THEIR NORMALISED FORMS .....	120
FIGURE 27: ALGORITHM SELECTION PROCESS.....	121
FIGURE 28: IMPLEMENTATION DETAILS OF STATIC ANALYSIS .....	127
FIGURE 29: STUDENT SOLUTION REPRESENTATIONS AT EACH STAGE OF THE ANALYSIS.....	129
FIGURE 30: EXAMPLE OF EXTRACTED AST REPRESENTATION OF GAP.....	130
FIGURE 31: STATIC ANALYSIS INTERFACE.....	130
FIGURE 32: STATIC ANALYSIS PACKAGES .....	131
FIGURE 33: A C# EXERCISE .....	132
FIGURE 34: AN EXAMPLE OF STATIC ANALYSIS CONFIGURATION.....	132
FIGURE 35: GAP CODE AND ITS ABSTRACT PSEUDO CODE FORM.....	136
FIGURE 36: EXERCISES AND STATIC ANALYSIS DATABASE RELATIONSHIPS .....	137
FIGURE 37: THE ELP AND DYNAMIC ANALYSIS INTEGRATION.....	148
FIGURE 38: TEST DIALOGS.....	149
FIGURE 39: DYNAMIC ANALYSIS ARCHITECTURE.....	149
FIGURE 40: BLACK BOX TESTING PROCESSES .....	151
FIGURE 41: STUDENT PROGRAM CONSTRUCTION PROCESS FOR BLACK BOX TESTING.....	153
FIGURE 42: GAP CHANGE PROGRAM CONTROL FLOW .....	154
FIGURE 43: EXAMPLE OF TEST OUTPUT ANALYSIS PROCESS.....	156
FIGURE 44: PROGRAM CONSTRUCTION FOR WHITE BOX TESTING.....	159
FIGURE 45: HIGH-LEVEL CONCEPTUALISATION OF WHITE BOX TESTING .....	161
FIGURE 46: GAP CHANGE CONTROL FLOW EXAMPLES .....	162
FIGURE 47: IDENTIFIER GAP AND ITS DEPENDENTS .....	163
FIGURE 48: THE DYNAMIC ANALYSIS ARCHITECTURE .....	166
FIGURE 49: BLACK BOX TESTING CONFIGURATION .....	168
FIGURE 50: WHITE BOX TESTING CONFIGURATION .....	170
FIGURE 51: BLACK BOX TESTING XML REPORT .....	171
FIGURE 52: WHITE BOX TEST XML REPORT .....	173
FIGURE 53: DATABASE STRUCTURE FOR DYNAMIC ANALYSIS .....	174
FIGURE 54: FEEDBACK PROVIDED FOR STUDENTS .....	182
FIGURE 55: EXAMPLE OF CUSTOMISED COMPILE ERROR MESSAGE.....	183
FIGURE 56: ELP EDITOR APPLET .....	184
FIGURE 57: CORRECT ANSWER VERIFICATION .....	186

FIGURE 58: CORRECT ANSWER ELABORATION .....	187
FIGURE 59: INCORRECT ANSWER VERIFICATION .....	187
FIGURE 60: ELABORATION ON INCORRECT ANSWER.....	188
FIGURE 61: PROGRAMMING PRACTICE HINTS.....	188
FIGURE 62: STUDENT COMPILE MONITORING.....	192
FIGURE 63: ANALYSIS FEEDBACK PROVIDED FOR STUDENTS.....	193
FIGURE 64: HIGH-LEVEL IMPLEMENTATION OF THE FEEDBACK GENERATION COMPONENT .....	194
FIGURE 65: BLACK BOX TEST RESULT - XML FILE.....	196
FIGURE 66: WHITE BOX TEST RESULT - XML FILE .....	196
FIGURE 67: STATIC ANALYSIS RESULT - XML FILE .....	196
FIGURE 68: UNITS EVALUATION HIERARCHY .....	203
FIGURE 69: FIRST EVALUATION QUESTIONNAIRE.....	204
FIGURE 70: SECOND EVALUATION QUESTIONNAIRE.....	205
FIGURE 71: THIRD EVALUATION QUESTIONNAIRE .....	209
FIGURE 72: STUDENT FEEDBACK ON THE EFFECTIVENESS OF “FILL IN THE GAP” EXERCISES .....	210
FIGURE 73: STUDENT AGREEMENT ON WHETHER THE ELP HELPS THEM START PROGRAMMING FROM DAY ONE.....	211
FIGURE 74: STUDENT AGREEMENT ON WHETHER THE ELP HELPS THEM LEARN TO PROGRAM .....	212
FIGURE 75: STUDENT AGREEMENT ON THE EASE OF USE OF THE ELP USER INTERFACE.....	216
FIGURE 76: STUDENT AGREEMENT ON THE HELPFULNESS OF THE “CHECK PROGRAM” FUNCTION .....	217
FIGURE 77: STUDENTS OPINIONS ON THE “CHECK PROGRAM” FUNCTION.....	217
FIGURE 78: STUDENT AGREEMENTS ON THE UNDERSTANDABILITY LEVEL OF PROGRAM ANALYSIS FEEDBACK.....	219
FIGURE 79: STUDENT OPINIONS ON THE UNDERSTANDABILITY LEVEL OF PROGRAM ANALYSIS FEEDBACK.....	219
FIGURE 80: STUDENT OPINIONS ON THE USEFULNESS OF THE “CHECK PROGRAM” FUNCTION FEEDBACK .....	220
FIGURE 81: STUDENT DISTRIBUTION FOR REASONS IF THE PROVIDED FEEDBACK IS USEFUL.....	220
FIGURE 82: FOCUS GROUP QUESTIONNAIRE .....	222
FIGURE 83: STAFF QUESTIONNAIRE FORM.....	225

## List of Tables

TABLE 1: DESCRIPTION OF BLOOM'S TAXONOMY LEVELS .....	17
TABLE 2: COMPUTER ASSISTED ASSESSMENT TOOLS .....	33
TABLE 3: STUDENTS' COMMON JAVA ERRORS SUPPORTED BY THE EXPRESSO SYSTEM.....	45
TABLE 4: COMPARISON OF OTHER WEB-BASED PROGRAMMING ENVIRONMENTS AND ELP .....	54
TABLE 5: COMPARISON OF OTHER PROGRAM ANALYSIS FRAMEWORKS AND THE ELP ANALYSIS FRAMEWORK.....	55
TABLE 6: MISSING PROGRAMMING ATTRIBUTES AMONG NOVICES .....	66
TABLE 7: NOVICES' DIFFICULTIES OVERCOME BY GAP-FILLING EXERCISES.....	73
TABLE 8: ELP DATABASE TABLES FIELDS DESCRIPTIONS.....	87
TABLE 9: COMMON STATIC ANALYSIS PROGRAM CODE METHODS .....	93
TABLE 10: SOFTWARE METRICS CLASSIFICATIONS AND THEIR ATTRIBUTES .....	95
TABLE 11: CYCLOMATIC COMPLEXITY THRESHOLD.....	97
TABLE 12: NOVICE STUDENTS' COMMON JAVA MISTAKES .....	99
TABLE 13: GAP TYPES .....	102
TABLE 14: GAP TYPES AND THEIR AVAILABLE VARIABLE SCOPES .....	104
TABLE 15: STATIC ANALYSIS FUNCTIONS, THEIR DESCRIPTIONS AND ADDRESSED MISTAKES .....	111
TABLE 16: CYCLOMATIC COMPLEXITY DEFAULT VALUES .....	113
TABLE 17: EXAMPLES OF REDUNDANT LOGIC EXPRESSION CHECK .....	114
TABLE 18: POSSIBLE SEMANTICS VARIATIONS BETWEEN TWO PROGRAMS .....	116
TABLE 19: HANDLED VARIATIONS BY THE STATIC ANALYSIS .....	123
TABLE 20: BLACK BOX AND WHITE BOX TESTING COMPARISON.....	145
TABLE 21: INTRODUCTORY PROGRAMMING EXERCISES CATEGORIES .....	146
TABLE 22: FUNCTIONS PROVIDED TO CHECK THE RESULT OF BLACK BOX TESTING.....	171
TABLE 23: DYNAMIC ANALYSIS DATABASE TABLES FIELDS DESCRIPTIONS .....	175
TABLE 24: DYNAMIC ANALYSIS - TEST OUTPUTS RECEIVABLE POSSIBILITIES .....	189
TABLE 25: TEST OUTPUTS CORRECTNESS POSSIBILITIES .....	189
TABLE 26: STATIC ANALYSIS CATEGORIES .....	190
TABLE 27: FIRST EVALUATION RESULTS .....	205
TABLE 28: EVALUATION OBJECTIVES AND ASSOCIATED SECOND QUESTIONNAIRE QUESTIONS .....	206
TABLE 29: EVALUATION OBJECTIVES AND ASSOCIATED QUESTIONS IN THIRD QUESTIONNAIRE .....	208
TABLE 30: THIRD EVALUATION PARTICIPANTS .....	210
TABLE 31: NUMBER OF STUDENTS WHO COMPLETED OBJECTIVE FOUR QUESTIONS .....	216
TABLE 32: RELATIONSHIP BETWEEN TEACHING STAFF EVALUATION OBJECTIVES AND QUESTIONS IN THE QUESTIONNAIRE FORM.....	224
TABLE 33: NUMBER OF STUDENTS AND STAFF PARTICIPATING IN THE EVALUATIONS .....	228
TABLE 34: ADAPTIVE LEARNING ENVIRONMENT CATEGORIES.....	238
TABLE 35: OPEN SOURCE GUI TESTING TOOLS.....	240
TABLE 36: GAPS TYPES AND THEIR BEHAVIOURS .....	254



## List of Abbreviations

<b>API</b>	Application Programming Interface
<b>AST</b>	Abstract Syntax Tree
<b>CAA</b>	Computer Assisted Assessment
<b>CMS</b>	Content Management System
<b>EBNF</b>	Extended Backus-Naur Form
<b>ELP</b>	Environment for Learning to Program
<b>FIT</b>	Faculty of Information Technology, QUT
<b>GUI</b>	Graphical User Interface
<b>HTML</b>	Hypertext Markup Language
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IDE</b>	Integrated Development Environment
<b>IO</b>	Input Output
<b>IPE</b>	Intelligent Programming Environment
<b>ITiCSE</b>	Innovation and Technology in Computer Science Education
<b>ITS</b>	Intelligent Tutoring System
<b>JAR</b>	Java ARchive
<b>JDK</b>	Java Development Kit
<b>JRE</b>	Java Runtime Environment
<b>OLT</b>	Online Learning and Teaching, QUT
<b>OO</b>	Object-Oriented
<b>PASS</b>	Peer Assisted Study Scheme
<b>PASS</b>	Program Assessment using Specified Solution
<b>QUT</b>	Queensland University of Technology
<b>SE</b>	Software Engineering
<b>XML</b>	eXtensible Markup Language
<b>XSLT</b>	eXtensible Stylesheet Language Transformation
<b>ZPD</b>	Zone of Proximal Development



## **Statement of Original Authorship**

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

**Signed** .....

**Date** .....



## **Previously Published Material**

The following papers have been published and contain material based on the content of this thesis.

1. Truong, N., P. Bancroft and P. Roe. 2002. ELP - A Web Environment for Learning to Program. In Australasian Society for Computers in Learning in Tertiary Education (ASCILITE), 661-670. Auckland, New Zealand. Available at: <http://ascilite.org.au/conferences/auckland02/proceedings/papers/082.pdf>
2. Truong, N., P. Bancroft and P. Roe. 2003. A Web Based Environment for Learning to Program. In Australasian Computer Science Conference, 255-264. Adelaide, Australia: ACM Press.  
Available at: <http://crpit.com/confpapers/CRPITV16Truong.pdf>
3. Truong, N., P. Bancroft and P. Roe. 2004a. Learning To Program in a Web Environment. In Online Learning and Teaching (OLT2004), 183-190. Brisbane, Australia: QUT. Available at:  
<http://oltfile.qut.edu.au/download.asp?rNum=1587222&pNum=1333785&fac=udf&OLTWebSiteID=OLT2004&CFID=6523793&CFTOKEN=38327948>
4. Truong, N., P. Bancroft and P. Roe. 2004b. Static Analysis of Students' Java Programs. In Australasian Computing Education Conference, 317-325. Dunedin, New Zealand: ACS.  
Available at: <http://crpit.com/confpapers/CRPITV30Truong.pdf>
5. Truong, N., P. Bancroft and P. Roe. 2005a. Automated Feedback for 'Fill in the Gap' Programming Exercises. In Australasian Computing Education Conference (ACE2005), 117-126. Newcastle, Australia: ACS.  
Available at: <http://crpit.com/confpapers/CRPITV42Truong.pdf>
6. Truong, N., P. Bancroft and P. Roe. 2005b. Learning to Program through the Web. In Annual SIGCSE conference on Innovation and technology in computer science education ITiCSE '05, 9-13. Lisboa, Portugal: ACM Press.
7. Truong, N., P. Bancroft and P. Roe. 2006. A Web Environment for Learning to Program. In Transforming IT education: Promoting a culture of excellence, eds. C. S. Bruce, G. Mohay, G. Smith, I. Stoodley and R. Tweedale, 205-222: Informing Science Press.



## **Acknowledgements**

First and foremost, I wish to express my deepest gratitude to Professor Paul Roe, my principal supervisor, for his inspiration and guidance not only for my research but also in other aspects of life. I also would like to thank Dr Peter Bancroft, my associate supervisor, for his support and constructive criticism throughout my PhD journey.

Many thanks to Dr On Wong and my friends Aaron, Jiro, Wen-Poh and Darren in the Programming Languages and Research Group at the Queensland University of Technology for their encouragement and for sharing their PhD experiences. Further thanks are extended to James Hallam, Tim Cater and other colleagues who have refined and evolved my research contributions.

Finally, but always first in my thoughts, I thank my grandparents, my parents for their unconditional support and encouragement, and my dear husband, Gavin, for his love and companionship.



# **Chapter 1 - Introduction**

*People learn best when engrossed in the topic, motivated to seek out new knowledge and skills because they need them in order to solve the problem at hand. (Norman and Spohrer, 1996)*

Teaching programming is a major challenge in computer science education. Novice programmers have difficulty in starting their first program and constructing abstract knowledge about programming (du Boulay, 1986; Robins et al., 2003). Teaching staff are often less successful in teaching students programming skills than they would expect (McGetrick et al., 2005). Introductory programming classes often have high attrition and low success rates (Morrison and Newman, 2001). An Innovation and Technology in Computer Science Education (ITiCSE) working group revealed that at the end of their first year, many students still do not know how to program (McCracken et al., 2001). This is because programming is difficult to learn and is one of the most difficult branches of applied mathematics (Dijkstra, 1982). Making learning to program easier and barrier-free for novices has presented long-term challenges to educators. When this is achieved, it can increase course success rates and change students' attitude towards programming.

One of the biggest problems with learning to program is that programming languages are artificial (Moser, 1997). While listening to lecturers and reading reference materials continue to be the primary means of acquiring knowledge at universities, programming knowledge cannot be directly transmitted from instructors to students to construct, as it must be acquired actively by students with guidance from teachers and feedback from other students (Ben-Ari, 2001). More importantly, repetition by practicing ensures the knowledge is retained. Therefore, to succeed in a programming course, students need plenty of practice. Unfortunately, students may lose their confidence and delay writing their own programs if they encounter many difficulties. Some common difficulties faced by novice programmers are:

1. Installing and setting up a programming environment,
2. Using a programming editor,

3. Understanding programming questions and using a programming language syntax knowledge to write code,
4. Understanding compilation errors, and
5. Debugging.

These difficulties are increased significantly when modern object-oriented (OO) programming languages such as C++, Java or C# are taught as the first programming language because these languages contain high-level abstraction and are designed especially for experienced programmers.

When students undertake programming practice, it is important that they are provided with support when they encounter difficulties. Feedback needs to be easily and immediately accessible to avoid misconceptions (Ben-Ari, 2001). This is because novice programmers do not have an effective model of computers that they can use to make a viable construction of knowledge based upon sensory experiences, such as reading, listening to lecturers and working with a computer (Ben-Ari, 2001). However, providing individual and immediate help for students in large class sizes together with heavy scheduled work represents a serious difficulty for tertiary educators.

Even for students who can overcome these starting hurdles and manage to write a program, research has shown that their programs are often poorly constructed. It has been noted that novices often do not pay attention to the quality of their programs (Joni and Soloway, 1986). They prefer to jump straight into coding as quickly as possible without enough consideration of the quality of their programs; nor do they iterate through the code to improve the program quality (Vizcaino et al., 2000). Helping students to consider the quality of their programs requires a significant amount of time and effort from teaching staff.

## **1.1 Aims and Objectives**

The aims of this research are: 1) to make learning to program easier for beginners by providing them with frequent and formative feedback and hence changing their attitude about programming; 2) to increase the success rate of introductory programming courses; and 3) to assist educators in teaching and administering large programming classes. A web-based programming environment is designed to help novice programmers learn to program in modern OO programming languages and to

provide immediate feedback to them about the quality and correctness of their programs. Students no longer have to spend time in installing and setting up a programming environment nor learning how to use a programming editor.

This research makes effective use of technology to propose flexible courseware which meets the needs of on-line and distance education and addresses a diverse range of students. The research facilitates effective educational transfer of knowledge and skills. Students' problem-solving skill is developed independently from programming language knowledge through exercises which address all six cognitive levels of Bloom's taxonomy (Bloom, 1956). Consequently, students are provided with a firm foundation upon which they can expand their skills and knowledge to keep up-to-date with the rapid change of technology.

For a long time, a large body of research has been conducted to help students learn to program. As a result of that, many tools have been developed. However, a majority of existing tools require students to write a complete program from scratch which is extremely difficult for them. Literature has revealed that beginning students often do not know where to start when they encounter a programming exercise due to the lack of problem-solving skill (Mayer, 1981; Perkins et al., 1986; Perkins et al., 1988). This skill is beyond any programming language syntax knowledge and needs to be developed independently (Perkins et al., 1988; Wiedenbeck et al., 1993). In addition, setting up and learning how to use an Integrated Development Environment (IDE) is not easy since a majority of available IDEs are designed for professional programmers.

As noted earlier, feedback is an important factor to facilitate interaction and motivation in students' learning when they undertake their practice. Many programming environments have built-in functionality to provide immediate feedback for students. However, a large proportion of those applications only assist students with syntax and semantics problems to get a program compiled rather than on quality and correctness of their programs (Deek and McHugh, 1998). Although some systems provide feedback on quality and correctness of students' programs, the feedback often does not have sufficient information to help weak students address issues in their programs. This is because a majority of the systems only carry out quantitative static analyses on program source, therefore they cannot indicate the areas of poor quality code and provide suggestions on how to improve them (Soloway et al., 1983). Similar to correctness feedback, the majority of existing

systems can only carry out functional testing, which only checks if a program has all functionality as specified in the requirements, as well as checking whether these functions produce correct results. Current available correctness feedback has been provided mainly based on analysing program outputs. Therefore, additional information, such as a possible location in a program that might cause a particular test to fail, information on reasons and how to fix the problem, is not provided for students.

From the perspective of teaching staff, most program analysis frameworks used for tutoring purposes require a large amount of time to set up exercises in order for analysis to be carried out (Deek and McHugh, 1998), while those used to assist teaching staff in the marking process are inflexible (Preston and Shackelford, 1999). Teaching staff are not able to adjust marks that are given for a particular criterion based on students' work, nor can they provide customised feedback or annotate a particular area in students' codes. Markers often feel "powerless" when using these systems (Preston and Shackelford, 1999). Therefore, educators are reluctant to use one in their classroom.

Hence, there is a need for a learning environment in which students are provided with an initial start for a programming problem so that they can successfully write their first program from 'day one'. The environment should focus on developing student problem-solving skill and also give formative feedback on both quality and correctness of students' programs. The provided feedback not only indicates if the student program is correct or not but also recommends alternative approaches to solve the problem, and gives information to improve the program quality or fix issues in the program.

## 1.2 Thesis Contributions

The contributions of the thesis are:

1. A constructive web-based Environment for Learning to Program (ELP) which supports "fill in the gap" exercises to help students to program successfully at an early stage in their learning.
2. A program analysis framework which is integrated into ELP for conducting both static and dynamic analyses to assess the quality and correctness of students' "fill in the gap" programs.

3. An automated feedback engine which provides customised compilation error messages, as well as timely and formative feedback for ELP students and their instructors on the program quality, structure, and correctness based on the program analysis results.

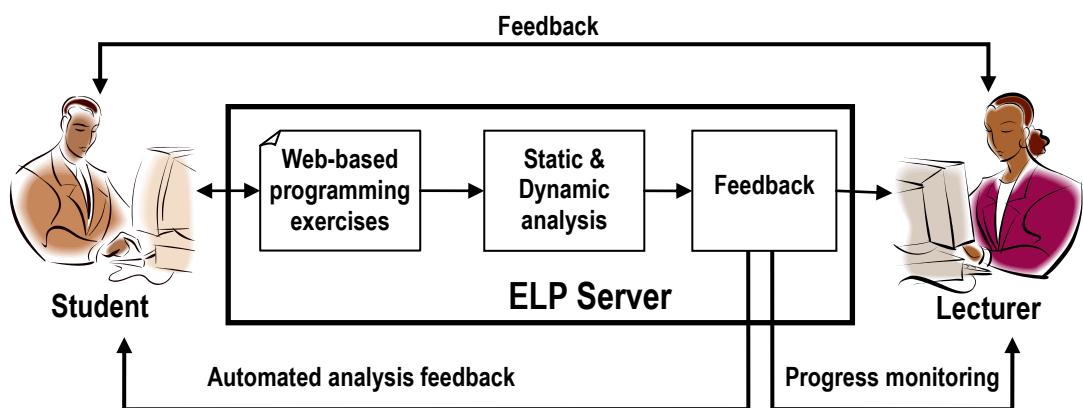
A key novel aspect of the ELP system and the program analysis framework is the capability in supporting small “fill in the gap” programming exercises. This type of exercise has been emphasised by many computer science educators and constructivists as a good way to teach beginning students to program and to develop their program comprehension (Ehrlich and Soloway, 1984; Entin, 1984; Hall and Zweben, 1986; Norcio, 1980a, 1980b, 1982a; Thomas and Zweben, 1986) and problem-solving skills (Garner, 2001). This is because students are required to understand the code provided for an exercise first before they can use their language syntax knowledge to complete it; hence, it will increase students’ comprehension skill. In addition, since novice programmers do not have domain specific knowledge, they cannot construct new knowledge based upon their experiences (Ben-Ari, 2001). At the same time, they also have difficulties in applying their existing knowledge to solve a new problem due to their lack of problem-solving skill (Affleck and Smith, 1999). Research has shown that provided code can be used as a concrete computer programming model which novices can build upon as a framework to incorporate new knowledge and apply their existing knowledge to new situation (Mayer, 1981).

Lister (2000) has claimed that this type of exercise addresses the lower four levels of Bloom’s taxonomy (Bloom, 1956) – namely, knowledge, comprehension, application and analysis - whereas the traditional exercises that require students to write a complete program can only address the last two levels of the taxonomy – synthesis and evaluation. Further, gap-filling programming exercises reduce the complexity in writing programs. The approach builds up students’ confidence; students are motivated and engaged in their learning process more actively.

The research contributions address the long-term challenges in teaching beginning students to program by:

1. Eliminating setting up hurdles which in turn makes programming easier to learn;
2. Providing a constructive learner-centred learning environment.

The ELP system changes the way students learn to program and how programming is taught in introductory programming courses. Typically, students work with program templates that focus their attention on the critical dimensions of a problem which needs to be solved, while at the same time receiving timely, appropriate, and directed feedback on their solutions. Through the ELP, teaching staff are able to monitor students' progress and identify their common problems. Hence, teaching material can be adjusted to fit the students' needs. Figure 1 shows the new teaching and learning style with the ELP system.



**Figure 1: New Teaching and Learning Style with ELP**

By using the ELP system, students are able to access as much tuition as they need. They are not limited to normal contact hours per week, nor are they limited to standard working hours. Students experience programming as a problem-solving activity from the earliest stage of learning. They are exposed to a variety of approaches to problem solving, the consequent alternative solutions, and the relative merit of such approaches through various feedback mechanisms. The system also provides a learning environment which meets various students' requirements, such as those who find attendance at tutorials difficult, those who require more time for tutorials, or those who are not used to normal tutorial situations.

As shown in Figure 1, in addition to the feedback from teaching staff, students are also provided with immediate feedback about the quality and correctness of their programs by the program analysis framework. This approach emphasises the problem-solving aspect of programming and enables finer-grained assessment to be used. In general, students learn by doing, getting feedback, and reflecting – all through the web. Learning is improved by providing early and frequent feedback.

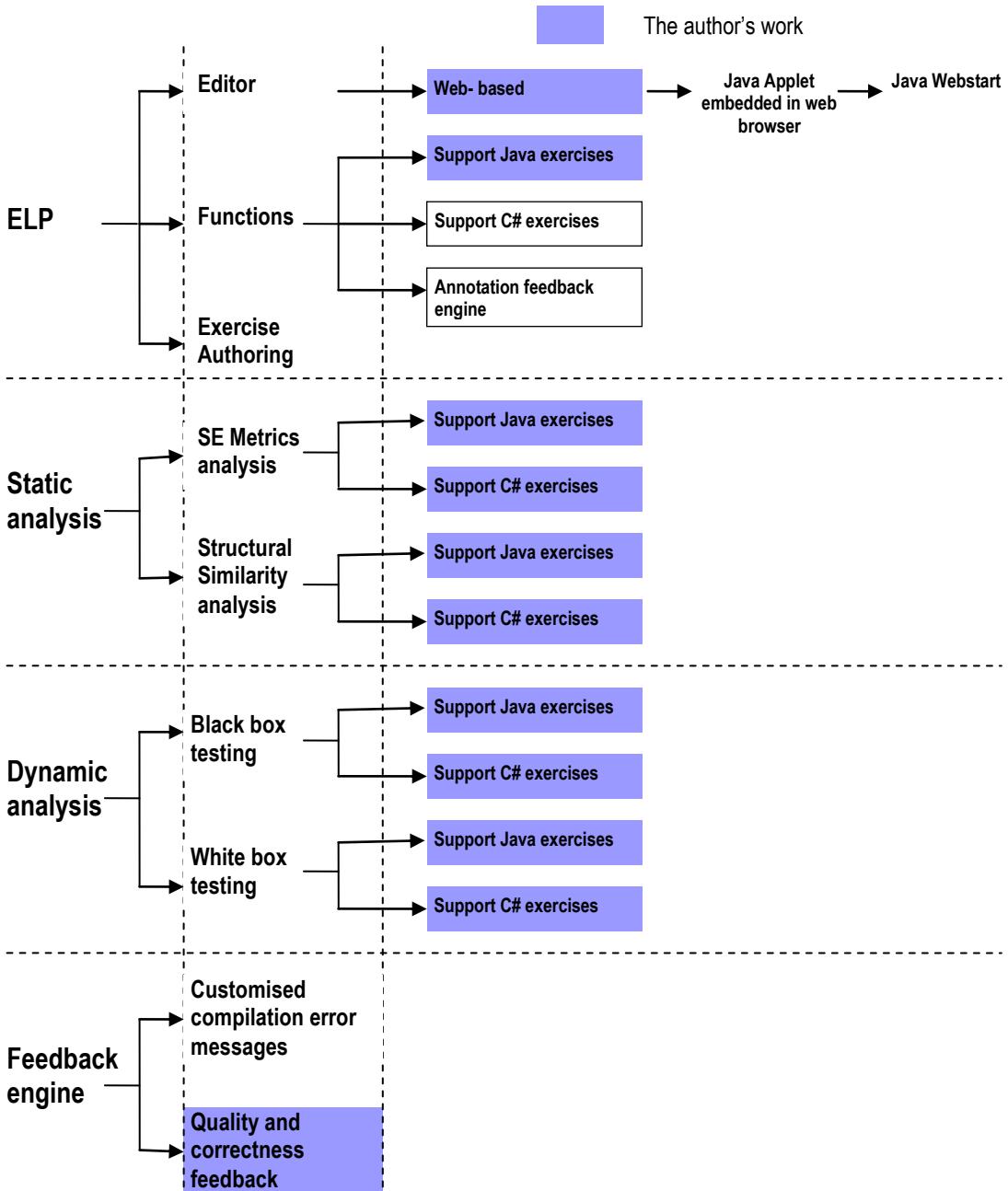
The program analysis framework carries out two types of analyses: static and dynamic. The static analysis is designed to judge the quality of student “fill in the gap” programs. In this analysis, both qualitative and quantitative analyses are conducted. Quantitative analysis is achieved by computing various cyclomatic complexity metric values of a program which indicate its overall quality. At the same time, a student’s program is checked against a set of common poor programming practices such as the use of literal values and lengthy methods. Qualitative analysis is achieved by comparing a program’s structure with the structure of expected model solutions to evaluate if it has the right structure. By checking poor programming practices and comparing its structure with the expected model solutions, the static analysis framework can provide detailed feedback for students on which section in a program is poorly coded or too complex, and approaches to improve the quality of the program can be recommended.

The dynamic analysis component of the program framework verifies the correctness of student “fill in the gap” programs. In this analysis, both *black box testing* which focuses on the functionality of programs and *white box testing* which is primarily concerned with the internal structure of the program, are carried out. Black box testing is the process of testing a program against its specification with testers having no knowledge of the program’s implementation while this knowledge is required for white box testing. White box testing identifies hidden errors in a program which are not revealed in black box testing. By conducting both black box and white box testing, the dynamic analysis framework can suggest possible reasons to students for why their programs fail certain tests and approaches to improve the performance of the program.

The static and dynamic analyses results are incorporated to provide feedback for students through the automated feedback engine: for example, “Your program has the right structure and passes the following tests, but not this test” or “Your program has passed all tests but the structure can be improved”. Static analysis feedback comprises comments on the quality of student programs, hints on how to improve the quality of the program and alternative solutions. On the other hand, dynamic analysis feedback reports on tests that were carried out together with the pass or fail results, specifies any runtime errors, identifies gaps that might contain errors, and provides hints on how the problem might be fixed.

The initial contributions of the research have been continuously evaluated by students and teaching staff. They are refined to address the users' needs. The ELP system has been used as an additional tutorial resource in seven subjects in the Faculty of Information Technology (FIT) at Queensland University of Technology (QUT) over the last four years. This is the achievement of the author and three other colleagues in the team in which the author played the key leadership role in researching, designing, implementing and evaluating the system. Figure 2 shows the evolution of the ELP system and the program analysis framework with the contributions of the author highlighted.

As illustrated in Figure 2, the author was the key investigator in designing and developing the first version of the ELP system which supports Java "fill in the gap" programming exercises presented on a web page. After the first evaluation, the editor of the system was redeveloped by other team members as a Java applet embedded in a web browser in order to provide more functions for the system. The ELP system was also extended by other team members to support C# programming exercises, give customised compilation error messages, and provide a communication mechanism between students and instructors through the annotation feedback engine and an exercise authoring interface. Each release of the system was researched by the author and the results were used as input for other team members to engineer the system.



**Figure 2: The Evolution of the ELP System and the Program Analysis Framework**

### 1.3 Outline of Thesis

The thesis is comprised of eight chapters; the remaining chapters are structured as follows:

Scaffolding teaching approach, constructivism learning theory and Bloom's taxonomy model of learning processes are reviewed in the second chapter. The chapter also gives a historical overview of research and development in computer science education; their current trends and future directions. It then reviews

applications which have been developed in computer science education to help students and teaching staff in teaching and learning programming. The main foci are web-based learning environments and automated program analysis systems.

The third chapter introduces the web-based Environment for Learning to Program (ELP) with its key novel aspect which supports “fill in the gap” programming exercises. The chapter first discusses difficulties encountered by novice programmers by analysing skills, knowledge, and strategies that are required in order to write programs. It highlights the skills, knowledge and strategies that among novices are weak or poorly developed. Based on this foundation, the design of the ELP system and how gap-filling type exercises help novices to overcome the barriers are discussed before detailing the implementation of the system.

The fourth chapter presents the static analysis component of the program analysis framework designed to assess the quality of students’ “fill in the gap” programming exercises. The chapter commences with an important discussion on approaches and software metrics that have been used to implement static analysis in computer science educational applications. It reviews poor programming practices of beginning students as reflected throughout the literature, together with the author’s findings as a foundation. The design and implementation of the static analysis are elaborated upon.

The fifth chapter introduces the dynamic analysis component of the program analysis framework and demonstrates how the correctness of students’ “fill in the gap” programs is ensured. The chapter starts by providing an overview of software testing theories followed by a discussion of the different types of programming exercises in CS1 courses which play an important role in the design of the analysis. The dynamic analysis framework is then elaborated upon.

The sixth chapter presents the design and implementation of the automated feedback engine which processes the results of analyses to provide more constructive feedback for students and teaching staff. Hence, learning and teaching programming is made easier. The chapter commences with a review of effective feedback in computer-based instruction. It then details the design and implementation of feedback provided for students and teaching staff.

The seventh chapter reports on the results of quantitative and qualitative evaluation of the ELP and the program analysis framework conducted among students and teaching staff. Evaluation objectives are first stated before describing

how each objective is evaluated and its results. The chapter is concluded by a discussion on the evaluation results.

The eighth chapter concludes the thesis by summarising the achievements and outlining areas of extension to this research. Appendices are provided to show some examples of beginning students' exercises in Java and C# which are currently available in the ELP system. These exercises have also been used to test and evaluate the system.



# **Chapter 2 - Literature Review**

## **2.1 Introduction**

Computers have been playing a significant role in education since their first use in the early 1960s. From that time on, there has been a continuing change in how computers are used in education. A comprehensive overview on the usage of computers in education can be found in Curtis (2000) and Kurland and Kurland (1987); Curtis' review pay special attention on the usage of computers in computer education.

In terms of hardware, in the 1960s expensive computer mainframes were used to teach programming and many other subject areas. In the 1970s, less expensive mini computers arrived. In the 1980s, stand-alone personal computers were introduced. Today, those personal computers are connected through the Internet, WWW and email.

In terms of software used over decades, in the 1960s, FORTRAN or assembler was used in programming classes. Programs were small and simple because of physical memory limitations. A decade later, structured programming was introduced, and Pascal was the main language taught in undergraduate courses. In the 1980s, strong type checking and structured programming languages such as Ada, Modula 2, ML and Prolog were introduced. In the 1990s and this decade, object-oriented languages such as C# and Java are predominant.

Teaching methods have also changed over decades. In the 1960s, computers were only used to teach programming. Programming courses offered to students only focused on details of the programming language features. Programming activities were limited to solving only intermediate problems with no attempt of generalising the concepts to solve subsequent problems (Curtis, 2000). In the 1970s problem-solving was emphasised in introductory programming courses and this emphasis has continued until now. In the 1980s, in parallel with the emphasis on problem solving, data abstraction was also emphasised. Programs became larger, more interactive and their input and output changed from text to graphics. In the current decade, programs are more complex and their input and output forms are more varied. Today, programming course content has grown quickly along with the number of

programming languages. Education providers must carefully select the scope and the areas to cover while still emphasising teamwork.

This chapter aims to review related research and development of applications in computer science education, especially in teaching introductory programming. The chapter consists of five sections:

- Section 2.2 reviews the scaffolding teaching approach, Bloom's taxonomy model of learning processes, and constructivist learning theory, which are used as the basis of the research and design of the ELP system.
- Section 2.3 describes the current state of the art of research in computer science education applications.
- Section 2.4 predicts the future trends of computer science education applications.
- Section 2.5 reviews web-based systems which had a great impact on the design and implementation of the ELP system.
- Section 2.6 details prior automated program analysis systems which assess the quality and correctness of student programs.

## **2.2 Education Background**

This section gives an overview of three important educational key concepts: the scaffolding teaching approach, Bloom's taxonomy model of learning processes, and constructivist learning theory, which the research and design of the ELP system are based on.

### **2.2.1 Scaffolding**

The scaffolding approach to teaching was introduced by Wood, Bruner and Ross (1976). The approach is based on Vygotsky's Zone of Proximal Development (ZPD) defined as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers" (Vygotsky, 1978). In general, this is the zone where the learner is just able to perform a task but needs to be supported.

In scaffolding instruction mode, learners are individually supported based on their ZPD by a more knowledgeable person. Learning activities which happen in this

mode of instruction are beyond the level of what the learner can do alone (Olson and Platt, 2004). The scaffolds facilitate a student's ability to build on prior knowledge and internalise new information. Learners are provided with more achievable tasks together with directions to achieve the goals. McKenzie (1999) identified eight characteristics of educational scaffolding. These characteristics are:

1. Provides clear directions
2. Clarifies purpose
3. Keeps students on task
4. Offers assessment to clarify expectations
5. Points students to worthy sources
6. Reduces uncertainty, surprise and disappointment
7. Delivers efficiency
8. Creates momentum

An important aspect of scaffolding instruction is that the scaffolds are temporary (Stuyf, 2002; Tobias, 1982) and adjustable (Rosenshine and Meister, 1992). As the learner's ability is increased, the scaffolding provided by the more knowledgeable person is progressively withdrawn. Finally, the learner is able to complete the task or master the concepts independently (Chang et al., 2002; Stuyf, 2002). Learners in scaffolding instruction are engaged, motivated and have minimum levels of frustration. This is extremely important in the case of special needs students who can become frustrated very easily then shut down and refuse to participate in further learning (Stuyf, 2002).

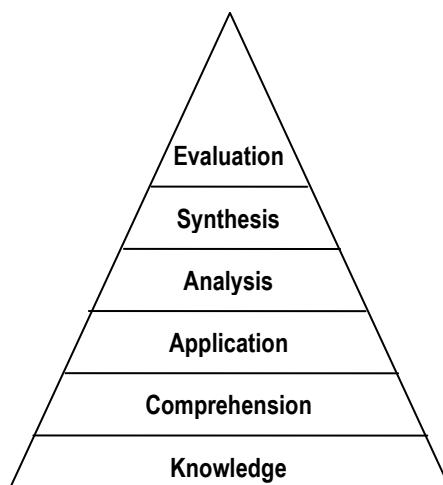
Although scaffolds apply to the teaching of all skills, they are particularly useful for teaching high-level cognitive strategies where many of the steps or procedures necessary to carry out these strategies cannot be specified (Rosenshine and Meister, 1992). Scaffolding in computer science education, particularly in programming, has been acknowledged to greatly benefit novice programmers: for example, the scaffolding teaching approach used to teach software design in introductory programming courses increases students' confidence and improves their problem-solving skills (Linder, Abbott and Fromberger, 2006). Eitelman (2006) reviews several research and development projects which use the scaffolding teaching approach in computer programming courses.

## 2.2.2 Bloom's Taxonomy

Bloom's taxonomy (Bloom, 1956) is a method of categorising cognitive skills by increasing order of complexity and can be used as a means to organise tasks and assessments in classrooms. There are six levels identified within the cognitive domain. The taxonomy is also used for analysing cognitive depth of performing a given task. It is a “concise model for the analysis of educational outcomes in the cognitive area and of remembering, thinking and problem solving” (Bloom, 1956). Figure 3 shows the levels of the taxonomy and Table 1 shows verbs used for describing objectives at each level.

Research has shown that when the taxonomy is included in the course, students become more engaged. They start asking deeper questions. They understand better what is expected of them. Instructors know better what to expect, how to teach the materials and how to assess student performance (Manaris and McCauley, 2004).

Although the taxonomy has been used in various education fields, it has been recently used as a basis for a number of studies in computer science education (Box, 2004; Lister, 2000; Lister and Leaney, 2003a, 2003b; McCauley, 2004; Rountree et al., 2005; Soh et al., 2005).



**Figure 3: Bloom's Taxonomy Levels**

<b>Level</b>	<b>Name</b>	<b>Description</b>	<b>Verb used for Objectivities</b>
1	Knowledge	Ability to recall facts	define, memorise, repeat, record, list, recall, name, relate, collect, label, specify, cite, enumerate, tell, recount
2	Comprehension	Understanding, translation, interpretation	restate, summarise, discuss, describe, recognize, explain, express, identify, locate, report, retell, review, translate
3	Application	Use of knowledge in a new context	exhibit, solve, interview, simulate, apply, employ, use, demonstrate, dramatise, practice, illustrate, operate, calculate, show, experiment
4	Analysis	Identification of relationships	interpret, classify, analyse, arrange, differentiate, group, compare, organise, contrast, examine, scrutinise, survey, categorize, dissect, probe, inventory, investigate, question, discover, text, inquire, distinguish, detect, diagram, inspect
5	Synthesis	(Re)assembling of parts into a new whole	compose, set up, plan, prepare, propose, imagine, produce, hypothesise, invent, incorporate, develop, generalise, design, originate, formulate, predict, arrange, contrive, assemble, concoct, construct, systematize, create
6	Evaluation	Making judgement	judge, assess, decide, measure, appraise, estimate, evaluate, infer, rate, deduce, compare, score, value, predict, revise, choose, conclude, recommend, select, determine, criticise

**Table 1: Description of Bloom's Taxonomy Levels**

### 2.2.3 Constructivism

Constructivist learning theory is the predominant teaching and learning paradigm in education today. The theory is based on the premise that learners must actively construct knowledge rather than passively waiting to be filled. Key contributors to the development of the theory include John Dewey (1933/1997), Jerome Bruner (1966), Piaget (1969), Vygotsky (1978) and Glaserfeld (1995).

Constructivism is a learner-centred pedagogy typified by experiential discovery learning through exploration. Teaching in a constructivist manner means that new information and experiences are presented in a way that places them into context and integrates them with knowledge the students already possess. Constructivist models of learning have been proved more effective than traditional settings; constructivism extends students beyond the information presented to them (Bruner, 2001) which leads to enhanced cognitive development (Glassman, 2001; Panitz, 1999) and increases motivation (Slavin, 1995), perceptions of skill development and solution satisfaction (Benbunan, 1997).

Boyle (2000) summarised the principles of constructivism in five main points:

1. Authentic learning tasks: learning tasks should be embedded in a problem-solving context that requires the learner to acquire relevant knowledge and skill to solve the problem.
2. Interaction: interaction must be the primary source for cognitive construction, focusing on the tutor – student and peer group relationship.
3. Encourage voice and ownership in the learning process: students are allowed to choose the problem they will work on, while teachers help students to generate problems which are interesting to them – this is a vital part of problem solving.
4. Experience with the knowledge construction process: learning how to learn, how to construct and refine new meaning.
5. Metacognition: the ability to transform one understanding to resolve other problems – the ultimate goal of the constructivist approach.

Although constructivism has been around for a long time, its applications in computer science education are quite recent (Ben-Ari, 1998; 2001) and have been increasing over the last five years. The exploration of the concept of constructivism in information technology education has resulted in a highly diverse body of practical examples (Ben-Ari, 2001; Booth, 2001; Fowler et al., 2001; Hadjerrouit, 1998; Kolikant, 2001; Lui et al., 2004; McKeown, 2004; Parker and Becker, 2003; Pullen, 2001; Soendergaard and Gruba, 2001; Van Gorp and Grissom, 2001).

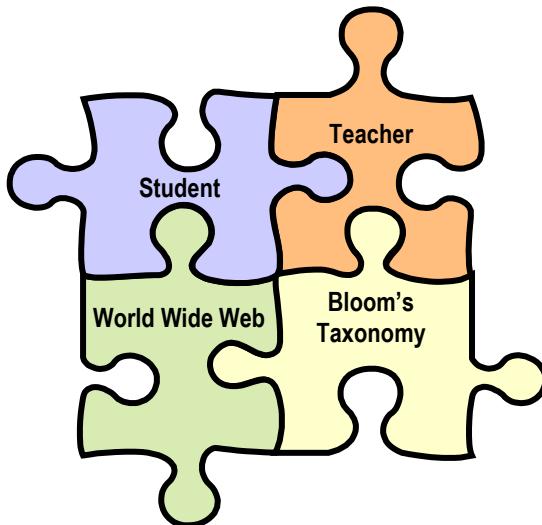
As Ben-Ari (1998) has pointed out, there are two main differences in the application of constructivism in programming than in other sciences. First, novice programmers have no ‘effective model’ of a computer; they do not have a cognitive structure that they can use to make viable constructions of knowledge based upon sensory experiences. Second, the computer forms an ‘accessible ontological reality’; that means students can access the “correct” answer easily.

Lui and colleagues (2004) provide a constructivist view on learning to program. Learning to program involves the mental construction of models of computer programming elements. A successful learner is one who can construct viable models that match the design models of computer programming elements.

With constructivism, the construction of mental models is a recursive process. New mental models appear after existing mental models are adjusted or dropped according to experience gained from interacting with the world. However, there are five major risks that students might encounter in their knowledge construction process. Those hazards are:

1. Programming models are inadvertently presented incorrectly to learners
2. High-level abstract concepts of common first programming languages, such as Java, C# or C, pose difficulties for learners to explore and understand the underlying programming model
3. The actual programming model is not continuously presented to learners due to the lack of feedback in a learning process
4. An incorrectly constructed knowledge is used to construct new knowledge
5. An unsuitable piece of knowledge is the basis on which to construct new knowledge

In brief, this section has reviewed three important educational concepts: the scaffolding teaching approach, Bloom's taxonomy model of learning processes and constructivist learning theory. The three concepts are closely related to each other. Bloom's taxonomy and scaffolding - together with the web - form a constructive dynamic learning environment as shown in Figure 4. In that learning environment, students must actively construct knowledge while teachers must carefully design their teaching material so that it scaffolds students' learning. Teachers must guide students through the exploration of content; they also have to provide opportunities for students to take ownership of their learning by presenting the knowledge in various techniques and addressing different learning styles. Further, teachers also have to constantly assess where students are and how to best address their needs. The next two sections discuss research and applications devoted to make teaching and learning to program easier.



**Figure 4: Dynamic Learning Environment**

(Adapted from Hobgood et al .<http://www.learnnc.org/articles/bloom0405-1>,  
accessed September 15, 2006)

## 2.3 Overview of Applications in Computer Science Education

Computers have been used to assist staff and students in teaching and learning computer science as early as the 1960s. Since then, many programming languages, teaching and learning environments have been developed. This section gives an overview of the use of computers for teaching and learning programming, with the focus on applications for novice programmers. The section has two sub-sections. In Sub-section 2.3.1, computer assisted instruction applications are discussed, while computer assisted assessment applications are detailed in Sub-section 2.3.2.

### 2.3.1 Computer Assisted Instruction

One of the earliest computer applications in education is the PLATO (Bitzer and Johnson, 1971) project which was initiated at the University at Illinois with the National Science Foundation in 1960. From that time on, many tools and applications have been developed; these systems overlap each other and there is no clear classification among them. Deek and McHugh (1998) grouped applications in computer science education for novice programmers into four groups based on their functionality:

1. Programming environments which support program construction, compilation, testing and debugging

2. Debugging aids which are used to detect and correct errors
3. Intelligent tutoring systems which can be used to provide adaptive individualised instruction, analyse student responses, determine correctness, guide, interact with students and provide feedback and advice
4. Intelligent programming environments which combine functionalities of programming environment and intelligent tutoring systems

Meanwhile, from a different perspective, Kelleher and Pausch (2005) classified introductory programming tools based on taxonomy of programming languages and environments designed to make programming more accessible to novice programmers of all ages. According to the authors, there are two main goals that programming tools are based on. First are those designed to teach programming, classified as teaching systems. Second are those designed to support the use of programming in pursuing of another goal, classified as empowering systems. Teaching systems can be divided into three sub-categories. The first category is called mechanics of programming which consists of systems designed to help students to understand and express their intentions to computers. Social learning is the second group of systems which allow students to learn in group, and the third group is called providing reason to program which includes systems that provide special activities as a starting point for students before programming. Similarly, empowering systems are also classified into two groups: mechanics of programming and activities enhanced by programming. Each of these groups is then subdivided into many small categories. Figure 5 overviews different groups of first year introductory programming tools based on their taxonomy. Kelleher and Pausch (2005) can be referred to for more information on each of these categories.

Teaching System	Mechanics of programming	Expressing programs	Simplify typing code	Simplify the language	
				Prevent syntax error	
			Find alternative to Typing program	Construct programs using objects	
				Create programs using interface	
				Provide multiple methods for creating programs	
			Structuring programs	New programming models	
				Making new models accessible	
			Understanding program execution	Tracking program execution	
				Make programming concrete	
				Models of program execution	
Social Learning	Side by Side	Side by Side			
		Networked Instruction			
	Providing Reasons to Program	Solve problems by positioning objects			
		Solve problems using code			
Empowering System	Mechanics of programming	Code is too difficult			
		Improve programming language			
	Activities enhanced by programming	Entertainment			
		Education			

**Figure 5 Novice Programming Systems and Language Taxonomy**  
*(Adapted from Kelleher and Pausch, 2005)*

Gross and Power (2005) grouped the existing novice programming tools into five groups:

1. Micro-worlds
2. Visual programming environments
3. Flow model environments
4. Object workbench environments
5. Algorithm realization environments

The remainder of this section is divided into two sub-sections. The first sub-section discusses programming tools designed to help novice programmers and instructors in teaching and learning programming which use artificial intelligence. The second sub-section discusses applications which do not use artificial intelligence.

### **2.3.1.1 Intelligent Applications**

Intelligent applications in computer science education can be divided into three major groups: Intelligent Tutoring Systems (ITS), debugging aids, and intelligent programming environments. Each of these groups will be discussed from a pedagogical view, with a focus on its usage in computer science education and its drawbacks, in detail in this section.

#### **Intelligent tutoring systems**

Research on ITS has been carried out for more than three decades aiming to provide instruction as effectively as a human tutor does but with lower cost (Anderson and Reiser, 1985; Ong and Ramachandran, 2000; Song et al., 1997). The goal of ITS is to “provide the benefits of one-on-one instruction automatically and cost effectively” (Ong and Ramachandran, 2000). Research has shown that ITS is highly effective in increasing students’ performance and motivation (Anderson and Reiser, 1985; Beck et al., 1996), deepening cognitive development, and reducing time taken by students to acquire skills and knowledge (Anderson et al., 1995; Woolf et al., 2001).

ITS often consists of three modules: expert, student and pedagogical (Song et al., 1997). The expert module contains knowledge of a particular domain, the student module represents the current knowledge state of a particular student and the pedagogical module contains information on how to teach. There are three common interactive approaches adopted by ITS to interact with students (Merrill et al., 1992). The first approach is tracking - the tutor follows students while they are doing their work, tries to figure out what is correct and what is buggy and provides directive feedback to ensure that students do not flounder during problem solving. The LISP tutor (Anderson and Reiser, 1985) and the genetic graph (Goldstein, 1982) are examples of systems using this approach. The second approach is coaching which offers students suggestions upon request or when the system determines that an error could be grossly counterproductive (Burton and Brown, 1982; Lesgold et al., 1992). The last approach is model tracing in which students’ problem-solving steps are compared with the reasoning of an underlying domain expert.

In computer science education curriculum, many ITS have been developed to help novice programmers learn to program. One of the earliest ITS is the LISP tutor (Anderson and Reiser, 1985); other well-known ITS include: RAPITS (Woods and Warren, 1995), the C tutor (Song et al., 1997), MoleHill (Alpert et al., 1995) and

SIPLeS (Chee and Xu, 1997). Recently, with the rapid development of communication technology, web-based ITS have also been developed which include: JITS (Sykes and Franek, 2004), CIMEL Java tutor (Blank et al., 2005) and ELM-ART (Weber and Brusilovsky, 2001).

Research has shown that ITS improve novice programmers' understanding and performance. In the LISP tutor experiment (Anderson et al., 1995), students who used the system scored 43% higher in the final exam than a control group that received traditional instruction. With a complex problem, students in the control group required 30% more time to solve the problem compared to those who used the system.

Even though ITS is successfully used to provide one-on-one tutoring for students, only a few of them are used in the classroom (Pillay, 2003). This is mainly due to the high cost in developing such a system (Pillay, 2003; Suthers, 1996). Compounding this is the lack of sharable components (Mizoguchi and Bourdeau, 2000); most of the existing ITS are developed from scratch. Murray and Woolf's investigation (1992) showed that 100 hours of development translate into one hour of construction. Beck, Stern and Haugsjaa (1996) suggested that the development of authoring tools to provide simple development environments will reduce the number of developers and a modularised intelligent tutoring system will allow new systems to reuse some existing components thus the cost of the development will be reduced. However, they also pointed out problems in modularising these systems including: differences in implementation languages; no unified standards for ITS modules and their contents; no protocol to communicate between those modules.

## **Debugging Aids**

Debugging aids are applications designed to allow students to test, observe program execution, detect and correct errors. These systems are designed to help novice programmers in the debugging task which is well acknowledged as one of the barriers that novice programmers encounter (Ahmadzadeh et al., 2005; Gugery and Olson, 1986). Some popular debugging aids developed for novice programmers are: LAURA (Adam and Laurent, 1980), PROUST (Johnson and Soloway, 1985), TALUS (Murray, 1988), CAP (Schorsch, 1995) and jGRASP (Cross II, 1999).

Debugging aid systems can be grouped based on the debugging strategy that a system uses which can be: filtering, checking computational equivalence of the

intended program and the actual one, checking the well-formedness of actual program and recognising stereotyped errors (Ducasse and Emde, 1988). Alternatively, they can also be classified based on the knowledge used: these classifications are knowledge of intended program, knowledge of actual program, understanding of the programming language, general programming expertise and knowledge of application domain.

### **Intelligent Programming Environments**

Intelligent programming environments (IPE) are systems which provide functions to edit, compile and run programs together with other artificial intelligent features such as adaptive instruction, monitoring and assessment of students' progress, feedback and advice tools that used in the program development process. These systems are designed to provide guided environments to help novices acquire programming skills through a series of problem-solving situations. In contrast, programming environments which do not make use of AI help novices in obtaining effective programming knowledge through a discovery-like process. Some noticeable intelligent programming environments for novices are: DISCOVER (Ramadhan, 2000), ELM-ART (Brusilovsky et al., 1996; Weber and Brusilovsky, 2001) and INTELLITUTOR II (Ueno and Inoue, 1999). The main drawback of IPE is that they tend to ignore the importance of pre-problem solving and fail to simulate novices understanding on programs' dynamic behaviour of the underlying computer system in relation to the programming language (Ramadhan, 2000).

Overall, this section has overviewed the pedagogical perspective and cognitive impact of ITS, debugging aids and IPE on novice programmers together with well-known applications for each category and its drawbacks.

#### **2.3.1.2 Non-Artificial Intelligent Applications**

In parallel with tools that make use of artificial intelligent there are applications which do not provide any intelligent functionality. This section places these into two large groups. The first group is programming environments and the second group contains other applications which do not support the full programming development process. These applications include: simplified programming languages, pedagogical IDEs and software visualisation tools.

## **Programming environments**

As described earlier, programming environments are applications designed to support the whole program development process (edit, compile, run and debug). However, unlike intelligent programming environments, these environments do not incorporate functionality to provide adaptive learning and other intelligent functions.

With the rapid development of the internet and communication technologies, the majority of programming environments nowadays are web-based and aim to provide ‘anywhere, anytime’ access for students. Some noticeable web-based programming environments for novice programmers which support high-level OO languages such as Java, C++ and C# are: WWW for teaching C++ (Hitz and Kogeler, 1997), web exercises for C++ (Elenbogen et al., 2000), ALECS (Carver, 1996), WIPE (Efopoulos et al., 2005), CourseMaster (2000) and Ludwig (Shaffer, 2005).

Recently, extreme programming methodology has become fashionable and “pair programming has become a hot topic both in academia and industry” (Cliburn, 2003). The main benefit of students working collaboratively is that they can undertake more complicated problems with better understanding of the material. This is because during the working process, students can discuss with their peers. They can explore a larger set of hypotheses, and this increases the social aspect of learning to program.

Evidences of positive effects of collaboration in computer science education have been shown in many studies (Cliburn, 2003; DeClue, 2003; Murphy et al., 2002). As stated by Murphy et al. (2002):

Incorporating active and cooperative learning into the computer science classroom can be challenging but well worth the effort, particularly because it addresses issues that are unique to computer science education; it provides a mechanism for coping with the ever-increasing complexity and change inherent in the discipline, can serve as a tool for dealing with the wide range of background and abilities exhibited by our students, and it can facilitate a safe and positive learning environment for students from underrepresented groups.

McDowell and colleagues (2002) have shown that students who work in pairs perform better on programming projects than students working alone. Chase and

Okie (2000) discovered that peer instruction and cooperative learning in introductory programming courses decreased the rate of withdrawal and failure in the course and that this is especially beneficial for weak students. For the coming decades, collaborative learning will be taking the lead and some of these collaborative learning environments will be: GREWPtool (Taneva et al., 2003), W4AP (Vazhenin et al., 2005), the Gild collaborative IDE (Cubranic and Storey, 2005), SICAS (Marcelino et al., 2004), OOP-Anim (Esteves and Mendes, 2003), COLLEGE (Mendes et al., 2005).

### **Other applications**

Besides programming environments, there are three other major types of tools developed to make learning to program easier including simplified programming languages, pedagogical IDE and software visualisation.

The main motivation behind simplified programming languages is that high-level OO industrial programming languages such as C++, Java and C# are not designed with novice programmers in mind (Kelleher and Pausch, 2005). They contain many abstract concepts and features that are not necessary for novice programmers. Rather, they are designed for maximising the productivity of professional programmers. Meanwhile, the aim of pedagogical IDE is to eliminate the time that novices have to spend on learning how to use a programming editor. Lastly, software visualisation is designed to help novice programmers to understand abstract concepts. The remainder of this section will discuss each group of tools, their aims and effect on students, together with examples of well-known applications in each group.

Industrial programming languages such as Visual Basic, C, C++ and Java have a variety of syntactic elements that can be difficult for novice programmers. Much research has been conducted to develop customised programming languages or libraries for novices. These novice programming languages have simpler syntax and are easy to learn, as well as customised libraries that perform abstract concepts helping students to understand it better. However, the key important factor in designing such novice programming languages and customised libraries is that they also need to make it easier for students to move from these teaching languages to industrial languages. Some well-known teaching programming languages for Java are Blue (Kolling and Rosenberg, 1996), JJ (Motil and Epstein, 1998), Kenya

(Chatley, 2001) and customised Java libraries developed to make a particular task easier such as TerminalIO, BreezyGUI (Osborne and Lambert, 2000), Java Power Tool (Proulx and Rasala, 2004) and Java Power Framework (Proulx and Rasala, 2004) and simpleIO (Wolz and Koffman, 1999).

Much research has shown that apart from battling to understand and remember programming language syntax, novice programmers are also having difficulty in using a programming editor to edit their programs. Professional editors such as Visual Studio or Eclipse are too complex for them to use. Due to that reason, many pedagogical IDE are designed to make it easier for novice programmers. Studies have shown that simplified pedagogical development environments can reduce anxiety and uncertainty that novice programmers have (DePasquale et al., 2004). Some well-known pedagogical programming editors for novice programmers are: BlueJ (Barnes and Kölling, 1999), DrJava (Allen et al., 2002), GILD (2005), BeanShell (Niemeyer, 2000), ProfessorJ (Gray and Flatt, 2003) and JavaMM (Bettini et al., 2004).

Software visualisation is used to represent abstract concepts in a graphic and interactive way (Brown, 1988; Stasko, 1990). Research and development in software visualisation is relatively young (Costelloe, 2004): it was first used in computer science education in the 1980s and many developments have been carried out in this area from the 1990s onward. In computer science education, software visualisation is often used for the following seven purposes (Bergin et al., 1996):

1. Clarifying complex concepts;
2. Building mental model of concepts;
3. Engaging students;
4. Increasing students' understanding;
5. Helping students to better comprehend concepts;
6. Handling students from different backgrounds;
7. Allowing instructors to cover more material in less time.

Research has shown that visualisation software improve students' learning outcomes (Naps et al., 2002). As stated by Naps and others (2002), visualisations "have cognitive potential which is effective if exploited by the learning environment to create a fructuous interaction between the student and the visualization".

According to Soloway (1986), novice programmers have difficulty in “putting the pieces of [code] together to solve a programming problem”. Visualisation can help learners understand many interactive abstract concepts which cannot be achieved through textbooks such as what a program does, why it does that, how it works and what the result of that process is (Wiggins, 1998). Visualisation helps beginners perceive how pieces of code are put together (Wiggins, 1998).

Existing software visualisation can be placed into three groups: program visualisation, algorithm animation and data visualisation (Bergin et al., 1996). Program visualisation such as ZStep94 (Lieberman and Fry, 1995), Alice (1999), EROSI (George, 2000), Jive (Cattaneo et al., 2004) and JKarelRobot (Buck and Stucki, 2001) focus on graphical representation of an executing program and are designed to help students understand program code. Algorithm animation such as The Sort Animator (Morris, 2004), AAPT (Sanders, 1991), VisMod (Jiménez-Peris et al., 1999), JHAVE (Naps et al., 2000), Java and Web-based Algorithm Animation (Pierson and Rodger, 1998) are used to show concepts of an algorithm. Data visualisation such as LIVE (Campbell et al., 2003), TANGO (Stasko, 1990), JVALL (Dershaw et al., 2002), SAMBA (Stasko, 1997) shows operations inherent to an abstract data type.

From the instructors’ perspective, even though research has proved that visualisation tools are beneficial for student learning, it might cause too much overhead to make it worthwhile to integrate one into the curricula (Naps et al., 2002).

In summary, this section has overviewed the research and development of tools designed to help novice programmers learning to program. ITS, debugging aids and IPE are three major research fields which make use of artificial intelligence while programming environment, simplified/subset of industrial programming languages, pedagogical IDE and software visualisation are the four large groups of research and studies which do not adopt artificial intelligence. In the next section, applications designed to help teaching staff in managing and marking students’ assessment are discussed.

### **2.3.2 Computer Assisted Assessment**

Computer Assisted Assessment (CAA) applications can be used to manage and mark various types of student assessments. These applications are beneficial for both students and teaching staff from both administrative and pedagogical perspectives.

From an administrative perspective, CAA applications reduce the amount of time and effort in the marking task, and increase objectivity and consistency of marking (Carter et al., 2003; Lewis, 2004; Mason and Woit, 1999; Seale, 2002). From the pedagogical perspective, CAA applications allow students to work at their own pace (Carter et al., 2003), monitor their own progress (Seale, 2002) and receive immediate feedback (Malmi et al., 2002; Seale, 2002). Additionally, assessments provided through CAA tools allow instructors to incorporate hints and other related materials easier; they can monitor students' progress and assign different learning activities based on their test results (Lewis, 2004; Seale, 2002). CAA allows students to learn from feedback, resubmit their work which supports their learning, carry out self assessment (Malmi et al., 2002), and it facilitates individual questions based on their levels. This fits with the educational principles of constructivism: allowing students to learn from their mistakes and future learning being contingent upon them (Malmi et al., 2002).

According to a survey conducted by the CAA Centre (University of Luton, 1998), CAA is most commonly used in computing, biomedical science, mathematics, engineering and modern languages (Seale, 2002). In computer science, CAA is commonly used in introductory programming courses rather than in intermediate and advanced courses and it is used for summative assessment more than formative assessment (Carter et al., 2003).

Automatic grading systems were used in computer science education as early as in the 1960s. One of the earliest automated grading systems is the Grader1 (Forsythe, 1964) designed to analyse ALGOL and its dialects; since then many applications and studies have been conducted in this area. Besides the advantages of CAA mentioned above, the use of CAA in computer science education also allows teaching staff to set up more assessments using different methods at various stages of a course to assess different range of knowledge despite large class sizes (Bull and McKenna, 2004; Hollingsworth, 1960). Additionally, CAA applications play an important role in motivating students with the capability of providing immediate feedback, increase the objectivity, and consistency in marking tasks.

Similar to computer assisted instruction, CAA research is also influenced by the increasing usage of information and communication technologies to enhance learning. Many on-line, web-based computer assessment tools have been developed. Besides all the advantages of web-based assessment inherited from web-based

learning mentioned above, web-based assessment provides great interactivity; these applications can provide instant formative feedback on progress (Foulkes and Thomas, 2001). Web-based assessment “offers opportunities to provide rich feedback which is then linked back to specific activities and can be used to raise issues for discussion in a linked on-line environment” (Foulkes and Thomas, 2001). Research has shown that careful planned on-line assessment can encourage students to focus on lower-level cognitive learning (James et al., 2002). In Australia, many universities adopt on-line assessments to diversify assessment tasks, broaden the range of skills assessed, and provide students with more timely and formative feedback on their progress. Others wish to meet student expectations for more flexible delivery and to generate efficiencies in assessment that can ease academic staff workload (Curtis, 2000).

Similar to computer assisted instructions, there are many ways to classify the vast number of existing CAA systems. These systems can be classified based on whether it fully automates or semi-automates the marking process, the types of questions that it supports, the type of analysis that it carries out or the type of assessment it is used for. According to Carter et al.’s study (2003), in computer science education CAA applications often support the following type of questions: (1) multiple choices, (2) textual answer, (3) programming assignment, (4) visual answers such as where students are required to translate a program into a flowchart and (5) peer assessment.

A majority of systems which support the programming assignment type of exercises adopt one or both types of analyses to assess students’ work: static and dynamic. Static analysis is the process of examining source code without executing it, while dynamic analysis involves executing a program through a set of test data. A recent study from Ala-Mutka (2005) has overviewed common approaches adopted by existing CAA applications in computer science education to assess students’ programming exercises. With dynamic analysis, students’ programs are checked on functionality, efficiency, testing skills and other features of programs (Ala-Mutka, 2005). With static analysis, students’ programs are assessed on coding style, programming errors, design skills, software metrics and other aspects of a program such as the use of certain functions of a library (Ala-Mutka, 2005). Table 2 summarises some recent CAA tools used to assess students’ programming exercises and provide immediate feedback for students during their learning process. As shown

in the table, these systems are compared based on supported types of questions, automation types, types of analyses, supported assessment types and whether it is an on-line or stand-alone application.

	Question types				Automation		Analysis		Assessment			Mode				
	Multiple choice	Textual answer	Programming	Visual answer	Peer assessment	Fully	Semi	Static	Dynamic	Formative	Summative	Self	Exam	Other	On-line	Stand-alone
ASSYST (Jackson and Usher, 1997)		✓			✓		✓	✓	✓							✓
CourseMaster (CourseMaster, 2000)		✓	✓			✓	✓	✓	✓							✓
BOSS (Luck and Joy, 1999)		✓				✓			✓	✓						✓
Online Judge (Cheang et al., 2003)		✓				✓			✓	✓						✓
Web-CAT (Virginia Tech, 2003)		✓				✓	✓	✓	✓	✓						✓
WebToTeach (Arnow and Barshay, 1999)		✓				✓			✓	✓		✓				✓
InSTEP (Odekirk-Hash, 2001)		✓				✓	✓	✓	✓			✓				✓
WWW for C++ (Hitz and Kogeler, 1997)		✓				✓			✓	✓		✓				✓
Datlab (MacNish, 2000b)		✓			✓				✓	✓						✓
GAME (Blumenstein et al., 2004a, 2004b)		✓			✓		✓		✓	✓	✓					✓
TRY (Reek, 1989)		✓			✓				✓	✓						✓
WebMCQ (Dalziel and Gazzard, 1999)	✓															✓
Quiver (Ellsworth et al., 2004)			✓			✓			✓	✓						✓
Automatic Exams (Frosini et al., 1999)	✓		✓			✓		✓	✓			✓				✓

Table 2: Computer Assisted Assessment Tools

Despite the benefits of CAA, it also has five major drawbacks which make it unpopular in the classroom. Firstly, it relies on technology therefore either downtimes or system failures might lead to corrupt or lost data (Carter et al., 2003). Secondly, instructors may be concerned about security and plagiarism issues when CAA is used for formative assessment purposes, especially in distance learning where it is difficult or impossible to verify the identity of the person who undertakes the assessment (Carter et al., 2003). With the growing demand of on-line and offshore learning, plagiarism has become a crucial research and development field. Some notable plagiarism detection applications are JPlag (Prechelt et al., 2000) and Moss (1994). Thirdly, it is time consuming and costly to set up exercises and assessment in CAA systems. Fourthly, the quality of feedback in some of the systems is quite poor. Lastly, some on-line marking systems are inflexible; markers feel powerless when assessing students' work especially when the marking criteria does not apply to students' work (Preston and Shackelford, 1999).

So far, this chapter has reviewed applications developed to assist students and educators in teaching and learning programming. In the next section, future directions of computer science education applications are discussed.

## **2.4 Future Direction of Applications in Computer Science Education**

Internet technology has led education into a completely new era. Learners have shifted from a “relative static, localized” learning world to an “interactive, dynamic and non-localized” world (Boroni et al., 1998). There is a radical shift in the teaching paradigm, from a teacher-centred to student-centred approach. Students’ expectations and how they learn have changed.

In the last decade, there has been a significant growth in the use of the internet in computer science education and other education areas. This adoption allows universities to make education more accessible, improve learners’ learning experience and reduce the cost of education (Ibrahim and Franklin, 1995; Le and Le, 2001; Owston, 1997). This trend will continue and become more and more popular. In the future, there will be more on-line courses, distance learning environment and e-learning. Predictions of the future of computer science education in the 21<sup>st</sup> century include:

Classroom lecture as the primary method of delivery of course information will decrease. Schools are opening up to students at greater distances who are not able to commute to the college campus. Greater emphasis will be placed on team efforts and capstone or real-world activities in courses and degree requirements. In introductory programming courses, team projects will take predominant. (Curtis, 2000)

The primary driver of change in our 21st century education system will be the creation of online courses that will remove the responsibilities for teaching academic subjects from teachers. (Schank, 2000)

Similar to other countries around the world, Australia is also currently in “the first major shift in teaching and learning methodology” where learning could be available for anyone, anytime, anywhere (DEST, 2006). Australia is acknowledged as one of the world leaders in distance education at all levels (DEST, 2006). However, providing accessibility and flexibility for students and teachers for their own specific teaching and learning needs remain key challenges (DEST, 2006).

E-learning provides five substantial gains in effectiveness, quality and cost benefits, including:

1. Classroom interactive learning, namely, interactions between students and teachers and among students;
2. Independent learning whereby students or teachers can learn and study alone in a variety of environments and modes including aspects of self directed lifelong learning;
3. Network learning through contact with groups, individuals and sources where different influences and experiences create qualitative differences to both standard and blended teaching and learning;
4. Organizational learning that creates learning communities, learning precincts and learning cities;
5. Managed learning whereby education technology is created through computer managed communication and learning management systems, which allows

teachers to provide individualised curricular and learning experiences for each student.

Some well-known e-learning systems that have been used in many institutions include Blackboard (1997), WebCT (Murray, 1995), Bodington (1997), Moodle (Dougiamas, 1998) and FirstClass (Open Text Corporation, 1990).

This chapter has overviewed the current trends and future directions of research and applications in computer science education. In the next two sections, prior web-based programming environments and program analysis applications which have great impact on the design of the ELP system and the program analysis framework of the research are discussed.

## **2.5 Web-Based Programming Environments**

As mentioned earlier, there are many applications which have been developed to support teaching and learning programming. This section describes programming environments which have been developed to support beginning students to learn to program and which have great impact on the design and implementation of the ELP system. Those systems are: CodeSaw (Liqwid Krystal, 1999), CourseMaster (University of Nottingham, 2000), CodeLab (Turingscraft, 2002), InSTEP (Odekirk-Hash, 2001), WebToTeach (Arnow and Barshay, 1999), WWW for Learning C++ (Hitz and Kogeler, 1997), W4AP (Vazhenin et al., 2005), VECR (Perry, 2004) and JavaMM (Bettini et al., 2004; Cecchi et al., 2003).

### **2.5.1 CodeSaw**

CodeSaw is a commercial on-line digital workspace designed to provide technology learners with a dynamic environment to connect new knowledge with real experience. Code snippets from an IT textbook are embedded within the CodeSaw framework, resident on the CodeSaw server. Learners can view, compile and run all these examples without the need to install a program development environment or having to type in code.

In order to use CodeSaw, students must register themselves at [www.codesaw.com](http://www.codesaw.com) and install the CodeSaw framework on their machines. After CodeSaw is installed, students can browse a list of books supported by CodeSaw on the system homepage. They can click on any example to load the code into the

CodeSaw system; compile and run it. Once compiled and run on the server, the program output is presented for the learner. Currently, CodeSaw supports “instantaneous” learning for C, C++, Java, Perl, Python, Ruby, XML, HTML, PHP, SQL, PL/SQL and JavaScript. With the current version of the CodeSaw, students will not be able to modify the code example.

## **2.5.2 CourseMaster**

CourseMaster (University of Nottingham, 2000) is a client-server system for delivering course-based programming. The system is an improved version of the Ceilidh (Foxley, 1999) system. It provides functions for automatic assessment of students’ work, detection of plagiarism in students’ programs, administration of the resulting marks, solutions and course materials. The marking component of the system is capable of marking Java, C++ programs, Object Oriented languages design, flowcharts and logical circuits (Higgins et al., 2002).

The marking system provides six tools. These are the:

1. Typographic tool which checks program sources for typographic layout;
2. Dynamic tool which tests the solutions’ behaviour by running them against test data;
3. Feature tool which is designed to inspect specific features of students’ programs;
4. Flowchart tool which converts students’ flowcharts to BASIC programs that can be subsequently marked with the use of the Dynamic Tool;
5. Object-oriented tool which checks the students’ OO diagrams for OO analysis and design;
6. CircuitSim tool which simulates the students’ logical circuits.

For programming exercises, student programs are checked for typographic layout, required features, and run against test data. Immediate feedback and results are provided for students. Feedback includes comments on how to improve the solution with links for further reading material. The marking scheme can be configured by teaching staff. They can specify if marks will be displayed in numeric or an alphabetic scale as well as the level of detail in the feedback.

Similar to CourseMaster, the ELP system supports multiple programming languages Java and C#. Unlike CourseMaster, the ELP system provides feedback for students about both quality and correctness of their program.

### **2.5.3 WebToTeach**

WebToTeach (Arnold and Barshay, 1999) is a web-based automatic homework checking tool for computer science classes. For students, it provides a self-paced experience with the fundamental elements of programming; it gives immediate feedback and hints on problem solving. WebToTeach supports Java, C, C++, FORTRAN, Ada, and Pascal and incorporates various types of exercises including writing code fragment, writing data for a test suit, writing a complete single source program and writing several source files.

Students are given either a single text area or multiple text areas on the web browser to provide the solution depending on the exercise type. Upon submitting the solution for the exercise, students are told immediately whether it is correct. In the case of failure, the student is given information about the cause of failure.

WebToTeach also provides facilities for instructors to forgive lateness, view the first and last correct submission of a student, directly send mail to an individual student, broadcast mail to students, set and edit the message of the day, and get statistics on homework completions. The two main advantages of WebToTeach are that it is able to analyse program fragments and it may easily be used in other computer science departments. The major drawback of the system is not providing feedback on the quality of a student's program.

WebToTeach is one of the early systems supports analysing “fill in the gap” programming exercises. However, in this system, the provided code of an exercise is hidden from students; hence, the system only supports one gap per exercise. Students are presented with a textbox to provide the missing code. The system has a built-in dynamic analysis framework to verify the correctness of students' programs but leaves the quality factor of a program unattended. With the program correctness feedback, the WebToTeach system does provide some mechanisms to reduce the sensitivity in comparing student program outputs with the expected model solution but does not have any way to separate the significant information and insignificant information in the program outputs.

#### **2.5.4 CodeLab**

CodeLab is a web-based-interactive programming exercise system designed to teach students programming in Java, C++, C and other languages. The system is the improved version of the WebToTeach system. It supports “fill in the gap” type programming exercises but the provided code is hidden. Students are presented with an editable textbox to enter their answers to programming exercises. Complete programs are built in the server from the student solution and compiled. If there is no compilation error, the program is checked for correctness, otherwise, customised compilation error messages are given to students. If the student solution does not produce the expected result, the student is provided with hints which are specific to the submission to fix the problem. Students can repeat an exercise several times or move on to another exercise after their first correct answer for the exercise.

Instructors can monitor students’ progress through an on-line roster of students registered in the course. They can view the most recent submitted solution for each of the assigned exercises of a student together with its correctness. However, instructors do not know the number of submissions a student entered or how long the student worked on a problem.

Currently, the system has more than 200 short exercises focused on various programming areas and organised from simplest to more difficult. The exercises range from as simple as declaring a variable or an assignment statement to more complex problems involving loops, functions or methods or even small class definition. Additional exercises can be added using the Instructor Exercise Editor. CodeLab determines the correctness of a program by comparing machine states of the model solution and the student one in special cases which are specified by the instructor. The system does not check for coding style and program efficiency. The system is available for trial at [www.turingscraft.com](http://www.turingscraft.com).

#### **2.5.5 InSTEP**

InSTEP is a web-based tutoring system designed to help students learn looping in C, C++ and Java. Students complete a program exercise by filling writable fields in a web form and submitting it for evaluation. InSTEP compiles the student’s solution in the server. If the student solution is successfully compiled, it is run against a set of test cases; the program’s outputs are captured to scan for mistakes. Otherwise, compilation error messages are returned; the student needs to fix those compilation

errors and resubmit. If the program outputs match with the expected outputs, no further analysis is performed; the program is considered to be correct and feedback is returned. Otherwise, the program's outputs are parsed into an output structure to analyse for a set of known common errors. If the error is not obvious from the program's outputs, each block of code that the student put into a field is analysed. When the analysis is completed, InSTEP sends the analysis back to the student as a web page.

A specialised module is required for each exercise in the InSTEP system. This module is used to generate test inputs, set maximum numbers of characters and line inputs, check the output for correctness, and analyse the output and code for errors.

Since only students' input fields are parsed to detect errors, InSTEP can misdiagnose a student's problem because it does not understand program code. Another drawback of the InSTEP system is that it cannot analyse arbitrary programming exercises.

### **2.5.6 WWW for Learning C++**

Web-based courseware was developed at University of Vienna to support an introductory programming course in C++ for first year business informatics students (Hitz and Kogeler, 1997). Called 'WWW for Learning C++', the system provides a seamless integration among textbook, references and practical exercises of course materials. The materials are organised into three semantic layers: main, reference and pragmatics. The main layer consists of a series of sequential lessons followed by a set of exercises. The reference layer contains syntactical details for each major feature of the language. The pragmatics layer is a series of auxiliary stand-alone pages which elaborate on the background of a specific feature dealt within the lesson.

Programming exercises in the main layer are provided in the "fill in the gap" form and presented on the browser as HTML form. After completing an exercise, students can submit the exercise to the server for compilation. If there is a compilation error, the exercise is displayed together with compilation error messages as close as possible to the line where the error occurs. Otherwise, students are prompted for inputs for the program. The program is run on the server and its output is presented to the student.

### **2.5.7 W4AP**

The web-based W4AP system (Vazhenin et al., 2005) is designed to support applied programming, grid computing and distance learning. The discussion here focuses on the distance learning component of the system; the applied programming and grid computing components are detailed in Vazhenin and Vazhenin (2001) and Vazhenin et al. (2002).

The W4AP programming environment supports all three stages of program development processes: edit, compile and run in numerous programming languages including Java and C. Users can work on a program alone or collaborate with others and execute the program. Functions available for collaboration are: message exchange subsystem, file/data sharing and CVS-mechanisms. After students complete their projects, they can submit them to the W4AP server for compilation and verification for correctness; the testing results are presented to students and teaching staff.

### **2.5.8 JERPA**

JERPA (Emory and Tamassia, 2002) is a web-based Java environment for remote programming assignments. It is designed to eliminate technical issues that students might encounter when doing their assignments, such as setting up environment variables and enable them to complete their assignments outside campus. The system also helps teaching staff in designing assignments with support materials. Instructors do not have to worry about compiling and distributing assignment material to students and explaining to them how to install it. Through the system, the assignment of courses and their associated support materials are accessible through the web.

The system consists of two main components: a student client and an administration client. The student client does not serve as a full Java development environment but provides functions that allow students to install and compile the assignment on their machines, run the demonstration of the completed assignment or the student's work, export all the provided code to a third party IDE, and prepare for submission where all the student codes are packed into a ZIP file.

The administration client is designed to allow course instructors and administrators to define the assignments and their support material, and install them on the JERPA server. It has four components: courses, assignments, support library and upload options. System administrators can create an unlimited number of courses

for storage in the JERPA server using the courses component. Similarly, they can add support libraries using the support library component. The assignment component allows teaching staff to set up assessment for a course. The file upload options component uses customised upload script to upload files from various platforms to the server.

### **2.5.9 VECR**

The View Edit Compile Run (VECR) web-based environment is developed to support students to perform C, Java and shell programming. Functions provided by the system include: view files, edit source code, compile, run (in debug mode, with output plotted), display output as a GIF image, display C processor output, display generated assembly code and insert compiler error messages as comments into the source code.

Students are required to authenticate themselves in order to use the system. After successful login, they are presented with all the projects that are assigned to them. Students can either do an exercise directly on the system or download it to work locally on their preferred IDE, then upload the exercise back to the server for testing. Exercises are presented to students in a web browser and students can provide their answers in the editable text area on the page. After completing an exercise, students can submit it to the server for compilation. On successful compilation, students can run the program either in normal mode or interactive mode, and debug it. Additionally, they can also plot the program outputs to a 2D or 3D GIF image.

The system also allows instructors to log in as a particular student to view all the student's projects. This is useful for helping students in debugging their programs and for grading finished projects.

### **2.5.10 Ludwig**

Ludwig (Shaffer, 2005) is a web-based system which allows students to edit their C++ programs in a controlled text editor, compile, analyse the quality and correctness of their programs and submit them for grading.

The programming editor is presented as a Java applet with cut and paste functions disabled. On successful compilation, student programs are checked for style such as program indentation, cyclomatic complexity, and global variables; these

style analyses are configured by instructors. If the program passes the style check, it is run against a set of test data specified by either students or instructors. The student program output is compared to the model solution outputs and differences are displayed to the student. The student then has the option to either continue modifying the program or to submit it.

Once the student submits the program for assessment, the system will create its own test data, re-run the student program and display the results to the student. Besides programming exercises, the system also supports four types of multiple choice questions: standard multiple choices in which all questions are displayed at once, multiple choices with one question displayed at a time, adaptive tests and word bank tests.

### **2.5.11 ALECS**

ALECS stands for ‘A Learning Environment for Computer Science’ (Carver, 1996); it is designed to assist teaching staff in teaching C++ programming language and fundamental concepts of computer science. ALECS provides lessons which consist of hypertext, graphics, voice annotations, animations, case studies and closed labs. The system provides feedback to students about their quizzes, practice exams and programming assignments which are tested against a set of instructor provided test data and analysed for coding style. The system tracks students’ progress and records this in the database.

## **2.6 Program Analysis and Automated Marking Systems**

In this section, some prior work on program analysis and automated marking systems are detailed. These systems are: ITPAD (Robinson and Soffa, 1980), Expresso (Hristova et al., 2003), TRY (Reek, 1989), BOSS (Luck and Joy, 1999; Joy et al., 2000), Datlab (MacNish, 2000a; 2000b), ASSYST (Jackson and Usher, 1997), Web-CAT (Virginia Tech, 2003), GAME (Blumenstein et al., 2004a, 2004b), HomeWork Generation and Grading Project (Morris, 2003), PASS (Thorburn and Rowe, 1997), IRONCODE (Bailey, 2005) and Online Judge (Cheang et al., 2003).

### **2.6.1 ITPAD**

The ‘Instructional Tool for Program Advising’ or ITPAD (Robinson and Soffa, 1980), is designed to assume some of the instructor’s duties. The system supports the FORTRAN programming language. It is able to keep student and assignment

profiles, detects plagiarism, and provides suggestions to students for improving their programs. ITPAD provides five main functions. Firstly, it examines students' programs for certain design structures. Secondly, it evaluates students' progress through a term. Thirdly, it evaluates the programming assignments. It also detects plagiarism. Lastly, it gives suggestions about how the student may be able to improve his or her program structure. Code optimisation, optimisation techniques and Halstead's software metrics are used to determine the characteristics of a student's program. The model answer of a programming exercise is available to ITPAD. By comparing the student program with the model solution, ITPAD can evaluate the use of control and data structures, and the conciseness of the implementation of the student program.

## **2.6.2 Espresso**

Expresso (Hristova et al., 2003) is designed to detect a set of common student Java syntax, semantic and logic errors and provides hints on how problems can be fixed. The Java common errors supported by the system were collected from 58 schools listed in the US News, World Report's Top 50 Liberal Arts Colleges for 2002 and members of the Special Interest Group on Computer Science Education. Overall, thirteen syntactic, one semantic and six logic errors were identified as the most common mistakes among novice programmers. These mistakes are shown in the table below.

Errors	
<b>Syntactic</b>	<ol style="list-style-type: none"> <li>1. confusing between = and ==</li> <li>2. confusing between == and .equals</li> <li>3. mismatching, miscounting and/or misuse of {}, [], (), " ", and ''</li> <li>4. confusing between &amp;&amp; vs. &amp; and    vs.  </li> <li>5. inserting ; after the parentheses defining if, for, or while conditions</li> <li>6. using , as separator in for loop instead of ;</li> <li>7. inserting the condition of if statement in {} instead of ()</li> <li>8. using keywords as method/variable names</li> <li>9. invoking method with wrong arguments</li> <li>10. forgetting () after method call</li> <li>11. incorrect ; at the end of a method header</li> <li>12. leaving a space after a period when calling a specific method</li> <li>13. &gt;= and =&lt;</li> </ol>
<b>Semantic</b>	<ol style="list-style-type: none"> <li>1. invoking class method on object</li> </ol>
<b>Logic</b>	<ol style="list-style-type: none"> <li>1. improper casting</li> <li>2. invoking a non-void method in a statement that requires a return value</li> <li>3. flow reaches end of non-void method</li> <li>4. confusion between declaring and passing method parameters</li> <li>5. incompatibility between the declared return type of a method and in its invocation</li> <li>6. class that implements some interfaces but missing methods that the interfaces must define and support.</li> </ol>

**Table 3: Students' Common Java Errors Supported by the Expresso System**

In order to identify these errors in a program, the program is parsed three times. The first time is to remove comments, record line numbers and store the resulting characters in a vector. White spaces are removed and the file is tokenised into words which are stored in another vector in the second parse. Errors are detected and error messages are generated in the final pass.

### 2.6.3 TRY

TRY (Reek, 1989) is a software package for the UNIX operating system that tests student programs from any programming languages. The system takes the student's source file, compiles and runs it on a predefined set of input files. The system automates the program execution by using UNIX script. The output produced from each run is compared with a predefined set of output files using a small awk program. Differences are reported to the student. A utility program is provided to do some filtering of blank spaces and line breaks to make the comparison less sensitive. The two main disadvantages of TRY are the requirements to write script to automate execution of the programs and that TRY is only available for the UNIX operating system.

## **2.6.4 BOSS**

The BOSS on-line submission system is a course management tool allowing students to submit their coursework on-line including programming work. With programming work, the system supports the submission, testing and marking of programming assignments, providing feedback to students and detecting plagiarism. The first version of the system is described in Luck and Joy (1999). In this section, the newer implementation of the programming marking component of the system, Boss2 (Joy et al., 2000), is detailed.

Students submit their programs through a web submission interface and an email is sent on successful submission. Automated testing is provided by using either shell scripts or JUnit (Gamma and Beck, 2000), an extreme regression testing framework for OO programs. When JUnit is used, the student program objects are compared with the expected result objects. If shell scripts are used, the program output is compared with the expected output by using the Linux `diff` command or hand-coded shell scripts. The test results are incorporated into the mark sheet directly. If the program passes all tests, full marks are given; otherwise, no mark is awarded. In this case, tutors or lecturers need to adjust the mark manually later on. The automated test function is also available for students so that they can test their program themselves before submitting it.

Additionally, the student program is also checked for plagiarism. This is done by comparing a pair of student programs against each other five times: (1) in their original form; (2) without white spaces; (3) without comments; (4) with all comments and without white spaces; (5) and lastly they are translated to a file of tokens. A token is a value such as name, operator, begin, loop, statement depending on the language (Joy and Luck, 1999). The system gives the student detailed feedback on how his or her program is marked. With assessment submission, extra individualised feedback is provided by teaching staff about the whole submission.

## **2.6.5 Datlab**

The datlab (MacNish, 2000b) system has been developed at the University of Western Australia for monitoring student progress in computer science laboratories and providing timely feedback on their Java programming work. The datlab system is continuously polling for requests at a regular interval. Students submit requests to the system by emailing the lecturer with datlab as the subject line. These emails are

then filtered and appended to the requests line of the datlab system. The system makes use of Java technologies to enable the process of running and static analysis of students' work. When the system finishes analysing student work, student records are updated and a report is emailed back to the student.

This technique has four major disadvantages, namely: (1) performance issues since long test sequences need to be carried out in order to discover errors; (2) lack of precision because on some occasions erroneous code may be unnoticed; (3) feedback is not formative; and (4) it is unable to monitor students' common errors (MacNish, 2000a). The current system is unable to predict why a method passes or fails.

### **2.6.6 ASSYST**

Assessment SYSTem, or ASSYST (Jackson and Usher, 1997), was developed to help instructors in marking Ada programs and to provide formative feedback to students. Students' programs are analysed with respect to five different aspects, namely, correctness, efficiency, style, complexity, and test data adequacy. The system performs both static and dynamic analyses.

The static analysis checks the style and complexity of students' compilable programs by adopting Berry and Meekings' (1985) style guidelines and McCabe's cyclomatic complexity (McCabe, 1976). With dynamic analysis, the program is executed against test data provided by both students and tutors. In addition, tutors produce a specification of the output that a correct program will be expected to generate. The UNIX Lex and Yacc tools are used to compare the student program output with the provided specification. The CPU is measured during the execution of the program to evaluate the efficiency. Lastly, ASSYST tests the adequacy of the test data provided by students by measuring the extent to which the test leads to statement coverage on execution. One major drawback of the system is that it is not platform independent since the dynamic analysis relies heavily on UNIX tools.

### **2.6.7 Web-CAT**

Web-CAT (Virginia Tech, 2003) is an on-line submission, automated grading and software testing teaching tool. Three main characteristics of the system are: pluggable services, customisable automated grading and feedback, and support for test driven assignments. The system is built on a plug-in architecture in which each

service is a sub-system, and additional services can be easily added. The system currently has four sub-systems: the Core, the Grader, the Admin and the Examples. The Grader sub-system is discussed in this section.

For each programming submission made to the Web-CAT Grader sub-system, the source file is compiled, and the generated executable is run against a set of test cases supplied by students to assess the correctness of their programs. If the program does not compile, error messages are returned. The student's submitted test cases are then run with the model solution to check the validity of these test cases. If any test case fails the model solution, it is considered to be an invalid test case. The model solution is also added with statement and branch coverage information to tests the student's submitted test cases code coverage. The system currently supports grading of Java and C++ programming exercises. With Java programming exercises, besides dynamic analysis, the system also makes use of other third party pluggable components such as Clover (Cenqua, Retrieved May 17 2006), CheckStyle (CheckStyle, 2006) and PMD (PMD, 2006) to carry out static analysis.

The Grader sub-system is also fully configurable by instructors. Teaching staff can adjust the automated grading and feedback generation processes to align with assignment objectives. Most importantly, the system support of test driven assignments allows students to submit many times as they develop their solutions; the number of submissions is controlled by teaching staff.

### **2.6.8 GAME: A Generic Automated Marking Environment**

The GAME (Blumenstein et al., 2004a; 2004b) system was developed at Griffith University to automate the marking of Java, C and C++ programming exercises. The system performs structural analysis and dynamic analysis. With structural analysis, three aspects of source code are examined: comment, indentation, and programming style. Commenting is measured by comparing total lines of code with the total number of comments. Indentation accuracy is calculated by comparing the number of key points which are correctly indented and the total number of key points in a program. Key points in a program are the beginning of functions, control statements and global variables. Programming style is checked by comparing the number of functions with the number of block comments and a tally of the number of “magic” numbers in a program. With dynamic analysis, the output of a student’s program is analysed for correctness using “keyword” or “ordered-keyword” strategy. The

“keyword” examines the student's program output for a keyword at any position while the “ordered-keyword” looks for an ordered set of keywords in the output. The summary of results for each student shows a total mark for programming style, structure and a total mark for correctness (compilation and execution) and a list of any warnings or compilation errors.

### **2.6.9 HomeWork Generation and Grading Project**

The HomeWork Generation and Grading system (or HoGG) (Morris, 2003) is developed to make the task of creating and marking student Java programming assignments more consistent with a low error rate. Assignment specification is given to students in the form of a software specification which lists a number of requirements. Each of these requirements is then translated into a Java method.

Students submit their program electronically via a dedicated machine. The student program is compiled on the server. If the compilation process is successful, the test driver then runs the student program's methods according to the test plan and the test output is stored in the database. The test output is evaluated by an evaluator and the result is also stored in the database. If the program is not successfully compiled, compilation error messages are stored on the database.

The reporter component of the framework is invoked to either generate an explanation to be sent to students or a report to be sent to teaching staff based on the evaluator results. The explanation for students describes the tests which were carried out, together with their score. The report for teaching staff contains the student's name, score and result for each of the class sections.

The system requires that the assignment specification and the test plan need to be written simultaneously. The system is implemented using Perl. It makes use of Java reflection to invoke the student programs and regular expressions to compare the student solutions with the model solutions.

### **2.6.10 PASS**

The Program Assessment using Specified Solution (PASS) (Thorburn and Rowe, 1997) is an automated assessment system designed to assist teaching staff in marking beginning students' C programs. The system analyses a compilable student program and provides information for staff about the correctness and design of a program; the marker then decides how many marks will be given for the program.

In order to evaluate the quality of a program, the provided model solution is translated into high-level pseudo-code called ‘plan’ which is represented by combining the hierachal function calls, together with each function description. The student’s program plan is extracted by identifying functions which are equivalent as those specified in the model solution plan. Two functions are considered as equivalent when they have the same signatures and produce the same outputs with the same inputs regardless of their names. Identified functions in the student solution plan are presented to the tutor. The function description is provided by tutors. It consists of the value ranges of each parameter and text description of the function. The output range is obtained by running the function against a set of random values in the input range.

### **2.6.11 IRONCODE**

IRONCODE (Bailey, 2005) is designed to help students write correct code at the earliest stage of their study by using the small programming problems approach. The system is currently capable of analysing C++ programming exercises. When students submit their solutions, which are small segment of code less than ten lines long, to the server, their solutions are compiled and linked with a test harness. If the solution fails to produce an executable program, the student is notified; students' answers together with error messages are presented. However, if the solution produces an executable program, the solution undergoes a series of test to evaluate its correctness. If the student program runs incorrectly, the students are not allowed to modify their answer but must seek help from teaching staff to complete the exercise.

### **2.6.12 Online Judge**

Online Judge (Cheang et al., 2003) is developed and used at the School of Computing Science at the National University of Singapore. It is designed to evaluate the correctness, efficiency and robustness of Java, C++ and C programs. Program correctness is assessed based on whether it produces outputs that matches with the pre-specified answer using a string matching technique. Program efficiency is determined by whether the program is able to produce its output within the limited time and memory resources. Extreme test cases are used to measure the robustness of the submission. Feedback provided to students from the system is currently quite limited; it cannot identify the source of error on a student’s submission.

## **2.7 Discussion and Summary**

In summary, this chapter has overviewed the most influential education concepts which have been used by many research and development endeavours in the computer science education field, including this research. These concepts include scaffolding, Bloom's taxonomy, and constructivism. The chapter has also overviewed major categories of tools designed to assist teaching and learning programming. It discussed the impact of technologies on the way students learn, the shift in teaching paradigm from a teacher-centred to learner-centred approach, and applications developed to support that shift. The last section of this chapter described web-based environments and program analysis tools which have a major impact on the research.

As stated in Chapter 1, there are three contributions of this thesis. The first contribution is the ELP web-based environment supporting “fill in the gap” exercises. The second contribution is a program analysis framework which can analyse “fill in the gap” exercises. The framework consists of two components: a static analysis and a dynamic analysis. The third contribution is an automated feedback engine which provides customised compilation error messages, and immediate, formative feedback about the quality and correctness of students’ programs. The feedback engine also gives information about students’ progress, their programming practices, and common logic errors which can be used to assist teaching staff.

The web-based environment is designed to reduce common novice programmer difficulties when they start learning to program, such as installing and setting up a compiler, and learning how to use a programming editor. The ELP facilitates constructivist learning through “fill in the gap” exercise, the web, and feedback. “Fill in the gap” type programming exercises address the lower three levels of Bloom’s taxonomy and enforces students’ program comprehension skills (Lister, 2000). This type of exercise allows instructors to introduce new programming concepts incrementally. It is recommended as a constructivist approach to teach computer programming (Wulf, 2005). The system is web-based which means all learning materials are accessible at anytime, anywhere and students can do their study at their own pace. This type of medium addresses different learning styles and allows information readiness for students to take ownership of their learning. With

the ELP, students are provided with individualised, timely and formative feedback about the quality, structure and correctness of their programs together with hints on how to improve them based on the program analysis results. The analysis results are processed by the automated feedback engine to be more ‘student-centric’ before being given to students. The engine also provides customised compilation error messages to assist students in achieving a compilable program. These various types of feedback allow students to learn and reflect in the context of the current exercise that they are working on. In addition, the feedback engine allows instructors to constantly assess students’ current progress. All these characteristics make the ELP system unique and distinguish it from other previous works.

Even though many tools have been developed, the ELP is the earliest integrated tool which addresses novice programmers’ common problems; it facilitates constructivism and scaffolds students’ learning through “fill in the gap” programming exercises; and it incorporates a program analysis framework which checks both program quality and correctness. The system aims to teach students procedural programming in the small approach. Table 4 compares all the well-known web-based programming environments discussed in Section 2 while Table 5 compares program analysis frameworks which are either stand-alone or integrated into a programming environment described in Section 2.5. As shown in Table 4, CodeLab, WebToTeach, InSTEP and WWW for Learning C++ are the few frontier systems supporting “fill in the gap” programming exercises even though the pedagogical benefits of this type of programming exercise has been identified in much earlier research.

CodeLab is the improved version of the WebToTeach system. However, both of these systems allow students to write small segments of code, rather than supporting gap-filling exercises. The provided code for an exercise is hidden from students. In addition, these systems only give program correctness feedback to students. The InSTEP system is designed to teach students looping concepts in C, C++ and Java. The system only supports a subset of these programming languages – loop syntax. In addition, a customised analysis module needs to be developed in order to check for program correctness; hence, it is time consuming to set up exercises. The WWW for Learning C++ supports gap-filling programming exercises, but does not provide feedback about program quality and correctness.

As also shown in Table 4, a majority of the programming environments provide help for students on correctness and compilation error problems. However, only a few systems assess the quality of student programs. This reflects the lack of attention to software quality in the curriculum which has been identified by Townhidnejad and Hilburn (2002).

Table 5 compares existing program analysis applications based on feedback type, supported languages, analysable exercises, provided analyses, and whether they are configurable by instructors and can be extended. As presented in Table 5, the ELP program analysis framework is one of only a few tools which conducts both static and dynamic analyses and is able to analyse gap-filling programming exercises. In the static analysis component of the framework, both quantitative (metrics) and qualitative (structural) analyses are carried out while the dynamic analysis component is the only one which offers automated white box testing. Conducting both quantitative and qualitative together with white box testing allows the framework to provide more constructive and detailed feedback for students. Further, the framework is configurable and extensible. These are the two crucial characteristics of a program analysis tool that enables it to be easily used in classrooms and have a long time of usage. Teaching staff can configure and control the analysis operations based on the requirement of an exercise; they can customise the feedback provided to students.

Overall, the literature has confirmed that the proposed ELP system is one of the early integrated web-based learning environments which facilitates constructivism and scaffolds student learning through “fill in the gap” programming exercises. The system also emphasises the importance of program quality in the software development process through the static analysis component of the program analysis framework. In the next three chapters, the design and implementation of the ELP system, and the static and dynamic analysis components of the program analysis framework are presented. In Chapter 6, how feedback is provided to scaffold students’ learning and allow teaching staff to assess students’ current progress is discussed.

	Web-based	Server compilation	“fill in the gap” exercises	Immediate feedback		
				Quality	Correctness	Customised Compilation Errors
CodeSaw	✓	✓				
CourseMaster	✓	✓		✓	✓	
CodeLab	✓	✓	✓		✓	✓
WebToTeach	✓	✓	✓		✓	✓
InSTEP	✓	✓	✓		✓	✓
WWW for Learning C++	✓	✓	✓		✓	✓
W4AP	✓	✓			✓	
JERPA	✓					
VECR	✓	✓				✓
Ludwig	✓	✓		✓	✓	
ALECS	✓	✓		✓	✓	
ELP	✓	✓	✓	✓	✓	✓

**Table 4: Comparison of Other Web-Based Programming Environments and ELP**

	Immediate Feedback	Multiple Languages	Analyse gap exercises	Static Analysis			Dynamic analysis			Configurable	Pluggable
				Metrics	Structural	Other	Black	White	Other		
ITPAD				✓	✓	✓					
Expresso	✓					Syntax, semantic, logic					
TRY		✓					✓				
BOSS	✓	✓							Unit test		
Datlab	✓					Syntactic parsing			Unit testing		
ASSYST				✓		Style	✓		Efficiency, test data adequacy		
WEB-CAT		✓				CheckStyle, Code coverage,			Unit testing	✓	✓
GAME	✓	✓				Style	✓				
Hogg							✓		✓		
PASS					✓		✓				
IRONCODE			✓				✓				
OnlineJudge	✓	✓					✓				
CourseMaster	✓	✓			✓	Plagiarism, maintainability	✓		Dynamic efficiency	✓	
WebToTeach	✓	✓	✓				✓				
CodeLab	✓	✓	✓				✓				
InSTEP	✓		✓			Tex parsing	✓				
WWW for C++	✓		✓				✓				
W4AP	✓	✓					✓				
JERPA	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
VECR	✓	✓									
Ludwig	✓			✓			✓			✓	
ALECS	✓					Style check	✓			✓	
ELP	✓	✓	✓	✓	✓		✓	✓		✓	✓

Table 5: Comparison of Other Program Analysis Frameworks and the ELP Analysis Framework



# **Chapter 3 - The Environment for Learning to Program (ELP)**

*It takes approximately 10 years to turn a novice [programmer] into an expert. (Winslow, 1996)*

## **3.1 Introduction**

It is generally acknowledged by IT educators that learning to program and teaching introductory programming courses are both difficult endeavours. Classes have high attrition and low success rates (Morrison and Newman, 2001), which are not helped by large class sizes and the need for multi-site and offshore delivery. Students in a large class with limited teaching resources are unlikely to get the individual attention they need, thus leading to an unsatisfactory learning experience and increasing possibility of failure (Anderson and Skawarecki, 1986).

Students need to go beyond explicit information to construct experiential and implicit knowledge in order to gain knowledge and become competent in the domain of computer programming (Affleck and Smith, 1999). One of the biggest difficulties is that programming languages are artificial - they contain a high level of abstraction (Moser, 1997). In order to program, students need to know the syntax of the language and be able to think abstractly. There is a great deal of implicit knowledge that is difficult or impossible for lecturers or instructors to make explicit. Students need to do a lot of programming practice to acquire this knowledge. Unfortunately, they generally encounter many difficulties when practising which often makes them lose confidence and become reluctant to write their own programs.

The contribution of this chapter is to introduce an innovative web-based learning environment, the ELP, designed to help students learn to program in Java, C and C#. The ELP is an on-line, active, collaborative and constructive web environment to support novice programmers. There are four distinguishing characteristics of the ELP system:

1. The system supports “fill in the gap” programming exercises and provides customised compilation error messages. This type of programming exercise

can help to develop required programming models, strategies, and skills for beginners to write programs. This type of programming exercise also reduces the complexity of writing programs, which therefore allows students to focus on the problem to be solved and actively engages them in the learning process.

2. The system provides a web-based interface which is easy to use for students. Students no longer have to spend time installing and configuring an IDE or a compiler before they start writing their first programs. This web-based interface facilitates a dynamic constructivist learning environment in which instructors can make learning materials ready for students to learn anywhere at anytime. It also allows smooth integration of programming exercises with lecture notes, tutorials and other web-based content.
3. The ELP incorporates a program analysis framework that provides students instant feedback about the quality and correctness of their programs. The framework facilitates a scaffolding instruction mode through which students are given clear directions, kept on track, and motivated.
4. The ELP system is exercise-centric. The exercise and its environment can both be configured for different stages of student learning.

Most first year IT students require extensive tutorial assistance as they often have difficulties in understanding programming concepts. By using the ELP system, students are able to access as much tuition as they need. They are not restricted to standard classroom contact hours per week nor standard working hours. Students can experience programming as a problem-solving activity from the earliest stages of their learning. They are exposed to a variety of approaches to problem solving, the consequent alternative solutions, and the relative merit of such approaches through various feedback mechanisms. The ELP system provides a learning environment which meets the diverse needs of students, for example those who find attendance at tutorials difficult, those who require more time for tutorials or those who are not used to formal tutorial situations.

Furthermore, the environment reflects constructivist learning theory which has the defining characteristic that knowledge cannot be transmitted from the teacher to the learner, but is an active process of construction (Ben-Ari, 2001; Hadjerrouit, 2005). In computer science education, constructivism is essential as beginning

programming students must actively build a mental model of language features to become proficient or accomplished programmers and designers (Astrachan, 1998). As discussed in Chapter 2, the five principles of a constructivist learning environment are: 1) providing authentic learning tasks, 2) promoting student-peer and student-tutor interactions, 3) encouraging ownership in the learning process, 4) allowing knowledge to be constructed and refined, and 5) supporting metacognition. With the ELP, each exercise consists of the requirements, other support materials and the program code. Students can choose to do exercises that are interesting to them as many times as they want, and they can communicate with tutors through the annotation feedback engine (Bancroft and Roe, 2006).

This chapter commences with a review of common difficulties that novice programmers encounter when they learn to program and then discusses how the ELP system addresses those hindrances. Finally, the architecture of the ELP system and its implementation are described in detail.

## **3.2 The Difficulties Encountered by Novice Programmers**

Dijkstra (1982) described programming as “one of the most difficult branches of applied mathematics”. This section discusses why programming is so hard to learn by analysing the skills and knowledge required to write a program, and will then elaborate on the difficulties that novice programmers encounter. This is followed by further discussion on specific problems of learning to program in modern OO programming languages such as Java, C++ or C# which are currently taught in most introductory programming courses.

### **3.2.1 The Programming Task**

Programming is not a single skill but a multi-layered hierarchy of skills, many layers of which are required to be active at the same time (du Boulay, 1986; Robins et al., 2003; Rogalski and Samurcay, 1993). According to Rogalski and Samurcay (1993), programming activity consists of four “interconnected” sub-tasks: problem representation, design, implementation and maintenance. Each requires different knowledge and strategies. In order to tackle those tasks, students need to have domain knowledge, design strategies, programming algorithms, methods knowledge, and debugging and testing strategies. They need to progress “simultaneously in

different domains” when they program so that the knowledge is acquired in a “spiral form rather than in a linear and accumulative form” (Rogalski and Samurcay, 1993).

From a different perspective, du Boulay (1986) describes five “overlapping” domains which are potential sources of difficulties among novice programmers including:

1. Orientation - what programming is for and the kinds of problem that can be tackled;
2. The notional of the machine - understanding general properties of machines;
3. Notation - the syntax and semantic of a programming language;
4. Structures - that is, how to acquire clichés or plans that can be used to solve sub-goals for a programming problem;
5. Pragmatics - the skills to specify, develop, test and debug.

A recent review by Robins and colleagues (2003) provides a good overview of the programming task. They developed a programming framework based on previous work in programming cognitive studies (shown in Figure 6). The framework illustrates three required attributes of a programmer (columns) at various stages of a program development process (rows). The authors emphasised that there is no clear distinction among these attributes, which should be considered “fuzzy”.

Program Development Processes	Attributes		
	Knowledge	Strategies	Models
Design	Of planning methods, algorithm design, formal methods	For planning, problem solving, designing algorithms	Of problem domain, notional machine
Generation	Of language, libraries, environments/tools	For implementing, algorithms, coding, accessing knowledge	Of desired program
Evaluation	Of debugging tools and methods	For testing, debugging, tracking/tracing, repair	Of actual program

**Figure 6: A Programming Framework**  
*(Adapted from Robins et al., 2003)*

### 3.2.2 The Difficulties of Learning Programming

Novices encounter many difficulties in designing, coding and evaluating a program, which leads to the lack of confidence and motivation to continue programming. There are many explanations available in the literature for why students struggle with

programming. This section expounds common novice programmers' difficulties which have been revealed in early work, based on the three attributes of a programmer shown in Figure 6.

### **3.2.2.1 Novice Knowledge**

Most studies on programming knowledge representation are based on the concept of "schema" which is an abstract solution to a programming problem that can be applied in many situations (Ehrlich and Soloway, 1984; Rist, 1986b). Similar concepts include "plans", "templates", "idioms" and "patterns" (Clancy and Linn, 1999).

Studies comparing the knowledge of experts and novices reflected that novice programmers' conceptual knowledge is superficial (Winslow, 1996), shallow (Lewandowski et al., 2005) and "fragile" (Winslow, 1996), which amounts to the state of 'half knowing something'. Various studies on types of fragile knowledge among novice programmers are discussed in Perkins et al. (1986), Perkins and Martin (1986) and Perkins et al (1988). According to Kurkand et al. (1989), novice knowledge tends to be context-specific rather than general. Their knowledge lacks five abstract characteristics of the mental representation of computer programs that experts have (Fix et al., 1993; Wiedenbeck et al., 1993). First, the mental representation of a program held by experts is hierarchically structured and multi-layered. Second, there are explicit mappings of code to goals in expert programmer mental representations. Third, the expert mental representation is founded on the recognition of basic patterns. Fourth, it is well-connected internally; experts understand how parts of the program interact with one another. Lastly, it is well grounded in the program text. This characteristic allows experts to locate information in a program quickly when it is needed the second time.

### **3.2.2.2 Novice Strategies**

Studies have identified four major shortfalls strategies among beginning programmers including: problem solving, program comprehension, code generation and debugging.

Many studies have indicated that lack of problem-solving skill is the central difficulty for novice programmers (Mayer, 1981; Perkins et al., 1986; Perkins et al., 1988). This skill is beyond the knowledge of syntax and semantic of a programming

language (Perkins et al., 1988; Wiedenbeck et al., 1993). According to McCracken and colleagues (2001), problem-solving is defined as a five step process. Firstly, the problem is abstracted from its description which involves identifying the relevant aspects of the problem statement and putting them in an appropriate abstraction framework. Secondly, the problem is broken down into many sub-problems. Thirdly, each sub-problem is transformed into sub-solutions by deciding the data structure, language construct and implementation strategy for each. After that, all the sub-solutions are recomposed to produce the solution for the problem. Finally, the solution is tested and debugged if there is any runtime or logic error. Unfortunately, novices encounter many problems in all steps.

Students have difficulties in abstracting the problem from its description and breaking it down into sub-problems (McCracken et al., 2001). Research by Winslow (1996) has concluded that students can solve the problem by hand but they cannot express the solution in the computer language. This is because beginning programmers do not have the appropriate mental models to decompose the solution into implementable sub-solutions. The sub-solutions are either still too complicated or do not suit the programming language used to solve the problem; this makes the programming task more difficult or impossible to solve (Perkins et al., 1986). Students also struggle in combining statements to generate the desired result because they do not think that designing an algorithm for a programming problem is important (Liffick and Aiken, 1996; Winslow, 1996). Novices also face many problems in testing and debugging their programs because their program comprehension skills are weak (du Boulay, 1986; Lister et al., 2004); they often use a “trial and error” approach to test their programs (Edwards, 2004).

Program comprehension skill is vital in testing and debugging processes. Research has shown that novices often comprehend a program line by line based on its physical order while experts start reading a program from its main section and move to the next statement based on its execution order (Jeffries, 1982; Mosemann and Wiedenbeck, 2001; Nanja and Cook, 1987). As a result of that limited approach, students have a fragmentary knowledge of the program and fail to connect the pieces in terms of the hierarchical structure, dynamic behaviour, interactions and purpose (Jeffries, 1982). They are “very local and concrete in their comprehension of programs” (Wiedenbeck et al., 1999), and they have difficulty in understanding the

accomplished task by groups of statements because they look at examples “using a microscope instead of just their own eyes” (Liffick and Aiken, 1996).

Studies have shown that the ability to comprehend a program and the ability to write one do not necessarily correspond (Rountree et al., 2005; Winslow, 1996). Winslow (1996) suggested that “both need to be taught along with some basic testing and debugging strategies”. However, there are more studies on program comprehension than code generation (Rountree et al., 2005). The most well-known code generation model was developed by Rist (1986a, 1986b, 1989, 1995). Based on Rist’s model, programming knowledge is represented as nodes in memory and a program is built by retrieving relevant nodes and linking them together to make a plan which performs a specific sub-task. The key difference between novices and experts in the code generation process is that experts retrieve existing plans from memory while novices must create plans. As discussed earlier, students have difficulty in retrieving their learnt knowledge and combining statements together to perform a desired task, hence code generation is difficult for them (Davies, 1991).

Novice students lack debugging and testing skills because their programming comprehension skill is weak. They often use the line by line approach to read a program. Compounding this problem, many novices do not track their code carefully. Perkins and colleague (1986) emphasise that the ability to read and understand what a piece of code does is an important skill for diagnosing bugs. This skill is mentally demanding and there are four potential factors causing failure in tracking a program (Perkins et al., 1986). First, students do not think that tracking is important and they are not confident in their tracking ability. Second, students might misunderstand how a particular programming language works. The third factor is the failure in building the mental model of a program, understanding its behaviour, and projecting intention on to the code. The fourth factor is the impact of program coding style and cognitive style differences.

### 3.2.2.3 Models

Mental models are “crucial to build understanding” (Winslow, 1996) and are required to write programs (Robins et al., 2003). When a programmer writes a program, he or she needs to have three models: the problem domain, the “concrete model” and the model of the program (Smith and Webb, 1995). The problem domain model is required so that students can understand the purpose of a program since

each program is designed to solve a problem (Anderson and Soloway, 1985). After understanding the goal to be achieved, the programmer needs to have some knowledge of how computer works – the so-called “concrete model” (du Boulay, 1986; Mayer, 1981). Finally, the programmer needs to use his or her programming knowledge to make the computer achieve the goal. Unfortunately, studies have indicated that many students lack all the above models (du Boulay, 1986; Kessler and Anderson, 1986; Mayer, 1981; Smith and Webb, 1995).

### **3.2.2.4 Motivation**

Students not only have fragile knowledge and lack essential models and strategies to write programs but also are not confident and motivated to pursue solutions to programming problems (Lewis and Watkins, 2001; Lui et al., 2004; Perkins et al., 1988). Perkins and colleagues established a “stopper-mover” continuum to characterise types of novice (Perkins et al., 1986; Perkins et al., 1988). The three different types of novice are the: “stopper”, “mover” and “tinkerer”. Students are classified as “stoppers” when they simply give up or show no hope of solving the problem on their own. This occurs when a student is unable to provide an answer. The student feels “at a complete loss” and is unwilling to continue with the programming problem (Perkins et al., 1986). “Movers” are students who keep trying, fixing their code and experimenting to solve the problem. “Tinkerers” are extreme “movers”; however, they are not able to track their code.

### **3.2.2.5 Learning to Program in Modern Object-Oriented Languages**

In the previous four sections, general difficulties in learning to program have been discussed. This section elaborates on obstacles that novices encounter in learning to program in modern OO programming languages such as Java, C#, C++ and Visual Basic with Java as the main focus since it is currently the most commonly used in introductory programming courses in Australia (de Raadt et al., 2003) and around the world. Even though Java is widely used in computer science courses, there are quite a few criticisms about its language constructs and the SUN Java Development Kit (JDK) which make the programming task even harder. These criticisms are detailed in the remainder of this section.

The processes a student undertakes to create a working program are edit, compile, debug and run. Hence, to produce their first Java programs, students need to

be able to use a program text editor, successfully install the Java compiler, know the Java language syntax, be able to compile a program and know how to debug the program using error messages generated by the compiler. More often than not, beginners encounter many difficulties in using the software tools that comprise the supporting development system (Lewis and Watkins, 2001). A majority of students have problems installing the JDK on their own computers because they have to modify two system environment variables, namely PATH and CLASSPATH, before they can compile and run programs. Students either have no knowledge of these variables or set them incorrectly in spite of detailed guidelines. When they have edited a program, students need to remember to save the source file with the same name as the class name in the source otherwise a compilation error message is generated (Muller, 2003).

In terms of the language constructs, Java contains high-level abstraction that is too complex to be introduced in the first few weeks of introductory programming courses and which would cause confusion for novices (Biddle and Tempero, 1998; Clark et al., 1998). Even a simple HelloWorld program already contains many abstract concepts such as “class”, “public”, “static”, “void” and “main”. Other discussions on choosing Java as the first teaching programming language and students’ difficulties in learning to program with it are set out in Andreae et al. (1998), Biddle and Tempero (1998), Dingle and Zander (2000), Garner et al. (2005), Tyma (1998) and West (2000).

In summary, this section has discussed difficulties that novices encounter when they first start learning to program based on the programming model developed by Robins et al. (2003), and shown in Figure 6. According to the model, knowledge, strategies and models are three attributes of an expert programmer; however, the literature has indicated that novices lack all three attributes, as well as motivation and confidence to pursue a programming problem. Table 6 summarises the missing attributes among novices discussed in this section. In addition, the section also discusses the difficulty in learning to program with high-level modern OO languages such as Java, C++ or C#.

Knowledge	Strategies	Models
1. Surface 2. Fragile 3. Missing hierarchical structure 4. Missing mappings of code to goals 5. Missing foundation to recognise learnt knowledge 6. Missing connection of knowledge 7. Not grounded with programming text	1. Problem solving 2. Program comprehension 3. Code generation 4. Debugging	1. Problem domains 2. Notation of machine 3. Notation of programming

**Table 6: Missing Programming Attributes among Novices**

Which of the problems mentioned above will be addressed by the ELP and how the system helps students to overcome those barriers will be discussed in the next section.

### 3.3 The Key Characteristics of the ELP

The ELP system is designed to accommodate difficulties that novice programmers encounter when they first start learning to program in Java, C and C#. The system has four distinguishing characteristics:

1. It supports “fill in the gap” programming exercises.
2. The system is completely web-based.
3. It provides timely and formative feedback for students.
4. It is exercise-centric.

This section is divided into four sub-sections; each sub-section will elaborate on one characteristic of the system and discuss how the characteristic can help novices to overcome those early barriers noted in the previous section.

#### 3.3.1 “Fill in the Gap” Programming Exercises

The ELP system provides students with “fill in the gap” programming exercises in which some parts of a program text are deleted. Students are required to fill in those parts to complete the program. This type of programming exercise has already been used in programming courses for some time. It is well-acknowledged as having potential in developing students’ comprehension, code generation, problem-solving skills and their programming mental model (Garner, 2001; Lister, 2000; Tuovinen, 2000); it also helps to overcome “fragile knowledge”. Once a novice’s comprehension skill is improved, the debugging and testing tasks will be much

easier, since those tasks are heavily reliant on this skill. As well, this type of exercise increases students' motivation since the complexity of writing the program is reduced. Till now, few teaching and learning tools support "fill in the gap" exercises. This type of exercise is often given to students either on paper or electronically in a format which precludes teaching staff from stopping students to modify the given code.

This section reviews studies which emphasise the impact of "fill in the gap" programming exercises in developing students' mental model for programming, comprehension, and problem-solving skills and motivation.

### **3.3.1.1 Develop Mental Models and Reduce Fragile Knowledge**

As noted in Section 3.2.2.3, there are at least three mental models required to be active at the same time in order to write a program: the problem domain (what is the problem that the program is trying to solve), the concrete model (how the computer operates), and the programming model (programming language syntax knowledge). "Fill in the gap" exercises help students to develop their problem domain and programming models.

The problem domain is normally given as an exercise requirement since students have difficulty in thinking in an abstract way and knowing where to start. The provided code in "fill in the gap" exercises gives students a starting point.

The combination of this type of exercise together with frequent practice will reinforce students' knowledge; hence, they will not have "fragile knowledge". This is because students must clearly understand the execution flow and the goal of the provided code in order to complete an exercise. Goals and plan mappings are developed in students' mental model with the goal being the task that a gap is trying to achieve, and plans are the segment of code that students need to provide. Progressively with practice, students will develop a hierarchical structure of goals and plans; they will notice familiar patterns and know how to apply them to new situations. Students' language syntax and semantics knowledge is also increased – forthwith, the programming model is developed. Chances that students develop fragile knowledge are reduced because they are constructing knowledge in small components. For each exercise, students are presented with the provided code (concrete model) so that they can use it to construct new knowledge based on that model. It also provides a context so that students can bridge and apply the new

knowledge to existing knowledge. In addition, the provided codes act as a scaffold to further help students.

### **3.3.1.2 Develop Comprehension and Code Generation Skills**

Program comprehension and code generation are two closely related skills but need to be taught explicitly (Lister, 2000; Winslow, 1996). “Fill in the gap” exercises can improve both students’ program comprehension and code generation skills. This is because in order to complete an exercise, students need to understand the given code first before they can use their programming language syntax knowledge to complete the program. This type of programming exercise has been used extensively in research to measure program comprehension skill (Ehrlich and Soloway, 1984; Norcio, 1980a, 1980b, 1982b). Studies have shown that this type of exercise is more effective in developing students’ program comprehension skill than asking students to understand a complete working example and annotation with information of what each section of code does (Van Merriënboer and Paas, 1990). This is because they provoke more “mindful” abstraction of programming plans in students (Van Merriënboer and Paas, 1990; Van Merriënboer and Krammer, 1987; Van Merriënboer et al., 1994) while with complete working examples “students tend to rush through the example” (Garner, 2001) and not fully understand it. Students might fail to comprehend a program, not because they do not understand the new concept, but because they “failed to understand previous concepts” used in the example (Liffick and Aiken, 1996).

With frequent practice of the same exercise pattern, the hierarchy of goal and plan mapping will be developed in the student’s mental model. This is one of the characteristics that novice mental models lack (Wiedenbeck et al., 1993). Van Merriënboer’s studies (1990; with De Croock, 1992) have shown that students who take “fill in the gap” programming assignments have a better understanding of a program compared to those who take the conventional programming assignments.

Most criticism of current introductory programming courses is centred on writing complete programs at the early stage of student learning, focusing on the technology but not the cognitive development of the student. This approach neglects the lower two levels - knowledge and comprehension - of Bloom’s taxonomy (Bloom, 1956) reviewed in the previous chapter. “Fill the gap” exercises enforce the lower four levels of the taxonomy while the top two levels are best addressed by

having students write complete programs (Lister, 2000). Lister (2000) also identifies four other benefits of “fill in the gap” programming exercises. Firstly, this type of programming exercise demonstrates good programming practice. Secondly, well-constructed provided code constrains students’ thinking to productive paths. Thirdly, the emphasis of “fill in the gap” is on the algorithm not the syntax. Lastly, the exercises concentrate on reinforcing new concepts from the most recent lecture, without forcing students to expend time writing mundane code.

“Fill in the gap” exercises also develop strong code generation skill (Bailey, 2005). Students need to use their programming knowledge to complete missing code. Therefore, with frequent practice, students’ mental model of programming will be developed progressively. With “fill in the gap” exercises, teaching staff can introduce new language constructs step-by-step since the size of gaps is controllable by teaching staff. This is extremely important when modern OO programming languages such as Java and C# are taught as first programming languages. Novices will not be overwhelmed by so many new language constructs. They will not have to understand and memorise multiple constructs in order to write a new program. Students only need to concentrate on new constructs that are introduced in that week of the lecture material.

### **3.3.1.3 Develop Problem-Solving Skill**

Problem solving is the well-acknowledged skill that novices lack. As noted in Section 3.2.2.2, this is a five stage process and students encounter problems in all stages due to the lack of two crucial mental models: the problem domain and programming domain. In Section 3.3.1.1, the advantages of “fill in the gap” exercises in helping students to develop those mental models were discussed.

Experiments carried out by Mayer (1981) have shown that the provided code in a “fill in the gap” exercise has a strong effect on the encoding and the use of new technical information by novices because they use that model to incorporate new information. It allows novices to develop “creative solutions” and to apply the learned knowledge to solve a new programming problem.

### **3.3.1.4 Increase Motivation**

Last but not least, “fill in the gap” programming exercises reduce the complexity of writing a program (Yousoof et al., 2006). This type of programming exercise makes

the programming task more achievable and easier for novice programmers at the earliest stage of their learning. Students will trust their ability and move on to more complicated exercises. The approach builds up students' confidence. They are motivated and engaged in the learning process more actively (Paas, 1992; Van Merriënboer, 1990; Van Merriënboer and De Croock, 1992). Studies have also shown that the increase in self efficacy (Bandura, 1986) together with a good mental model of programming will help novices to acquire knowledge better, and hence, increase course performance (Ramalingam et al., 2004).

In addition, students only have to type in small segments of codes in gap areas. This reduces typing errors and thus syntax errors. Programming is not only problem-intensive but also precision-intensive: students can pass a spelling test with 90% while a program with 10% spelling error will fail and make the debugging task frustrating (Perkins et al., 1988).

In brief, this section has discussed the benefits of “fill in the gap” programming exercises in helping students to develop their problem domain and mental model of programming, their comprehension, code generation, problem-solving skills, and in motivating and engaging students in the learning process. This type of exercise also has the potential to ease the testing and debugging process since students’ comprehension skill is improved.

### **3.3.2 Web-Based**

Web-based courses make learning more effective because they facilitate a student-centred learning environment (Al-Ayyoub, 2004; Blackboard, 2000; Slay, 1997). In web-based learning environments, different students’ learning styles and needs are accommodated. Students can express their ideas freely without shyness. They can access learning materials and explore new concepts anywhere and at anytime. Web-based courses also reduce the necessity for physical attendance.

The ELP system can be accessed through a web browser. This chosen user interface has brought five major benefits for the ELP system, as follows:

1. Requiring no installation: students neither have to spend time installing and setting up the compiler and IDE nor learning how to use a programming editor before they start writing their first program.

2. Providing a familiar and comfortable environment: a web browser is considered to be the simplest interface to use. This interface also increases the level of flexible delivery and overall accessibility of the system. Students and staff are able to use the system anywhere at anytime.
3. Allowing seamless integration of lecture and tutorial notes together with practical exercises embedded in web pages.
4. Being server-based allows teaching staff to track and monitor students' progress.
5. Embedding in any Content Management System (CMS) - with this design, the ELP system is ready to be integrated or co-exist with other CMS such as Blackboard (1997) or WebCT (Murray, 1995)

### **3.3.3 Provide Timely and Frequent Feedback**

Learning is facilitated by feedback (Dihoff, 2004). Feedback makes learning more effective and motivates students. Lacking a ready answer to a problem makes students not only feel completely lost but unwilling to explore the problem any further (Perkins et al., 1986). Much research has been carried out to investigate the type and timing of feedback and its impact on learning. It is acknowledged that in all learning, individualised on-demand feedback is the most effective because it meets each student's needs. This type of feedback has great benefits for weak students (Anderson and Reiser, 1985; Bloom, 1984; Heaney and Daly, 2004).

The ELP system provides various types of on-demand feedback for students including customised compilation error messages, and feedback on quality and correctness of their solutions.

Novice programmers have difficulty getting a program to successfully compile. This is caused by the lack of syntax and semantics knowledge of a programming language and the precise nature of programming. Novices are often shocked by hundreds of unfriendly compiler error messages which might be caused by simply misspelling a variable name or missing a closing bracket. The ELP system provides students with customised, friendly error messages. This makes it easier for students to fix errors in their programs.

Immediate feedback on quality and correctness of students' programs allows students to learn from their mistakes and reflect on their solutions in the context in which they are currently engaged. This mode of feedback is much more effective

than if it were given a few weeks afterwards, because the students have already forgotten the context of the programming exercise. Based on the given feedback, students can carry out further experiments to investigate what was wrong in their previous attempt and try to fix the problem. This approach of learning is called “reflection in action”, and first described by Schon (1983). According to Edwards (2004), this learning approach to introductory programming courses allows students to learn to program more successfully. Edwards also noted that timely feedback will move students away from the “trial by error” testing strategy.

### **3.3.4 Exercise-Centric**

The ELP system is exercise-centric. Each exercise in the ELP system is a Java applet embedded in a browser. The applet contains all the programming code of an exercise while the textual content and other support material for the exercise are included in the web page. The editor applets can be configured like a normal IDE to suit each student’s preferences; students’ attempts for each exercise can be analysed and their progress can be easily tracked.

To summarise, the present section has discussed four key characteristics of the ELP system, namely, that it supports “fill in the gap” programming exercises, incorporates a feedback engine, is web-based and is exercise-centric. “Fill in the gap” programming exercises are effective in developing novices’ knowledge, strategies, and models as shown in Table 7. The integrated feedback engine facilitates constructivist learning and provides scaffolding for students through customised compiler error messages and comments on quality and correctness of students’ programs. The web-based user interface means no installation is required, provides flexible learning (meeting the needs of on-line and offshore delivery), and allows seamless integration of exercises and teaching materials. The system is exercise-centric: exercises in the system are Java applets which allow the system to be readily integrated into any CMS. Furthermore, users are also can customise the ELP programming editor.

<b>Problem</b>	<b>ELP</b>
Surface knowledge	✓
Fragile knowledge	✓
Missing hierarchical structure	✓
Missing mapping of code to goals	✓
Missing foundation to recognise learnt knowledge	✓
Connecting knowledge	✓
Grounding in program text	✓
Problem solving	✓
Comprehension	✓
Debug	✗
Problem domains	✓
Notational of machine	✗
Actual program	✓

**Table 7: Novices' Difficulties Overcome by Gap-Filling Exercises**

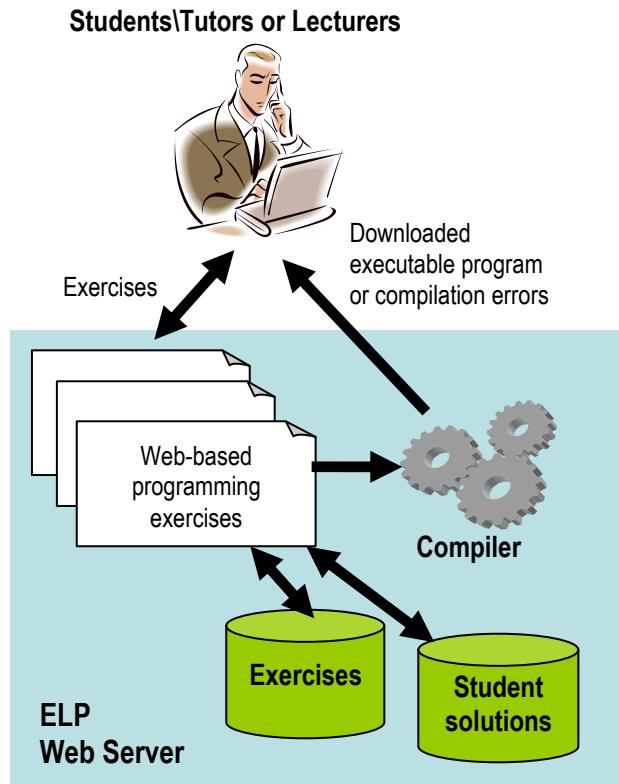
The forthcoming sections will describe how these characteristics are designed and implemented in the ELP system.

### **3.4 The Architecture of the ELP**

This section starts with an overview of the ELP system before describing the system user interfaces. It then discusses the representation of a “fill in the gap” exercise and the system functionalities.

#### **3.4.1 System Overview**

The ELP is a web-based client-server system. The ELP client runs in a web browser as an applet using the Java Runtime Environment (JRE). All exercises are stored in a database on the server. At present, the ELP supports console and graphical GUI Java, C and C# exercises. The system recognises three types of users: students, tutors and lecturers. Students are presented with a programming exercise as a web page. They complete the exercise and submit their answer to the server for compilation. The executable program is downloaded and run on the student's own machine. Tutors can monitor the progress of students in their classes. Lecturers and Unit coordinators can perform exercise administration through the ELP administration applet. Figure 7 gives an overview of the ELP.



**Figure 7: An Overview of the ELP System**

Applets have been adopted in many educational tools because they are highly interactive and can be embedded in a web page. The use of Java applets as the client interface for the ELP system has yielded major benefits for the system. Firstly, the system will be platform independent and portable because applets are run on the web browser. Java applets provide an open distributed and expendable platform for courseware (Ray, 2004; Wie, 1998). Secondly, the system is capable of communicating with a wide range of applications on the server since it supports various communication mechanisms. Thirdly, the system is readily integrated into any CMS since applets can be embedded into a web page. Lastly, the use of applets support exercise centricity. The applet can be included in a web page together with other resources such as text, images, and voice. This will enhance the exercise and make it more interesting.

As Wie (1998) suggested for maximum use of applets in educational tools, instructors should be able to include applets in a web page as easily as including an image. With the ELP, only two lines of HTML are required to enable the ELP client applet to reside in a web page (see Figure 8). The first line identifies the exercise on the ELP system that the user would like to load. The second line specifies a loading script designed to allow the applet to load on different browsers. In this example, the

HelloWorld exercise in the Week 1 topic of the ITB610 unit will be loaded in the ELP editor applet.

```
<html>
<body>
    <input type="hidden" id="exercise" value="/ITB610/Week 1/HelloWorld"/>
    <script src="http://www.elp.fit.qut.edu.au/load.js">
</body>
</html>
```

**Figure 8: Example of HTML Source to Include ELP Applet**

Even though the ELP system supports three types of users - students, tutors and lecturers - this thesis only discusses the student and lecturer components. The student and lecturer views of the ELP system will be described in the forthcoming two sections. In Section 3.4.4, the representation of the ELP exercises, how they are stored in the database and how the exercise solution and a student program are constructed will be discussed. The functionality provided by the ELP system is detailed in Section 3.4.5.

### **3.4.2 Student View**

Students can access the ELP system through a web page or a local CMS. At QUT, the ELP can be accessed through the Online Learning and Teaching (OLT) system, a local CMS, as shown in Figure 9. Programming exercises are embedded in web pages. Each exercise is comprised of the question text, other support materials contained in the web page and a separate Java applet which implements the ELP. The students log in and are authenticated for the subject by the ELP applet. Thus the ELP is used for programming activity only. This is fundamental in the design of the system as it means that the ELP can readily be integrated into any CMS and can leverage other CMS features.

The screenshot shows a blue header bar with the text "OLT" and "online learning & teaching". Below the header, the URL "OLT Trail: Software Development 3 > Tutorials >" is visible. The main content area has a white background and features a large bold title "Week 5". Below the title, the text "Object-orientation, C# properties" is displayed. A section titled "Tutorial Questions" contains a link "week5.doc" and a note "File Size: 102.50 KB". Below this, four questions are listed: "Question 1" (ELP - creating and using a Person class), "Question 2" (ELP - extending a Person class), "Question 3" (ELP - creating and using a Fraction ADT), and "Question 4" (ELP - creating and using a Money class).

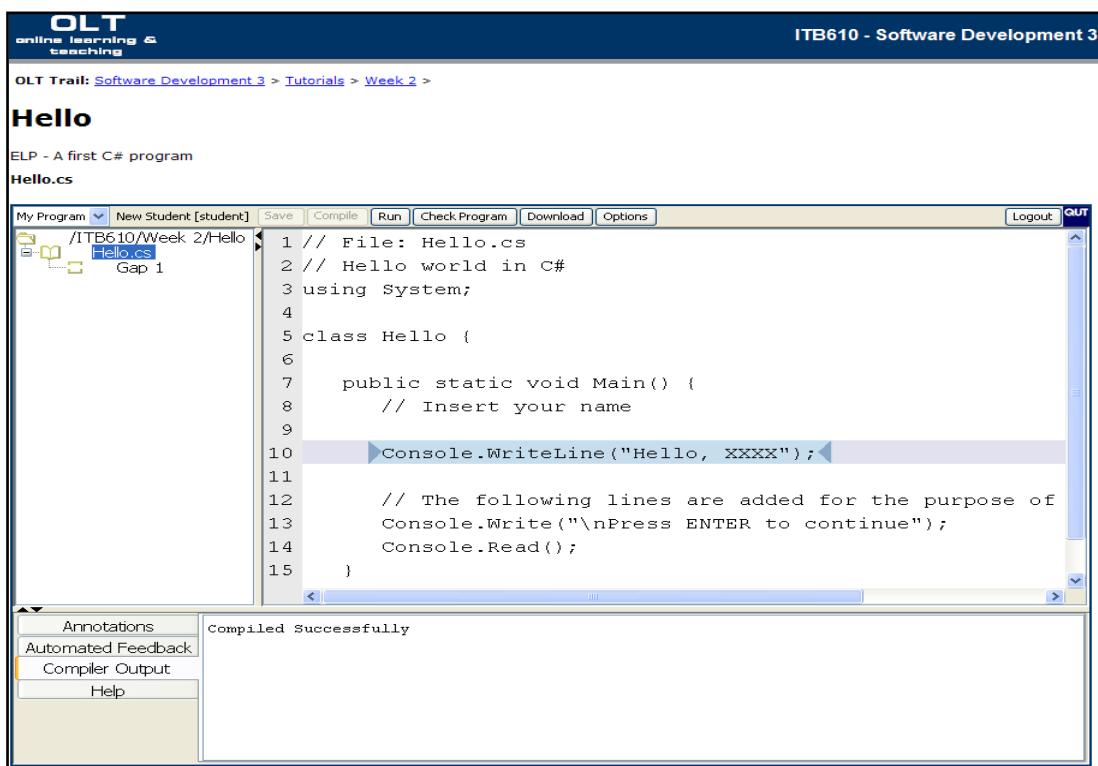
**Figure 9: The Integration of ELP into OLT**

An ELP exercise consists of a program with some missing lines of code which students must complete. The missing lines (gaps) are writable fields in the ELP editor while the enclosing code is non-editable. Making the provided code un-editable prevents students from modifying it. The gaps usually contain helpful hints describing the missing code. Figure 10 illustrates the ELP applet for an exercise with one gap.

Exercises are organised in increasing level of difficulty. Some early exercises have no gaps; the student simply compiles and runs the exercise. This allows the student to experience how the ELP works and to see some complete, working programs. A gap can be as small as an expression or as large as a complete program. Exercises may contain more than one gap and may even incorporate multiple classes and multiple file programs.

The student completes a program by filling in all the writable fields of the exercise template. The student needs to save the exercise in order to compile it. This will ensure that students will always see their latest attempt. In a conventional IDE, beginning students often forget to save their programs and become frustrated by not seeing their most recent code. The exercise is sent to the server for compilation and the results are displayed on the console pane below the program text. If there are no errors, the students may download and run the executable program on their own machines. Otherwise, customised compilation error messages are displayed for

students. These error messages are customised with friendly messages and include links so that students can easily navigate to lines in the program which have problems. Figure 7 summarises students' interaction with the system. Model solutions are released through the ELP at the lecturer's convenience. The "Options" button allows the student to set the font size of the editor and the applet size to suit different requirements. Students can save exercises locally and work off-line, if they desire.



**Figure 10: Student View of the ELP System**

On successful compilation of the exercise, students can check the quality and correctness of their program and receive immediate feedback through the "Check Program" function. The feedback they receive comments on code quality, the overall structure and the correctness of their programs. They are also given hints on how to fix problems or how to improve the quality.

There are three different views in student mode: hint, solution, and My Program. Students work on the exercise in the My Program view. They can switch to the hint or solution view to have a look at hints or the solution for gaps. If this is the first time the student attempts the exercise, all gaps are occupied by hints in the My Program view, otherwise, the student's previous attempt is displayed. Students

cannot edit the provided code, hints and solutions in either the hint view or solution view. They can only edit the gaps in the My Program view. This gives great control for teaching staff.

Using the ELP for Java exercises means there is no need to worry about installing the JDK and setting PATH and CLASSPATH environment variables on the student's machine. Students also do not have to spend time learning how to use a program editor. They only have to focus on learning to program and receiving immediate feedback about their work. The ELP interface is simple and designed specially for novices unlike other IDEs such as Eclipse and Visual Studio .NET which are designed for professional programmers and quite complicated to use.

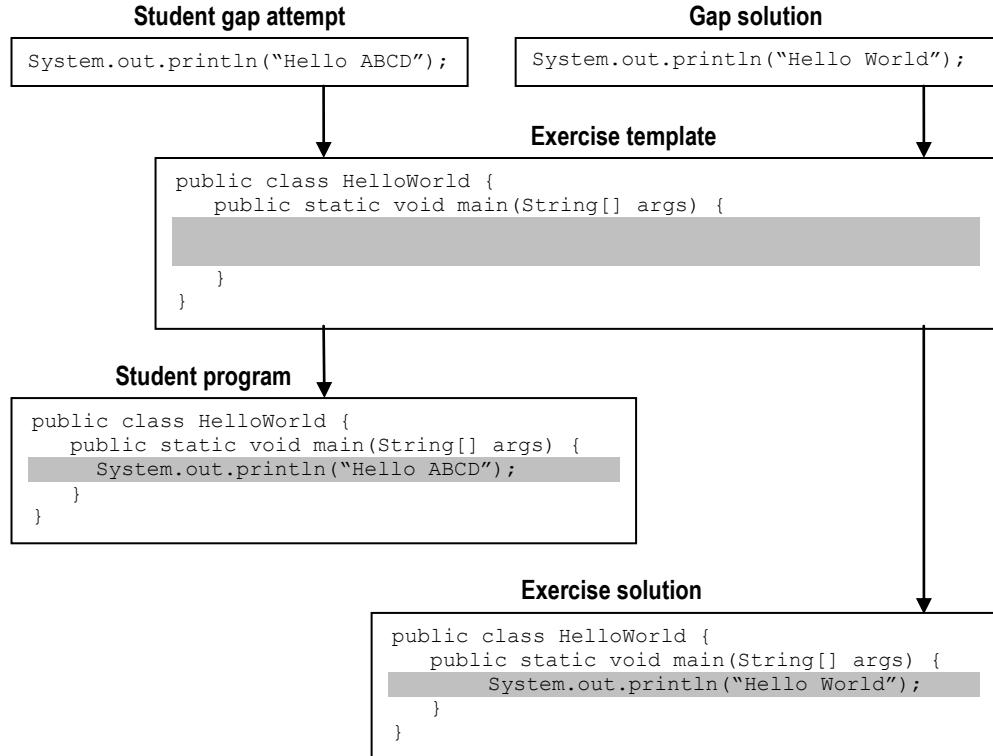
### **3.4.3 Lecturer View**

Lecturers can access the ELP through a web page. When lecturers log in to the ELP, they are presented with an enhanced view of the editor applet with increased functionality for creation and management of exercises. There are four views in the lecturer mode of the ELP system. These are: template, hint, My Program and solution views. Lecturers can create and remove gaps for an exercise in the template view. They can edit hints for a newly created gap or modify existing hints by switching to the hint view in the editor. The solution view shows the exercise with model solutions occupying all gaps. After editing an exercise, the lecturer can change to the My Program view to check how the exercise will be displayed for students. Additionally, they can add and remove files in an exercise, change file names and specify a date of release for the solution.

### **3.4.4 The Exercise Representation**

An ELP exercise is comprised of two components: an exercise template and a number of exercise instances. The exercise template is the provided code, while an exercise instance is the gap content which can be hints, solutions or students' attempts, known as My Program. Thus, one exercise has one hint instance, one or many solution instances and many My Program instances. With this design, the hint and solutions of an exercise can be independently created, edited and modified. Exercises are stored in the database with minimum redundant data. For all exercises, only solutions, hints and student attempts for gaps are stored rather than storing the complete program. To create an exercise solution, the gap content in the solution

instance is inserted into the exercise template. The same approach is used to create the exercise with hints for gaps and students' programs. Figure 11 gives an example of how an exercise solution and a student program are constructed.



**Figure 11: The Construction of Various Exercise Views**

### 3.4.5 Functionalities of ELP

The ELP system provides facilities for students to save, compile and run programming exercises and receive immediate feedback on their work through the “Check Program” function. Alternatively, they can download exercises to work off-line. These functions are also available for lecturers and tutors together with additional functions for manipulating gaps and files in an exercise and monitoring students’ progress.

In student mode, the “Save” function simply stores a student’s current attempt in the database on the server. All students’ attempts are saved for monitoring purposes but students only see their latest saved attempt the next time they revisit the exercise. In lecturer and tutor modes, the “Save” function stores all changes that have been made to the exercise.

When an exercise is compiled, customised error messages are provided to ease the debugging process. The interface forces students to save their work before compiling to reinforce the edited, compile run programming process.

When a program compiles successfully, the executable format of an exercise is downloaded and run on students' machines. This function is only available when the exercise is successfully compiled. With Java programming exercises, the resulting .class files of an exercise together with other necessary libraries are packed in a JAR file; while an .exe file is used for C and C# programming exercises.

The "Download" function allows students to download their work. Exercise sources and other necessary libraries are packed together in a ZIP file. The start and end of each gap is marked in the downloaded sources. This function enables students to do the exercise off-line or they can download their work and submit it to any electronic submission system.

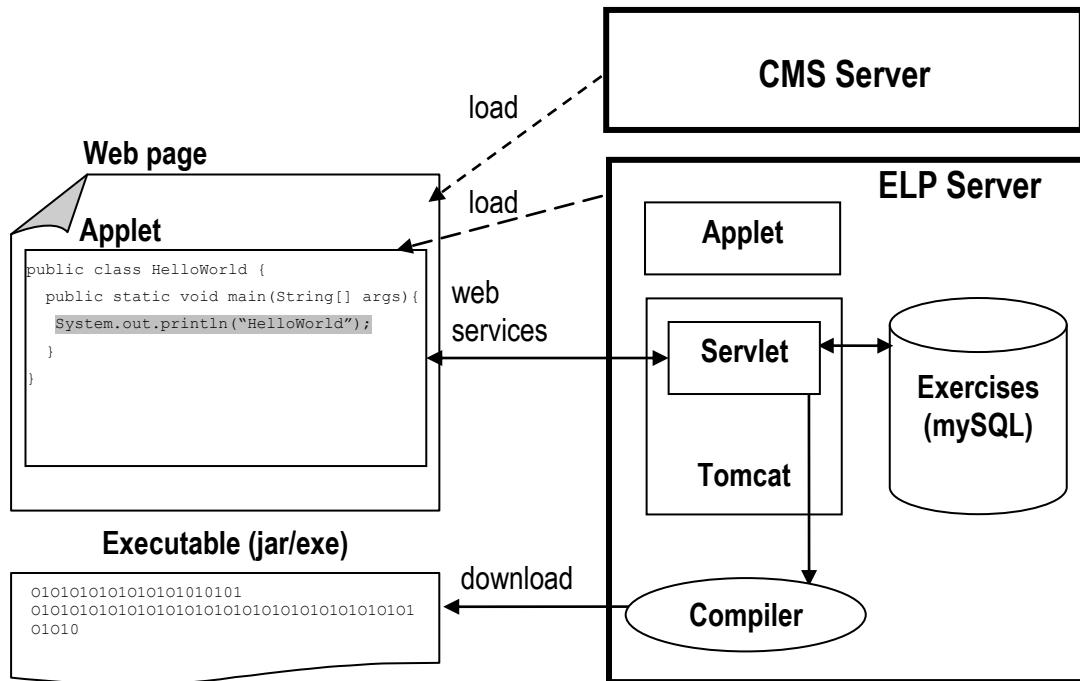
The "Check Program" function provides immediate feedback for students about the quality and correctness of their program based on the static and dynamic analyses results detailed in the next two chapters. Their code will be checked against a set of software engineering metrics and good programming practice guidelines to judge the quality. Students' programs are also executed against a set of test data, and test outputs are compared with the expected output to evaluate the correctness of their programs.

In summary, the ELP is a web-based client-server system. The client is implemented as a Java applet embedded in a web page. The system provides functions for students to edit, compile, run and receive immediate feedback about the quality and correctness of their programs. It also provides functions for lecturers and tutors for monitoring students' progress. A set of functions is also designed for lecturers to create and manipulate exercises. Each exercise in the ELP system has three views: hint, solution, and My Program. In the hint view, the instructor's hints are occupied gap areas. Similarly, gap solutions are occupied in the solution view and students' attempts are occupied gap areas in the My Program view. In the next section, the implementation of the system will be detailed.

### **3.5 The Implementation**

The ELP is based on a web page access to a client applet which connects to a Linux server running Apache Tomcat container. The client applet communicates with the

server through HTTP requests and responses. The system makes extensive use of eXtensible Markup Language (XML). All exercises information is stored in a MySQL database as XML strings; HTTP requests and responses between client and server are sent and received in XML string format. The system supports Java, C and C# programming exercises. The standard JDK is used to compile Java programming exercises, the gcc compiler is used for C programming exercises and Mono (Mono, 2003) is used to enable C# compilation on the server. Each user has a temporary directory on the server to perform the compilation and analysis process. An HTTP session mechanism is used to perform user tracking. Figure 12 illustrates the high-level implementation of the ELP.



**Figure 12: High-Level Implementation of the ELP System**

XML is a way to define data formats (Usdin and Graham, 1998). It is a universal format for structured documents and data on the web (W3C, 2002). It is the standard for how all computer systems exchange information (Maschhoff, 2000). Four fundamental principles which have empowered XML are: 1) separation of content from format or processing, 2) hierarchical data structures, 3) embedded tagging, and 4) user-definable structure.

If a piece of information can be separated from how it is presented, the information can be reused and displayed with different user views. XML tags can be

embedded in the data without the need to specify in advance how long any piece of information will be. This provides great flexibility for application in terms of how much information they will receive. The most important feature of XML is the user-definable tags. Information will be more readable, and information searching and retrieval will be much easier.

The use of XML provides several benefits to the ELP system. Firstly, the system can communicate and share information with other different databases such as a database on a CMS. Secondly, by sending requests and responses from client to server in XML strings format rather than Document Object Model (DOM), the application is able to communicate with a wide range of applications on the server independently from their platform and language implementations. Thirdly, exercises on the ELP system can be reused. The exercise can be displayed in the applet with different customised user views or can be formatted and displayed in an HTML page.

This section discusses how an exercise as described in Section 3.4.4 is represented in XML and the database structure which stores all information in the ELP system.

### **3.5.1 The Exercise XML Format**

As mentioned in Section 3.4.4, an ELP exercise consists of an exercise template which is the provided code and an exercise instance which is gap content. The exercise template consists of two files: the program code text file and an XML document which specifies the start position of each gap in the program code text file. Exercises may be comprised of multiple files; and gaps in those files are specified by character start positions. By specifying the gap start character positions, the system is able to support numerous gap types such as block type gaps and expression type gaps. In the ELP system, a gap can be any segment of a program, however a gap must be well-formed in order to carry out analysis. The well-formed gap concept and gap categories are discussed in detail in Chapter 4.

Exercise instances specify gap content. There are three different types of instance: hint, solution and My Program. Instances are also stored in XML file format in the database. Hint instance stores lecturers' instructions for all gaps in an exercise. Similarly, solution instances store all gap solutions and My Program stores a student's attempt for all gaps in an exercise. Examples of the exercise XML template, exercise solution, and student's attempt instances for the HelloWorld

exercise in Figure 11 are shown in Figure 13. Figure 14 illustrates how the exercise solution can be constructed from the exercise template, program source code, and the solution instance. The same process is applied to construct the hint view and the My Program view of the exercise from the hint instance and My Program instance.

```

<code>
<file mainclass="true" name="HelloWorld.cs" language="csharp">
    <gap id="1" charpos="20"/>
</file>
</code>

```

(a) XML Template

```

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

```

(b) Program Source Code

```

<code><file name="HelloWorld.cs">
<gap id="1">
    <![CDATA[//Display Hello World]]>
</gap>
</file>
</code>

```

(c) Hint Instance

```

<code><file name="HelloWorld.cs">
<gap id="1">
    <![CDATA[System.out.println("Hello World");]]>
</gap>
</file>
</code>

```

(d) Solution Instance

```

<code><file name="HelloWorld.cs">
<gap id="1">
    <![CDATA[System.out.println("Hello ABCD");]]>
</gap>
</file>
</code>

```

(e) My Program Instance

**Figure 13: Files that Make up an ELP Exercise**

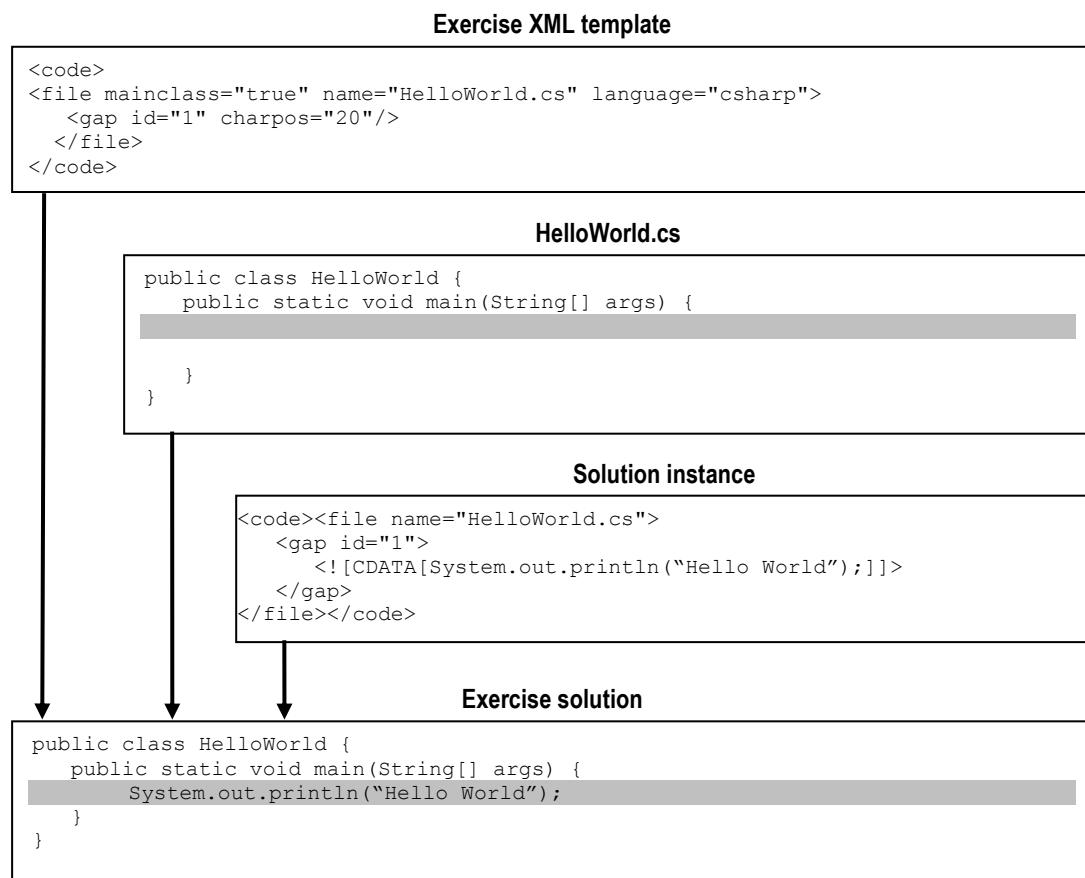
In order to construct the exercise solution, all gap start character positions are obtained from the exercise template XML file. Gap content from the exercise instances is inserted into the program code text file at the character positions specified in the XML template file. A similar approach is used to construct exercise with hints or student attempts.

With reference to Figure 13 (a), each gap in an exercise has a unique ID in the gapId attribute. This ID is used to identify the gap and help manage exercise

versioning. When an existing gap in an exercise is deleted and replaced by a newly created gap, the new gap will be assigned the gapId as the next number from the largest gapId value in the exercise regardless of how it is displayed in the editor.

As well as gap start positions, the exercise XML template also stores other important information about the exercise. Firstly, the class will be the main execution point for a multiple class exercise and is identified by setting the mainclass attribute to true, otherwise it is set to false. Secondly, the programming language for each class in the exercise is identified as either Java or csharp by setting the language attribute. Thirdly, the file name is set in the name attribute. This file name is required for Java exercises since in Java the class and its stored file name need to be the same.

As can be seen in all the XML files shown in Figure 13, CDATA is used to faithfully represent arbitrary program text.



**Figure 14: Detailed Process of Generating Exercise Solution**

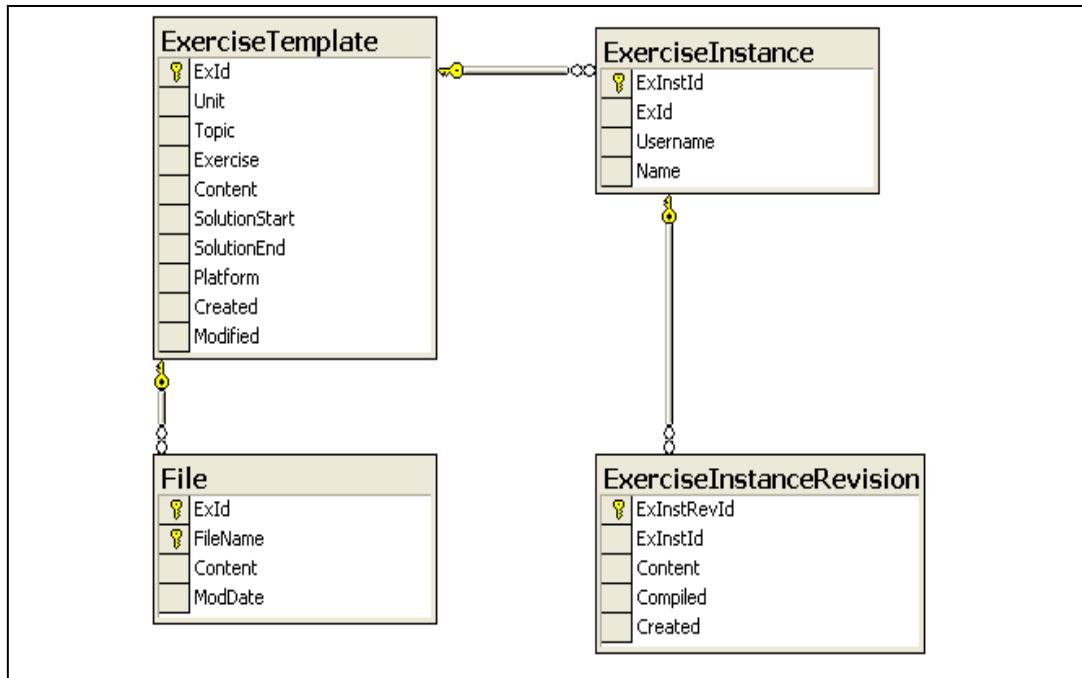
### **3.5.2 Database Structure**

As mentioned earlier, an ELP exercise consists of two components, namely, an exercise template and instances. Exercises and all students' saved attempts are stored in four separate tables in the database. These tables are ExerciseTemplate, ExerciseInstance, ExerciseInstanceRevision and File. Each exercise in the ExerciseTemplate table is uniquely identified by an exercise ID (ExId) or by combining the unit, topic and exercise name. Each exercise is associated with one hint instance, at least one solution instance and zero or more My Program instances in the ExerciseInstance table. The exercise is also associated with one or many files in the File table.

An instance in the ExerciseInstance table is uniquely identified by an exercise instance ID (ExInstId) or through the combination of three keys (ExId, Username and Name). The instance name can be hint, solution or My Program.

All students' saved attempts are stored in the ExerciseInstanceRevision table. Each save attempt is stored as a revision which is uniquely identified by the ExInstRevId. Each exercise instance ID is associated with one or more exercise instance revision IDs. Each revision is stored with a special flag to indicate if the revision is compiled successfully. The timestamp of the revision is saved for the purpose of monitoring students' progress.

The ExId field in the File and ExerciseInstance tables is used to identify which exercise a particular file or instance belongs to. Figure 15 details the relationship among tables in the database, while Table 8 details fields, field descriptions and their data types in those tables.



**Figure 15: The Database Relationship among Tables in the ELP System**

Name	Data Type	Description
ExId	int(4)	Unique ID of an exercise
Unit	varchar(100)	Unit that an exercise belongs to
Topic	varchar(100)	Topic that an exercise belongs to
Exercise	varchar(100)	Name of an exercise
Content	longtext	Exercise XML template
SolutionStart	bigint	First date of solution availability
SolutionEnd	bigint	Last date of solution availability
Platform	varchar(100)	Platform that the exercise has to be run in (either Java or C#)
Created	datetime	Template creation time
Modified	timestamp	Last modified time

#### ExerciseTemplate Table Fields Descriptions

Name	Data Type	Description
ExInstId	int(4)	Unique ID of an instance
Username	varchar(100)	Student number for My Program instance, author for hint and solution instance
ExId	int(4)	Identifies the template to which this instance belongs
Name	varchar(100)	Name of the instance which can be hint, solution or My Program

#### ExerciseInstance Table Fields Descriptions

Name	Data Type	Description
ExInstRevId	bigint	Unique ID of revision
ExInstId	int(4)	Exercise instance ID that the revision is related with
Content	longtext	Student attempt
Compiled	tinyint	Flag if the revision is successfully compiled <ul style="list-style-type: none"> <li>• null if student has not attempted compiled</li> <li>• 0 if attempted but failed and 1 if successfully compiled</li> </ul>
Created	timestamp	Time when the revision is stored

#### ExerciseInstanceRevision Table Fields Descriptions

Name	Data Type	Description
ExId	int(4)	Identifies which exercise template this source belongs to
FileName	varchar(100)	Class file name; unique to this exercise
Content	longtext	Source code; except gaps
ModDate	timestamp	Date file was last saved

**File Table Fields Descriptions**

**Table 8: ELP Database Tables Fields Descriptions**

### 3.6 Summary

In summary, this chapter has presented the ELP system which is designed to help beginning programmers to more easily learn programming in Java and C#. The system has four novel characteristics, namely: 1) it supports “fill in the gap” exercises, 2) it has a web interface, 3) it provides timely and quality feedback, and 4) it is exercise-centric. The system provides students with a scaffolding instruction mode through immediate individual feedback. It also facilitates constructivist learning through “fill in the gap” exercises which address lower four levels of Bloom’s taxonomy and a web-based user interface.

“Fill in the gap” exercises help to develop three attributes: knowledge, strategies and models that novice programmers are often lacking when compared to expert programmers. In terms of knowledge, this type of programming exercise together with frequent practice will develop goal and plan mappings. It allows students to construct a small chunk of new knowledge in a context, therefore minimises the chances of students developing fragile knowledge. In terms of strategies, gap-filling programming exercises develop students’ comprehension, code generation and problem-solving skills. Comprehension skill is improved as students need to carefully understand the provided code in order to complete the exercise. As students are offered exercises to write small, quality segment of code at the beginning of their courses, they will be able to progressively develop the right skills to complete a quality program while developing more programming knowledge. Students are also presented with effective illustrations that depict how to abstract a programming problem from its requirements and break a problem into sub-problems. In terms of the model attribute, this type of exercise also develops students’ knowledge of the problem domain. In addition, “fill in the gap” exercises reduce the complexity of writing programs since students only need to provide a small segment of code; as such, they are motivated and engaged in the learning process more

actively. When high-level OO programming languages such as Java or C# are used for teaching programming languages, “fill in the gap” exercises allow teaching staff to introduce new concepts gradually and reduce the complexity of writing programs.

A web-based user interface makes the ELP system easy to use and can be accessed at anywhere, anytime. This interface eliminates start-up hindrances that often discourage students who have difficulties in installing and learning how to use a compiler or an IDE. Therefore, the ELP enables students to start writing a program from ‘day one’. The web-based interface also allows a smooth integration between learning materials and practical exercises; because the system is readily integrated into an existing CMS.

Through the ELP, students have access to immediate feedback such as customised compilation error messages and comments on the quality and correctness of their programs. Providing immediate feedback for students means they will not feel lost and de-motivated after encountering difficulties. Customised compilation error messages help students to fix bugs more easily. Immediate feedback on the quality and correctness of programs will encourage students to reflect on and learn from their mistakes.

The system is exercise-centric. Each exercise in the ELP consists of all necessary learning notes and additional resources, together with practical exercises. The system also supports a user customisable editor and allows teaching staff to analyse students’ performance for each exercise in the system. Apart from supporting students’ learning, the system incorporates lecturer and tutor views. Lecturers and tutors can communicate and monitor students’ progress in either view. Additional functionalities are added in the lecturer view to manage exercises.

# Chapter 4 - Static Analysis

*One of the major problems in teaching programming is trying to implant good programming habits.(Ulloa, 1980)*

*No matter how careful one is to introduce clean programming habits at the very beginning of students' programming experience, students too easily fall into 'quick and dirty' programming.*  
(Cherniak, 1976)

In the previous chapter, the design and implementation of the ELP system and how the system helps novice programmers learn to program were discussed. In this chapter, the static analysis component of the program analysis framework which is designed to ensure the quality of students' "fill in the gap" programs is presented.

## 4.1 Introduction

One of the major requirements of a good programmer is the ability to write correct and good quality code. According to a recent industry survey cited by Townhidnejad and Hilburn (2002), more than 55% of a project budget is spent on improving software quality. Good quality software will consume fewer resources when it runs and reduce the complexity and cost of maintenance.

Despite the importance of software quality in the industry, computer science courses do not pay much attention to it. The topic is not introduced continuously and covered in-depth during the course (Sanders and Hartman, 1987), therefore students do not have a clear understanding about software quality (Ala-Mutka and Jarvinen, 2004). They often neglect the process of designing the program and analysing alternative solutions before coding (Howles, 2003; McGill and Volet, 1995). In Howles' study (2003), only 5% among 168 first and second year computer science and engineering students always design their code before implementing it.

Static analysis is a process of analysing a program without executing it. The analysis can give students a fine level of detail about the quality and structure of their programs (Mengel and Ullans, 1999). It also assists teaching staff in the marking process (Mengel and Ullans, 1999). Static analysis reveals program attributes such as

complexity, size, understandability, and structure of the code, as well as why the program would not perform well under dynamic analysis (Ball, 2001; Mengel and Ullans, 1999). Many static analysis tools have been previously developed either to assist instructors in the marking process or to provide immediate feedback to students about their programs. The survey conducted by Ala-Mutka (2005) grouped existing static analysis tools based on their purposes into five groups: 1) checking coding style (Jackson and Usher, 1997; Software Quality Institute, 1991; Rees, 1982; Zin and Foxley, 1991); 2) detecting possible programming errors (ESC/Java, 1990; Schorsch, 1995); 3) measuring characteristics of computer programs using software metrics (Hung et al., 1992; Jackson and Usher, 1997; Mengel and Yerramilli, 1999; Zin and Foxley, 1991); 4) assessing the design of a program (MacNish, 2000a; Saikkonen et al., 2001; Thorburn and Rowe, 1997); and 5) checking for certain features in a program (Saikkonen et al., 2001; Zin and Foxley, 1991). However, few existing systems support “fill in the gap” exercises. This type of exercise enforces the lower four levels of Bloom’s (1956) taxonomy and makes learning to program easier, as discussed in the previous chapters.

The contribution of this chapter is the proposal of a static analysis framework which is designed to check the quality and structure of “fill in the gap” programs. The framework facilitates scaffolding and constructivist learning; and has five key characteristics:

1. Analysis of “fill in the gap” exercises - the analysis can be carried out on the whole program. However, a finer level of feedback is provided for “fill in the gap” exercises because only parts of codes created by students (gaps) are checked. This makes the analysis process more accurate, simpler, and produces improved feedback as gaps in a program usually have well-defined purposes.
2. Support for multiple languages - the framework currently analyses Java and C# programming exercises and can be easily extended to support other programming languages.
3. Configurability - the framework provides a set of functions to assess the quality of a program. The operations of these functions are fully configurable by teaching staff to meet the objectives of an exercise. Thus, the analysis is flexible.

4. Extensibility - the framework is designed so that additional analyses can be plugged in easily. It can also be extended to analyse other programming languages exercises.
5. Performance of both quantitative and qualitative analysis - unlike existing static analysis tools, the framework's static analysis performs two complementary groups of analyses: Software Engineering (SE) metrics (quantitative analysis), and structural similarity analysis (qualitative analysis). The quality of a program is indicated through a set of computed SE metrics values, while structural similarity verifies SE metrics results by identifying high complexity and poor quality code areas in a program.

The key novel aspects of the static analysis are the ability to analyse “fill in the gap” programs and perform both quantitative and qualitative analyses. The analysis is integrated into the ELP system to primarily focus on evaluating student codes for the gaps. This allows the analysis to provide more detailed feedback for students since gaps in the ELP often have a well-defined purpose. As discussed in Chapter 3, gaps in gap-filling exercises often represent plans or sub-solutions which can be used by students to solve subsequent problems. Providing feedback about the quality of gap codes can help students to develop “bullet proof” code. This also reduces the complexity in setting up analyses for exercises from an instructor’s perspective.

The static analysis provides two complementary groups of analyses, namely SE metrics and structural similarity. SE metrics analysis makes use of software complexity metrics and good programming practice guidelines to evaluate the quality of students’ programs. The analysis indicates if a program is too complex or poorly coded based on various computed complexity values. However, it cannot identify the specific parts of the program which are too complex. In contrast, the structural similarity analysis verifies the result of SE metrics analysis by comparing the high-level algorithmic structure of a program with the model solution to provide detailed, quality information on high complexity and poor code areas in a program.

The static analysis facilitates a scaffolding and constructivist learning environment for the ELP. As discussed in Chapter 2, scaffolding is the most effective learning mode and constructivism is essential in teaching and learning to program. The framework provides immediate and individualised feedback on the quality and

structure of students' programs when they do their practice. The static analysis feedback allows students to reflect on their choice of algorithm and construct knowledge in the context of the current exercise, which therefore encourages them to consider alternative better quality solutions (Sanders and Hartman, 1987). Student learning is enforced and program quality is constantly addressed on the day-to-day basis. The static analysis is integrated into the web-based ELP system that enables students to receive feedback at anytime and anywhere. Constructivist learning is facilitated. Finally, the analysis augments the grading process performed by instructors and teaching assistants. It ensures the consistency in marking and helps to reduce their workload.

This chapter is divided into seven sections. Section 4.2 gives an overview about common static analysis methods and software metrics adopted in both industrial and educational applications to measure various aspects of a software product. Section 4.3 discusses novice programmers' common mistakes. Section 4.4 discusses different gap types for an ELP exercise; a well-formed gap is also defined together with examples. The design of the static analysis is detailed in Section 4.5. Section 4.6 discusses required configurations from teaching staff in order to carry out static analysis for an ELP exercise and finally, the analysis implementation is described in Section 4.7.

## **4.2 Background**

This section gives an overview about static analysis and software metrics. It consists of two sub-sections. Sub-section 4.2.1 gives an overview about automated static analysis techniques, while Sub-section 4.2.2 reviews software metrics then summarises common software complexity metrics.

### **4.2.1 Overview of Static Analysis Approaches**

As mentioned earlier, static analysis is the process of analysing a program without executing it. Static analysis ensures that program code complies with its requirements, minimising coding errors and ensuring the quality of code (Ball, 2001; German, 2003). Static analysis can be carried out on requirement, or design documents, or program code itself. However, analysis carried out on project documents is often informal and difficult to automate. The main focus of this section

is to review analyses carried out on program code, which is the most common form of static analysis.

According to Howden (1981, 1982), static analysis methods can be classified into two groups: formal and informal. The formal methods are often used to reveal various attributes of a program, and detect specific errors or anomalous constructs in a program. The informal methods may perform line by line checking of a program code for errors in an error checklist, or simulate execution of the program with sample data. Table 9 summarises common static code analysis methods in scientific environments identified by Howden (1982) and German (2003).

<b>Formal methods</b>	
<b>Name</b>	<b>Description</b>
Subroutine interface analysis	Ensuring the consistency between formal and actual parameters of called and calling methods.
Control flow analysis	Analysing the sequence of control transfers to detect incorrect or inefficient constructs.
Data flow analysis	Analysing variables in a program to ensure that there is no execution path which attempts to access an un-initialised variable; or detect assigned variables that are not used.
Information flow analysis	Identifying how execution of an unit code creates dependency between inputs to outputs from that code.
Statement analysis	Analysing program source to search for localised errors and to check the completeness of certain classes of required comments. This method is often language dependent.
<b>Informal methods</b>	
<b>Name</b>	<b>Description</b>
Check list code inspection	Inspecting program source line by line to detect errors and classifying those errors against a list of known errors checklist.
Simulated execution, symbolic evaluations	Replacing numeric variables or values with symbolic values which can be variable names, expressions in numbers, arithmetic operators or other symbolic values to understand what the code does in all circumstances for the whole range of input variables for each program section.
Compliance Analysis	Ensuring the consistency between program code and its documentation by comparing comments (in machine readable format) in a program against its documentation.

**Table 9: Common Static Analysis Program Code Methods**

In computer science education, automated static analysis is often used to check coding style, detect programming errors, measure characteristics of a program using software metrics, assess the design of a program or to check for some special features such as if a student program makes use of certain libraries or programming structures (Ala-Mutka, 2005). The analysis process is either carried out on the program source directly using various string matching techniques or adopts one of the formal methods such as control flow, data flow, or information flow analysis. In

contrasting with the industrial environment, programs in educational environments are often small in size, have a clear specification and solutions are known in advance.

So far, various common static analysis techniques and their goals have been discussed. The next section reviews software metrics which is a widely acknowledged form of static analysis used to measure various aspects of software.

#### **4.2.2 Software Metrics**

Software metrics is a well-known quantitative approach to measure various aspects of software. Software metrics can be classified into three groups: processes, products and resources (Fenton, 1991; Fenton and Neil, 2000). Process measurements are designed to evaluate software related activities and often involve a time factor. Product measurements are designed to measure any deliverables products or documentations which arise out of the software development processes. Resources are inputs to processes. Each group consists of many entities which can be evaluated by either internal or external attributes, defined as follows:

- Internal attributes of a product, process, or resource “are those which can be measured purely in terms of the product, process, or resource itself” (Fenton, 1991, 43);
- External attributes of a product, process or resources “are those which can only be measured with respect to how the product, processes, or resource relates to its environment” (Fenton, 1991, 43).

Table 10,, which appeared in Fenton (1991), illustrates the major entities in each software metrics group together with their common internal and external attributes.

ENTITIES		ATTRIBUTES	
		Internal	External
<b>Products</b>			
Specifications	Size, reuse, modularity, redundancy, functionality, syntactic correctness	Comprehensibility, maintainability	
Designs	Size, reuse, modularity, coupling, cohesiveness, functionality	Quality, complexity, maintainability	
Code	Size, reuse, modularity, coupling, functionality, algorithmic complexity, control flow structure	Reliability, usability, maintainability	
Test data	Size, coverage level	Quality	
<b>Processes</b>			
Constructing specification	Time, effort, number of requirements changes	Quality, cost, stability, ...	
Detailed design	Time, effort, numbers of specification faults found	Cost, cost-effectiveness	
Testing	Time, effort, number of coding faults found	Cost, cost-effectiveness, stability	
<b>Resources</b>			
Personnel	Age, price	Productivity, experience, intelligence	
Teams	Size, communication level, structured-ness	Productivity, quality	
Organizations	Size, ISO Certification, CMM level, ...	Maturity, profitability, ...	
Software	Price, size	Usability, reliability	
Hardware	Price, speed, memory size	reliability	
Offices	Size, temperature, light	Comfort, quality	
.....	.....	.....	

**Table 10: Software Metrics Classifications and Their Attributes**  
*(Adapted from Fenton, 1991 p.44)*

In computer science education, software metrics are often used to measure internal attributes of student programs to provide feedback to students about the quality of their programs or to assist teaching staff in the marking process (Hung et al., 1992; Jackson and Usher, 1997; Leach, 1995) giving them a better indication of students' performance (Mengel and Yerramilli, 1999). Complexity metrics are most commonly used in computer science education tools because this group of metrics provides detailed information on the internal structure of the code which instructors are trying to point out to students (Watson and McCabe, 1996). These metrics will help to make students aware of the quality and structure of their code. Common software complexity metrics measurements are described below.

#### 4.2.2.1 Software Complexity Metrics

As Fenton (1991, 153) states, complexity is “commonly used as a term to capture the totality of all internal attributes”. For example, the complexity of software at runtime is its execution time and storage required to perform the computation; for the

programmer it is the difficulty of coding, debugging, testing or modifying the software (Kearney et al., 1986). There are many complex aspects of a software product, however, structural, computational, logical, conceptual and textual complexities are the most important ones (Ejiogu, 1985). Knowing those complexity values will allow programmers to predict the cost of testing, maintenance and number of errors; assess the development time, effort, cost (Kokol et al., 1996); and ensure the quality of the software.

Software complexity measures for programs can be divided into three groups: bulk, control flow and data flow (O'Neal, 1993). Program bulk measures determine the physical size of a program based on its actual text. The underlying assumption of this approach is that the complexity of a program is directly related to its physical size. The number of lines of code measurement is the simplest bulk measurement but it is also one of the most important ones because it relates to productivity. There is no clear definition of lines of code. The most common definition of a line of code is any line of program text that is not a comment or a blank line regardless of the number of statements or fragments of statements on the line (Conte et al., 1986). This includes all lines containing program headers, declarations, executable and non-executable statements.

The complexity measurement proposed by Halstead (1977) is a well-known technique in the bulk class. The measures are based on numbers of tokens, which are the basic syntactic units distinguishable by a compiler, derived directly from a program's source code:

n1	the number of distinct operators (things which affect the value or ordering of operands)
n2	the number of distinct operands (variables or constants)
N1	the total number of operators
N2	the total number of operands

From these numbers, five measures are derived:

Measure	Symbol	Formula
Program length	N	$N = N1 + N2$
Program vocabulary	n	$n = n1 + n2$
Volume	V	$V = N * (\log_2 n)$
Difficulty	D	$D = (n1/2) * (N2/n2)$
Effort	E	$E = D * V$

There are five major advantages of the bulk metrics. They indicate the overall quality of programs, and predict the rate of errors without requiring in-depth analysis of programming structure. Further, these measurements are simple to calculate and are language independent. However, the main drawback of this measurement is that it can only be carried out on the complete program code.

Control flow measures assess the complexity of a program based on its control flow graph. The number of loops, branches and conditional statements within a program are factors considered to contribute to the complexity of the program. One of the most well-known control flow measurements is the cyclomatic complexity, introduced by McCabe (1976). Cyclomatic complexity measures the amount of decision logic in a single software module. It is based on the structure of the module's control flow graph. Cyclomatic complexity can be calculated in the following way:

1. As the number of regions in the control flow graph
2. As the number of edges – the number of nodes + 2

$$v(G) = e - n + 2$$

where G = graph, e = edges and n = nodes

3. As the number of decision nodes + 1 where a decision represents a conditional branch statement

$$v(G) = d + 1$$

where G = graph, d = decisions

Table 11, which appeared in VanDoren et al. (1997), illustrates the relationship between the cyclomatic complexity value of a software product and the error risk in it.

Cyclomatic Complexity	Risk Evaluation
1-10	A simple program, without much risk
11-20	More complex, moderate risk
21-50	Complex, high risk program
> 50	Untestable program (very high risk)

**Table 11: Cyclomatic Complexity Threshold**  
(Adapted from VanDoren et al., 1997)

Data flow measures are concerned with the amount and type of information that is passed between program components. These components can be as large as

modules and procedures in a large software system, or as small as statements within a procedure. Data flow measures are typified by the work of Henry and Kafura (1981). Their measures depend on procedure size and the flow of information into the procedure (the fan in) and out of the procedure (the fan out). The complexity of a procedure is defined as:

$$\text{length} \times (\text{fan-in} \times \text{fan out})^2$$

The main advantage of this metric is that it can be computed at the design stage, before the coding process is carried out. However, it can give a complexity value of zero if a procedure has no external interactions. This is considered to be the main drawback of the metric.

In summary, this section has reviewed common static analysis approaches; software attributes that can be measured by software metrics were outlined, followed by an in-depth discussion of common software complexity metrics. In the next section, novice programmers' common mistakes will be discussed. Understanding students' errors will guide the analysis design.

### **4.3 Novice Programmers' Common Mistakes**

Novice programmers encounter many difficulties when they first start learning to program. The three kinds of difficulties faced by most beginners are grouped as syntactic, semantic and conceptual difficulties. Beginners often fail to get their programs to compile or produce correct results. This does not help with their poor programming practices which, concomitantly, make it harder to locate errors in the program. Research has shown that students do not follow the coding standards that are given by instructors (Li and Prasad, 2005). Program quality is not considered important by students (Zaidman, 2004).

Many studies have been carried out to investigate beginning students' difficulties so that additional help can be provided to overcome those hurdles. Similarly, many coding standards are also set and given to students to ensure the quality of their programs. The section is divided into three sub-sections. Sub-sections 4.3.1 and 4.3.2 review students' common semantic and conceptual mistakes, and their programming practices reported in prior work. Sub-section 4.3.3 details the results of the research investigation on common mistakes among novice programmers at QUT.

### **4.3.1 Semantic and Conceptual Mistakes**

Semantic errors are errors in usage of a programming language. Not all semantic errors can be detected by the compiler, which leads to cases where a program is compiled and run but produces runtime errors. Similarly, logic errors can lead to the program producing an incorrect result. These types of errors are much harder to debug than syntax errors.

As mentioned in Chapter 3, Java is most commonly used in large introductory programming courses. In 2001, Microsoft introduced C# which is “simple, modern, type safe, and object oriented” (Microsoft, 2001), and has many overlaps with Java. However, since it is relatively new, it has not been popularly used as an introductory programming language in CS1 and CS2 courses. Pohl (2003), Reges (2002) and Alm et al. (2002) authored the few published works which discuss the advantages and disadvantages of C# over Java in CS1 and CS2 courses at the time of writing this thesis. Therefore, the discussion of this section is limited to Java.

Since the framework only analyses compilable programs, the following table summarises semantic and logic errors which are not reported by the compiler and which cause unexpected behaviour in Java programs based on the work of Ahmadzadeh et al. (2005), Hristova et al. (2003), Jackson et al. (2004), Jadud (2005), Taylor (2005) and Ziring (2001).

<b>Java mistakes</b>	
1	Forgetting Java is 0 index based
2	Accessing method/attribute on <code>null</code> reference variable
3	Comparing two objects using <code>"=="</code> instead of <code>.equals</code>
4	Variables are not initialised properly
5	Array index out of bound
6	Forgetting to initialise object arrays
7	Naming local method variables same as class attributes
8	Forgetting to call super class constructors
9	Treating strings as input/output parameters
10	Declaring constructors as methods

**Table 12: Novice Students’ Common Java Mistakes**

### **4.3.2 Programming Practice**

Industry leaders have criticised the lack of emphasis on software quality in introductory programming courses. Novice programmers often have poor programming practices and do not pay much attention to the quality of their programs. Throughout the literature, there are many investigations into students’

common syntax and logic errors; however, very few of these detail students' programming practices. Lewis and Mulley (1997) are among the few to have reported on students' programming practices. They have discovered three types of poor programming practices among their first year students: unused variables, identifiers similar to keywords and identifier names in different scopes. Students often have "deeply nested" selection and looping statements in their programs which is a reflection of poor design and inadequate planning.

Conversely, there are many good Java and C# programming practice guidelines set out for students in the literature (Levow, 1997; Li and Prasad, 2005; Poole and Meyer, 1996; Zaidman, 2004). Through these standards, a program's readability is affected by four factors: documentation, choice of type and identifier names, complexity of algorithm, and consistency of coding style (Zaidman, 2004). All class members and method parameters should be documented. Identifier names should be less than 12 characters long and meaningful. Variables should be declared as `final` if their values do not change throughout a program. A line of code should not be more than 80 characters long.

#### **4.3.3. QUT Students' Common Mistakes**

As mentioned earlier, only compilable programs are analysed by the static analysis. Additionally, its main focus is to judge the quality of students' programs and to provide quality feedback to aid students in fixing their logic errors; hence, logic errors, especially those subtle errors, in beginning students' programs are the main concern of the investigation.

Based on the findings of previous studies, a small survey was conducted among undergraduate introductory students in the Peer Assisted Study Scheme who were learning to program. PASS is a student mentoring scheme in which second and third year undergraduate students run additional tutorial sessions for first year students. The survey set out to discover common logic errors beginning students often make and their programming practices. The PASS group of students was chosen because they are struggling with their study and willing to improve their understanding of the subject by enrolling in the PASS scheme. Twenty-three students were interviewed in week 9 of a second semester 2002 subject. Five common logic errors were revealed, namely:

1. Forgetting the “break” statement in a case block
2. Forgetting the “default” case in switch statements
3. Confusing instance and local variables
4. Forgetting to initialise object arrays
5. Forgetting to call the super class constructor.

A separate study made the same enquiries regarding students’ programming and common bugs but asked teaching staff who have been teaching introductory programming units in the FIT at QUT for more than two years. The study revealed eight common poor programming practices among novice students listed below. This finding was verified by reviewing students’ hand-in exercises and assignments throughout the semester.

1. Too many loop and conditional statements
2. Too few methods
3. Too large methods
4. Use of global variables rather than parameters to a method
5. Use of magic numbers (literals)
6. Unused variables
7. Unnecessary checking with Boolean expressions
8. Use of inappropriate access modifiers.

To sum up, this section has reviewed students’ common semantic and logic errors in Java, and their coding practices reported in prior work. The section also discussed common mistakes and coding practices of novice programmers at QUT. In the next section, the concept of well-formed gap and different gap categories are discussed. The design of the analysis is detailed in Section 4.5.

## 4.4 Gap Categories

In the ELP system, a gap can be any piece of code or comments in a program. However, only well-formed gaps can be analysed by static and dynamic analyses. A well-formed gap is a gap which is one of the following:

1. A statement
2. A block of statements

3. A method
4. A complete class.

Figure 16 gives an example of an ill-formed gap. This gap is considered to be an ill-formed gap because it crosses between two methods.

```
using System;

class Factorial {
    public static void Main() {
        string input;
        int numb;

        Console.WriteLine("\nEnter an integer less than 17: ");

        // Read the input, parse and assign to "numb"
        input = Console.ReadLine();
        numb = int.Parse(input);
        Console.WriteLine("{0} Factorial is {1}", numb, MyFactorial(numb));
    }

    public static int MyFactorial(int number) {
        int factorial = 1;

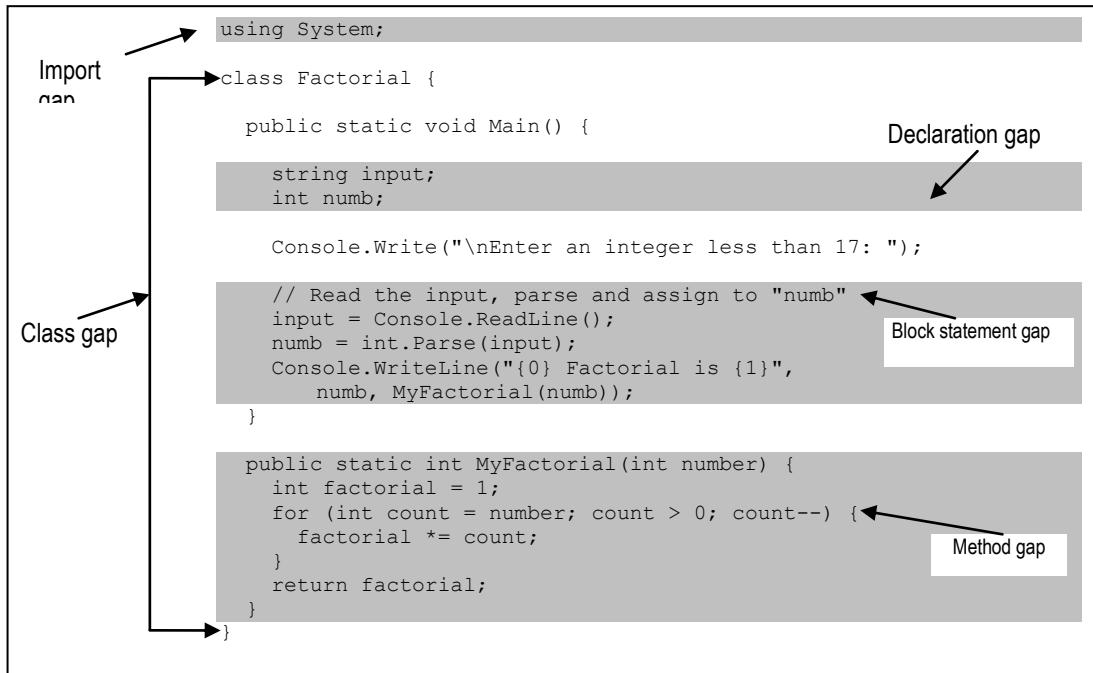
        for (int count = number; count > 0; count--) {
            factorial *= count;
        }
        return factorial;
    }
}
```

**Figure 16: Example of Ill-Formed Gap**

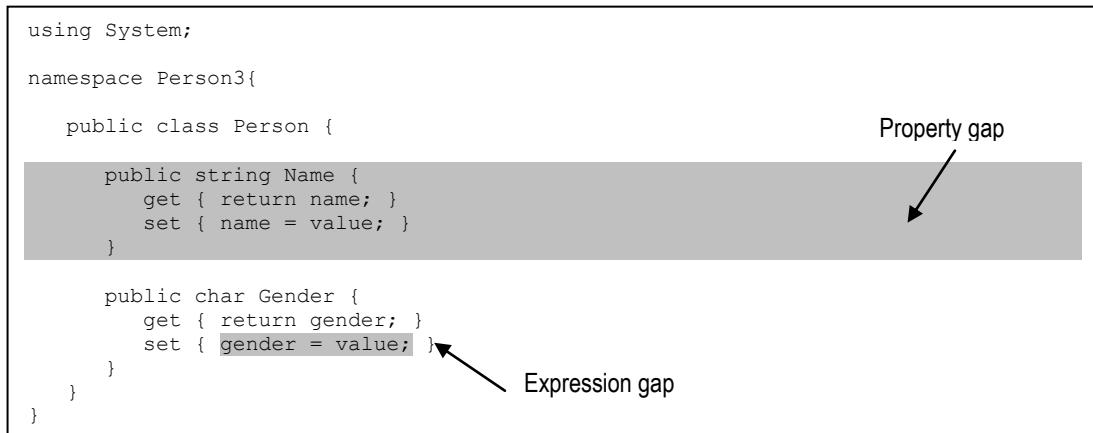
There are six identifiable well-formed gap types for Java exercises and seven for C# exercises. Table 13 summarises all gap types available in both Java and C# exercises, together with their descriptions. As shown in the table, the properties gap type is only available for C# exercises. Figure 17 and Figure 18 give an example of each gap type.

	Gap Type	Description	Language	
			Java	C#
1	Expression	Gap is an expression in a program	✓	✓
2	Block	Gap is either a statement or a block of statements	✓	✓
3	Method(s)	Gap is a method in a program	✓	✓
4	Class	Gap is a complete class	✓	✓
5	Declaration(s)	Variables declaration	✓	✓
6	Import	Import statements in a program	✓	✓
7	Properties	Properties function in C# program	✗	✓

**Table 13: Gap Types**



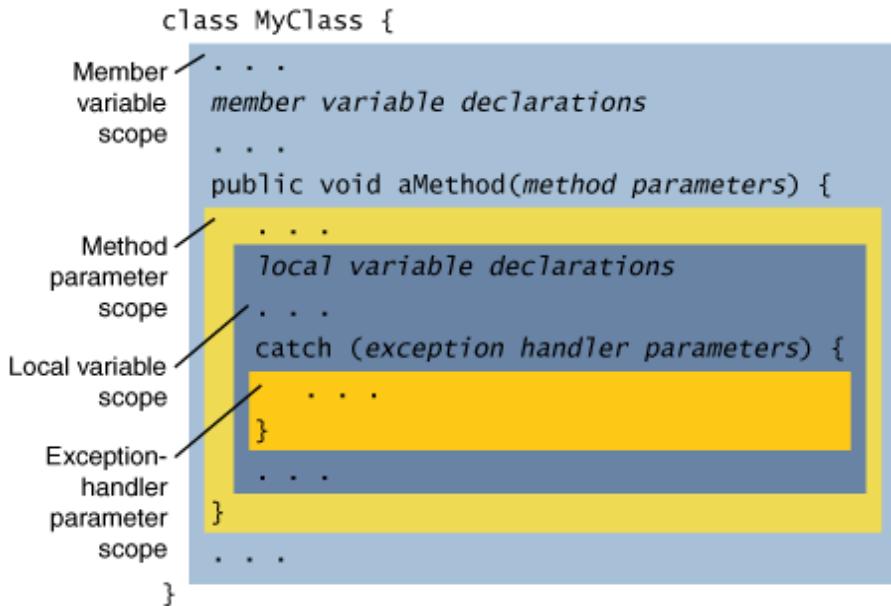
**Figure 17: An Example of Well-Formed Gaps**



**Figure 18: Examples of Gap Types**

With the declaration gap type, the analyses perform slightly different assessments depending on the scope that a variable is declared. There are four different scope types in one class (Sun, 2005). These are member variables, method parameters, local variables, and exception handler parameters.

Figure 19 (from Sun, 2005), shows all possible variable scopes in a program. However, exceptions are not taught in the first year course at QUT; hence, there are only three variable scopes. Table 14 summarises all available variable scopes in each gap type. Feedback provided to students from this analysis is comprised of variable name, its type and in which gap it is located.



**Figure 19: Variable Scopes in a Program**  
*(Adapted from Sun, 2005)*

	Member scope	Method parameter scope	Local scope
Expression	✗	✗	✗
Import	✗	✗	✗
Declaration	✗	✗	✗
Block	✗	✗	✓
Method	✗	✓	✓
Properties	✗	✓	✓
Class	✓	✓	✓

**Table 14: Gap Types and Their Available Variable Scopes**

## 4.5 The Design

In this section, the design of the static analysis is presented. The section is divided into three sub-sections. Sub-section 4.5.1 gives an overview of the analysis while Sub-sections 4.5.2 and 4.5.3 respectively detail the design of the SE metrics analysis and the structural similarity analysis.

### 4.5.1 Overview

The goal of the static analysis component is to assess the quality of students' "fill in the gap" programs. The static analysis component consists of two complementary groups of analyses: SE metrics and the structural similarity. SE metrics analysis judges the quality of student programs based on a set of computed complexity metrics and a checklist of good programming practice guidelines. It indicates whether or not a program is high quality and well coded. This is an absolute analysis.

The structural similarity analysis aims to verify the result of the SE analysis by identifying high complexity and poor programming practice areas in the program. It relatively compares the structure of a student's program against a set of expected model solutions to identify differences. Structural differences between the student solution and model solutions can be used to provide better feedback to students if their solutions ultimately result in an incorrect output in dynamic analysis. Thus, structural similarity analysis closes the gap between static and dynamic analyses which exists in earlier related research.

The key novel aspects of the static analysis are: analysing “fill in the gap” programming exercises, performing both quantitative and qualitative analyses, and having the capacity to be configured or extended. The analysis facilitates a learner-centred learning environment in the ELP system, and provides scaffolding for students. Students do their gap-filling exercises and receive immediate feedback about its quality. Analysing only gap codes of introductory programming exercises and conducting both qualitative and quantitative analyses reduce syntactic and algorithm variations which are two major obstacles identified in prior research; this allows the analysis to provide detailed, formative feedback for students. Students are shown which areas in a program need to improve with regard to quality, and recommendations on how to improve it and suggestions for alternative solutions. The analysis allows teaching staff to add or remove analyses for each gap in an exercise and configure how each analysis is carried out to address the exercise objectives. All analyses provided by the static analysis component are designed as pluggable components. Therefore, additional analyses can be easily added; the static analysis is extensible.

Neither static nor dynamic analyses can be used to diagnose programs which have syntax errors. Programs must compile first. Additionally, the static analysis only analyses gap codes. Even though a gap can be any piece of code in a program in the ELP system, only well-formed gaps are analysed by the program analysis framework to ensure that there is enough information about the context. As discussed in the previous section, there are six Java well-formed gap types and seven C# well-formed gap types. The gap categories are expression, block, method(s), class, declaration(s), import, and properties (only in C#).

When an exercise is successfully compiled, the “Check Program” button on the ELP editor applet, of which the interface and architecture are detailed in the

previous chapter, is enabled. Students can check for the quality of their programs by clicking on the button. The static analysis process is carried out on the ELP server, as shown in Figure 20. After all the analyses are carried out, the results are processed by the automated feedback engine, as will be discussed in Chapter 6, and feedback is returned to students in the format shown in Figure 21, and Figure 22. As illustrated, students are presented with a summary of all analyses that were carried out on their program. These analysis names are presented as links, together with a tick or cross image indicating if the analysis is a pass or fail. Students can click on each link to view a more detailed report of a particular analysis. With the structural similarity analysis, if the structure of the student's program is not the same as the model solution, only the high-level structure of gaps that do not match the high-level structure of the corresponding model solution gap is presented as part of the returned feedback (as illustrated in Figure 22), otherwise a congratulatory message is presented. As shown in the figure, both the structure of the student's gap solution and the structure of model solution gap are displayed so that the student can identify the differences.

By presenting high-level structures of the model gap solutions for only unmatched gaps, the analysis prevents students from adopting a learning approach whereby they first try to submit their programs and then use the model solution template to generate a more plausible solution without understanding how the code works. In addition, after static analysis is carried out, student programs will undergo dynamic analysis (discussed in the next chapter); hence, the chances of a student's plausible solutions passing both analyses are low. Students need to understand the exercise requirements and how the program works. Feedback presented to students, discussed in Chapter 6, is the combination of both static and dynamic analyses results.

In the next two sections, the design of the SE metrics and structural similarity analyses are presented, while a more detailed discussion on how feedback is generated occurs in Chapter 6.

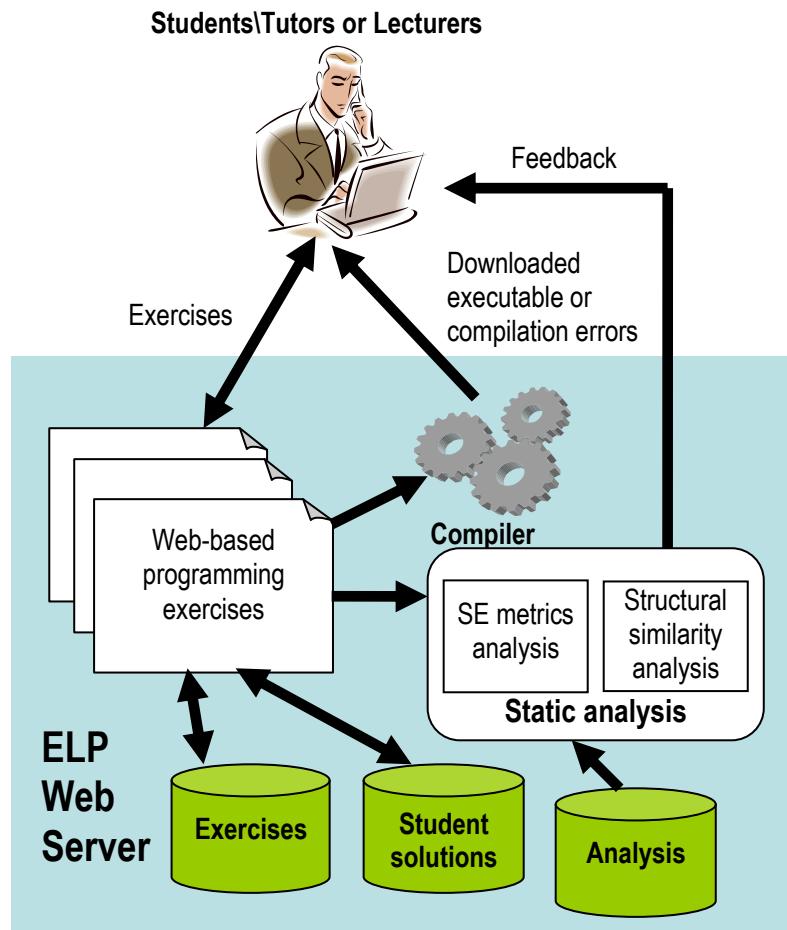


Figure 20: The Architecture of Static Analysis

Screenshot of a web-based static analysis tool interface:

- Top Bar:** My Program, Test Student [student], Save, Compile, Run, Check Program, Download, Options, Logout, OUT.
- Left Sidebar:** Annotations, Automated Feedback, Compiler Output, HELP.
- Main Content - Programming Practice:**
  - ✓ Redundant Logic Expression Check
- Main Content - Program Structure:**
  - ✓ Solution Similarity
- Report Section - Redundant Logic Expression Check Report:**

Class	Gap	Feedback
IterativeFindMax.cs	3	Great! None of the boolean expression in this gap has the following format $x==true$ (where x is an boolean expression) Keep up the good work.
- Bottom Content - Solution Similarity:**

File IterativeFindMax.cs  
 Congratulations! your program has a similar structure to the model solution.

Figure 21: Static Analysis Feedback for a Correct Attempt

The screenshot shows a software interface for static analysis. At the top, there's a menu bar with 'My Program', 'Test Student [student]', 'Save', 'Compile', 'Run', 'Check Program', 'Download', 'Options', 'Logout', and 'QUIT'. On the left, a sidebar has links for 'Annotations', 'Automated Feedback', 'Compiler Output', and 'hELP'. The main content area is titled 'Program Structure and Quality'.

- Programming Practice:**
  - Redundant Logic Expression Check:** An error message with a red X icon.
- Program Structure:**
  - Solution Similarity:** An error message with a red X icon.

Below this is a section titled 'Redundant Logic Expression Check Report' with a table:

Class	Gap	Feedback
IterativeFindMax.cs	3	There are 1 conditional expression(s) of the if statement(s) in your program perform redundant boolean expression check.

A 'Top' link is at the bottom of this section.

Under 'Solution Similarity', it says 'File IterativeFindMax.cs' and 'Gap 3'. It compares 'Model Solution' and 'Your Solution' side-by-side:

Model Solution	Your Solution
If condition is true do 1 assignment statement	If condition is true do 1 assignment statement otherwise do 1 assignment statement

A 'Top' link is at the bottom of this section.

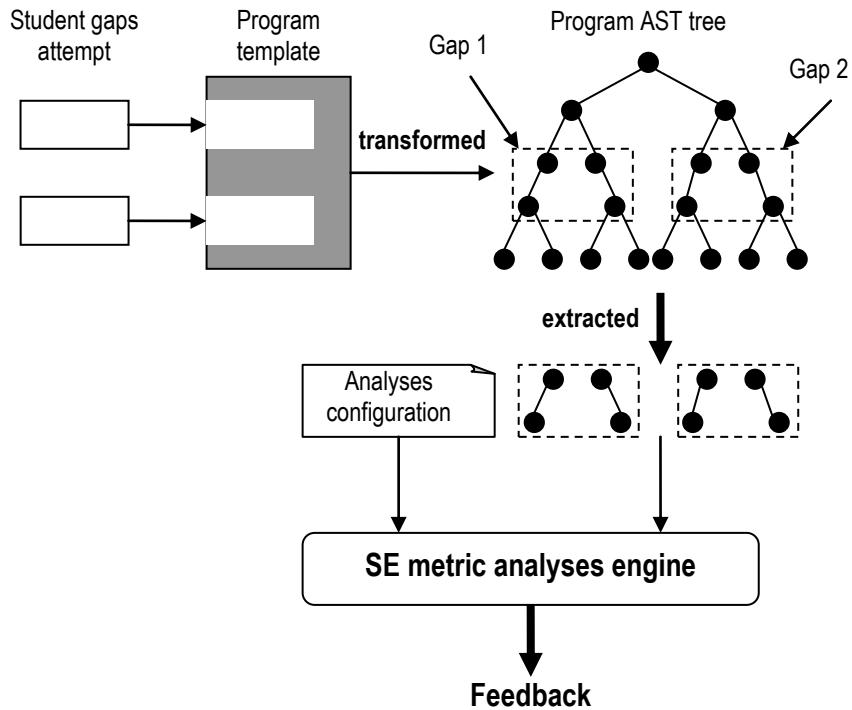
**Figure 22: Static Analysis Feedback for an Incorrect Attempt**

#### 4.5.2 Software Engineering Metrics Analyses

The SE metrics analysis group is designed to judge the quality of students' Java or C# "fill in the gap" programs based on software complexity metrics values and good programming practice guidelines. Analyses in this group either search for poor programming practices in a student program or compute the value of a particular software engineering metric. It gives comments about the overall quality of a program, however, it does not identify areas in the program which need to be improved.

All analyses in this analysis group only analyse student gap code. Student gap solutions are inserted into the program template with the beginning and end of the gap code marked. The completed student program is converted into an abstract syntax tree (AST), which captures the structure of a program. The beginning and end of each gap in the exercise are marked in the AST so that the AST representation for gaps can be easily extracted. For each gap, the gap type is identified; additional information about the gap is also obtained, before the AST representation for gap

code is then extracted from the complete program AST for analysis. Figure 23 summarises the SE metrics analysis process.



**Figure 23: An Overview of Software Engineering Metrics Analysis**

Currently, there are eight check functions provided in this analysis group which aim to address common logic errors and poor programming practices among introductory students at QUT, as discussed in Sub-section 4.3.3. Each analysis is designed as a separate component and only loaded at runtime if it is specified to be carried out for a gap through the Java Dynamic Class loading mechanism. Therefore, additional analyses can be easily plugged into the static analysis. The provided functions are:

1. Program statistics
2. Cyclomatic complexity
3. Unused parameters
4. Redundant logic expression
5. Unused variables
6. Magic numbers
7. Methods and variables access modifiers
8. Switch statement (only for Java).

Table 15 shows the description of each analysis and gap types that it operates on, together with the mistakes it aims to address.

Check	Description	Students' mistakes	Supported Language		Gap Types					
			Java	C#	Properties	Import	Declaration	Block	Method	Class
Program statistics	Count the total number of variables, statements and expressions in a gap.	Too few methods Too large methods	✓	✓	✓	✓	✓	✓	✓	✓
Cyclomatic Complexity	Count the number of logic decisions in a program.	Too many loop and conditional statements	✓	✓	✓	✗	✗	✓	✓	✓
Unused parameters	Check if there are any unused parameters in a method.	Unused parameters	✓	✓	✗	✗	✗	✗	✓	✓
Redundant logic expression	Detect redundant logical expressions check e.g. expressions “x==true”.	Unnecessary checking with Boolean expression	✓	✓	✓	✗	✗	✓	✓	✓
Unused variables	Check if there are any unreferenced variables in a specific scope.	Unused variables/Unused parameters	✓	✓	✓	✗	✗	✓	✓	✓
Magic numbers	Ensure student solutions do not have hard coded numbers or string literals.	Use of magic numbers (literals)	✓	✓	✓	✗	✗	✓	✓	✓
Methods and variable access modifiers	Ensure variables and methods have the correct modifiers.	Use of inappropriate access modifiers	✓	✓	✗	✗	✗	✗	✓	✓
Switch statement	Ensure that all switch statements have “default” case and in each case block there is a “break” statement.	Forgetting break statement in a case block Forgetting default case in switch statement	✓	✗	✗	✗	✗	✓	✓	✓

**Table 15: Static Analysis Functions, Their Descriptions and Addressed Mistakes**

A program statistics function is designed to ensure that a student program does not contain methods that are too large. Research has shown that student programs often have long methods and there is a strong relationship between a program's size and its error rate (Endres, 1975; Khoshgoftaar and Munson, 1990). The analysis counts the number of variables, statements, and expressions in a gap and the total of those values. These values are then compared with those from the expected model solution, or checked if they are in an acceptable range specified by teaching staff to identify if the program is too complicated. For method(s) and class gap types, the statistics is calculated for each method in the gap, otherwise it operates in the gap block code. Figure 24 shows an example of the program statistics outputs for a gap.

```

int temp;
if (num == 2) {
    // Base case
    return 1;
}
else {
    // recursive step
    arf = num / 2;
    return (1 + divideByTwo(arf));
}

```

Variables	1
Conditional statement	1
Return statement	2
Method call	1
Expression	3

**Figure 24: An Example of Program Statistics Output**

The check cyclomatic complexity analysis aims to ensure the student solution for a programming problem is not too complex. The analysis measures the number of linearly independent paths through a program module based on the McCabe cyclomatic complexity discussed in the previous section. This value reflects the number of conditional and loop statements in a student solution. This metric is adopted because it provides useful information about the structure of a program. It helps to detect complex code in a student solution since the analysis can be carried out for gaps in a class rather than the complete program. Additionally, the use of this metric can be extended to test the thoroughness of a student's submitted test data in dynamic analysis. Depending on gap types, complexity value is calculated per

method for method(s), and class gap types, otherwise it is only calculated to the block of statements in gaps.

From the complexity value, a student program is judged as to whether or not it is too complex using a set of threshold values. A customised configurable threshold is set up for the analysis, since the threshold (mentioned in Sub-section 4.2.2) is not applicable due to the simplicity of the type of programming problems that novice programmers have to do. Currently, if a gap has less than five decisions, it is considered very easy to understand, and from five to nine is considered easy; otherwise, students are recommended to rewrite their solution. These are default values from the analysis; teaching staff can set their own customised value if needed.

Complexity Values	Feedback
< 5	very simple
5 – 9	simple
> 9	too complex, required to rewrite

**Table 16: Cyclomatic Complexity Default Values**

The check unused parameters analysis is designed to check if any parameters in a method declaration are declared but not referenced in the method. Unreferenced parameters are not detected by the compiler but make the debugging task harder. As shown in Table 15, this function is available in method and class gap types. Similarly, check unused variable analysis is designed to detect variables which are declared but not referenced in a program. Even though compilers do provide warning messages on unused variables in the program, novice programmers tend to overlook these messages. This analysis can be set to carry out in block, method and class gap types.

The redundant check logic expression analysis is designed to ensure that students do not perform unnecessary Boolean expression checks. Similar to the other two analyses, a redundant check logic expression does not cause compilation error but makes the debugging task harder. Excluding the import and declaration gap types, this analysis can be applied for all other gap types. Some examples of students' poor Boolean expressions are shown in Table 17, together with their correct forms. The analysis reports to students expressions which perform redundant Boolean checks and their line numbers in the program.

Example	Poor programming practices	Good programming practices
Ex1	<code>if (a==true) { }</code>	<code>if(a) { }</code>
Ex2	<code>if(a){ } else if(!a){ }</code>	<code>if(a){ } else{ }</code>

**Table 17: Examples of Redundant Logic Expression Check**

The check magic numbers analysis is designed to detect any hard coded numbers or string literals in a gap code. Hard coded numbers and string literals in the programs will reduce the maintainability of a program. The default permitted values are 0, 1, 0.0, 1.0, null and “”. However, teaching staff can specify other allowable values or string literals in the configuration. Similar to the redundant logic expression check, the check magic numbers analysis also reports a list of hard coded numbers or string literals and their line numbers in the program.

The access modifier analysis checks the access modifiers for both global variables in an OO class and method declaration. The analysis is fully configurable by teaching staff. They can specify all variables or methods in an OO class to have the same access modifiers or a certain number of public, private, or protected methods or variables in the class. Furthermore, teaching staff can also enforce the number of methods, variables or their names in a class.

The check switch statement analysis ensures that every case block in a switch statement has a break statement and every switch statement has a default case. If a switch statement does not have a default case and none of the cases in a switch statement is met, subtle errors might occur. This function is only available for Java programming exercises because the Java JDK compiler does not enforce this, while the C# compiler does.

In summary, this section has discussed the design of the SE metrics analysis. Analyses in this group identify poor programming practices and point out high complexity gaps in a program which need to be revised. However, it does not provide further information on areas where the complexity lies and hints on how to improve the quality of the gap. As Soloway and colleagues (1983) commented regarding checking the quality of student programs using software metrics (quantitative analysis) only is not sufficient to explain to students why a program is harder to understand than the others. Comparing their programs with the expected model solution (qualitative analysis) will pinpoint trouble areas in the code and

provide information on how to improve it. The next section will discuss the design of structural similarity analysis, a qualitative analysis to judge the quality of students' programs.

### **4.5.3 Structural Similarity Analysis**

Structural similarity analysis is a qualitative analysis designed to verify SE metrics analysis results by comparing the structure of a student solution with a set of model solutions. SE metrics analysis results indicate if a program is complex or poorly coded, while the structural similarity analysis locates high complexity, lengthy or poor programming practice codes in the program. The two analyses complement each other.

In this analysis, the structural comparison between a student's solution and the model solution can be considered as comparing two versions of a program, since the two programs are trying to achieve the same outputs. Only the gaps codes are different. The section is comprised of two sub-sections. In Sub-section 4.5.3.1, the challenges in comparing two versions of a program that have been identified in previous research are presented. The discussion is limited to a comparison of student programs which are small and have a clear program specification. In Sub-section 4.5.3.2, the design of the analysis is detailed, together with discussion of how existing challenges are addressed.

#### **4.5.3.1 Challenges in Identifying Differences between Two Programs**

Identifying differences between two versions of a program is often required in order to debug or diagnose student programs. It can be carried out directly from the program code using a string matching technique or based on the program dependence graphs. The literature has reflected that syntactic and algorithm variations are two main challenges in comparing two versions of a programs (Rich and Wills, 1990; Song et al., 1997; Xu and Chee, 2003). Overall, there are thirteen possible semantics preserving variations that can occur between two programs identified by Xu and Chee, as summarised in Table 18.

	<b>Differences</b>	<b>Description</b>
1	Algorithms	Different algorithms used to solve a problem but produce same output
2	Source code formats	Different in spaces, indentation and so on
3	Syntax forms	Use different syntax forms to express a certain program element such as an output message can be cascaded in one print statements or separated into multiple print statements
4	Variable declarations	Use different temporary variables, can be declared at method level or inside the scope of the block.
5	Algebraic expression forms	Use different algebraic expression forms
6	Control structures	Use different control structures in expressing the same control
7	Boolean expression forms	Different Boolean expression forms
8	Temporary variables	Different numbers of temporary variables or block temporary variables
9	Redundant statements	Programs might contain dead code or statements for debugging purposes
10	Statement orders	Statements are placed in different order
11	Variable names	Different parameters names, temporary variables names
12	Program logical structures	Different program structures
13	Statements	Different statements but results in same computation

**Table 18: Possible Semantics Variations between Two Programs**

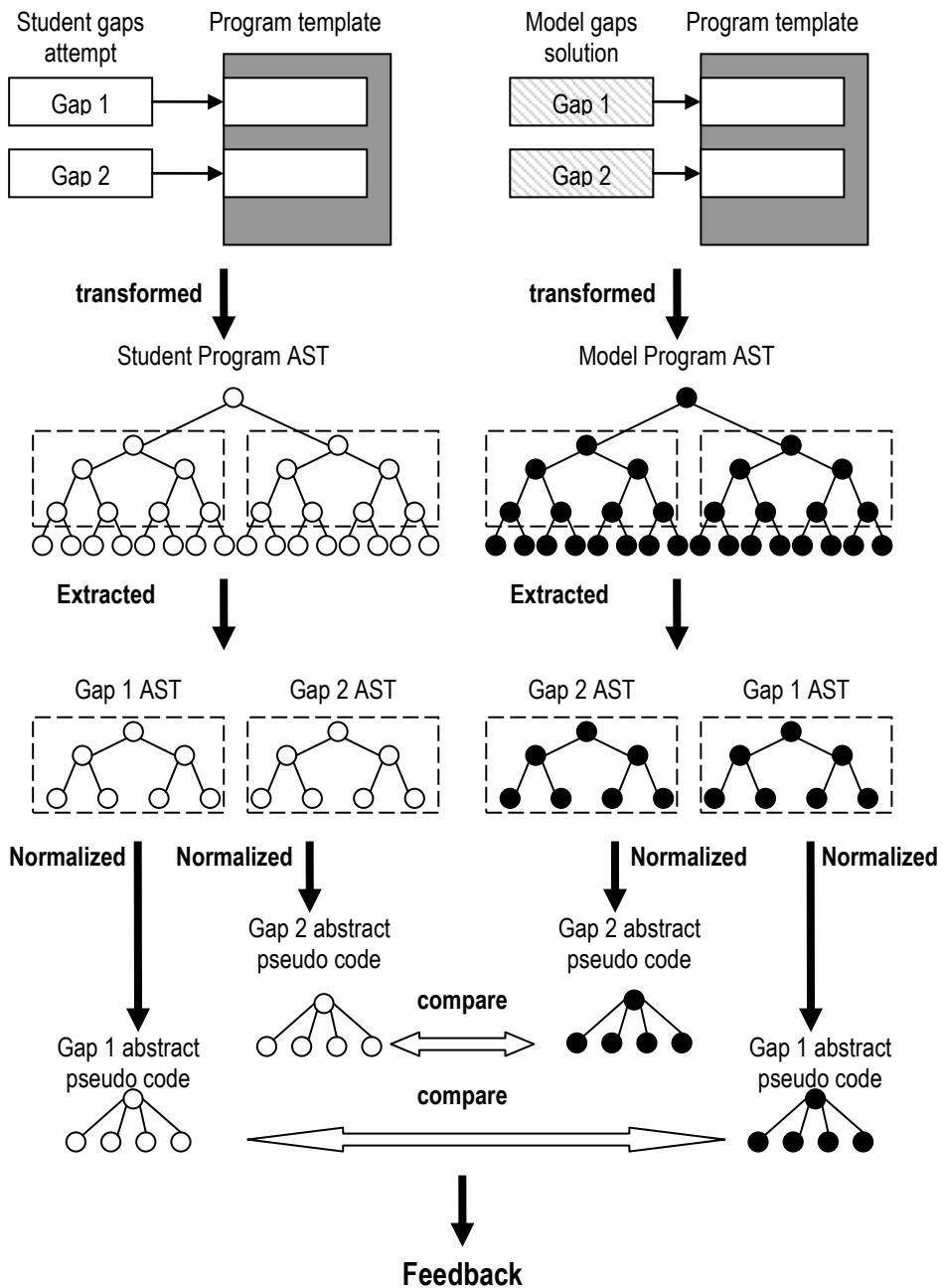
Due to these variations, comparison based on program text using Unix `diff` or other text comparison tools cannot accurately pinpoint the syntactic differences between the two programs (Horwitz, 1990; Yang, 1991). However, comparing two versions of a program based on its program dependence graph can address those variations. This is the most commonly used method to identify differences between two programs. In this approach, variations are removed through program transformation before the comparison process is conducted. There are four important types of program transformation used in an educational application to compare a student solution with the expected one: change of indentation, transformation of a program fragment into an equivalent fragment in the same language, modification of a program fragment so that it performs some other tasks, and translation of a program from one language to another (Czejdo and Rusinkiewicz, 1985).

LAURA (Adam and Laurent, 1980), TALUS (Murray, 1986), Programmer's Apprentice (Rich and Waters, 1990) and SIMPLEs-II (Xu and Chee, 1999) are some well-known debugging systems which perform program transformation to remove semantic variations before carrying out the comparison process. LAURA (Adam and Laurent, 1980) translates source codes into flowchart-like graphs. The graphs are standardised to remove variation with regard to multiple assignments, intermediate

variables and so on. TALUS (Murray, 1986) transforms source codes from an extended Lisp dialect to a simpler dialect and normalises selection structures to remove variations. Programmer's Apprentice (Rich and Waters, 1990) translates source codes into plan diagrams that are free from syntactic variations and non-contiguity of clichés. SIPLes-II (Xu and Chee, 1999) performs comparison based on an augmented object-oriented program dependence graph which is the result of parsing programs into AST before transforming and producing control flow graphs and further transformation.

#### **4.5.3.2 The analysis design**

Similar to the SE metrics analysis group, the structural similarity analysis is only carried out on gap codes in a program based on program AST. Both student and model solutions are transformed to AST and extracted as in the SE metrics analysis group. However, these AST are then normalised to abstract a pseudo code form to minimise the syntactic variations before the comparison process is conducted. Figure 25 illustrates the four steps that are carried out in the structural similarity analysis. In the first step, student programs are transformed from program source to AST. The second step involves extracting gap AST from the program AST. The gap AST is normalised to abstract a pseudo code form in the third step before the student normalised gap AST are compared with the associated model ones.



**Figure 25: Overview of Structural Similarity Analysis**

The first two processes in this analysis are identical to those in the SE metrics analysis, hence this section details the design of the analysis focusing on the normalisation and comparison processes which are the two important operations. It presents techniques used to analyse different gap types, together with discussion on how the syntactic and implementation variations are addressed in the analysis design. Last but not least, it also elaborates on how the two key characteristics of the static analysis component - configurable and extensible - are designed in this analysis.

As mentioned in Section 4.4, there are seven different gap types in ELP exercises: expression(s), block, method(s), class, declaration(s), import and properties. All of those can be analysed by this analysis except for the expression gap type. Among those analysable gap types, structural similarity analysis for declaration and import gap types does not require a normalisation process because these gap types are simple. The AST of these gaps are traversed and compared directly. The normalisation process is carried out for the remaining gap types.

Normalisation is the process of generalising programming language constructs which perform the same operation or abstract away unimportant information. In this analysis, the normalisation process is designed to accommodate syntax variations. It makes all different syntaxes of loops, conditionals, input and output statements become one. These normalisations are provided as functions so that additional normalisations can be added in later; the analysis is extensible. Figure 26 gives an example of a gap code and its normalised form. As can be seen in Figure 26, the normalisation process retains the program structure but only generalises the syntax of loop, conditional statement and sums the total for each type of statement in a gap such as number of expressions, and read and write statements. Consequently, program source 1 and source 2 have the same normalised form even though program 1 uses the `while` loop and program 2 uses the `do while` loop; similarly `println` statements are used in source 1 but `print` statements are used in source 2.

Program source 1	Normalised version
<pre>guess = reader.readInt("Guess a number " +                       "between 1 and 100 "); while(guess != secret){     if(guess &lt; secret){         writer.println("Your guess is low");     }else {         writer.println("Your guess is high");     }     guess = reader.readInt("Guess a " + "number                            between 1 and 100 "); }</pre>	<pre>1 assignment statement 1 read method call Loop {     Conditional statement     True branch         1 write method call     False branch         1 write method call     1 assignment statement     1 read method call }</pre>
Program source 2	Normalised version
<pre>guess = reader.readInt("Guess a number " +                       "between 1 and 100 "); do {     if(guess &lt; secret){         writer.print("Your guess is low");     }else {         writer.print("Your guess is high");     }     guess = reader.readInt("Guess a " + "number                            between 1 and 100 "); }while (guess != secret);</pre>	<pre>1 assignment statement 1 read method call Loop {     Conditional statement     True branch         1 write method call     False branch         1 write method call     1 assignment statement     1 read method call }</pre>

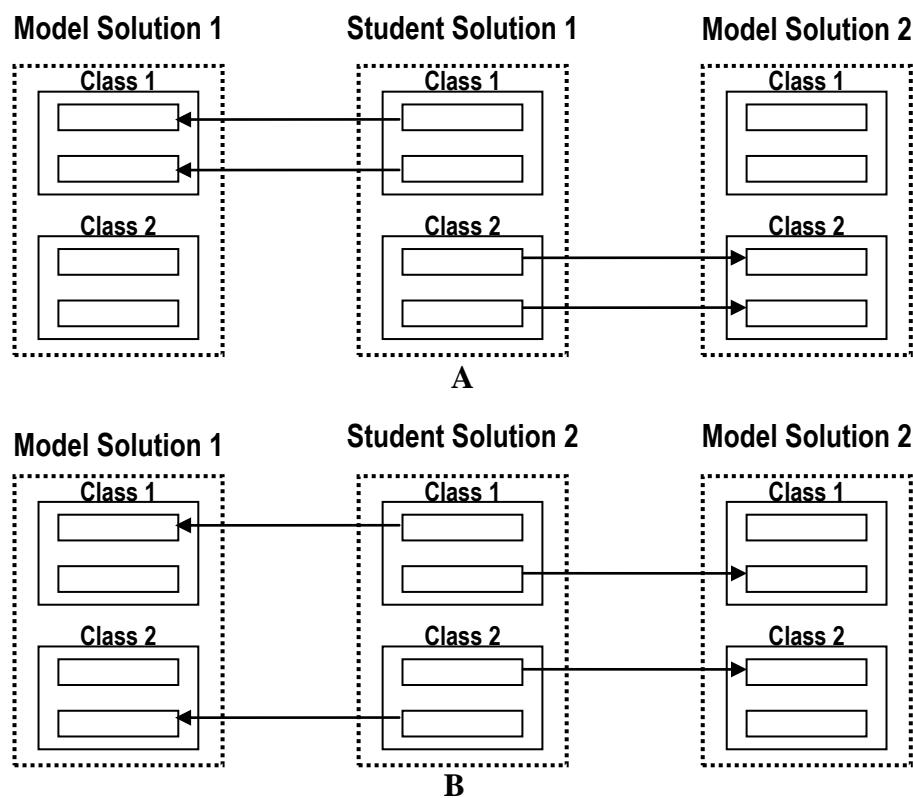
**Figure 26: Gaps and Their Normalised Forms**

After the AST of gaps are normalised, the comparison process is carried out. This involves selecting a set of model solutions which have a similar structure to the student solution and comparing these solutions with the student solution. In order to identify a set of model solutions which have a structure similar to that of the student one, the algorithmic structure of the student gap solution is compared with the algorithmic structure of all available model solutions of the exercise. For one class exercise, a model solution is selected from a list of solutions to carry out the analysis if, and only if, algorithmic structures of all gaps in the student solution are matched with those in the model solutions. For a multiple classes exercise, a solution is selected if all gaps in a class in a student's solution are matched with the corresponding one in the solution. Figure 27 illustrates how the algorithm selection is conducted for an exercise that has two classes and where there are two gaps in each class. In Figure 27A, both model solution 1 and 2 are selected to carry out further comparison for student solution 1 but not for student solution 2. This is because not all gaps in a class in student solution 2 match all gaps in the corresponding model solution class.

After a set of model solutions with a structure similar to the student solution is selected, an extensive comparison process is carried out. The numbers of input, output statements, expressions, method calls, and variables in a gap are compared

with those in the corresponding model solution gap. Differences in number in those values are recorded, together with the gap ID in the program in the result which will be processed by the feedback engine to provide feedback to students. By comparing these values, the analysis can identify gaps which contain redundant or poor code structure. Feedback provided to students identifies gaps that have poor structure, together with the expected solution structure for that gap.

If the system cannot find a match between the student solution and all available model solutions, the student solution is stored in a special location in the server which is then later reviewed by teaching staff. If the instructor recognises that it is another allowable solution for the exercise, it can be added to the model solution list. This will, step by step, build up a comprehensive list of possible solutions for a programming problem: that is, the system can learn. This also helps the analysis to address syntactic and algorithm variations.



**Figure 27: Algorithm Selection Process**

The algorithm selection process described above is quite constrained with larger gap types - such as method and class gap types - because it enforces the level of nested loop and conditional statements and the order of those statements in a

program. If the approach is used, large amounts of possible gap solutions are required. To accommodate large gap types, the analysis allows teaching staff to configure the comparison process to be less strict. They can specify how many methods, loop and conditional statements are required in a correct solution.

So far, this section has presented the processes of structural similarity analysis. The remainder of the section details how syntactic and algorithm variations - which are two major challenges in comparing two versions of a program - are addressed in the design.

Research has been carried out to develop techniques to tackle those variations (Hattori and Ishii, 1996; Horwitz, 1990). However, most of these techniques are based on control flow or data flow or both graphs' representation of a program, which are all more suitable for large and complex programs. Introductory programs are often less than 50 lines of code and consist of two or three methods. In this analysis, only gap codes of introductory programs are used. Hence, the main conjecture is that the algorithm and syntax variations are small, and analysis based on an abstraction of the AST is sufficient.

Syntax variations are eliminated by allowing configurable normalisation and matching processes. These enable the analysis to be used with different types of exercises. As noted in Sub-section 4.5.3.1, there are thirteen semantic variations among two programs identified by Xu and Chee (2003); Table 19 shows which of those variations can be addressed by the analysis, together with techniques used to address them.

Differences	Default	Configurable	Not Handle	Techniques
Algorithms	✓			<ul style="list-style-type: none"> <li>Collect a set of model solutions for a problem</li> <li>Make the system learn</li> </ul>
Source code formats	✓			<ul style="list-style-type: none"> <li>Operate on AST tree, hence program indentation is ignored</li> </ul>
Syntax forms	✓			<ul style="list-style-type: none"> <li>Normalisation process</li> </ul>
Variable declarations		✓		<ul style="list-style-type: none"> <li>Teaching staff can configure to ignore the number of variables or specify the requirement</li> </ul>
Algebraic expression forms			✓	
Control structures	✓	✓		<ul style="list-style-type: none"> <li>The normalisation process generalises all different syntax of loop and conditional statements</li> </ul>
Boolean expression forms			✓	
Temporary variables		✓		<ul style="list-style-type: none"> <li>Teaching staff can configure to ignore the number of variables or specify the requirement</li> </ul>
Redundant statements			✓	
Statement orders	✓	✓		<ul style="list-style-type: none"> <li>By default, the normalisation process sums all the same statements before, inside and after control statements</li> <li>Can be configured to sum all despite control statements</li> </ul>
Variable names	✓	✓		<ul style="list-style-type: none"> <li>The normalisation has abstracted away variable names</li> <li>Can be configured to retain certain interested variable names</li> </ul>
Program logical	✓	✓		<ul style="list-style-type: none"> <li>Collect a set of model solutions for a problem</li> <li>Make the system learn</li> </ul>
Statements		✓		<ul style="list-style-type: none"> <li>Teaching staff can configure to ignore non-important statements</li> </ul>

**Table 19: Handled Variations by the Static Analysis**

As can be seen in the table, among thirteen semantic variations, ten are addressed by the static analysis. Algebraic expression forms, Boolean expression forms and redundant statements are the three variations not handled by the analysis. By collecting all the possible solutions for a programming problem and making the system learn, the analysis can handle widely different algorithm and program structures that students may have used to solve a programming problem. Since the analysis operates on the AST, it eliminates all differences in spaces, indentation and other formats in code. The normalisation process addresses some of the variations.

The trade-off of this analysis is that the more detailed feedback that educators would like to provide to students, the more configuration is required to set up the analysis. Similarly, with larger gaps such as methods and classes, the more flexible the algorithm variation, the less detailed feedback the static analysis can provide. The main drawback of the analysis is that it cannot tackle differences in the redundancy of statements or variables because none of the data flow is analysed.

Overall, this section has discussed the design of both SE metrics and structural similarity analyses and explained how the two analyses complement each other. The SE metrics analyses aim to provide an overview about the quality of student solutions using SE metrics and poor programming practices reported in the literature. The structural similarity analysis verifies the results of SE metrics by identifying high complexity and poorly coded areas in a program. This is done by comparing the high-level structure of a student solution with an expected model solution. Analyses results are then further processed by the automated feedback engine to provide feedback to students. The next section will discuss required configurations for existing well-formed gap ELP exercises in order to carry out the static analysis.

## 4.6 Author Configuration

One of the key characteristic of the static analysis is its flexibility. This section discusses in detail the configuration that is required for each analysis in the SE metrics analysis and the structural similarity analysis.

With the SE metrics analysis, many different analyses can be carried out for a gap in an exercise. In order to set up an SE metric analysis for a particular gap in an exercise, teaching staff only need to specify the analysis name and the gapId. An example of the configuration is shown in Figure 34. However, there are five analyses in this group which can be customised to address particular exercise requirements. Available configurations for those analyses are discussed according to the following areas:

1. Program statistics - designed to check if a student program has too few or too large methods. This can be done by either comparing the statistics of a student program with a set of expected model solution statistics or checking if the statistics conform to a range of values.

2. Cyclomatic complexity - counts the number of logic decisions in a program to evaluate if a student solution is too complex. As for the program statistics analysis, teaching staff can customise the acceptable number of logic decisions in a program in three ways. The first approach is using the system default value ranges (as shown in Table 16) and no configuration is required. The second approach is to customise the default value ranges to suit an exercise. In the third approach, the complexity value of a student solution is compared with the model solution.
3. Check magic numbers - detects if a student solution has hard coded numbers or string literals. As mentioned earlier, the system has a set of default allowable values in a program which are 0, 1, 0.0, 1.0, null and "", otherwise considered magic numbers and string literals. Teaching staff can add additional allowable values and string literals in a program if required.
4. Check unused variables - identifies unreferenced variables in a specific scope. By default, the analysis searches for unreferenced variables in all the available scope in a program. However, teaching staff can optimise the analysis to search for only a specific scope in the program.
5. Check access modifiers - ensures variables and methods have the correct modifiers. There are two configuration options available for this analysis. Teaching staff only have to provide the number of public, private or protected methods or variables in a program in the first option. In the second option, both access modifiers and the associated methods or variables name are required.

Unlike SE metrics analysis, structural similarity analysis can only be performed at the class level of an exercise to minimise the number of possibilities of dependency among gaps in a class. In order to set up structural similarity analysis for a class in an exercise, teaching staff only need to specify the analysis name in the configuration of the first gap in the class. For multiple classes exercises, teaching staff can select which classes will undergo structural similarity analysis.

## 4.7 The Implementation

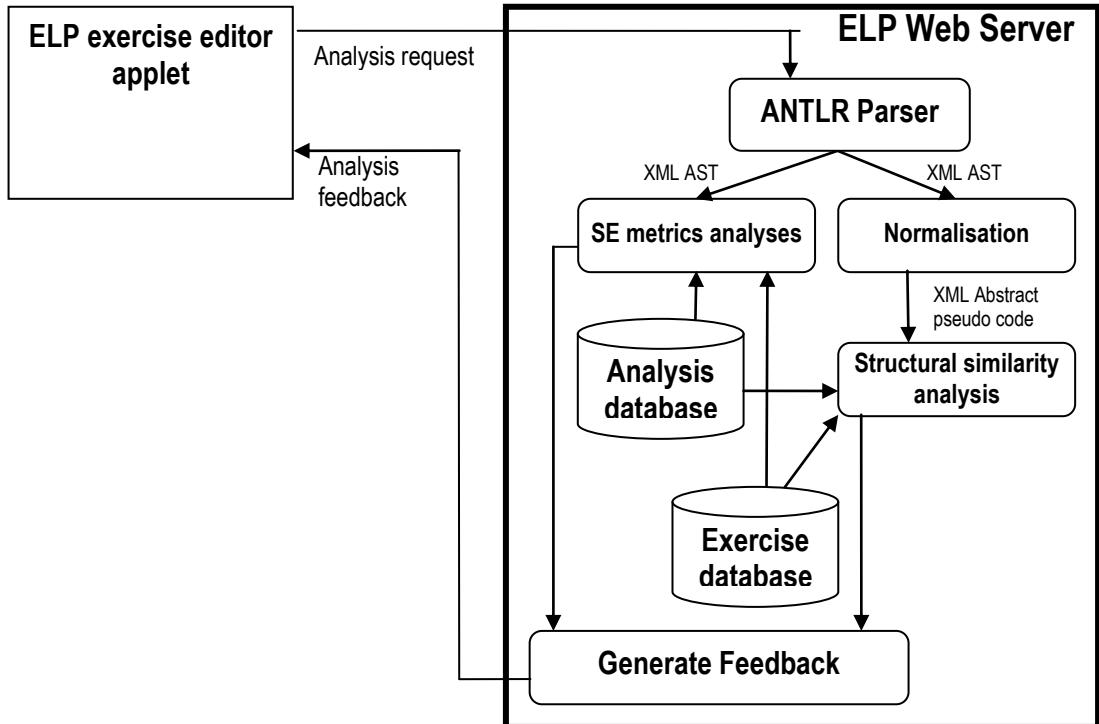
In the previous sections, the design and configuration of SE metrics and structural similarity analysis are discussed. The focus of this section is to describe how the designs of those analyses are implemented.

When an exercise is submitted for analysis, it is first converted to an XML marked-up AST using the ANTLR (Parr, 2003) parser. The SE metrics and structural similarity analyses operate on the program XML represented as AST. All specified SE metrics analyses are carried out on students' gap solutions. In order to conduct structural similarity analysis, the abstraction of the gap model solution AST and student gap AST are obtained from the normalisation process. These two abstract documents are compared with each other to identify high complexity and poor code structure in gaps. These analysis results are stored on the analysis database which will be used by the automated feedback engine to provide feedback to students. Figure 28 illustrates the detailed implementation of the static analysis.

There are three key novel aspects of the analysis implementation. First, the analysis makes use of XML to represent program ASTs, analyses, configurations and results. This makes the traversal, access and normalisation of tree nodes much easier and more efficient with other well-supported technologies such as XPath, XQuery and XSLT. Analyses configurations and results are stored in XML format in a database on the server. This makes analyses configurations more readable and easier to define, while the use of XML to store analyses results enables the feedback provided to students to be easily customised to suit students' needs. This is because XML separates content from its representation.

The second key novel aspect in the analysis implementation is the adoption of an extensible and flexible parser to parse Java and C# programs to AST. The ANTLR parser was chosen as the program parser for the static analysis because of its flexibility. The parser is able to parse programs written in Java, C++, Sather, C# and many other programming languages. Furthermore, the tool is able to retain comments and indentation of a program. Therefore, it is easy to extend the analysis to check for program coding style. By default, the AST generated by ANTLR is in extended Backus–Naur form (EBNF). A tree walk program was written to convert the EBNF AST to XML AST and store it in a Document Object Model (DOM). The third key

novel aspect is that a Java Dynamic Class Loading mechanism is used to load analyses at runtime.

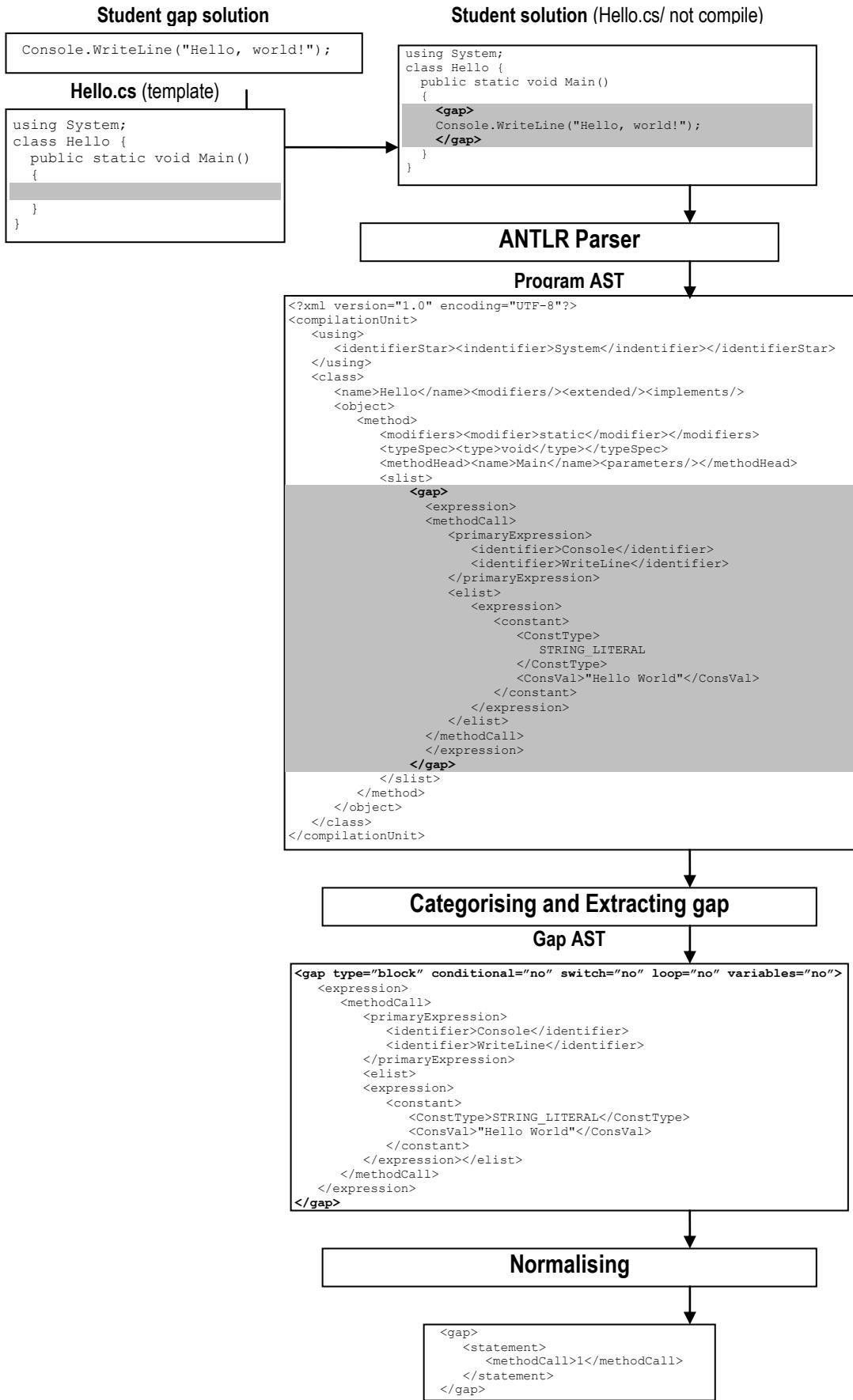


**Figure 28: Implementation Details of Static Analysis**

As mentioned earlier, only gap code in a program is analysed. In order to differentiate gap code from the provided code, the `<gap>` and `</gap>` tags are inserted at the beginning and end of each gap during the complete program code construction process. Hence, the resulting class does not compile. Since the ANTLR parser only processes syntactically correct programs, modifications have been made to the Java and C# grammars of the parser to parse and retain the two special tags in the resulting AST and therefore the AST of gap codes can be easily extracted from the complete program AST for analysis. Figure 29 shows the HelloWorld C# program with gaps marked and the resulting XML AST.

Before the AST representations of gaps are extracted from the AST of the complete program, gap AST are processed to determine gap types which can be either expression, block, method(s), class, declaration(s), import(s) or properties. At the same time, information about the number of conditional, switch, loop statements and number of variables in the gap code is also recorded. This information is checked before any analysis is carried out; this enhances the efficiency of the analysis. For

example, if the check redundant logic expression may be specified to be carried out in a gap, however, if the information indicates that there is no loop and conditional statement in the gap, no analysis will be carried out. Figure 30 shows an example of the AST representation of an extracted gap which pre-processes information recorded in <gap> tag.



**Figure 29: Student Solution Representations at Each Stage of the Analysis**

```

<gap type="block" conditional="no" switch="no" loop="no" variables="no">
    <expression>
        <methodCall>
            <primaryExpression>
                <identifier>Console</identifier>
                <identifier>WriteLine</identifier>
            </primaryExpression>
            <elist>
                <expression>
                    <constant>
                        <ConstType>STRING_LITERAL</ConstType>
                        <ConsVal>"Hello World"</ConsVal>
                    </constant>
                </expression></elist>
            </methodCall>
        </expression>
    </gap>

```

**Figure 30: Example of Extracted AST Representation of Gap**

As noted earlier, one of the key characteristic of the static analysis is extensibility. All analyses implement a common StaticAnalysis interface, shown in Figure 31; they are stored in the “staticanalysis” directory on the server and only loaded at runtime when they are specified as one of the required analyses for a gap using the Java Dynamic Class Loading mechanism. SE metrics analyses are invoked by calling the analyse method of the interface while the similarity method is invoked for the structural similarity analysis. The short and long descriptions of an analysis can be retrieved by invoking the getLongDes and getShortDes methods accordingly. This mechanism allows new analysis which can be in the form of a Java class file or JAR package to be added to the static analysis component easily by saving it to the located folder. The only requirement for the new component is that it needs to implement the StaticAnalysis interface.

```

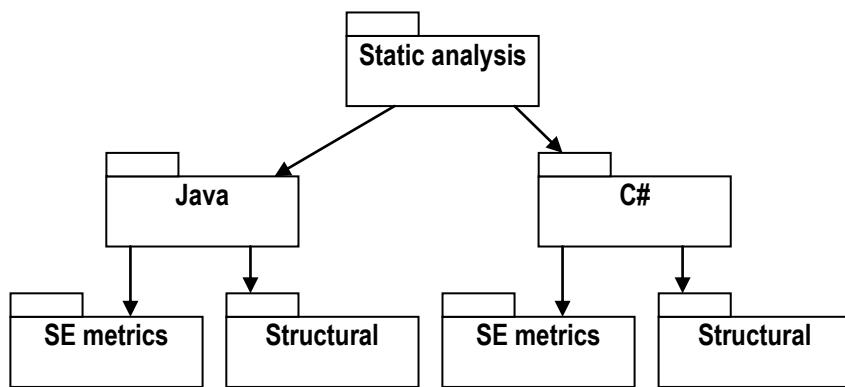
public interface StaticAnalysis {
    public String getShortDes();
    public String getLongDes();
    public Document analyse(Element gap, Document configDoc,
                           Document solution);
    public Document similarity(Document studSol, Document modelSol,
                               Document configDoc);
}

```

**Figure 31: Static Analysis Interface**

Further, from easily adding new analyses, the static analysis component can be extended to support multiple programming languages. Currently the analysis is capable of analysing students’ Java and C# programs. Analyses for each of these programming languages are grouped together in the same software packages structure as illustrated in Figure 32. Classes in the top level static analysis package

check the programming language of the analysing exercise request and invoke the appropriate analyses for that language. Therefore, in order to support new programming languages, SE metrics and structural similarity analyses need only implement the common StaticAnalysis interface described above and organised in the same software packages structure.



**Figure 32: Static Analysis Packages**

So far this section has explained how an exercise is converted to AST XML representation, how gap AST is extracted from the complete program and its normalised form. The extensibility characteristic of the analysis in supporting multiple programming languages has also been elaborated. As noted earlier, the process of converting program source to AST XML representation and gap extract are used in both SE metrics and structural similarity analysis. The following two sections will discuss how the SE metrics and structural similarity analysis are carried out after the AST representation of gap codes are extracted from the complete program AST and gap type is identified.

#### 4.7.1 Software Engineering Metrics Analysis

The main aim of SE metrics analysis is to judge the quality of student programs based on a set of computed complexity metrics values and a checklist of good programming practice guidelines. The key characteristics of analyses in this group are analysing only students' gap codes, supporting multiple programming languages exercises, and providing configurability and extensibility. The extensibility characteristic of the static analysis component is achieved by implementing a common StaticAnalysis interface which is discussed in the previous section. This section will describe in detail how student gap codes are analysed, the

implementation and required configuration of all analyses provided in this group through a C# exercise which has two block gap types as shown in Figure 33; the static analysis configuration for the exercise is illustrated in Figure 34.

```
using System;

class Power {

    public static void Main() {
        int power;

        // Call the Power method
        power = divideByTwo(256);
        Console.WriteLine("\n256 is 2 to the " + power);
    }

    public static int divideByTwo(int num) {
        int arf;

        int temp;
        if (num == 2) {
            // Base case
            return 1;
        }
        else {
            // recursive step
            arf = num / 2;
            return (1 + divideByTwo(arf));
        }
    }
}
```

**Figure 33: A C# Exercise**

```
<analysis>
    <class name="Power">
        <gap id="1">
            <structuralSimilarity/>
            <softwareEng>
                <CheckMagicNumbers using="customized">
                    <allow>
                        <number>256</number>
                    </allow>
                </CheckMagicNumbers>
            </softwareEng>
        </gap>
        <gap id="2">
            <softwareEng>
                <ProgramStatistics/>
                <CyclomaticComplexity/>
                <CheckRedundantLogicExpression/>
                <CheckUnusedVariables/>
            </softwareEng>
        </gap>
    </class>
</analysis>
```

**Figure 34: An Example of Static Analysis Configuration**

As shown in Figure 34, *CheckMagicNumbers* is configured to be carried out in the first gap of the exercise while *ProgramStatistics*, *CyclomaticComplexity*, *CheckRedundantLogicExpression*, *CheckUnusedVariables* are configured to be carried out in the second gap.

Since the *CheckMagicNumbers* analysis is configured for the first gap, the analysis will traverse through the AST representation of the first gap to detect any hard coded numbers or string literals in the gap code. As noted earlier, the default permitted values for this analysis are 0, 1, 0.0, 1.0, `null` and “”. However, in the example shown in Figure 34, 256 is the additional hard coded value allowed in the gap.

The program statistics analysis is set for the second gap in the example illustrated in Figure 34 which is a block gap type. Hence, the analysis counts the total number of variables, statements and expressions in the gap; the results are shown in Figure 24. For method and class gap types, the statistics are provided on a method basis. The statistics information consists of a number of conditional, loop, method call, return statements and expressions.

The cyclomatic complexity analysis is designed to check how easy it is to understand a student gap solution. This analysis counts the number of logic decisions in a gap. The following factors in a program are counted: loop operations such as `while`, `for` and `do while` statements, conditional operations such as `if else` and `switch` statements, and compound Boolean expressions such as `a = true && b = false` with `a` and `b` are two Boolean expressions. Depending on gap types, complexity value is calculated per method for method and class gap types, otherwise it is only calculated for the block of statements in gap.

With the `if else` and `switch` statement, the `else` branch and the `default` case do not count as one decision branch in a program. Therefore, the complexity value for the second gap in the above example is 2. It is important to mention that there is no consistency among existing metrics analysis programs on how the `default` case of the `switch` statement is calculated. It is not considered to contribute to the complexity of the `switch` statement in JavaNCSS (Clemens, 1997) and JMetric (Cain, 2000) but it is counted as one logic decision in the Krakatau (Power Software, 2000) program.

The redundant check logic expression analysis is designed to ensure that students do not perform unnecessary Boolean expression checks. Currently, only the first form of the redundant logic expression check, shown in Table 17, can be detected by the function. In order to be able to check the second form, code

optimisation and information flow is required. There is no redundant check logic expression in the above example.

The check unused variables analysis is designed to check if any variables are declared but not referenced in a program. This is carried out by computing the number of times a variable is used in the program. A list of variables which have not been used is reported to the student. Like the check unused variables analysis, the check unused parameters analysis counts the number of times that a parameter is referenced in a method and reports all unreferenced parameters to the students. The *CheckUnusedVariables* is specified to be conducted for the second gap in the above example. As can be seen in Figure 33, the variable `temp` is declared but not referenced in the second gap. Hence, the analysis will report the variable name and the line number in which the variable is declared in the code as the result.

Additional analyses which are not configured in the example are the check access modifier and the check switch statement analyses. The access modifier analysis checks the access modifiers for both global variables in an object-oriented class and method declaration. This analysis allows great flexibility. Teaching staff can select all variables or methods to be public, private or protected. They can configure a certain number of public, private or protected methods or variables in an OO program. They can also enforce the number of methods, variables or their names in a class.

The check switch statement analysis ensures that every case block in a switch statement has a break statement and every switch statement has a default case. This function is only available for Java programming exercises because the JDK compiler does not enforce this, while the C# compiler does.

In summary, this section has discussed in detail the implementation of eight functions currently provided by the SE metrics analysis. The operations of these analyses are fully configurable by instructors. As mentioned earlier, the key characteristic of the analysis is configurability and extensibility, hence, additional functions can be added to check other poor practices of novice programmers. In the next section, the implementation of structural similarity analysis will be discussed.

## 4.7.2 Structural Similarity Analysis

Structural similarity analysis aims to verify the result of SE metrics analysis by providing insights into high complexity areas in a program. It performs a relative comparison between the structures of a student's solution with the expected solution.

As with other analyses in the SE metrics analysis group, the structural similarity analysis is only loaded if it is required for an exercise. As noted earlier, unlike analyses in the SE metrics analysis group, this analysis is carried out at class level to reduce the number of implementation variations. If a class in an exercise is configured to carry out structural similarity analysis, the configuration is set in the first gap of that class, as shown in Figure 34. For one class exercises, students need to complete all gaps in an exercise in order to carry out the analysis, while with multiple classes' exercises, the analysis is only conducted for those classes which have the gaps completed.

After the students' gaps, AST are pre-processed and extracted from the completed program. These gap AST are transformed into abstract pseudo code form and compared with the model solution abstract pseudo code forms stored in a database in the server to select a set of solutions which have matching structures with classes in the exercise. Then, comprehensive analysis is carried out to compare other statements and expressions in gaps. If a student solution has a matching structure, a congratulatory message is returned. Otherwise, the student solution is marked as structurally different. The feedback highlights all the differences between the student and model solutions, together with the instructor's suggestions of how the problem should be solved.

Program abstraction is achieved by adding generic nodes to the AST. For example, a generic loop node is used to represent any form of loop (`for`, `while`, `do while` and `foreach`). Similarly, there are generic expressions and selection nodes. Other generic nodes represent statement counts. Figure 35 illustrates a gap for a block of statements and its normalised form. This normalisation process also helps to limit the variation of possible solutions for a problem.

```

guess = reader.readInt("Guess a number " + "between 1 and 100 ");
while(guess != secret){
    if(guess < secret){
        writer.println("Your guess is low");
    }else {
        writer.println("Your guess is high");
    }
    guess = reader.readInt("Guess a " + "number between 1 and 100 " );
}

```

```

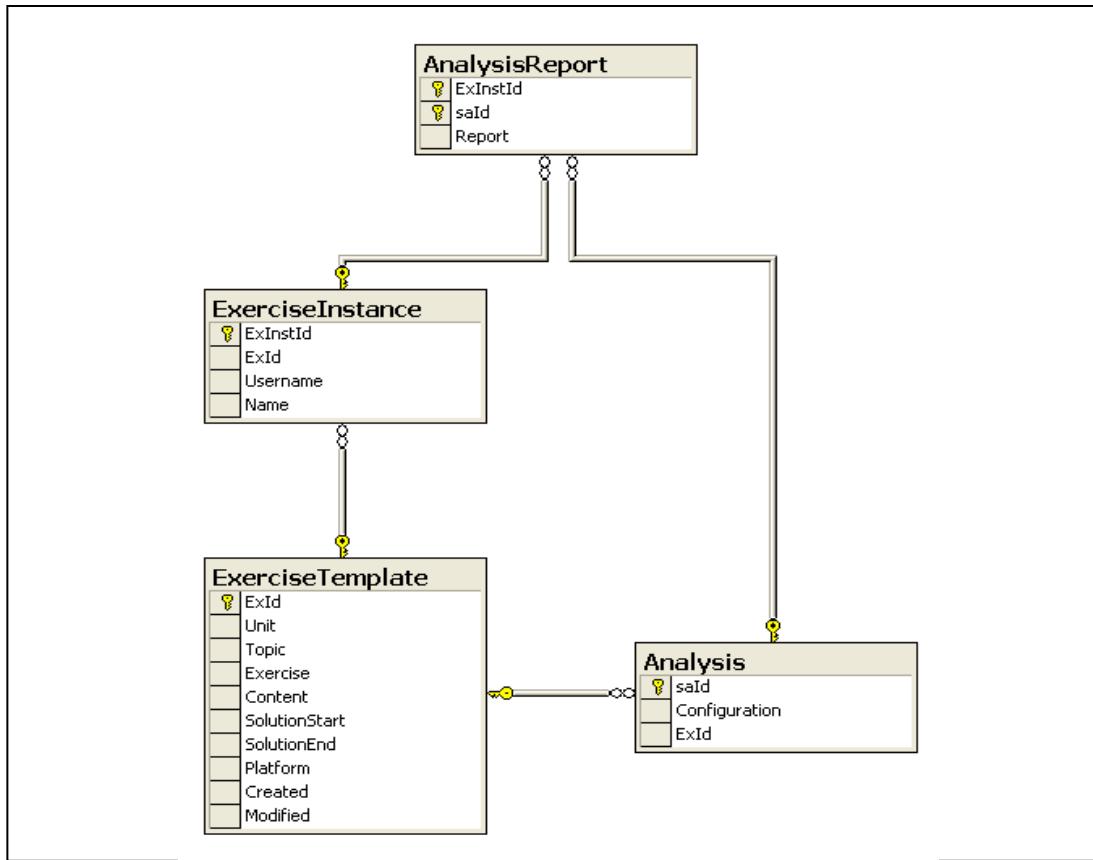
<gap>
<statements>
    <assignment>1</assignment>
    <methodCall>1</methodCall>
    <loop>
        <condition>
            <trueBranch>
                <methodCall>1</methodCall>
            </trueBranch>
            <falseBranch>
                <methodCall>1</methodCall>
            </falseBranch>
        </condition>
        <assignment>1</assignment>
        <methodCall>1</methodCall>
    </loop>
</statements>
</gap>

```

**Figure 35: Gap Code and Its Abstract Pseudo Code Form**

#### 4.7.3 Database Structure

The static analysis information is stored in an analysis database which contains two tables: Analysis and AnalysisReport. Each exercise template in the exercise database is associated with one analysis configuration stored in the analysis table. The AnalysisReport table stores the analysis result of a student attempt. These tables are designed so that when an exercise template is modified all analysis configurations and analysis results associated with the exercise template become invalid. Figure 36 demonstrates the relationship between the exercise and static analysis databases.



**Figure 36: Exercises and Static Analysis Database Relationships**

Name	Data Type	Description
saId	int(4)	Unique ID of the static analysis configuration
Configuration	text	Static analysis XML string configuration
ExId	int(4)	Unique ID of the exercise that using the analysis configuration

#### Analysis Table Fields Descriptions

Name	Data Type	Description
saId	int(4)	Unique ID of the static analysis configuration
ExInstId	int(4)	Unique ID of student gap instance that the analysis carried out
Report	text	XML string analysis result

#### AnalysisReport Table Fields Descriptions

## 4.8 Summary

In summary, the chapter has presented a static analysis framework designed to assess the quality of student Java and C# “fill in the gap” programs. The static analysis framework promotes scaffolding instruction and constructivist learning principles in the ELP. This is because the framework feedback is based on individual student’s gap attempts, and scaffolding instruction. The feedback allows students to reflect on their choice of algorithm and refine their knowledge. The analysis feedback is discussed in detail in Chapter 6.

The five key characteristics of the static analysis are that it: 1) supports “fill in the gap” type exercises; 2) is able to analyse multiple programming languages exercises; 3) is configurable; 4) is extensible; and 5) performs both quantitative and qualitative analyses.

The static analysis framework consists of two complementary groups of analyses: SE metrics and structural similarity. SE metrics analyses are quantitative analyses. Analysis in this group judges the quality of student programs based on computed cyclomatic complexity and line of code metrics values, together with a checklist of poor programming practices in a program. The structural similarity analysis verifies the result of the SE metrics analyses by comparing the structure of the student solution with the structure of the model solution to identify high complexity and poor code areas.

Only students’ well-formed gaps codes are analysed by the static analysis component. There are seven well-formed gap types: expressions, properties (only for C# exercises), import, declaration(s), block, method(s) and class. Static analysis can be conducted for all well-formed gap types apart from expression gaps. SE metrics analyses are performed for each gap in an exercise, whereas the structural similarity analysis is carried out for all gaps in a class of an exercise. The analysis can be extended to support additional programming languages such as C and C++. With SE metrics analyses, each analysis is provided as a pluggable component; hence additional analyses can be added in easily. The analyses are fully configurable by teaching staff, thus they can be adjusted to suit exercise objectives. The analysis was evaluated among a few hundred students over two semesters, and the evaluation results are discussed in Chapter 7

# **Chapter 5 - Dynamic Analysis**

Chapter 3 presented the ELP system and discussed how it supports novice programmers in learning to program. Chapter 4 introduced the static analysis component of the program analysis framework and explained how it helps to increase the quality of beginning student programs. In this chapter, the design and implementation of the dynamic analysis which is used to assess the correctness of students' programs is elaborated upon.

## **5.1 Introduction**

Edwards (2004) has described four phenomena associated with novice programmers when they undertake programming exercises:

1. Once the compiler accepts their code without complaining, they think they have removed all the errors.
2. Once their code produces the output that they expect on one or two test values, they assume that it will work well all the time.
3. Once students think the code is “correct” but it produces the wrong answer, they will use a “trial by error” approach, switching around a few things to see if they can make the problem go away.
4. Once their code gives the correct answer for the instructor’s sample data, they are finished.

Dynamic analysis or automated testing is the process of executing a student program through a set of test data. Dynamic analysis gives students immediate feedback about the correctness of their programs, thus allowing them to learn and improve their programs throughout the process of achieving the solution for a programming problem. This process will encourage students to move from a “trial by error” learning approach to “reflection in action” (Schon, 1983) which enforces learning. In addition, automated testing also makes students aware of the software testing process (Jones, 2001b) which has had only little focus in computer science curricular (Edwards, 2004; Jones, 2001b).

GRADER (Forsythe, 1964) was the first automated grading machine used in computer science education as early as 1964 at Stanford University. Since then,

many systems have been developed. Some of the systems were described in Chapter 2 and include CourseMaster (CourseMaster, 2000), InSTEP (Odekirk-Hash, 2001), WebToTeach (Arnow and Barshay, 1999), TRY (Reek, 1989), BOSS (Luck and Joy, 1999), Datlab (MacNish, 2000b), ASSYST (Jackson and Usher, 1997), Web-CAT (Virginia Tech, 2003), GAME (Blumenstein et al., 2004a), CMERun (Etheredge, 2004) and HomeWork Generation and Grading Project (Morris, 2003).

However, there are three major drawbacks to existing systems. Firstly, the process of comparing student program output with model solution output in most existing systems cannot differentiate significant information which contributes to the program correctness and information that does not (Blumenstein et al., 2004a; Jackson, 1991). This leads to an inaccurate judgment of the correctness of students' programs. Secondly, the generated feedback is not detailed enough to help students learn or to solve their problems. Thirdly, a large amount of time and effort is often required in order to set up testing. Other drawbacks include: only test programs written in a particular programming language and limited types of programming exercises are supported (Blumenstein et al., 2004a).

In this chapter, a dynamic analysis which can be used to verify the correctness of student "fill-in the gap" programs is presented. The analysis brings benefits for both students and teaching staff. From the student point of view, the analysis can provide immediate individual feedback on the correctness of their programs which helps to reduce frustration and misconception. From the teaching staff point of view, the analysis can be used to assist them in the marking task by obtaining the analysis results and in providing additional feedback for students. In addition, the analysis results also allow them to monitor students' progress.

The dynamic analysis together with the static analysis, discussed in Chapter 4, promotes scaffolded instruction and constructivist learning in the ELP. The two analyses allow students to do their practice on the ELP anywhere at anytime and receive timely and quality feedback on both the quality and correctness of their programs. Students can learn and reflect on their choice of solution for a programming problem while teaching staff can monitor students' progress. Furthermore, the integration of the two analyses distinguishes the ELP system with previous web-based learning environments which provide either static or dynamic analysis, but not both. The dynamic analysis feedback is the incorporation of automated tests results and customised feedback from teaching staff. It identifies

gaps that might have errors, together with hints on how to fix them. More information on analysis feedback is discussed in the next chapter.

Unlike most existing automated testing systems, the dynamic analysis conducts both black box and white box testing which are two of the most common software testing strategies. Black box testing is used to check if a program performs all the functions correctly as specified in the requirements; it does not require knowledge of program implementation. In contrast, white box testing requires knowledge of program implementation and therefore, it is difficult to automate the testing process. In this analysis, white box testing is carried out to refine the results of black box testing; it detects any possible errors which have not been revealed by black box testing. In addition, the result of white box testing allows the analysis to provide more detailed qualitative feedback for students.

The analysis has four key characteristics which distinguish it from other previous work:

1. Supports testing “fill in the gap” programming exercises: This type of programming exercise reduces the complexity of automated testing. It allows both black box and white box testing to be carried out on an exercise and therefore more detailed feedback is provided for students. Gaps that might have errors and possible causes are presented to students as part of the testing feedback.
2. Is web-based: This user interface makes the analysis easy to use and more accessible to students and teaching staff. The analysis allows students to do their exercises on-line and receive feedback at anytime, anywhere.
3. Is configurable: Teaching staff can configure different levels of test, giving rising to a granular score for a program. More importantly, they can specify customised feedback associated with each test assisting students to fix their problems when a test fails.
4. Is extensible: The analysis supports multiple programming language exercises. Currently, it is able to test Java and C# “fill in the gap” programming exercises and can be extended to test other programming language exercises. In addition, functions used to compare test outputs are loaded at runtime through dynamic class loading and therefore additional functions can be added easily to address the exercise objectives.

The chapter is comprised of five sub-sections. Section 5.2 gives an overview about software testing concepts, while Section 5.3 examines different types of novice programming exercises as the foundation for the design of the dynamic analysis. Sections 5.4, 5.5 and 5.6 respectively detail the design, required configuration and the implementation of the analysis.

## 5.2 Overview of Software Testing

Testing is the process of executing a program with the intention of finding errors (Myers, 1979). Software testing can be carried out at three different levels: unit, integration, and system. At the unit testing level, individual software component is tested to ensure that the software operates correctly. Test stubs and drivers are used to exercise the component based on test cases. After unit testing is carried out, several software components are combined, and integration testing is conducted. System testing incorporates functional and non-functional requirements of the software. At all levels, black box and white box are the two main strategies which are used. The remainder of this section is divided into two sub-sections. Sub-section 5.2.1 discusses black box testing, its advantages and disadvantages, while white box testing is detailed in Sub-section 5.2.2.

### 5.2.1 Black Box Testing

Black box or functional testing is a specification-based approach. The main purpose of black box testing is to ensure that all required functions of a software product are implemented correctly. It is carried out without knowledge of program implementation. The testing process consists of three steps: identifying required functions, developing test data to check if the functions are implemented and specifying test oracles to determine the correctness of test outputs.

There are five test case design techniques for black box testing which are listed as follows:

- *Equivalence partitioning*: the inputs domain of a program is partitioned into a finite number of equivalent classes such that a representative value of each class is equivalent to any other value in the same class.
- *Boundary value*: test inputs are situated directly on, above, and beneath the edges of input equivalence partitioning classes mentioned above.

- *Case-effect graphing*: test input and output conditions are described by Boolean operators.
- *Error guessing*: possible errors and test cases for those errors are listed and test data based on those test cases are constructed.
- *Random testing*: test inputs are randomly selected; this is the weakest test case design methodology.

An important advantage of black box testing over white box testing is the highly reused nature of test cases once they are designed and therefore it is less expensive to carry out than white box testing. For example, test cases can be reused for customer acceptance testing when the program implementation is changed. In addition, black box test cases can be developed in parallel with the implementation since knowledge of program implementation is not required (Heiser, 1997). However, black box testing also has two major disadvantages. Firstly, the design of test cases can be difficult and time consuming when the software requirement is not clear and concise. Secondly, this type of testing may leave many segments of code untested and it cannot identify complex code areas in a program (Koundinya, 2003). More information on black box testing can be found in Beizer (1995), DeMillo et al., (1987) and Myers (1979).

### **5.2.2 White Box Testing**

White box testing is a program-based testing process. The main purpose of white box testing is to discover hidden errors in code. It uses the internal structure of the code to generate test cases and is concerned with the degree to which test cases exercise or cover the logic of the program.

There are five test case design techniques for white box testing detailed as follows:

- A *statement coverage* test case ensures each statement in a program is executed at least once.
- A *decision coverage* test case ensures that each branch decision in a program is traversed at least once.
- A *condition coverage* test case is designed so that each condition in a decision takes on all possible outcomes at least once.

- A *decision/condition coverage* test case guarantees that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once.
- A *multiple condition coverage* test case ensures that all possible combinations of condition outcomes in each decision, and all points of entry, are invoked at least once.

There are two advantages of white box testing including that it reveals errors in hidden code and ensures the quality of a software product. However, white box testing is expensive to carry out since test plans and test inputs cannot be reused once the program implementation is changed. Furthermore, it requires the code to be implemented first before testing can take place and testers might miss cases which are omitted in code. Further information on white box testing can be found in Bryson (2003), Cole (2000), Culwin (1995) and King et al. (1995).

To summarise, this section gives an overview of software testing and discussed in detail two software testing strategies: black box and white box. Table 20 outlines differences between the two strategies in terms of technique, aims and their advantages and disadvantages. The three key differences between automated testing in a computer science educational environment and in an industrial environment are that beginning student programs are often small in size, program efficiency is not obvious and the test oracle exists in the form of a model solution. Most automated assessment and tutoring systems only adopt a black box testing strategy rather than both strategies to verify the correctness of student programs because understanding the implementation of student programs is complex. However, by conducting only black box testing, these systems cannot point out to students what is wrong in a program and which areas in code can be improved or optimised.

	<b>Black box testing</b>	<b>White box testing</b>
<b>Technique</b>	Does not require the knowledge of program implementation	Requires knowledge of program implementation
<b>Aims</b>	<ul style="list-style-type: none"> <li>• Reveals missing functions</li> <li>• Focuses on user functionality</li> <li>• Discovers faults of omission</li> </ul>	<ul style="list-style-type: none"> <li>• Reveals hidden errors and optimisation</li> <li>• Focuses on quality of code</li> <li>• Discovers faults of commission</li> </ul>
<b>Advantages</b>	<ul style="list-style-type: none"> <li>• Inexpensive due to high level reuse of test cases</li> </ul>	<ul style="list-style-type: none"> <li>• Reveals errors in hidden code</li> <li>• Ensures software quality</li> </ul>
<b>Disadvantages</b>	<ul style="list-style-type: none"> <li>• Might leave many code segments untested</li> <li>• Cannot identify complex code areas in a program</li> <li>• Test cases are hard to design when specification is ambiguous</li> </ul>	<ul style="list-style-type: none"> <li>• Expensive</li> </ul>

**Table 20: Black Box and White Box Testing Comparison**

In the next section, introductory programming exercises categories are discussed. Since white box testing requires knowledge of program implementation, an understanding of the types of programming exercises used in introductory programming courses is crucial. This knowledge forms a foundation for developing a generic technique to carry out white box testing for those exercises.

### 5.3 Novice Programmer Exercise Categories

Understanding different types of introductory programming exercises is vital in designing the dynamic analysis. This knowledge helps to design the analysis which is able to test a wide range of programming exercise categories; allowing the analysis to provide more meaningful feedback. This knowledge was obtained through investigation of exercises in introductory programming courses at QUT and the CS1 curriculum guidelines (The Joint Task Force on Computing Curricula, 2004).

Programming exercises in five introductory subjects taught at QUT were collected over the last three years. These subjects were: CTB210, ITB111, ITB112, IB610 and ITN600. CTB210 and ITB111 were two introductory Java programming subjects for undergraduate students in the first semester of their undergraduate degree. ITB112 was a second Java programming subject taught in the second semester and ITB610 was a second year third semester C# programming subject. ITN600 was the first introduction to programming subject for coursework Master students used C# as the teaching programming language.

Through investigation, there are four main novice programmer exercises categories. They are console, OO, graphical user interface (GUI) and file IO. These

categories and their subtypes are shown in Table 21. In accordance with the CS1 curriculum guidelines (The Joint Task Force on Computing Curricula, 2004), a majority of the exercises are console IO programs and OO concepts. Therefore, these types of programming exercises are investigated first. At the present time, the program analysis framework supports console IO and OO programming exercises only.

As discussed in the previous chapter, there are seven gap types, namely, expression, block, method(s), class, declaration(s), import(s) and properties. Additionally, it was discovered that gap code in those exercises either changes the value of variables in a program or modifies a program output or both.

Exercise Categories			
Console	Object Oriented	GUI	File IO
Only modify the format of the output	Concerned with access modifiers	A user interface is provided	Read from or write to provided files
Only perform arithmetic or string literal operations and display the result	Concerned with user defined types	A user interface is not provided	Read from or write to students files

**Table 21: Introductory Programming Exercises Categories**

The following sections describe the design and implementation of how the analysis carries out tests for console IO and OO programming exercises. Techniques to test GUI and file IO programming exercises are discussed in Chapter 7.

## 5.4 The Design

This section is divided into three sub-sections. Sub-section 5.4.1 gives an overview of the dynamic analysis; Sub-sections 5.4.2 and 5.4.3 respectively discuss in detail the design of black box and white box testing.

### 5.4.1 Overview

The dynamic analysis carries out both black box and white box testing. The main purposes of black box testing are to check if a program performs correctly all the functions as specified in the requirements and to detect any runtime error, while white box testing is carried out to refine black box testing results. It identifies gaps that have logic errors which lead to incorrect outputs in black box testing and reveals

any hidden errors which are not discovered by black box testing. The two testing strategies are supporting each other.

Testing “fill in the gap” programming exercises is the key novelty of the analysis. Like the static analysis, only well-formed gaps can be tested by the dynamic analysis and, depending on the gap types, different testing techniques are used. There are seven well-formed gap categories namely, expression(s), block, method(s), class, declaration(s), import and properties (only available for C#) exercises.

In order to conduct black box and white box testing for an ELP exercise, a test plan and test inputs are required. Optionally, teaching staff can add customised feedback for tests which will be incorporated with the testing results to provide feedback for students. Further information on testing configuration requirements is discussed in Section 5.5.

The dynamic analysis adopts client-server architecture. When an exercise is successfully compiled, a student can test it by clicking on the “Check Program” button in the ELP editor applet, as shown in Figure 37. The executable test driver is generated on the ELP server which is subsequently downloaded and run on the student’s machine. The testing running progress is presented to the student in a pop-up window (as illustrated in Figure 38) and it can be cancelled at anytime. When each test finishes running, the test output is sent back to the ELP server to compare with the expected output. The comparison process can be carried out on the ELP server or forwarded to another server but comparison results are stored in a database on the ELP server. Figure 39 illustrates the architecture of the dynamic analysis.

Students are notified when all tests finish running; they can view test feedback by switching to the automated feedback panel on the ELP editor applet. The test feedback reports all tests that were carried out, their descriptions, their results, any missing test outputs during the transmission process, and any runtime errors. If the student’s program passes all tests, a congratulation message is displayed; otherwise, for each failed test, an indication of gaps that might have problems and hints on how to fix the problems are presented. Figure 37 gives an example of the test feedback for an exercise which passes all tests. A detailed discussion on how the test results are processed to provide feedback to students is discussed in Chapter 6.

With the chosen client-server architecture, the analysis is secure, flexible and efficient. Running student programs on their own machines or a separate one

prevents the execution of malicious code and reduces the load on the server. Further, testing student programs on a separate machine also increases the flexibility of the analysis. This allows the analysis to be used for both tutoring and assessment purposes. For tutoring purposes, the testing process can be set up to run on the student's machine; students can control the testing process. For assessment purposes, the testing process can be built to be carried out on a dummy machine to avoid interruption of students.

The screenshot shows the ELP Administrator interface. At the top, there is a menu bar with options like My Program, Open..., Save, Compile, Run, Check Program, Download, Options, Logout, and GUI. Below the menu is a toolbar with icons for file operations. The main workspace contains a code editor and a testing results panel.

**Code Editor:**

```

11  public static void Main()
12  {
13      // declare variables
14      double inches, cms;
15      string input;
16      const double CMS_TO_INCHES = 2.54;
17
18      // print a message to the screen
19      Console.WriteLine("\nEnter your height in centimetres: ");
20
21      input = Console.ReadLine();
22
23      cms = Double.Parse(input);
24
25      inches = cms / CMS_TO_INCHES;
26
27      Console.WriteLine("Your converted height is: {0,10:F5} inches", inches);
28

```

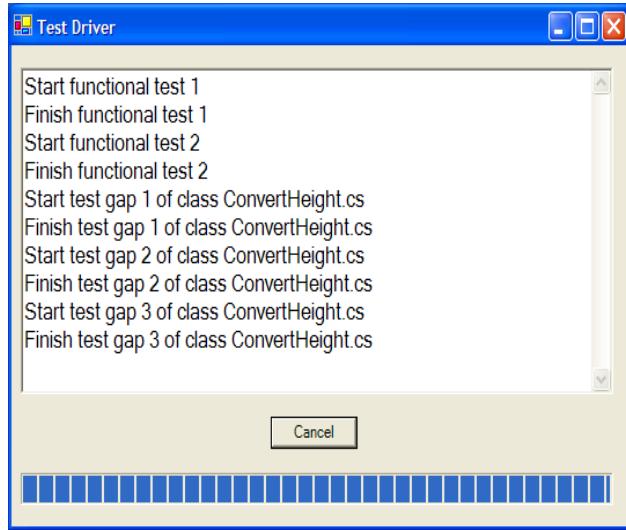
**Testing Results:**

**Program Testing Results**

Congratulations, your program produces correct results for all tests.  
Following are tests that were carried out on your program.

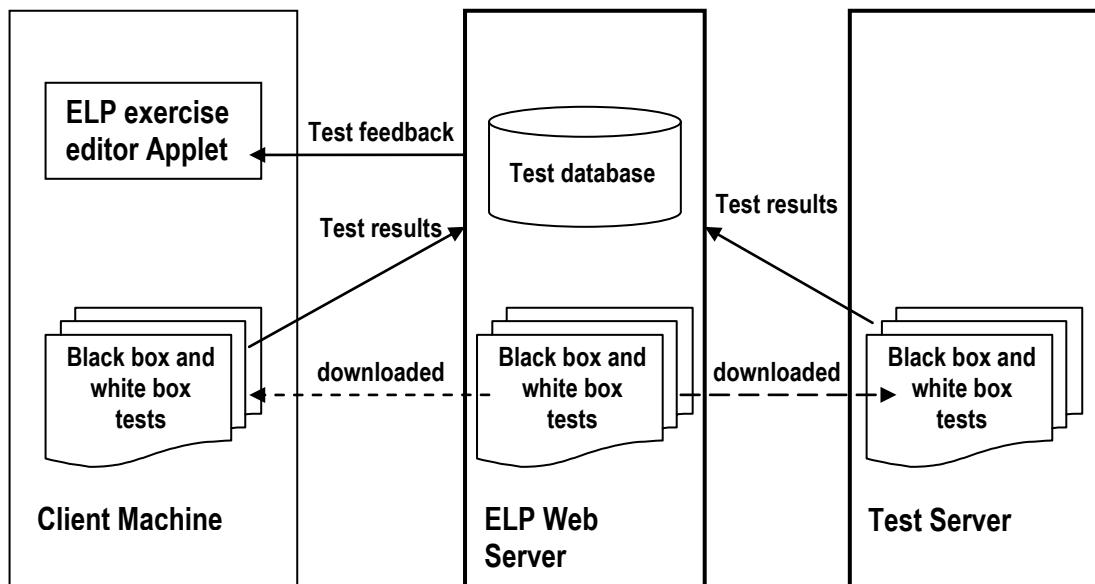
Simple Tests
✓ convert 156cm to inches
✓ convert 172cm to inches

**Figure 37: The ELP and Dynamic Analysis Integration**



**Figure 38: Test Dialogs**

As shown in Table 21, there are four beginning student program categories namely, console IO, OO, file IO and GUI exercises. The analysis currently supports “fill in the gap” console IO and OO programs written in Java and C#. However, the analysis can be extended to test file IO and GUI exercises, as will be discussed in Chapter 8.



**Figure 39: Dynamic Analysis Architecture**

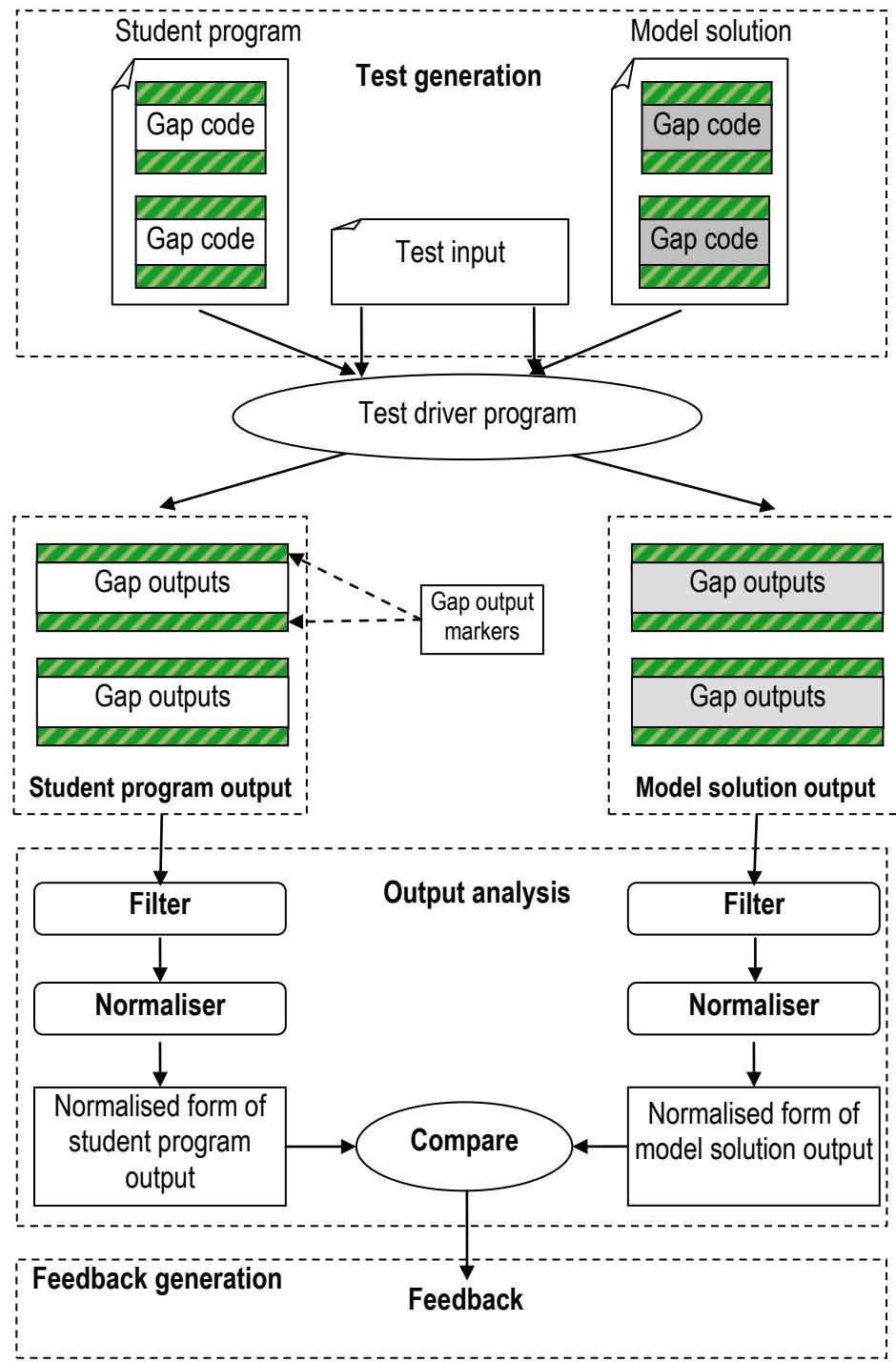
#### 5.4.2 Black Box Testing

The black box testing is designed to check if a student program operates correctly as specified in the exercise requirements. The student program and the exercise model

solution are executed with the same test inputs and the two program outputs are then compared with each other to check if the student's program output is same as the output of the model solution.

The novelty aspect of this black box testing is that it can test gap-filling programs. The black box testing in this analysis is slightly different from the conventional black box testing in that part of a program implementation - the provided code - is known by testers. With "fill in the gap" exercises, only implementation of gaps is unknown but outputs from gaps can be differentiated from the provided code output by inserting a print statement at the beginning and the end of each gap code. Further, as noted in Section 5.2, one of the key differences between automated testing in the industrial environment and in the educational environment is that a programming problem solution is available in the educational environment. The expected execution results of the gaps are known from the exercise model solution. Student gap outputs can be separated from the whole program outputs to compare with the corresponding gap model solution outputs; gaps whose output does not match those of the associated model solution can be identified. This information is used to provide better feedback to students, as they are shown gaps which might have logic errors. This is one of the major benefits of testing "fill in the gap" programs.

Figure 40 gives an overview of black box testing processes including test generation, program output analysis and feedback generation. The test generation process involves constructing the student program and the exercise model solution from the student gaps attempt, exercise template and gap model solution. These programs are constructed so that output from gaps can be separated from the provided code outputs. Output analysis involves comparing the extracted student gap outputs with the gap model solution outputs. The comparison results are then processed to provide feedback to students in the feedback generation step. The design for each of the processes is discussed in detail below.



**Figure 40: Black Box Testing Processes**

In the test generation process, both the student gap attempt and the gaps model solution are inserted into the exercise template to generate the student program and the model solution. Depending on gap types, gap markers may also be inserted into the exercise template together with gap codes so that gap outputs can be separated from the provided code outputs. As noted earlier, there are seven well-formed gap types. Black box testing can be conducted for all gap types apart from

expression and import gaps. Black box testing for the expression gap type has not been developed and it is not necessary to test the import gap type. This is because the analysis only tests compilable programs and a program needs to have all correct imported libraries to be compiled. Among supported gap types, special print statements used as gap output markers can only be inserted before and after block or local variable declaration gap codes to separate gap outputs from the provided code outputs. For other gap types, only gap code is inserted to the program template. Gap outputs cannot be separated from the provided code outputs unless a customised test drive program is used.

For block and local variable declaration gap types, a gap is first processed to detect if it contains program branching control flow statements including `return`, `exit`, `break`, `label`, `continue` and exception handling statements such as `try-catch-finally` and `throw` before gap markers are inserted. If a gap contains those statements, inserting gap markers after the gap code will cause compilation errors. This is because the end gap marker will not be executed; hence, end gap output markers are missing. In this case, gap markers are inserted at the start of the gap and before the change control statements.

Figure 41 shows the student program construction process for an exercise whose gaps do not contain branching or exception handling control flow statements, while Figure 42 gives an example of a program whose gaps contain branching control flow statements. Gap markers are inserted in both examples. As can be seen in Figure 41, markers are inserted at the beginning and the end of the gap block together with gap code into the program template. In contrast, markers are inserted before the `return` statements in gaps 2 and 3 in Figure 42. These gap markers are output in open and closed XML tag format for easy processing purposes.

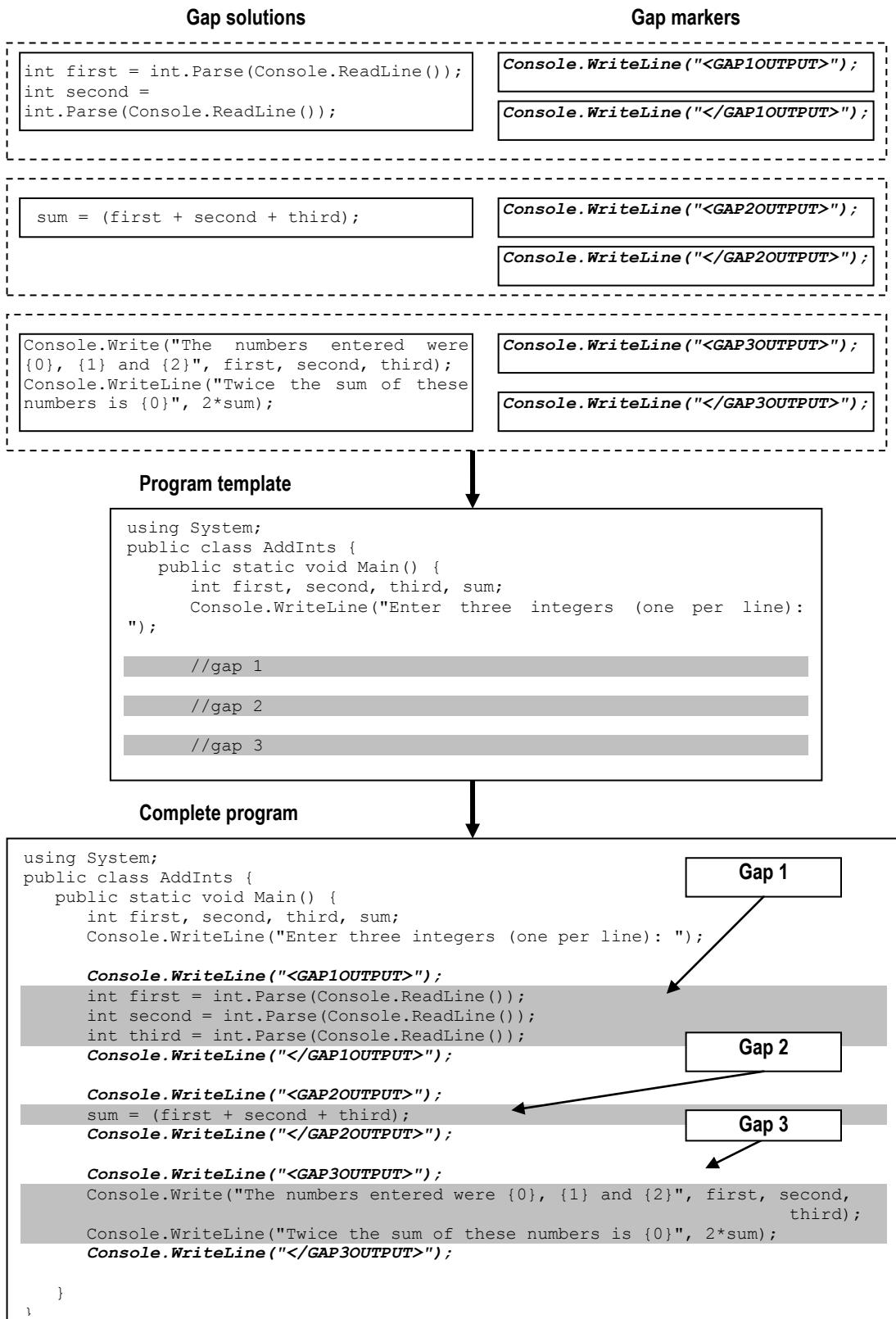


Figure 41: Student Program Construction Process for Black Box Testing

```

        using System;

class Power {

    public static void Main() {
        int power;

        Console.WriteLine("<GAP1OUTPUT>");
        // Call the divideByTwo method for 256
        power = divideByTwo(256);
        Console.WriteLine("</GAP1OUTPUT>");

        Console.WriteLine("\n256 is 2 to the " + power);
    }

    public static int divideByTwo(int num) {
        int arf;

        if (num == 2) {
            Console.WriteLine("<GAP2OUTPUT>");
            Console.WriteLine("</GAP2OUTPUT>");
            return 1;
        }else {
            Console.WriteLine("<GAP3OUTPUT>");
            // recursive step
            arf = num / 2;
            Console.WriteLine("</GAP3OUTPUT>");

            return (1 + divideByTwo(arf));
        }
    }
}

```

Gap 1

Gap 2

Gap 3

**Figure 42: Gap Change Program Control Flow**

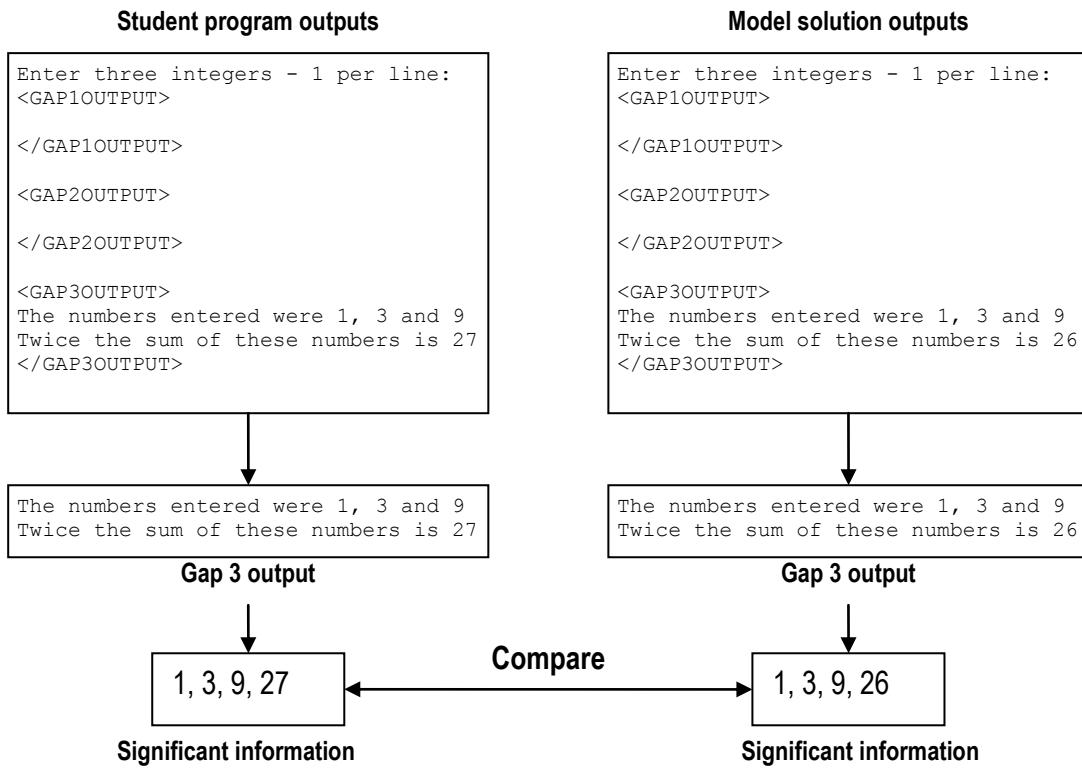
In the test output analysis process, gaps outputs are separated from the provided code outputs using gap markers in both student program output and the model solution output. A set of filters and normalisers first process the gap outputs before the comparison process is carried out. The main purpose of filtering and normalising is to distinguish between important items that are relevant to the correctness of the solution and those that are irrelevant. This allows the analysis to scope important items that are in a different order, giving a great deal of flexibility for it. A student program will not be judged incorrect because it has misspelled words, additional white spaces, or new line characters in the outputs.

Filters and normalisers make use of regular expression. Filters can be used to extract all the important keywords or values that are related to correctness and insert them into a set data structure. The student set and the model solution set may then be matched, taking order into account or not. Normalisers allow comparison of the outputs of a student's solution with those of the model solutions while ignoring insignificant differences such as spaces or tabs. Configurability and extensibility are two among four key characteristics of the dynamic analysis. With black box testing, filters and normalisers are provided as functions which are only loaded at runtime

using a dynamic class loading mechanism. This mechanism allows additional filters and normalisers to be added in the future.

Figure 43 gives an example of the test output analysis process for the exercise shown in Figure 41. In the example, a numbers filter is used to separate significant items and those not important ones from the gap outputs. As can be seen in Figure 43, there is no output for gaps 1 and 2; hence the correctness of the program is decided by the gap 3 output. The third gap output is extracted from the complete program output and all numbers in the output are filtered into a set data object. Each of the numbers in the student solution set is then compared with the corresponding one in the model solution set.

The process of comparing test output is fully configurable by teaching staff. They can set up the format of program outputs, the occurrence of certain keywords or values which are important in deciding the correctness of the program. This allows various correctness levels of a programming problem rather than just ‘right’ or ‘wrong’: for example, your program produces correct results but the output format is not right. This also means that the analysis can provide more detailed feedback for students. The feedback identifies sections in the program which are incorrect to help students more easily locate errors in their program. This is one of the advantages of learning to program with gaps. Making the analysis fully configurable gives teaching staff a great level of control over the feedback and the assessment process. With the example shown in Figure 43, teaching staff can configure the comparison process between the student solution numbers set and the model solution set, either in order or disregarding the order of those numbers.



**Figure 43: Example of Test Output Analysis Process**

As previously mentioned, there are three main stages in black box testing: test generation, test output analysis and feedback generation. So far, the test generation and test output stages have been discussed. A detailed black box testing report based on test results consists of:

- A list of conducted tests and their description in increasing levels of difficulty
- Number of passed and failed tests and indication of gaps that might have logic errors
- Any runtime errors
- Any missing test outputs due to failure in transferring the output from the client to the server or early test process termination by users.

More details on how feedback is generated based on the testing report are discussed in the next chapter. In the next section, the design of white box testing is presented.

### **5.4.3 White Box Testing**

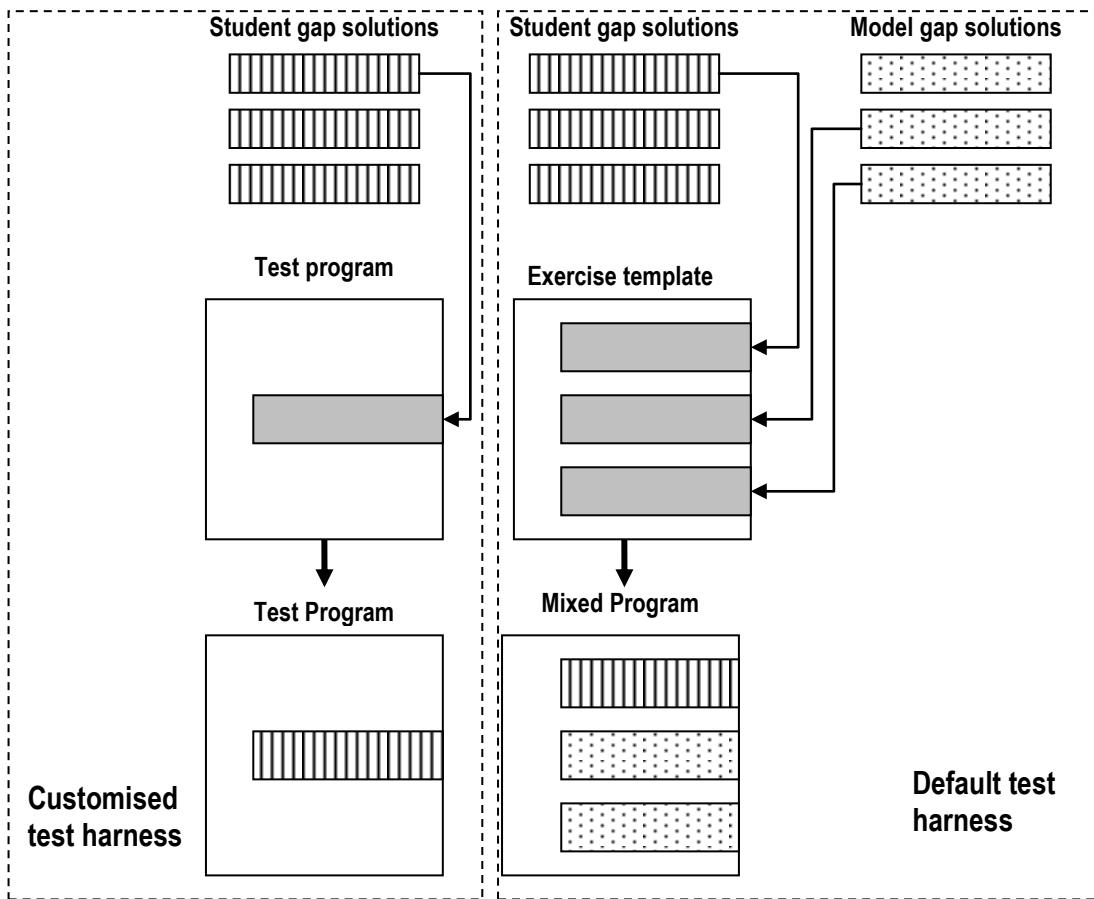
The main purposes of white box testing are to discover any possible logic errors which have not been revealed in black box testing and to refine the results of black box testing. Different from black box testing, white box testing requires the knowledge of program implementation. It also requires more effort for the tester to set up than in the case of black box testing. In this analysis, white box testing tests each student gap solution separately to identify error. It is carried out by inserting student gap solutions, one at a time, into a test harness program. The outputs of each gap are compared with the outputs produced by the corresponding gap solution. A non-matching pair of outputs indicates the gap might be one of the possible causes of the black box testing failure. This helps novice programmers to more easily locate errors in the program in the debugging process.

In black box testing, print markers are used to separate gap outputs from the provided code outputs; this allows the test to identify which student gap outputs do not match with the corresponding gaps' model solution outputs. However, due to the dependency among gaps, it is not always true that gaps whose outputs do not match the corresponding model solution outputs identified in black box testing are those that have logic errors. The incorrect outputs might be caused by other gap codes which are invoked in that gap. Figure 43 gives an example of an exercise which has dependency among gaps. In the exercise, the student gap 3 output does not match the corresponding model solution output because the calculation in the student gap 2, invoked in gap 3, is incorrect. Therefore, the testing of one gap at a time in white box testing refines the results of black box testing and gives a better indication of which gap might contain errors. Black box and white box testing are supporting one another.

In white box testing, gaps are considered to be units of a program which cannot be executed by themselves. They need to be embedded into a program template or a separate test program in order for the testing process to be carried out. Gaps in an exercise can depend on each other: for example, in an exercise which has other gap referencing variables declared in the first gap of the exercise. If there is no dependency among gaps, each student gap code is inserted into a test harness program for testing, otherwise, groups of gaps that are dependent on each other are inserted at the same time.

A test harness is a program skeleton which is used to test a unit that is dependent on it. The white box testing supports two types of test harness programs: a default test harness, which is the exercise template; and a customised test harness, which is provided by teaching staff. If the default test harness program is used, white box testing makes use of a regression testing technique in which the tested student gap solution is considered to be a new change for the program, and the model gap solutions occupy the remaining gaps in the template. With this approach, gaps which have errors can be identified. In contrast, unit testing is adopted when a customised test harness program is adopted because only a student gap solution is tested alone. Figure 44 illustrates how the complete program for an exercise which has three gaps is constructed to carry out white box testing for the first gap in two scenarios. The first case is when the default test harness program is used and the second case is when the customised test harness is adopted.

As can be seen in Figure 44, when the default test harness is used, the student first gap solution is inserted into the program template while the second and third gaps are occupied by the model solution to produce the mixed program. In contrast, only the first gap from the student solution is inserted into the test harness program to build the complete program when the customised test harness program is adopted.



**Figure 44: Program Construction for White Box Testing**

JUnit (Gamma and Beck, 2000) is a well-known unit testing framework which has been adopted in many teaching and learning tools such as BlueJ (Barnes and Kölking, 1999), WebCAT (Virginia Tech, 2003) and JAM\*Tester (Wittry and Poll, 2005) to automate the testing process. JUnit is not adopted in this dynamic analysis because it is designed to test OO programs rather than small procedural introductory programs. Even though the dynamic analysis can be extended to use JUnit for testing OO programs later in the course, the discussion will not be covered in the thesis.

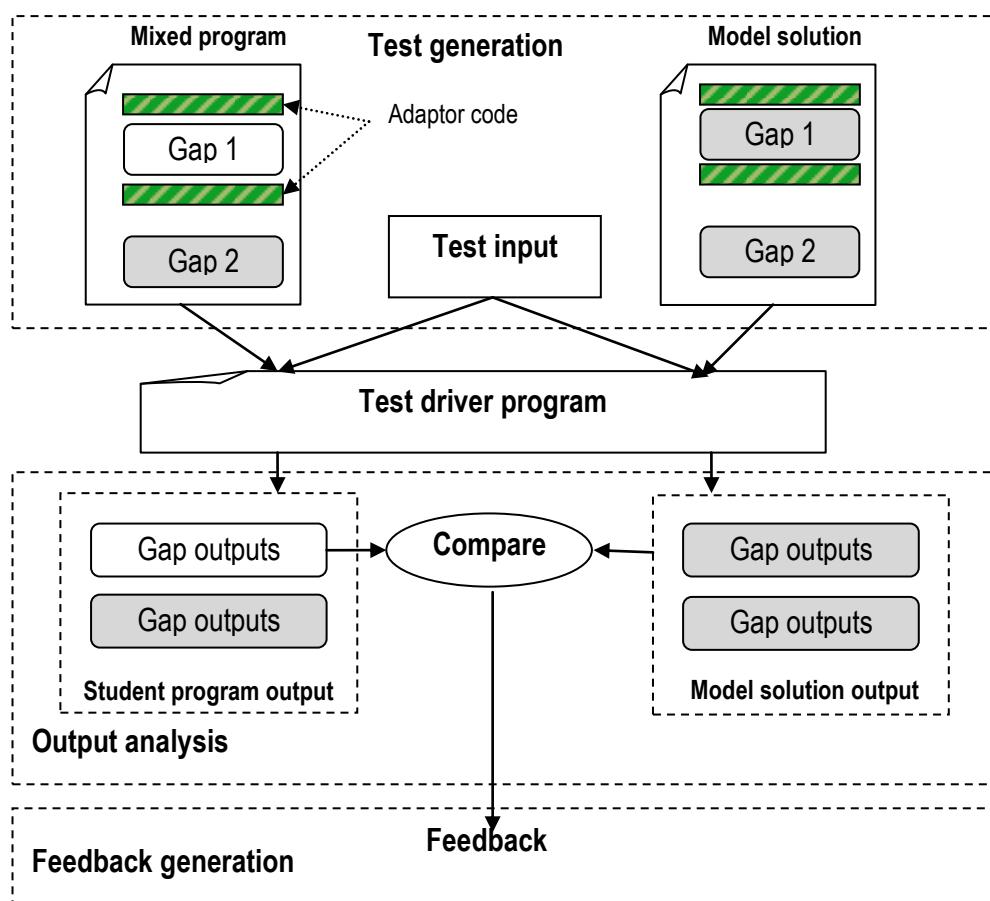
As well as the test harness program, white box testing requires each gap to be wrapped in an adaptor code to provide more information about its internal code. Through investigation, two types of gap behaviours are identified: gaps in which states of variables are changed, and gaps which only modify the program output. To accommodate these gap behaviours, the dynamic analysis provides two types of adapter code for white box testing: variable state dump and print. Each type of adaptor code is a mechanism to carry out white box testing. A variable state dump

can be used to monitor the state of the variables before and after execution of gaps in which the states of variables are changed. This mechanism provides a set of functions to record the type and value of a variable at runtime. The print method is used to separate gap outputs from the provided code outputs as in black box testing; it is used for gaps which only modify program output. Gap outputs are extracted and analysed.

Program assertion is an alternative method to carry out white box testing for gaps which have variable state changes. It can be used to test certain conditions that need to be met either before or after gap execution. However, this method is not adopted in the dynamic analysis because it only returns ‘true’ or ‘false’ without further information on why a condition is not met, which is important for students to fix errors in their programs. With the variable state dumps approach, by comparing type and values of variables in the student gap solutions with those in the model gap solution, the analysis can provide detailed feedback for students: for example, the variable *x* of type integer in the model solution has value of 5 but it has a value of 7 in the student solution. The variable state dumps approach is more useful when students encounter variables of more complex types such as array type or user-defined type. The approach can point out to students which values in their array do not have the right order, or which fields in their defined type are missing or do not have the correct access modifiers.

Figure 45 shows the high-level conceptualisation of the white box testing when the default test harness is used. As shown in the figure, there are three steps in the white box testing process: test generation, output analysis and feedback generation. In the test generation process, the student gap solution and its adaptor code are one at a time inserted into the exercise template (default test harness) while the model gap solutions occupy the remaining gaps to produce a mixed program. For exercises having gaps depending on each other, all dependent gaps and their adaptor code are inserted into the test harness program at once. The mixed program is then compiled and executed by the test driver as in the exercise model solution, using the same set of inputs. This process is slightly different when a customised test harness program is used. Instead of producing the mixed program, two test harness programs are generated: one contains the student gap solution and the other contains the gap model solution. These test harness programs are then run by the test driver.

The output of the student's tested gap is compared with the outputs of the model solution gap to provide feedback to the student. For gaps which only modify the program output, the output analysis step involves output extraction, filtering and normalising as in black box testing. For gaps with variable state changes, the states of those variables are compared with states of the associated variables in the model gap solution in this step. The black box and white box testing results are incorporated with lecturers' customised feedback by the automated feedback engine, as will be discussed in Chapter 6, to provide feedback to students.



**Figure 45: High-Level Conceptualisation of White Box Testing**

As mentioned previously, there are seven different types of well-formed gap: block, class, methods, expression(s), declaration(s), import and properties (for C# exercises). There are four different forms of variable declarations, namely, member scope variables, method scope parameters, local scope variables and exception handling parameters. White box testing can be conducted for six gap types apart from the import gap type. Similar to black box testing, the variable state dump and

the print codes can only be inserted at the beginning and the end of block and local variable declaration gap types. For block gaps which contain branching control flow statements such as `exit`, `break`, `label`, `continue`, `try-catch-finally`, and `throw`, the states of variables are dumped before and after these change control flow statements. If the branching control flow change statement is a `return` statement, the return expression value is also dumped. Figure 46 gives an example of how the variable state dumps can be carried out for the second and third gaps which change the program flow of control. If the print mechanism is used, print statements are inserted at the beginning and before those change control flow statements as in black box testing.

```
using System;

class Power {

    public static void Main() {
        int power;
        // Call the divideByTwo method for 256
        power = divideByTwo(256);

        Console.WriteLine("\n256 is 2 to the " + power);
    }

    // pre : num is a positive integral power of two
    public static int divideByTwo(int num) {
        int arf;

        if (num == 2) {
            // Base case
            return 1; → return dump(1);
        }
        else {
            // recursive step
            arf = num / 2;
            return (1 + divideByTwo(arf));
        }
    }
}
```

**Figure 46: Gap Change Control Flow Examples**

The strength of this approach is that it provides the ability to identify bugs in a particular gap. White box testing can point out which variables in a gap do not have the correct value as expected. However, it does not work well when there is dependency among gaps which occurs in two cases: first, in the case of gaps which require students to define their own variables and other identifiers that are used in

other gaps in a program as shown in Figure 47; and second, where gaps require students to declare their own methods.

In the first case, even though the required variable names that need to be declared are specified in the exercise requirement, there are still chances that students do not follow the instruction, or misspell or mistype the suggested names. Similarly, students might misspell method names or declare a method with a different signature in the second case. This means the generated mixed program between the student solution and model solution cannot be compiled as there are missing variables or methods. These dependency issues are previously discussed in the work of Morris (2003).

The use of reflection or parsing a student program source can detect all variable names and method signatures in a program. However, it is impossible to associate a student variable name with its equivalent variable in the model solution so that the comparison can be carried out.

```
import TerminalIO.*;

public class Change {
    static ScreenWriter writer = new ScreenWriter();
    static KeyboardReader reader = new KeyboardReader();

    public static void main (String [] args ){

        // Declare an integer named 'tendered' and 2 doubles named
        // 'price' and 'change'
        int note;
        double price, change;

        // Complete the following line to get the price
        price = reader.readDouble("Please enter the price: $");

        // Complete the following line to get the amount tendered
        note = reader.readInt("What note was provided?: $");

        // Calculate change
        change = note - price

        writer.println("The amount of change required is $" + change);
    }
}
```

**Figure 47: Identifier Gap and Its Dependents**

Similar to black box testing, white box testing is fully configurable and extensible by teaching staff. They can configure the type of test harness program that the white box testing uses, the gap testing order and the groups of gaps that are tested at the same time. They can customise test feedback and the comparison process in the test output analysis step. Functions used to compare variable states make use of

dynamic class loading at runtime and therefore additional functions can be easily added.

To summarise, this section has described the design of both black box and white box testing. The two testing strategies are only performed for well-formed gaps. The results of black box and white box testing subordinate each other. In black box testing, all student gap solutions are tested at once and print markers are used to separate gap outputs from the provided code outputs. Gap outputs are extracted, filtered and normalised before comparison with the corresponding expected outputs. In white box testing, gaps in an exercise will be tested one at a time to identify hidden errors or which gap might cause black box testing to fail. Testing student gap solutions one at a time can be done by either using the exercise template, default test harness, or a customised one provided by lecturers. To obtain more information about students' gap implementation, variable state dumps or print code can be appended at the start and end of gap codes when it is inserted into the test harness program. A variable state dump is used for gaps which have state of variables change at runtime while the print marker codes is applied for gaps which only modify program output. The next section will discuss the required configurations from teaching staff in order to carry out testing for an exercise.

## 5.5 Author Configuration

Research has shown that the two major reasons for the irregular usage of most existing automated checking systems in the classroom are the lack of flexibility (Preston and Shackelford, 1999) and the time consuming nature of setting up exercises. Taking that into account, the dynamic analysis provides a greater level of flexibility for teaching staff by making it fully configurable. This section discusses basic required and additional configurations in order to carry out black box and white box testing.

In order to enable black box testing, lecturers need to provide a good test plan which consists of test classes, test cases and test inputs. A good test case is one which has a high probability of making a faulty program fail. The main benefit of providing a good test plan is that these test cases can be reused in the white box testing. In addition to the test plans and test inputs, instructors can configure filter, normaliser, and matching rules to be used in the comparison process for each test. This will increase the level of flexibility when comparing student solution outputs

with the expected model solutions. If no matching rules are configured, an exact match is applied. Customised feedback for each test can be set by staff and is incorporated with the test output analysis results in the returned feedback for students.

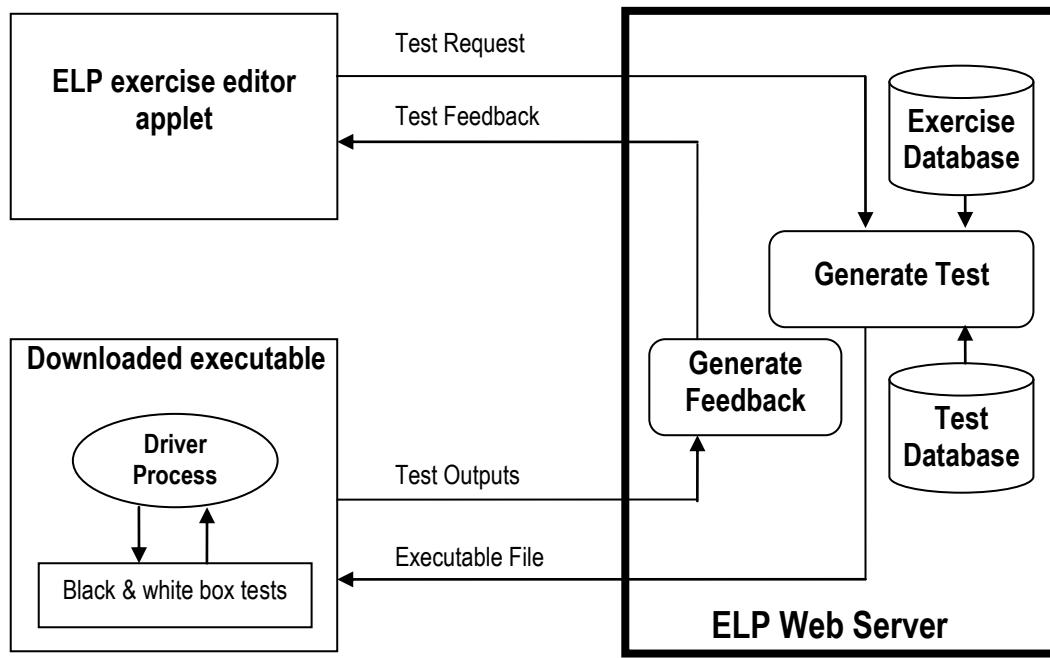
To enable white box testing, lecturers need to provide a test plan and test inputs. For exercises in which both black box and white box testing are carried out, a majority of test inputs are reused from black box testing. However, additional test inputs can be applied. As discussed in the previous section, there are two mechanisms to carry out white box testing: state dump and print. If the state dump mechanism is used, instructors have to provide the name of the variable whose state will be recorded at runtime. When the default test harness is not sufficient, lecturers have to provide customised test harness programs in order to carry out white box testing. Optionally, they can specify matching rules for the comparison process applied to each white box test, and customised feedback.

Finally, it is important to note that in order to provide good feedback for students in any automated testing system, tests and exercise specification need to be developed concurrently (Jones, 2001a; Morris, 2003). Jones (2001a) has developed a SPRAE model to guide the development of exercise requirements which make the automated testing process most effective and provide good feedback to students.

## **5.6 The Implementation**

In Sub-sections 5.4.2 and 5.4.3, the design of black box and white box testing were discussed. This section examines technologies and techniques used to implement those designs.

As noted in Sub-section 5.4.1, the dynamic analysis makes use of client-server architecture. When a student clicks on the “Check Program” button in the ELP editor applet, a test request is sent to the server to create the executable test driver which is subsequently downloaded and run on the student’s own machine. When each test is finished, the test output is sent back to the server for analysis. The test analysis result is stored on a database in the server. When all tests are finished, the student can click on the “Automated Feedback” tab in the editor applet to request test feedback. The test feedback which incorporates analysis results for all conducted tests is presented to students. Figure 48 illustrates the architecture of the dynamic analysis.



**Figure 48: The Dynamic Analysis Architecture**

There are three key novel ideas in the dynamic analysis implementation. First, the analysis makes use of client-server architecture. Second, XML is used to mark up test configurations, test analysis results, HTTP requests and responses between clients and the server and to serialise object states in white box testing. Third, dynamic loading is used to load up test inputs and test harness programs, filters and normalisers. Similar to static analysis, all test configurations and test analysis results are stored in XML format in the test database on the ELP server. This means the information is more readable, easier to search and retrieve, and can be presented differently. The HTTP request and response mechanism is used to communicate between clients and the server. These XML marked up requests and responses are sent from the client to the server as strings. By sending these messages as strings, the framework can be extended to test exercises written in different programming languages with minimal modification on the client side while the server side remains the same. Serialising object states into XML allows state comparison to be more accurate and robust. Loading test inputs and other necessary libraries dynamically at runtime reduces the initial test start-up time. Dynamic loading allows additional filters and normalisers to be easily added.

This section is divided into four sub-sections. The implementation of the test driver generation process, black box testing and white box testing are presented in

detail in Sub-sections 5.6.1, 5.6.2 and 5.6.3. In Sub-section 5.6.4, the database structure for the dynamic analysis is detailed. The feedback generation process is briefly discussed in this section; a more detailed discussion appears in Chapter 6.

### **5.6.1 Test Driver Generation Process**

The test driver generator component performs three tasks. First, it creates the test driver program. Second, the student program, exercise model solution and the test harness program are constructed. Finally, it packages all the test related classes into an executable file which is sent back to the student.

In order to generate the test driver program, test configurations and test inputs are queries from the test database. Figure 49 and Figure 50 give an example of the black box and white box test configurations for the C# exercise `Power.cs` shown in Figure 46. These configurations are used together with an XML test driver template to generate a test driver program which is stored in the user designated temporary directory in the ELP server. In the test driver program, each test is set up to run on a separate thread. The input stream of these threads is redirected to get the test input from a file, while the output stream is redirected to a customised stream which subsequently is sent to the server when each test finishes. The error stream is also captured and sent back to server to report on any runtime errors. The thread is set to invoke the main execution point of the student program specified by teaching staff in the test configuration. Inputs for each test are written to a file which is stored in the user's temporary directory.

As can be seen in Figure 49, black box testing is organised into test classes and test cases. Test classes divide the input domain into multiple equivalent partitions, while test cases are samples of the input domain that they belong to. Tests are organised based on their difficulty level; each test class has a description which will be displayed to students as part of the returned feedback, with optional customised feedback from lecturers and one or more test cases. The `id` attribute of test classes and test cases are the tests' execution order.

Each test case consists of a description, a set of inputs representing the input partition that it belongs to, and expected outcome. The main execution entry for the program is specified in the `runnable-class` node in the XML. Different from Java, there are two formats of the `main` method in C# (one with parameters and one without parameters). The attribute `hasParam` is used to indicate if the main method

has parameters or not. This information is checked when generating the test driver program. Gaps in the `Power.cs` exercise are block type gaps; print statements which display delimiters in the program output are inserted at the beginning and end of each gap to separate gap outputs from the provided code outputs. These delimiters are generated from the hash code of the model solution and time when the test is configured. This will ensure a different delimiter is produced not only for each gap in a program but also for gaps which have the same solutions but the constructed times are different. These delimiters are output in an XML-like format as shown in Figure 42. An open tag style delimiter is inserted at the beginning of the gap and the closed tag style is inserted at the end of the gap.

```

<?xml version='1.0' encoding='utf-8'?>
<blackbox-test>
    <runnable-class hasParam="false">Power.cs</runnable-class>
    <delimiters>
        <gap id="1"><! [CDATA[<G12345679>]]></gap>
        <gap id="2"><! [CDATA[<G123C456F79>]]></gap>
        <gap id="3"><! [CDATA[<GDF12JH34EFGH>]]></gap>
    </delimiters>
    <testclass id="1" diffLevel="simple" name="simple input output">
        <description><! [CDATA[normal case]]></description>
        <feedback>
            <! [CDATA[Your program pass all simple test]]>
        </feedback>
        < testcase id="1">
            <description>
                <! [CDATA[calculate power of 5]]>
            </description>
            < input id="2"/>
            < output id="99"/>
        </ testcase>
    </ testclass>
</ blackbox-test>

```

**Figure 49: Black Box Testing Configuration**

As discussed previously, white box testing is carried out for each gap or a group of dependent gaps in an exercise. A gap can undergo multiple tests and each test can contain multiple inputs. There are two white box testing mechanisms: the print and the variable state dump. With the print mechanism, a gap output delimiter will be inserted at the beginning and at the end of each gap. Similarly, with the dump mechanism, the dumping code is inserted according to the configuration from teaching staff.

Figure 50 gives an example of a white box testing configuration for the first gap of the `Power.cs` exercise which is a block type gap. There are two tests configured for the gap. The first test uses the print mechanism specified in the

technique attribute, while the dump state is adopted in the second test. Similar to black box testing, the delimiter used in the print mechanism is generated by hashing the code of the gap model solution and the generated time when the test is configured. The dumping code for the second test is inserted at the end of the gap code specified by using the post-code node in the configuration. The pre-code node would be used to indicate if the dumping code should be inserted at before the gap code.

To generate the student and model solutions, the test generation component queries the exercise database for the exercise template, files, solution instance and the student saved attempt instances. For black box testing, all student gap solutions are inserted into the program template and delimiters are embedded at the beginning and end of each gap in the exercise to construct the student solutions. For white box testing, only the student tested gap is inserted into the program template while the gap model solutions are occupied other gaps. When the print mechanism is used, delimiters are inserted at the beginning and end of the gap code while gap is wrapped with variable state dump code if the state dump mechanism is adopted.

After all the programs are constructed and stored in the appropriate student temporary folders in the server, these programs are compiled. With C# programming exercises, the test driver program is compiled into an executable file (.exe) using the Mono (2003) compiler. Test inputs, student program and other necessary testing classes for each test are packed into a .dll file. Students only download the executable test driver which uses dynamic loading to retrieve the .dll file from the server for each test at runtime. The downloaded test .dll is then loaded into the client's machine assembly cache to run. When the test finishes, the .dll is unloaded and the test thread is destroyed. On-demand test inputs and other necessary libraries downloads reduce the initial test start-up time; when the testing process starts, only the executable test driver is downloaded. With Java programming exercises, the test driver and student programs are compiled using the Sun Java compiler. They are then packed into a JAR; the driver program is set as the JAR main class in the manifest file. A custom class loader is used to load the student program and run it. If there is a compilation error, students are advised to contact their instructor for assistance.

```

<?xml version="1.0" encoding="utf-8" ?>
<white-box-test>
<class name="Power.cs">
<gap id="1">
    <test id="1" name="test output format" technique="print">
        <driver in-line="true">
            <delimiter><![CDATA[ABCDEFGH1HAA1B4CEF]]></delimiter>
        </driver>
        <inputs>
            <set id="1">
                <input id="2" />
                <output id="100" />
            </set>
        </inputs>
    </test>
    <test id="2" name="test the calculation" technique="dump">
        <driver in-line="true">
            <post-code>
                <![CDATA[DumpState.dump(power, "power.so");]]>
            </post-code>
        </driver>
        <inputs>
            <set id="1">
                <input id="2" />
                <output id="101" />
            </set>
        </inputs>
    </test>
</gap>
</class>
</white-box-test>

```

**Figure 50: White Box Testing Configuration**

### 5.6.2 Black Box Testing

In black box testing, the student program and the exercise model solutions are invoked with the same set of test inputs. Depending on gap type, gap output markers are inserted before and after the gap code to mark the outputs produced by the gap. The student gap outputs are extracted from the complete program output and compared with the corresponding model gap solution for correctness.

As noted earlier, the test execution process is conducted in the student's machine; when each test finishes running, the test output is sent back to the server for analysis. Test output is parsed into a DOM so that output from gap codes and non-gap codes can be separated easily. This can be done because gap output markers are inserted in XML format; beginning gap outputs have the form of an opening tag and end gap outputs has the form of a closing tag. The comparison process is fully configurable by teaching staff. Filter and normaliser rules can be applied to each gap output to extract all important keywords or values in it. These keywords or values are then compared with the set in the gap solution to provide feedback to students.

Filters and normalisers are implemented using Java regular expressions. Currently, there are two filters, two matchers, and two normalisers implemented to compare students' program outputs with that of the model solution. Table 22

summarises all filters, matchers and normalisers together with their sub-options. Dynamic class loading is used for the filters, matchers and normalisers; this makes the framework extensible as additional filters and matchers can be easily plugged in.

<b>Filters</b>	<ul style="list-style-type: none"> <li>• Number filter</li> <li>• Keyword filter           <ul style="list-style-type: none"> <li>◦ Case sensitive</li> </ul> </li> </ul>
<b>Matchers</b>	<ul style="list-style-type: none"> <li>• Exact match</li> <li>• Match disregard the order</li> </ul>
<b>Normaliser</b>	<ul style="list-style-type: none"> <li>• Space or tab           <ul style="list-style-type: none"> <li>◦ Leading and trailing space</li> <li>◦ Space or tab between words</li> </ul> </li> <li>• New line character           <ul style="list-style-type: none"> <li>◦ Leading and trailing new line character</li> </ul> </li> </ul>

**Table 22: Functions Provided to Check the Result of Black Box Testing**

A test report is generated based on the comparison result which is marked up and stored in XML format in the database on the server. Figure 51 gives an example of a black box testing XML report for the Power.cs exercise based on the configuration shown in Figure 49 in which no filters and normalisers are used in the output analysis process. As shown in Figure 49, the report identifies the matching results of both gap code and non-gap code outputs. In the example case, the student output is matched with the expected solution, otherwise, the student gap output and the corresponding model solution are attached in the gap matching-result node in the XML. Feedback is generated for students by incorporating all test reports.

```

<?xml version="1.0" encoding="UTF-8"?>
<compare-result>
<exercise-name>/ITB610/Week 4/Power</exercise-name>
<tested-category>1</tested-category>
<tested-case>1</tested-case>
<input-id>2</input-id>
<input-file>input2.txt</input-file>
<test-error/>
<matches>
    <matching-result gapId="nongap">true</matching-result>
    <matching-result gapId="1">true</matching-result>
    <matching-result gapId="2">true</matching-result>
    <matching-result gapId="3">true</matching-result>
</matches></compare-result>

```

**Figure 51: Black Box Testing XML Report**

### **5.6.3 White Box Testing**

White box testing is carried out by mix matching student gap solutions and model solutions to identify errors in gaps. The student's gap solutions are one at a time inserted into the test harness programs while the model gap solutions occupy the remaining gaps to produce the mixed programs. The two test harness programs - one mixing student gap solutions and model gaps solutions, and the other one containing the gaps model solution - are then compiled and executed with the same set of test inputs. The student's gap outputs and the model gap outputs are compared.

Similar to black box testing, the white box testing process takes place on the students' machines and the test outputs analysis process is carried out on the server. A test report is generated and stored on the database for each test. The test report will be incorporated with other test reports to provide feedback to students.

The analysis supports two types of test harness program, namely, the default test harness program, which is the exercise template, and a customised one, which is provided by teaching staff. If the default test harness program is used, multiple versions of the student program are generated; one version for each tested gap in which the tested gap is replaced with the student solution and surrounded with tested code as specified in the test configuration. If there are dependencies among gaps in the exercise, one or more gaps can be tested at the same time.

There are two different mechanisms which can be used to perform white box testing: variable state dumps and print. The variable state dumps approach is used to track changes of variables in gaps which have variables whose states are changed at runtime. In contrast, the print approach is applied to test gaps which only modify the program output; it is used to separate gap outputs from the test harness program outputs. The output of the gap is extracted and compared with the output of the model gap solution using the same matching mechanism as for black box testing.

In the variable state dump approach, a StateDump class is implemented to provide a set of overloaded dump methods which dump the state of different variable types at runtime. This class also provides a set of overloaded dumpExpression methods to dump the values of different expression types. States of variables in a program are captured using reflection and are serialised into XML string to be sent from the client to the server. This allows the analysis to perform testing for any programming language. There are four possible types of variables that can appear in

a program: primitive, reference, array (either primitive or user defined object) and user-defined. With a user-defined type, the recorded information includes object type, classes and interfaces that the user-defined type extends and implements, and all fields in the object's class. Modifiers, type and the name and value of a field are recorded. Teaching staff can choose to perform a deep serialisation which will recursively serialise a user-defined class or array of user-defined classes which is referenced in the current class. Each element of an array of user-defined type is serialised. Examples of XML serialized format of primitive type, array of primitive type and user-defined type are shown in Appendix B.

The value comparison process for serialised objects is configurable and extensible. Teaching staff can configure the comparison process to only check the type, format or range of numerical value. With an array type variable, it is possible to compare only values in the array regardless of an array element's position. With variables of user-defined type, teaching staff can choose to check only fields whose values would be changed after gap execution. Figure 52 gives an example of an XML test report result.

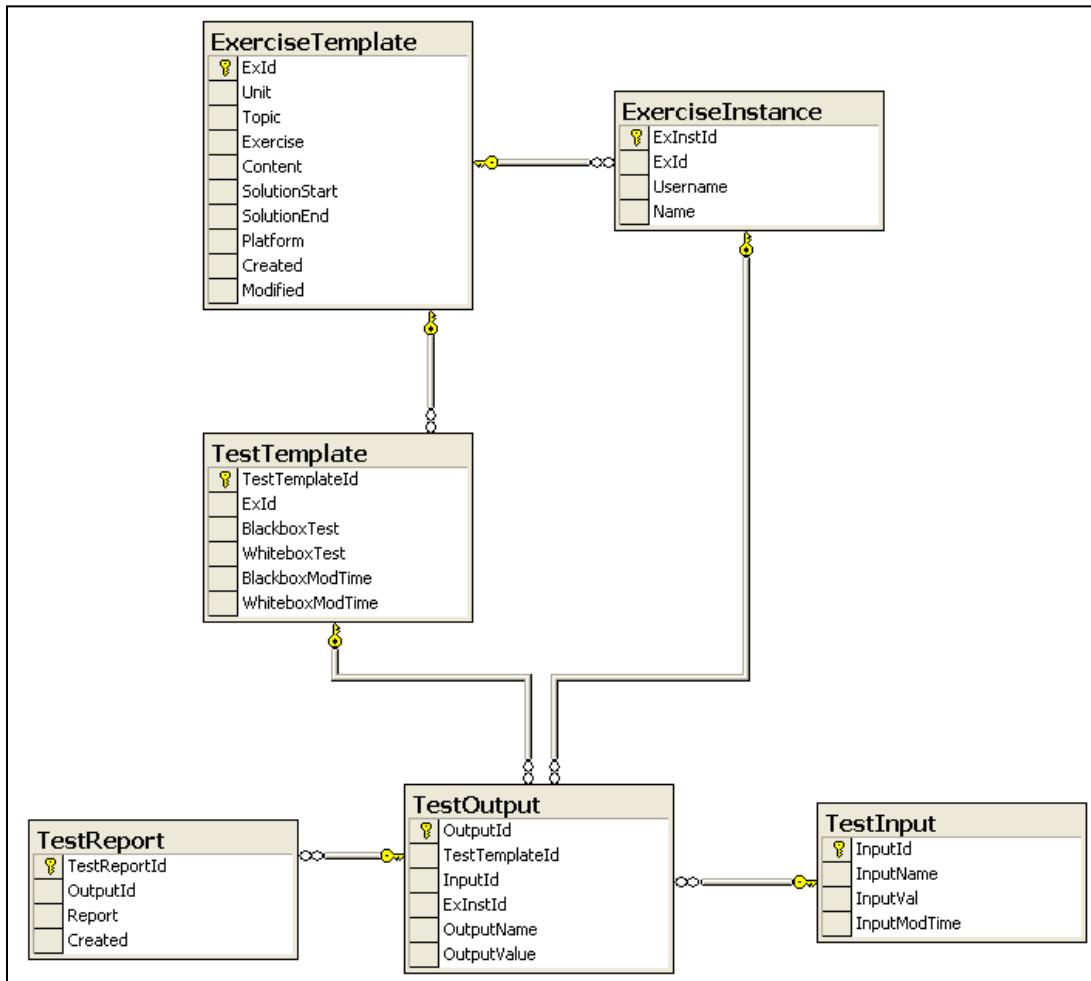
```
<?xml version="1.0" encoding="UTF-8"?>
<compare-result>
    <exercise-name>/ITB610/Week 4/Power</exercise-name>
    <tested-class>Power.cs</tested-class>
    <tested-gap>1</tested-gap>
    <test-id>1</test-id>
    <input-file>input2.txt</input-file>
    <input-id>2</input-id>
    <test-error/>
    <matches>
        <matching-result><matched>true</matched>
        <differences/>
        <model-output>
            <primitive-int name="power.so"><! [CDATA[8]]></primitive-int>
        </model-output>
        <student-output>
            <primitive-int name="power.so"><! [CDATA[8]]></primitive-int>
        </student-output>
        </matching-result>
    </matches>
</compare-result>
```

**Figure 52: White Box Test XML Report**

#### 5.6.4 Database Structure

The testing information is stored in a test database which contains four tables: TestTemplate, TestInput, TestOutput and TestReport. Each exercise in the ExerciseTemplate table is associated with a test template stored in the TestTemplate table. Each test template has one or more test inputs; the information of these inputs

is stored on the TestInput table. Each output produces by executing the model solution instance of the exercise with the input specified in the test template stored on the TestOutput table. The TestReport table stores the comparison result of the student solutions output with the model solution. When there is an update in the exercise template or test template, all existing test outputs and test reports are invalidated. Figure 53 shows the relationship among tables in the test database.



**Figure 53: Database Structure for Dynamic Analysis**

Name	Data Type	Description
TestTemplateId	int(4)	Unique ID of a test template
ExId	int(4)	Unique ID of an exercise that using the test template
BlackboxTest	text	Black box test configuration
WhiteboxTest	text	White box test configuration
BackboxModTime	timestamp	Last modified time of black box test configuration
WhiteboxModTime	timestamp	Last modified time of white box test configuration

#### TestTemplate Table Fields Descriptions

Name	Data Type	Description
InputId	int(4)	Unique ID of an input
InputName	varchar	Name of the input file
InputVal	varchar	Input value
InputModTime	timestamp	Last modified time of an input

**TestInput Table Fields Descriptions**

Name	Data Type	Description
OutputId	int(4)	Unique ID of the output
OutputName	varchar(50)	Output file name
OutputValue	varchar(200)	Output value
TestTemplateId	int(4)	Template that referencing the output
InputId	int(4)	Input ID that is used to generate the output
ExInstId	int(4)	Unique ID of the model solution instance

**TestOutput Table Fields Descriptions**

Name	Data Type	Description
TestReportId	int(4)	Unique ID of the report
OutputId	int(4)	OutputId that this report is generated on
Report	text	Comparison result report
Created	timestamp	Time that the report is created

**TestReport Table Fields Descriptions**

**Table 23: Dynamic Analysis Database Tables Fields Descriptions**

## 5.7 Summary

In summary, this chapter has presented the dynamic analysis which has four key characteristics. Firstly, it is able to analyse “fill in the gap” Java and C# programming exercises. Secondly, the analysis is web-based, allowing students to undertake exercises and receive feedback at anytime, anywhere. Thirdly, the analysis is configurable to support great flexibility in comparing the outputs of student programs with those of the model solution. It also allows teaching staff to specify customised feedback. Lastly, it is extensible; it can be extended to test different programming languages and types of exercises.

The dynamic analysis carries out both black box and white box testing and only student gap codes are tested. The main purpose of black box testing is to ensure that an exercise implements all the required functions correctly, while white box testing is performed to refine the results of black box testing and to detect any possible errors which have not been revealed in black box testing. With black box testing, the student gap attempt is inserted into the exercise template to construct the student program. This program is then executed with the model solution using the same set of test inputs. Student gap outputs and the corresponding gap model solutions outputs are compared with each other. A set of filters, normalisers and

matchers are provided to differentiate significant and insignificant information in the program outputs.

In white box testing, each student gap or a group of dependent gaps are inserted into a test harness program and tested one at a time. Two different mechanisms are provided to carry out white box testing: variable state dumps and program prints. These mechanisms are designed to accommodate two types of gap behaviours: gaps in which the states of variables are changed, and gaps which only modify the program output. Variable state dumps are used to check the state of variables before and after gap execution, while the print method is used to check those programs whose gaps only modify the output. White box testing supports two types of test harness programs: a default test harness which is the exercise template, and a customised one which is provided by teaching staff. When the default test harness is used, student gap solutions are one at a time inserted into the template while the remaining gaps are occupied by gap solutions. In contrast, with the customised harness, only the student gap solution is inserted into the program.

The analysis makes use of client-server communication architecture whereby the execution of students' programs takes place on the students' own machines, while the correctness evaluation is carried out on the server. This architecture ensures the security and increases the flexibility and efficiency of the analysis since it prevents the execution of malicious student code and reduces the load on the server. Lecturers can set customised feedback in both black box and white box testing. The analysis is currently not able to test for the correctness of exercises which perform file input and output operations and GUI exercises. Approaches to extend the analysis to support this type of exercise are discussed in Chapter 8.

# **Chapter 6 - ELP Feedback**

The previous three chapters presented the innovative web-based programming environment - the ELP - and the static and dynamic analyses components of the program analysis framework. The ELP system is designed to help novices more easily learn to program in modern OO programming languages. The static analysis assures the quality of students' programs, and the dynamic analysis verifies the correctness of students' programs. This chapter discusses the various types of automated feedback that the ELP system and the program analysis framework provide for students and teaching staff.

## **6.1 Introduction**

Feedback is a critical factor in a student's learning process (Bangert-Drowns et al., 1991; Milton, 2001; Whyte et al., 1995). It facilitates learning (Dihoff, 2004), helps learners to identify errors, and eliminates misconceptions. In addition, feedback is also a significant factor in motivating further learning (Whyte et al., 1995; Kwong, 2001). Research in feedback started as early as in the 1950s (Pressey, 1950; Skinner 1968). Most of research on feedback at that time was conceptualised within a behavioural framework in which feedback was considered as a contingent event that reinforces or weakens responses. Positive comments were provided on correct responses; errors did not receive positive feedback and were weakened. For a comprehensive review of those early studies refer to Kulik and Kulik (1988). From the 1970s onward, with the presence of information processing theory (Miller, 1956), errors were no longer considered as mistakes but regarded as an expected part of learning; they provide important information for teaching and learning. Providing feedback based on this theory gives hints to learners on how to correct their errors, helps them determine the performance expectations, judges their level of understanding, and eliminates misconceptions. Most current research in feedback is built on this information processing theory.

Among other aspects in feedback research, feedback timing has the most substantial history. Even though there are many debates on the effectiveness of immediate and delayed feedback on students' learning, most researchers agree that immediate feedback has great benefits for weak students in many areas including computer science education (Hundhausen and Brown, 2005). This is because these

students are unable to achieve correct answers based on their existing knowledge. The answers are out of their Zone of Proximal Development (ZPD), as discussed in Chapter 2. Hence, they need the support of a more knowledgeable person, relative to their ZPD. They might interpret the cause of their problems incorrectly which leads to misconception. Additionally, as discussed in Chapter 3, weak students often lack confidence and have low motivation; the delay in feedback might cause frustration for them. Unfortunately, with the typical current class setting, students often have to wait for days or weeks to receive feedback from teaching staff about their work due to the heavy schedule and large class sizes.

The main contribution of this chapter is to present the ELP feedback engine which provides feedback for students and teaching staff. For students, the engine gives customised compilation error messages and comments on quality, structure and correctness of their programs based on static and dynamic analyses results. One of the biggest learning difficulties for novice programmers is achieving a compilable program because they lack the programming language syntax knowledge. This is not helped by unfriendly compiler error messages. Customised compilation error messages are designed to make syntax and semantic errors easier to find, more user-friendly and less cryptic. These error messages indicate the line numbers which are located in a program together with examples on how they can be resolved. For staff, the system allows them to track students' progress, common logic errors, and programming practices. Based on this information, staff can provide additional help for weak students and adjust the teaching material to address students' needs.

Students' learning becomes more interactive, flexible, and effective when feedback about their program quality, structure, and correctness is provided immediately. They no longer have to endure delays in obtaining feedback from teaching staff about their work. They do not have to come to university and be constrained within the standard consultation time in order to check if their work is correct. Most importantly, students can reflect on the feedback in the context of the exercise that they are currently working on; thereby their knowledge is reinforced. Based on the provided feedback, students can further experiment with alternative solutions to resolve a programming problem. This process gives students deeper understanding of programming concepts and therefore allows them to transform the new constructed knowledge to solve other programming problems.

The system facilitates constructivist learning. Teaching staff can make all learning materials available on the ELP system for students to actively take ownership of their learning. Staff can design lessons which scaffold students' learning. More importantly, staff can also constantly assess learning progress and adjust the teaching material to best address students' needs. Active students can evaluate and synthesise information in the provided feedback and transform this information to become his or her understanding of a particular topic.

This chapter is divided into three sections. Section 6.2 gives a brief overview of research on computer-based instruction feedback, and highlights the criteria of effective feedback. Section 6.3 discusses the design of several types of automated feedback that the ELP system provides to students and staff. Section 6.4 presents the implementation of those automated feedback mechanisms.

## **6.2 Feedback in Computer-Based Instruction**

As noted earlier, there was a significant change in research on feedback after the information processing theory was introduced. The present section reviews the research on effective feedback in computer-based instruction after the appearance of that theory.

According to Kulhavy and Stock (1989), two critical components of effective feedback are *verification* and *elaboration*. *Verification* is a simple judgment of whether an answer is correct or incorrect. *Elaboration* is an informational component providing relevant hints to guide a learner towards a correct answer. It is well acknowledged that successful feedback must include both *verification* and *elaboration* (Bangert-Drowns et al., 1991; Kulhavy and Stock, 1989). The combination of the two components can highlight errors in answers, give correct answer options and provide information that strengthens correct answers and makes them memorable (Bangert-Drowns et al., 1991).

There are three types of *elaborations* namely: *informational*, *topic specific* and *response specific* (Kulhavy and Stock, 1989). *Informational elaboration* does not address individual answers but provides information from which the correct answer can be drawn. *Topic specific elaboration* provides specific information about the correct answer but does not address incorrect answers. *Response specific elaboration* addresses both correct and incorrect answers. It explains why an answer is incorrect and provides information about the correct answer. This type of elaboration enhances

students' achievement more than other general forms of feedback (Whyte et al., 1995).

Depending on the detail level of verification and elaboration, feedback provided for students can take many forms. Researchers (Kulhavy and Stock, 1989; Merrill, 1987; Overbaugh, 1994; Schimmel, 1988) have identified eight common levels of feedback which are listed below:

1. *No feedback*: only contains the learner's score; it has no verification or elaboration.
2. *Knowledge of response*: specifies if the learner's answer is correct or incorrect but does not provide information about possible errors.
3. *Answer until correct*: has the same information as *knowledge of response*; however, the learner has to continue doing the exercise until the correct answer is achieved.
4. *Knowledge of correct response*: indicates if the learner's answer is correct or incorrect and provides correct answer information. However, elaboration is not provided.
5. *Topic contingent*: indicates whether the learner's answer is correct or incorrect. If the answer is incorrect, general elaborative information is provided for the learner to draw the correct answer.
6. *Response contingent*: provides both verification and elaboration; explains why an answer is correct or incorrect.
7. *Bug related*: relies on a bug list to identify and correct the learner's common errors. This type of feedback provides no information about the correct answer but helps learners to identify errors.
8. *Attribute isolation*: indicates if the learner's answer is correct or incorrect and highlights important attributes of the correct answer.

Research has shown that among eight levels of feedback, *response contingent* feedback significantly improves students' learning, and is more successful than the other levels (Gilman, 1969; Waldrop et al., 1986; Whyte et al., 1995). This is because learning is enhanced in feedback which has more *elaborative* information (Clariana, 1990; Clariana et al., 1991; Gilman, 1969; Morrison et al., 1995; Pridemore and Klein, 1991, 1995; Roper, 1977; Waldrop et al., 1986; Whyte et al.,

1995). The extra elaboration information in the feedback provides students with enhanced knowledge from which they can correct their misunderstandings, or decide about the correctness of an answer (Pridemore and Klein, 1991; Roper, 1977).

As well as the feedback forms and detail levels of verification and elaboration, the tone of feedback also has a great impact on student learning. Feedback should begin with a positive comment, have the right balance between negative and positive comments, and be written in an informal and conversational tone. All criticisms should be turned to positive suggestions (Rust, 2002).

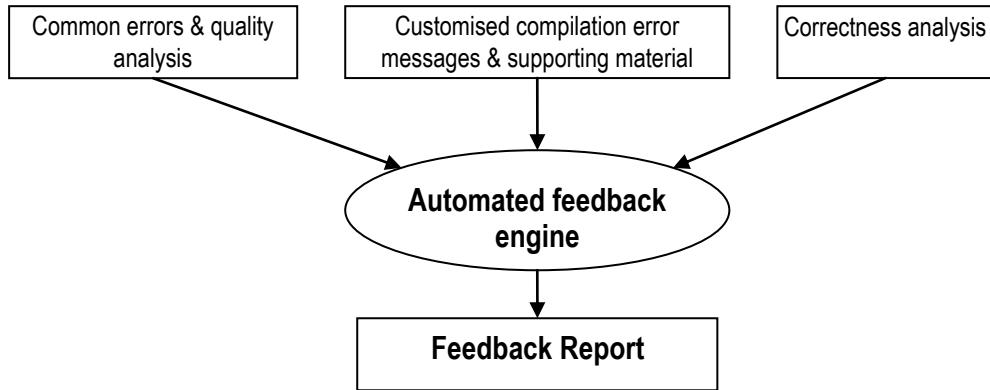
In short, feedback is of most benefit to weak students when it is immediately provided and consists of *verification* and *elaboration* components. For higher ability students, the effectiveness of feedback is influenced by many other factors such as their achievement levels, the depth of their knowledge and their attitude toward feedback. In the next section, how these good design attributes are adopted to provide feedback for students in the ELP system will be discussed.

### **6.3 Feedback for Students and Staff**

The section comprises two sub-sections. Sub-section 6.3.1 presents how customised compilation error messages and quality, structural and correctness feedback are provided for students, while Sub-section 6.3.2 discusses how students' progress information is reported to teaching staff.

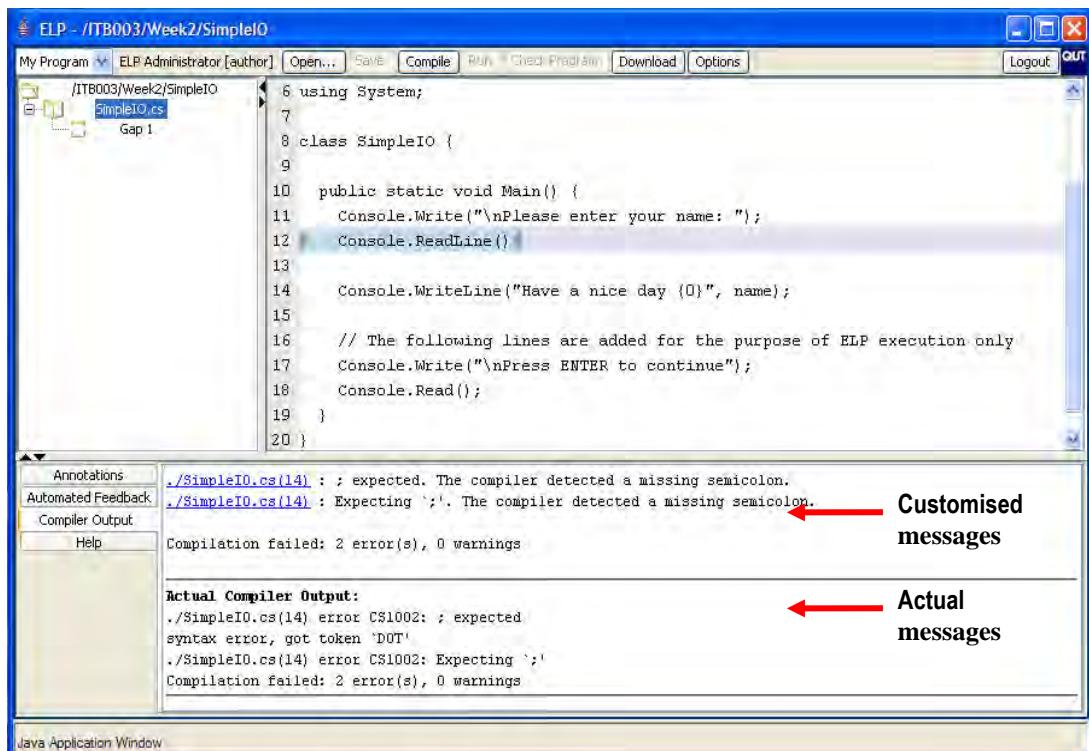
#### **6.3.1 Feedback for Students**

In the ELP system, students are supported in getting a program to compile with customised compilation error messages. They are also provided with instant comments on the quality and correctness of their programs, generated by combining static and dynamic analyses results. Figure 54 summarises different types of feedback provided for students by the automated feedback engine.



**Figure 54: Feedback Provided for Students**

Understanding compiler error messages has always been a major difficulty for novice programmers because these messages are cryptic and not user friendly. Forgetting to import a library can lead to having up to hundreds of error messages returned. In the ELP system, compilation error messages generated by Java and C# compilers are processed to make the errors more apparent and to include additional instructions on how to fix them. Students are informed if their programs are compiled successfully or not. If a program is successfully compiled, compilation feedback contains only a *verification* component which congratulates the student; otherwise it has both *verification* and *elaboration* components. The level of feedback is *topic contingent*, the fifth form of feedback discussed in the previous section. The *elaboration* component of the feedback consists of processed user-friendly compiler error messages together with examples of how to fix them and the actual compiler output. In addition, the actual compilation error messages are also displayed for advanced students to refer to and for weak students to become familiar with it step by step. Figure 55 gives an example of ELP customised compilation error messages when a semi-colon is missing in a program.



**Figure 55: Example of Customised Compilation Error Message**

As well as having been helped in getting programs to compile, students can check the quality and correctness of the programs by using the “Check Program” function on the ELP editor, as shown in Figure 56. This feedback can be viewed in the “Automated Feedback” tab on the ELP editor. The feedback provided to students is a combination of both static and dynamic analyses results. As discussed in Chapter 4, there are two groups of analyses in the static analysis component: SE metrics and structural similarity analysis. The SE analyses evaluate the quality of students’ programs using complexity metrics and good programming practice guidelines. Analyses in this group provide quantitative information about the quality of students’ programs. The structural similarity analysis is designed to verify the results of SE metric analyses. It identifies high complexity areas in codes by comparing the high-level algorithmic structure of student solutions with that of the model solutions. The two analyses are supporting each other. With dynamic analysis, as presented in Chapter 5, black box and white box testing are carried out to verify the correctness of students’ programs. Black box testing checks if student programs perform all required functions correctly, while white box testing reveals any errors which have not been identified by black box testing. The black box and white box testing results are supporting each other.

The screenshot shows a software interface for editing and running programs. At the top, there's a menu bar with 'My Program' (selected), 'New Student [student]', 'Save', 'Compile', 'Run', 'Check Program', 'Download', and 'Options'. On the right side of the menu bar, there are 'Logout' and 'OUT' buttons. Below the menu, the code editor displays a C# script with numbered lines from 12 to 34. Lines 26 through 31 are highlighted in light blue, indicating a conditional block. Line 34 contains a comment: '// The following lines are added for the purpose'. The status bar at the bottom left shows 'Annotations', 'Automated Feedback' (highlighted in orange), 'Compiler Output', and 'Help'. A red arrow points to the 'Compiler Output' tab. In the main window area, the message 'Compiled Successfully' is displayed.

```

12 // Get user input of an integer
13 Console.WriteLine("Enter an integer < 10,000: ");
14 input = Console.ReadLine();
15 number = int.Parse(input);
16
17 // Let user choose between finding the integer's inverse or square
18 // User to input 'i' or 'I' for inverse; 's' or 'S' for square
19 Console.Write("Find the inverse or square? i/s: ");
20 input = Console.ReadLine();
21 choice = char.Parse(input);
22
23 // Use an IF/ELSE block to determine which method to call
24 // Call that method
25 // Print the result
26 if (choice == 's' || choice == 'S') {
27     Console.WriteLine("The square of {0} is {1}", number, findSquare(number));
28 } else if (choice == 'i' || choice == 'I'){
29     Console.WriteLine("The inverse of {0} is {1}", number, findInverse(number));
30 } else
31     Console.WriteLine("Invalid choice");
32
33
34 // The following lines are added for the purpose

```

**Figure 56: ELP Editor Applet**

The quality and correctness feedback provided by the ELP system consists of both *verification* and *elaboration* components and the feedback level is *response contingent* which is one of the eight levels of common feedback, as discussed in the previous section. The feedback is written in informal and student-centric language. The *verification* component of the feedback confirmed if an answer is correct or incorrect, while the *elaboration* component explains why an answer is correct or incorrect. As discussed in the previous section, *elaboration* in the *response contingent* level of feedback enhances student learning the most, compared to other forms of elaboration. Providing quality and correctness feedback for students based on their answers will address individual students' needs. The system facilitates effective learning environments. Students' learning is scaffolded to overcome their difficulties and to achieve correct answers for programming problems. They can interact with the ELP system, experiment with various approaches to tackle a programming problem, construct knowledge, and receive immediate feedback in the context of the current exercise. This process will reinforce the new constructed

knowledge which students can then use to solve other problems. The system facilitates constructivism.

The quality and correctness feedback is comprised of three sections. The first section is a short summary about the overall quality, structure and correctness of a program. This is the *verification* part of the feedback. The second section is feedback about the program correctness based on dynamic analysis results, and the third section is comments on the program structure and quality based on the static analysis results. Both the second and third sections have a short summary on the overall result (*verification*), followed by more details (*elaboration*). Figure 57 and Figure 58 give examples of feedback provided for a correct gaps attempt, while feedback for an incorrect answer is shown in Figure 59 and Figure 60.

Program correctness feedback is provided in the form of *response contingent* feedback and consists of both *verification* and *elaboration*. The feedback starts with a short summary (*verification*), indicating whether a program passes all tests together with all conducted test names which are displayed in increasing level of difficulty. If a program passes all tests, a congratulation message is presented as the summary; otherwise it provides information on the number of failed tests, runtime errors and missing test outputs due to a connection issue, or the testing process is cancelled. A tick or a cross image is displayed next to each test name to indicate if a test is passed or failed. With failed tests, test names are displayed as links which students can click on to obtain more information (*elaboration*), as illustrated in Figure 59. In that case, students are given hints on which gaps might cause the test to fail together with an additional explanation for that particular test which is specified by teaching staff. Any runtime errors are also reported.

Similarly, *response contingent* form of feedback is also adopted to provide comments on program structure and quality to students. All conducted analysis names shown as links together with a tick or a cross image to indicate if the student code passed or failed the analysis. An example of this feedback is shown in Figure 58. Students can click on those links to obtain more detailed explanation *elaboration*, on why their programs failed that analysis, hints on how to improve the program. All analyses feedback is provided in gap-based format as shown in Figure 58 and Figure 59. In the Figures, the detailed feedback for the check logic expression and solution similarity analyses of static analysis is displayed in tabulated format; the feedback is provided only for gaps in the exercise for which these analyses are configured.

General hints, shown in Figure 61, are provided for students to improve the quality of their programs.

The presentation of the feedback has been described; the remainder of this section will look closely at how static and dynamic analyses results are incorporated to generate program correctness, quality and structure feedback respectively.

The screenshot shows a software interface for managing student programs. At the top, there's a menu bar with options like 'My Program', 'New Student [student]', 'Save', 'Compile', 'Run', 'Check Program', 'Download', 'Options', 'Logout', and 'QUIT'. Below the menu is a code editor window displaying C# code:

```
19     Console.WriteLine("Find the inverse or square? i/s: ");
20     input = Console.ReadLine();
```

To the left of the code editor is a sidebar with links: 'Annotations', 'Automated Feedback', 'Compiler Output', and 'Help'. The main content area is titled 'Summary' and contains the following text:

You have written a good program; it works and has good structure. Below is a detailed report on tests and analyses that were carried out on your program.

**Program Testing Results**

Congratulations, your program produces correct results for all tests.. Following are the tests that were carried out on your program

Simple Tests
✓ find square of 9
✓ find inverse of 9
✓ find square of 25
✓ find inverse of 25

**Program Structure and Quality**

Programming Practice

✓ [Redundant Logic Expression Check](#)

Program Structure

✓ [Solution Similarity](#)

**Figure 57: Correct Answer Verification**

The screenshot shows a programming interface with a code editor and a feedback panel.

**Code Editor:**

```

My Program New Student [student] Save Compile Run Check Program Download Options Logout QUIT
/ITN600/Week 3/Lecture3Ex1.cs
19     Console.WriteLine("Find the inverse or square? i/s: ");
20     input = Console.ReadLine();
21     choice = char.Parse(input);
~~

```

**Feedback Panel:**

### Program Structure and Quality

- Programming Practice
  - ✓ [Redundant Logic Expression Check](#)
- Program Structure
  - ✓ [Solution Similarity](#)

---

#### Redundant Logic Expression Check Report

Class	Gap	Feedback
Lecture3Ex1.cs	1	Great! None of the boolean expression in this gap has the following format <code>x==true</code> (where x is an boolean expression) Keep up the good work.

[Top](#)

#### Solution Similarity

File Lecture3Ex1.cs

Congratulations! your program has a similar structure to the model solution.

[Top](#)

**Figure 58: Correct Answer Elaboration**

The screenshot shows a programming interface with a code editor and a testing results panel.

**Code Editor:**

```

My Program New Student [student] Save Compile Run Check Program Download Options Logout QUIT
/ITN600/Week 3/Lecture3Ex1.cs
38     Console.ReadLine();
39   )
40
41 // Method to find the square of an integer
~~

```

**Testing Results Panel:**

### Summary

Your program does not seem to work. It fails 2 automated tests but it has good structure. Below is a detailed report on tests and analyses that were carried out on your program.

#### Program Testing Results

Your program produces incorrect results for 2 tests. There might be problem with gap(s) 1.. Following are the tests that were carried out on your program

Simple Tests
✗ <a href="#">find square of 9</a>
✓ <a href="#">find inverse of 9</a>
✗ <a href="#">find square of 25</a>
✓ <a href="#">find inverse of 25</a>

**find square of 9**  
 This test fails because of gap 1. This might be because the calculation/logic in your gap code or gaps which are called by this gap are incorrect; or your print out statements do not follow the instructed format

**find square of 25**  
 This test fails because of gap 1. This might be because the calculation/logic in your gap code or gaps which are called by this gap are incorrect; or your print out statements do not follow the instructed format

**Figure 59: Incorrect Answer Verification**

The screenshot shows a programming interface with the following details:

- Code Editor:** Displays the file `Lecture3Ex1.cs` containing C# code:
 

```
1 using System;
2
3
4 class Lecture3Ex1
5 {
6     public static void Main()
7 }
```
- Annotations:** A sidebar on the left lists "Annotations", "Automated Feedback", "Compiler Output", and "Help".
- Program Structure and Quality:** A section with two items:
  - Programming Practice:** Shows a red X icon next to "Redundant Logic Expression Check".
  - Solution Similarity:** Shows a green checkmark icon next to "Solution Similarity".
- Redundant Logic Expression Check Report:** A table showing the results of the check:
 

Class	Gap	Feedback
Lecture3Ex1.cs	1	There are 1 conditional expression(s) or the IF statement(s) in your program perform redundant boolean expression check.
- Solution Similarity:** A section indicating similarity with the model solution:
  - File Lecture3Ex1.cs:** Shows a green checkmark icon.
  - Congratulations!** Your program has a similar structure to the model solution.
  - Click here:** For further help on how to improve the quality of your program.

**Figure 60: Elaboration on Incorrect Answer**

The screenshot shows a programming interface with the following details:

- Code Editor:** Displays the file `Lecture3Ex1.cs` containing C# code:
 

```
1 using System;
2
3
4 class Lecture3Ex1
5 {
6     public static void Main()
```
- Annotations:** A sidebar on the left lists "Annotations", "Automated Feedback", "Compiler Output", and "Help".
- Help - Analysis Hints:** A large central panel titled "Help - Analysis Hints" containing a table of hints:
 

Analysis Name	Hints
Code Complexity	If your program is too complex, you should consider reducing the number of conditional and loop statements in it.
Redundant Logic Expression Check	Change any of the boolean expressions in the following format if (x == true) or if (true == x) to if (x)
Check Magic Numbers	You should not have hard coded numbers in your programs. You should consider declaring them as constants.
Check Switch Statement	Check all the switch statements in your program and make sure that each case in a switch statement has a break statement and each switch statement has a default case.
Check Unused Parameters	Check if there are any unused parameters in any of your methods.
Check Unused Variables	Check if there are any unused variables in your program.
Check Access Modifiers	Make sure you use "public", "private" and "protected" access modifiers correctly as specified in the exercise requirements.
Solution Similarity	The structure of your solution is not similar to the suggested solution, but there are many ways to solve a problem. However, if your program failed all the automated tests, you should consider modifying your solution and checking it again.

**Figure 61: Programming Practice Hints**

### 6.3.1.1 Program Correctness Feedback

As discussed in Chapter 5, black box testing and white box testing are used to verify the correctness of students' programs. The testing process is carried out on the student's machine and test outputs are sent back to the server for analysis. That means there is the possibility that test outputs are not received by the server due to

connection errors or early termination of the testing process. There are three possible test output scenarios which can occur for an exercise in which only either black box or white box testing is conducted, while there are seven possible scenarios when both testing strategies are applied. These scenarios are shown in Table 24.

<b>Only black box or white box testing</b>
1. receiving all test outputs
2. receiving some test outputs
3. missing all test outputs
<b>Black box and white box testing</b>
1. receiving all black box and white box test outputs
2. receiving all black box tests and receive some white box tests outputs
3. receiving all black box and missing all white box tests outputs
4. receiving some black box and receive all white box tests outputs
5. missing all black box and receive all white box tests outputs
6. receiving some black box and receive some white box tests outputs
7. missing all black box and missing all white box tests outputs

**Table 24: Dynamic Analysis - Test Outputs Receivable Possibilities**

For all test outputs received by the server, there are three possible scenarios in terms of their correctness for an exercise to which only black box or white box testing is applied. There are seven possible scenarios that can arise for exercises that make use of both black box and white box testing. Table 25 summarises the test output correctness cases. Among those scenarios, two cases should be noted: the first case is that all black box tests are correct and all white box tests fail - this can occur if test inputs are not comprehensively designed and the number of conducted tests is not sufficient to reveal hard coded value in the student's program; the second case is that all white box tests are correct and all black box tests are incorrect - this probably occurs when program output format is an important factor in the expected results.

<b>Black box or white box testing</b>
1. all tests are correct
2. some tests are correct
3. all test are incorrect
<b>Black box and white box testing</b>
1. all black box tests are correct and all white box tests are correct
2. all black box tests are correct and all white box tests are incorrect
3. some black box tests are correct and all white box tests are correct
4. some black box tests are correct and some white box tests are correct
5. some black box tests are correct and all white box tests are incorrect
6. all black box tests are incorrect and all white box tests are correct
7. all black box tests are incorrect and all white box tests are incorrect

**Table 25: Test Outputs Correctness Possibilities**

As noted previously, one of the key characteristics of feedback provided by dynamic analysis is to point out to students the gaps that might cause a test to fail. Depending on the adopted testing strategies (black box, white box or both), a gap which is considered to make a test fail is determined differently. With black box testing, gaps which do not produce a similar output as the expected output will be considered as the cause. With white box testing (which is carried out per gap base), if white box tests of a particular gap are failed, the gap is considered as the cause and is notified to users. When both black box testing and white box testing are conducted, a list of gaps whose outputs do not match the corresponding model solution outputs in the black box testing is combined with a list of gaps that either have unmatched output or variable states in the white box testing.

### 6.3.1.2 Program Quality and Structural Feedback

The program quality and structural feedback consists of both *verification* and *elaboration*. With the elaboration component, students are presented with a list of conducted analyses with a tick or a cross image displayed next to it indicating if the analysis is passed or failed. These analyses are displayed as a link so that students can click to receive further feedback. A ‘general hints’ page is included to explain to students what each of the analyses does and what might cause the analysis to fail, as shown in Figure 61. Analyses are displayed in groups according to their purposes as shown in Figure 57. There are four analysis categories: program structure, program comprehension, programming practices and object orientation; Table 26 summarises the provided analyses and their categories.

Program structure	Program Comprehension	Programming Practices	Object Orientation
<ul style="list-style-type: none"> <li>• Structural similarity</li> <li>• Gap statistics</li> </ul>	<ul style="list-style-type: none"> <li>• Code complexity</li> </ul>	<ul style="list-style-type: none"> <li>• Check magic numbers</li> <li>• Check unused variables</li> <li>• Check unused parameters</li> <li>• Redundant logic expression check</li> <li>• Check switch statements</li> </ul>	<ul style="list-style-type: none"> <li>• Check methods modifiers</li> <li>• Check variables modifiers</li> </ul>

**Table 26: Static Analysis Categories**

With the check unused parameters, redundant logic expression, unused variables and magic numbers analyses, the provided feedback indicates which parameters, variables or hard coded numbers are not supposed to be in the program together with their line numbers. Comments on the use of access modifiers for variables and methods in an OO program are provided for students in access modifiers analyses.

With structural similarity analysis, students are congratulated (as shown in Figure 58) when their programs have structures similar to the model solution; otherwise, the high-level structure of the student solution and the model solution are displayed to students to identify differences. A useful addition to this analysis is to incorporate the analysis results with testing results to predict reasons why a program fails dynamic analysis. This incorporation closes the gap between static and dynamic analyses and allows a finer level of feedback to be provided to students. Further discussion on how this can be developed is presented in Section 8.2.

In summary, the ELP system provides customised compilation error messages to assist students to achieve a compilable program and provides immediate feedback on structural, quality and correctness of their programs. The quality and correctness feedback is *response contingent specific* and consists of both *verification* and *elaboration*. With correctness feedback, students are presented with a list of conducted tests and their results. For each failed test, hints on gaps that might cause a test to fail, suggestions on how problems can be fixed and customised feedback from teaching staff are presented. Similarly, a list of analyses used to check the quality and structure of a student program is shown to the student. Feedback is provided for each gap in the program. For each failed analysis in the program quality analysis group, good examples for the analysis are shown so that students can build on those examples to improve their programs. If the student solution structure does not match the expected model solutions, the structures of the student and model solutions are presented to students for comparison.

### 6.3.2 Feedback for Teaching Staff

Through the ELP system, teaching staff can obtain valuable information about their students' progress on an exercise such as the number of compilation attempts before a compilable program is achieved, the time interval between compilation attempts, the correctness of each attempt and the student's programming practices. Figure 62

gives an example of the exercise browser interface provided for teaching staff to monitor students' compilation attempts. As shown in the Figure, the students' most recent saved attempt date, number of saved attempts, number of compilation attempts, and the success of the latest compilation attempt are presented to teaching staff. A student's most recent attempt for an exercise can be viewed by clicking on the exercise links.

Through the program analysis results, teaching staff can have a better understanding about their students' programming practices and any common logic errors. The information also gives them insight into their students' progress. Therefore they can provide additional help for weak students, and adjust teaching material to emphasise good programming practices or logic errors.

Student Attempts						
Exercise	Username	Name	Student	Tutor	Compiled	Saved
/ITB610/Week_2/ConvertHeight2	student	Test Student			✓ (2)	1/08/05(3)
/Demo/Java/HelloWorld	guest	Guest User			✓ (3)	1/08/05(4)
/Demo/CSharp/RomanToArabic	guest	Guest User			✗ (1)	9/08/05(3)
/ITB003/Week2>Hello	student	Test Student			(0)	25/07/05(3)
/ITB610/Week_2/ConvertHeight	n1892959	STEVEN HARRAP			✓ (8)	27/07/05(12)
/ITB610/Week_2/TestCast	n5461936	PEY LOW			✓ (6)	27/07/05(7)
/ITB610/Week_3/Arrays	n1892959	STEVEN HARRAP			✓ (3)	30/07/05(4)
/ITB003/Week2/ConvertHeight	student	Test Student			✓ (3)	2/08/05(3)
/ITB003/Week2/SimpleIO	n5399432	JUNBIN LIU			✓ (1)	14/08/05(3)
/ITB610/Week_4/TestLecturer	n1892959	STEVEN HARRAP			✓ (2)	6/08/05(4)
/ITB610/Week_4/Palindrome	n1892959	STEVEN HARRAP			✓ (1)	6/08/05(13)
/ITB610/Week_5/Question3	n1892959	STEVEN HARRAP			✓ (24)	13/08/05(32)

ID  Name

**Figure 62: Student Compilation Monitoring**

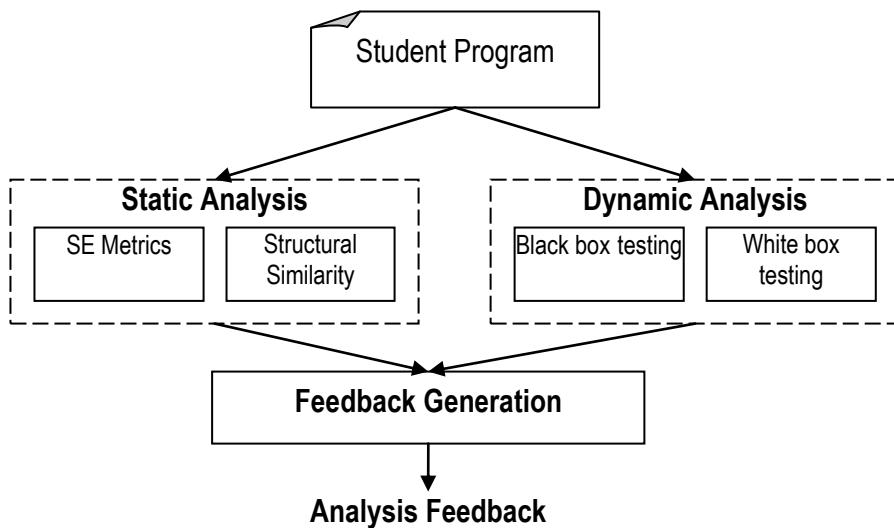
## 6.4 The Implementation for Feedback Generation

This section discusses the implementation of feedback provided by the ELP and the program analysis framework to students and staff. It is divided into two sub-sections. In Sub-section 6.4.1, implementation of feedback provided for students is presented while Sub-section 6.4.2 details how feedback for teaching staff is implemented.

### 6.4.1 Feedback for Students

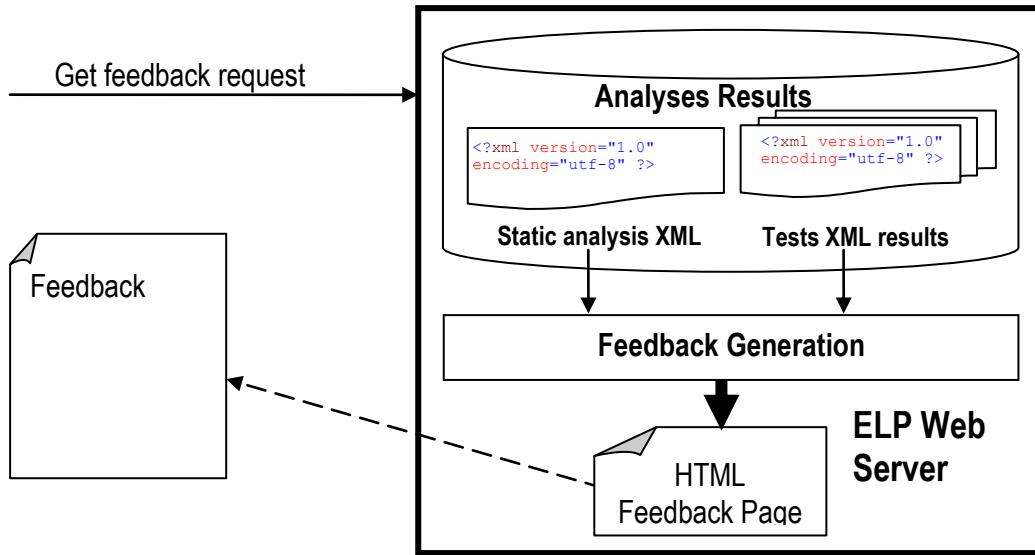
The ELP system provides customised compilation error messages and feedback on the quality and correctness of exercises for students. In order to provide students with user-friendly error messages, two text files were written. One contains friendly error messages for the Sun JDK compiler while the other one rephrases all error messages from the Mono (Mono, Retrieved June 1 2003) compiler. Error messages from the compiler are parsed to obtain the error code which is matched with the associated friendly error message in the text file.

With feedback on program quality and correctness, a feedback generation component is implemented to incorporate static and dynamic analyses results, as shown in Figure 63. The implementation of the component is discussed in the remainder of this section.



**Figure 63: Analysis Feedback Provided for Students**

When all tests and analyses finish running, students can switch to the “Automated Feedback” tab in the ELP editor which sends a ‘get feedback’ request to the server. When the request is received, all analysis and tests results associated with the user are retrieved from the database and sent to the feedback generation component to produce a feedback HTML page, as shown in Figure 64. As described in Chapters 4 and 5, these analysis results are stored as XML files in the database on the ELP web server. With static analysis, all analyses results for an exercise are stored in one single file, while with dynamic analysis each test result is stored in a separate file.



**Figure 64: High-Level Implementation of the Feedback Generation Component**

As described in the previous section, the program quality and correctness feedback consists of three sections: summary, program correctness and program quality and structure. The summary section reports on the overall quality of a program, its structure and correctness. In addition, the number of failed tests, runtime errors and missing test outputs are also presented. With static analysis, the value of the `pass` attribute for each analysis element (as shown in Figure 67) indicates if a gap passes or fails an analysis. With dynamic analysis, the number of failed tests is calculated by counting the number of `pass` attributes in the `matches` node which is set to ‘false’ in all black box and white box XML test results, as shown in Figure 65 and Figure 66. Similarly, the number of runtime errors is obtained by counting the number of `test-error` nodes which has children in all the test results files. The number of missing tests outputs is calculated by comparing the total number of configured tests in the test configuration files and the number of received test outputs.

The program correctness feedback also starts with a short summary on overall performance of the program in the testing process followed by a list of the test descriptions that were conducted. The summary reports on the number of failed tests, runtime errors, and missing test outputs. Test descriptions are obtained from the test configurations and displayed together with icons to indicate if the test passed, failed, has runtime errors or is missing test outputs.

In black box testing, a test is passed if all gap outputs in a student program are matched with those in the corresponding model solution; the matched node in the XML result file must be set to ‘true’ as shown in Figure 65. If a particular gap output does not match the model solution output, the output of the student’s gap and that of the corresponding model are appended as children of the differences node for that particular gap as shown in the XML. The gap id, the student and model solution outputs of that gap are recorded and presented to students as detailed feedback for that test. Similarly, in white box testing, the value of the matching result element indicates the success or failure of a test (Figure 66). At the same time, information on tests that produce runtime errors is also obtained. The `test-error` element in the XML result files contains all runtime errors which occurred during the test run. These errors will also be shown to students.

To generate the program quality and structure report, each analysis in the static analysis has a corresponding class responsible for generating the feedback for that analysis. These classes are loaded at runtime using Java Dynamic Class loading. Each class accepts the XML element result for their correspondence analysis as an input and returns a meaningful HTML feedback string. These strings are combined to make up the report. For example, in Figure 67 the `GenerateCyclomaticComplexityFeedback` and `GenerateStructuralSimilarityFeedback` classes will be invoked with `CyclomaticComplexity`, `StructuralSimilariy` XML elements and all their children.

```

<?xml version="1.0" encoding="UTF-8" ?>
<compare-result>
    <exercise-name>/ITB610/Week 3/RomanToArabic</exercise-name>
    <tested-category>1</tested-category>
    <tested-case>7</tested-case>
    <input-id>15</input-id>
    <input-file>inbb13.txt</input-file>
    <test-error />
    <matches pass="true">
        <matching-result gapId="nongap">
            <matched>true</matched>
            <differences/>
        </matching-result>
        <matching-result gapId="1">
            <matched>true</matched>
            <differences/>
        </matching-result>
        <matching-result gapId="2">
            <matched>true</matched>
            <differences/>
        </matching-result>
        <matching-result gapId="3">
            <matched>true</matched>
            <differences/>
        </matching-result>
    </matches>
</compare-result>

```

**Figure 65: Black Box Test Result - XML File**

```

<?xml version="1.0" encoding="UTF-8" ?>
<compare-result>
    <exercise-name>/ITB610/Week 3/RomanToArabic</exercise-name>
    <tested-class>RomanToArabic.cs</tested-class>
    <tested-gap>1</tested-gap>
    <test-id>1</test-id>
    <input-file>input18.txt</input-file>
    <input-id>16</input-id>
    <test-error />
    <matches pass="true">
        <matching-result>
            <matched>true</matched>
        </matching-result>
        <differences/>
    </matches>
</compare-result>

```

**Figure 66: White Box Test Result - XML File**

```

<?xml version="1.0" encoding="UTF-8" ?>
<static-analysis-result>
    <class name="RomanToArabic.cs">
        <gap id="1">
            <CyclomaticComplexity>
                <result pass="true" gapType="statements"/>
            </CyclomaticComplexity>
        </gap>
        <gap id="2"/>
        <gap id="3" />
        <StructureSimilarity>
            <result pass="true">
                <attempAllGaps>yes</attempAllGaps>
                <matched>yes</matched>
                <set id="1" match="true"/>
                <set id="2" match="true"/>
                <set id="3" match="true"/>
            </result>
        </StructureSimilarity>
    </class>
</static-analysis-result>

```

**Figure 67: Static Analysis Result - XML File**

### **6.4.2 Providing Feedback for Teaching Staff**

The ELP system enables teaching staff to monitor students' compilation attempts, the correctness of their attempts, and their programming practices. Information on student compilation attempts is obtained by counting the total number of saved attempts of the student for an exercise stored in the `ExerciseInstanceRevision` table, as described in Chapter 3. For each successfully compiled attempt, the `compiled` field in the table is set to 1. This field is set to `null` if the attempt has not been compiled and to 0 if there are syntax errors in the attempt. As shown in Table 8, the timestamp when the attempt was saved is also recorded in the database. This information is used to provide information on the latest saved attempt. In addition, through the database, teaching staff can also obtain the full history of a student's working progress on an exercise. At the same time, the teacher can also identify students' programming practices and common logic errors in each of the saved attempts.

## **6.5 Summary**

To conclude, this chapter discussed the design and implementation of the three types of feedback that the ELP system provides. The feedback is used by students to facilitate constructivist learning and by teaching staff to help monitor student progress. The provided feedback for students consists of both *verification* and *elaboration information*. The feedback is response-specific and individual to each student's needs.

For students, the system provides customised compilation error messages, and feedback on quality, structure, and correctness of their programs. The customised compilation error message helps students to more quickly achieve a syntactically correct program by making errors more apparent. Feedback on quality and correctness is designed to reinforce the importance of program quality among students and check the correctness of their programs. The feedback is a combination of static and dynamic analyses results as it provides comments on students' programming habits and their program correctness. All generated runtime errors are reported to students. The feedback points out which gaps might contain errors and provides hints on how to fix the problem.

The feedback can be used by teaching staff to monitor students' compilation attempts on an exercise and to gain a better understanding of students' programming practices and their common logic errors.

# **Chapter 7 - ELP Evaluations**

*Evaluation is an educational process, not an end in itself; we learn in order to help our students learn. (Almstrup et al., 1996)*

In the previous chapters, the design and implementation of the ELP system, the program analysis framework and how feedback is provided to students and teaching staff were discussed. This chapter presents evaluation results of the ELP system from both the perspective of students and teachers.

## **7.1 Introduction**

Evaluation is a critical process in education. It is often carried out either to assess the effectiveness of courses as a whole, or to identify the impact of various contextual factors such as the impact of a new teaching and learning tool. For research which introduces technology in education, evaluations often aim to investigate: 1) if the technology improves students' learning experience; and 2) the possible difficulties in using the new technology. Evaluations in this type of research are often difficult because there is involvement of people (Almstrup et al., 1996).

An ITiCSE working group (Almstrup et al., 1996) summarised six groups of approaches to conducting evaluations, namely:

1. Formative vs. summative
2. Quantitative vs. qualitative
3. Field studies vs. laboratory studies
4. Experimental vs. observation
5. Incremental vs. longitudinal
6. Data-driven analysis vs. theory-driven analysis

Further information on these evaluation approaches can be found in Almstrup et al. (1996) and Fritze (2003).

In this research, formal quantitative evaluations are used to produce numerical data for statistical analysis and qualitative evaluations, which obtain descriptive data. The evaluations were used to assess the ELP system and its integrated program analysis framework. The ELP system supports "fill in the gap"

programming exercises to make learning to program much easier and give frequent and formative feedback for students about the quality and correctness of their programs based on the program analysis results. Therefore, the research evaluations aim to assess the students' learning to program experience with the ELP system, and to obtain teaching staff opinions about their experience in using ELP and how the system benefits students. Quantitative evaluations were conducted to reveal the overall opinions of students and staff about the system, while qualitative evaluation results provide more in-depth information to subordinate and clarify the numerical data obtained in quantitative evaluations.

The contribution of this chapter is to discuss evaluations conducted among students and teaching staff to evaluate the ELP system and the program analysis framework together with their results. The chapter is comprised of three sub-sections. The evaluation objectives are discussed in Section 7.2. In Section 7.3, how the evaluations were conducted together with their results are reported, followed by a discussion of the evaluation outcomes in Section 7.4.

## **7.2 Objectives of the Evaluations**

As stated in the first chapter, there are three contributions of this thesis:

1. The innovative and constructive web-based learning environment – the ELP - that supports “fill in the gap” programming exercises to help novice students program successfully from ‘day one’.
2. A configurable and extensible program analysis framework capable of analysing students’ “fill in the gap” programs to ensure the quality, structure, and correctness of the programs.
3. An automated feedback engine that provides students with customised compilation error messages and immediate, constructive and user-friendly feedback on the quality and correctness of their programs based on the analysis results.

The three key aims of the research project are:

1. Make learning to program in high-level OO programming languages like Java and C# easier, since these programming languages contain many abstract concepts;

2. Remove all early hurdles and allow beginning students to program successfully at the earliest stage of their learning;
3. Provide students with a constructive, one-on-one learning environment when they undertake programming exercises.

The ELP system offers students with small “fill in the gap” programming exercises. As discussed in Chapter 3, this type of programming exercise gives a starting point for a programming problem with the code provided which reduces complexity of the programming task and addresses the lower four cognitive levels of Bloom’s taxonomy. It allows students to focus on the problem areas to be solved. Students do their programming exercises through a pedagogical programming editor embedded in a web browser and the compilation process is carried out on the server. Therefore, they do not have to spend their time in learning how to use a programming editor or setting up the compiler, which are common barriers among novice programmers. Students are also given individualised feedback on the quality and correctness of their programs.

Evaluations were carried out among students and teaching staff to assess whether the research project achieves its aims. For students, the objectives of the evaluation can be expressed in four key questions:

1. Does the ELP “fill in the gap” style exercise make learning to program easier?
2. Does the ELP system eliminate the programming barriers in learning to program?
3. Is the ELP system easy to use?
4. Does the analysis feedback support students’ learning?

The main objectives of the evaluation conducted among teaching staff were:

1. Identify whether the ELP system is easy to use;
2. Identify whether it is time consuming to set up exercises on the ELP system;
3. Investigate benefits of the ELP and the program analysis for teaching staff;
4. Identify whether the ELP supports constructivist learning through exercises;
5. Reveal benefits of the ELP and the program analysis framework for students from the teaching staff point of view.

Evaluations of the research project were carried out to assess the long-term impact of the ELP system rather than the immediate impact. The system had been used as a supporting tool for weak students who encountered difficulties in learning to program, or for those who would like to do more practice in their own time.

For teaching staff, the aim of the ELP is for it to be used in conjunction with normal face-to-face classroom teaching. Therefore, the evaluations did not put a strong focus on evaluating whether teaching staff workload was reduced when the ELP was introduced in the classroom, but rather on the ease of use and the pedagogical benefits of the system for students from their point of view.

In the next section, the evaluations conducted among students and staff are described in detail together with their results, while Section 7.4 discusses the results and concludes the chapter.

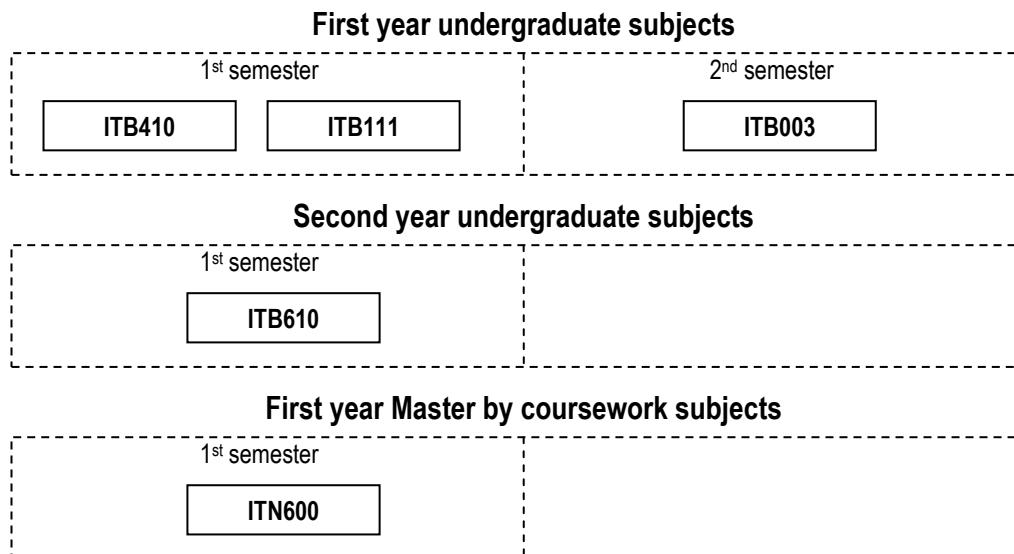
## **7.3 Evaluations and Results**

In order to evaluate the ELP system and the program analysis frameworks against the objectives outlined in the previous section, both quantitative and qualitative data were collected from students and teaching staff during the ELP development. This section consists of three sub-sections. Sub-sections 7.3.1 and 7.3.2 respectively detail how quantitative and qualitative data were collected among students. In Sub-section 7.3.3, qualitative feedback from teaching staff who used the ELP in their classrooms is discussed.

### **7.3.1 Quantitative Evaluations among Students**

The research was divided into three phases and quantitative evaluations were carried out using questionnaire forms at the end of each phase. In stage one, the first version of the ELP system was released without the program analysis framework. In stage two, the ELP system and the static analysis were available, while in stage three, the ELP system, static and dynamic analyses were designed and implemented.

Students in five subjects in the Faculty of Information Technology at QUT were involved in the quantitative evaluations. Those subjects were ITB410, ITB111, ITB003, ITB610, and ITN600. Figure 68 shows the hierarchical structure of those subjects in undergraduate and postgraduate IT courses at QUT together with their objectives and prerequisite.



<b>Unit</b>	<b>Teaching language</b>	<b>Prerequisite</b>	<b>Objectives</b>	<b>Notes</b>
ITB410	Java	None	Introduction to programming	No longer offered
ITB111	Java	None	Introduction to programming	Replaced ITB410 subjects
ITB003	C#	Undertook one programming subject	Developing students' problem-solving skill with advanced OO concepts (e.g. encapsulation, polymorphism, inheritance)	First time offered when evaluation conducted
ITB610	C#	Undertook two programming subjects	Teach student Abstract Data Type (ADT)	Offered for software engineering specialised students
ITN600	C#	None	Introduction to programming	Designed for Master by coursework students

**Figure 68: Units Evaluation Hierarchy**

There are 13 weeks in a semester at QUT. The first evaluation was conducted among ITB410 students at week five of their semester. Similarly, the second evaluation was held in the fifth week for ITB610 students. The third evaluation was carried out in the fourth week for ITB610 students and the sixth week for ITB003 and ITN600 students. This section is divided into three sub-sections which each detail the evaluations conducted at each stage of the research.

### 7.3.1.1 Stage 1 – ELP Only

In this evaluation, 30 first year ITB410 students were tested midway through their semester. These students found themselves having difficulty with Java programming

and some had not even been able to install the JDK compiler on their machines at home.

This was the first time the ELP system was presented to the students; they experienced the ELP in a lab for two hours then filled in an on-line feedback form, shown in Figure 69. Only the ELP system was available at this stage, hence the main emphasis of the evaluation was to discover if the ELP system's support of “fill in the gap” programming exercises and its chosen programming editor design made learning to program in Java easier.

1. The ELP system makes it easier for me to write Java programs
  - A. I agree
  - B. I disagree
  - C. I neither agree nor disagree
2. Using the ELP system helps me to understand Java programs
  - A. I agree
  - B. I disagree
  - C. I neither agree nor disagree
3. I would like to continue to use the ELP for the rest of this subject
  - A. Yes
  - B. No

**Figure 69: First Evaluation Questionnaire**

Even though 30 students participated in the evaluation, only 12 students completed the evaluation form. All the respondents reported that they enjoyed using the ELP and agreed that the system makes compiling and writing programs much easier with “fill in the gap” type exercises and customised compilation error messages. They believed that using the ELP is a good way to learn programming because “fill in the gap” exercises make programming exercises easier and allow them to concentrate on the main parts of the problem. The evaluation result is shown in Table 27.

All 12 students mentioned that they had been currently having problems in learning to program. Two students in the group had never written a Java program as they thought they could not do it. One student in the group had problems in running the Java command line compiler and therefore, the student had never been able to compile a program. With the ELP system, the students felt that they could complete a program, compile and run it easily and therefore, they were more confident to progress to other exercises.

	<b>Agree (Yes)</b>	<b>Disagree (No)</b>	<b>Neither agree nor disagree</b>
Question 1	11	0	1
Question 2	12	0	0
Question 3	12	0	0

**Table 27: First Evaluation Results**

### 7.3.1.2 Stage 2 – ELP System and Static Analysis

This evaluation took place among 46 students at Week Five of the Software Development 3 (ITB610) subject. In this evaluation, both the ELP and static analysis were evaluated. Students were required to do their practical exercises in the first five weeks of the subject using the ELP system. The survey form, shown in Figure 70, was handed out at the end of week five.

1. Have you had any programming experience before taking this subject?  
 Yes       No
2. Please comment on the ELP system.
3. Do you think the program analysis function helps you to learn to program?  
 Yes       No  
 Please specify the reason for your answer
4. Which features of the program analysis did you find helpful? Why?
5. Did you find the output of the program analysis easy to understand?  
 If not please explain why and how it can be improved
6. Please comment on each of the following program analysis features  
 Code Complexity  
 Check Switch Statement  
 Redundant Logic Expression Check  
 Solution Similarity
7. Suggestions for improvement to the program analysis framework

**Figure 70: Second Evaluation Questionnaire**

The evaluation was conducted to measure attainment of the four objectives set out in Section 7.2; however, only the static analysis feedback was evaluated in the fourth objective. Both qualitative and quantitative data were collected in the evaluation with qualitative data obtained through open-ended questions in the survey. Table 28 shows the relationship between the evaluation objectives and the associated questions in the questionnaire form which address each of the objectives. The discussion of this section is divided into two sections: the first section reports on

students' comments about the ELP system and the second section details students' feedback on the static analysis.

Objectives	Associated Questions
1. Do “fill in the gap” style exercises make learning to program easier?	2
2. Does the ELP system eliminate the following barriers in learning to program: complex editor, programming environment installation, compiler installation?	2
3. Is the ELP system easy to use?	2
4. Does the analysis feedback support students' learning?	4, 5, 6

**Table 28: Evaluation Objectives and Associated Second Questionnaire Questions**

### The ELP System

A total of 46 students participated in the evaluation and 35 (more than 71%) gave positive feedback about the ELP system. Most respondents agreed that the ELP is useful in helping them learn to program – for example, a typical comment described it as a “very helpful, friendly environment as opposed to setting up and IDE locally on own PC” - and the web-based user interface of the system was perceived to be the most useful feature because it allows the students to do practical exercises anywhere, anytime since many of them encounter difficulties in using Visual Studio as their IDE. The survey respondents also acknowledged that “fill in the gap” type programming exercises is a good way to teach a “not so good” programmer learn to program. In this regard, the following is a typical comment:

*“I think it’s a very useful system to help refresh student about programming skills. It’s a good way for student to practice programming cause it’s fast and convenient, so lazy student somehow don’t feel too burden doing it. I think currently, the ELP is so far the best solution for “not-so good” student to practice programming cause they don’t have to write whole program.”*

However, survey respondents commented that the ELP should provide syntax highlighting and have better support for different browsers.

### The Static Analysis

Sixty-three percent (63%) of the students responded that the static analysis is useful in providing them feedback about how their solutions compare to the model solution. This helps them think about alternative solutions when doing exercises. They

reported that feedback from these analyses is easy to understand. For example, respondents commented that:

*“The analysis helps to point out which part of our program is un-similar.”*

*“It clarifies some mistakes [an] amateur programmer would have made.”*

*“It suggests a better way to improve the code construct, make you more aware of good coding structure.”*

*“The [static] analysis is good because it helps me to work out all the problems in my code without looking at the model solution.”*

The student respondents agreed that all analyses in the static analysis are useful but solution similarity analysis was considered to be the most useful one. The redundant logic expression check “allows” students to “practise compacting logic expression”. Check code complexity and the program statistics analyses show students if their code is “understandable”, while the check unused parameters and variables “remind” the students about the variables or parameters they have declared but “forget to use”, and help to “tidy up the code”. Solution similarity is considered to be the most useful analysis because it lets students know if they are currently on the right track, points out to them “unnecessary code that they may have added” and show them how “to code efficiently assuming model answer is efficient”. One respondent commented that the function is “*good but also annoying since there are valid solutions which do not pass*”. This is because the analysis only compared the structure of students’ solutions against the structure of one model solution instead of a set of solutions. This single solution restriction was subsequently addressed in a later release of the system.

### **7.3.1.3 Third Evaluation – ELP System, Static and Dynamic Analyses**

The third evaluation was carried out among three subjects: ITB003, ITB610 and ITN600 which all used C# as the teaching language. As discussed earlier, ITB003

was the second software engineering subject for first year IT students. ITB610 was taught in the second year as the third software engineering subject for specialised software engineering undergraduate students. ITN600 was the first introduction to programming for Master by coursework students at QUT. This was a conversion subject for non-IT graduates.

In all three subjects, students used ELP to do their practical work every week in the labs and at home in conjunction with normal classroom tutorials. The questionnaire forms, shown in Figure 71, were handed out to ITB610 students in week four of the semester in lecture time and week six of the other two subjects in lab sessions. Table 29 summarises the association between the objectives stated in Section 7.2 and questions in the survey.

Objectives	Associated Questions
Do “fill in the gap” style exercises make learning to program easier?	4, 5
Does the ELP system eliminate the following barriers in learning to program: complex editor, programming environment installation, compiler installation?	3, 4
Is the ELP system easy to use?	6, 7,8
Does the analysis feedback support students’ learning?	9, 10, 11

**Table 29: Evaluation Objectives and Associated Questions in Third Questionnaire**

1. In which unit are you currently enrolled?  
ITB610                    ITN600                    ITB003
2. In which of the following programming languages have you had experience?  
Java      C#      C      VB      Other      None
3. ELP helps me to start writing programs from ‘day one’.  
Strongly disagree   Disagree   Undecided   Agree   Strongly Agree
4. The ELP system is very useful in helping me learn to program.  
Strongly disagree   Disagree   Undecided   Agree   Strongly Agree  
Please give reasons for your answer
5. Fill-in-the-gap programming exercises make it easier for me to start writing programs.  
Strongly disagree   Disagree   Undecided   Agree   Strongly Agree
6. The interface of ELP is user friendly.  
Strongly disagree   Disagree   Undecided   Agree   Strongly Agree
7. Please specify any other programming editor (i.e. Visual Studio, Eclipse, JCreator, Notepad) that you have experienced.
8. How does ELP compare with those programming editors?
9. The “Check Program” function helps me learn to program.  
Strongly disagree   Disagree   Undecided   Agree   Strongly Agree  
Please give reasons for your answer by checking the appropriate box(es)

<input type="checkbox"/> Have not used it	<input type="checkbox"/> Do not need to use it	<input type="checkbox"/> The function does not work properly (encounter exception)
<input type="checkbox"/> Gives immediate feedback	<input type="checkbox"/> Gives feedback about the correctness of my program	<input type="checkbox"/> Gives feedback about the structure of my program
Other reasons		

10. I understand the “Check Program” feedback.  
Strongly disagree   Disagree   Undecided   Agree   Strongly Agree  
Please give reasons for your answer by checking the appropriate box(es)

<input type="checkbox"/> Have not used it	<input type="checkbox"/> Do not need to use it	<input type="checkbox"/> The function does not work properly (encounter exception)
<input type="checkbox"/> Complicated and hard to understand	<input type="checkbox"/> Clear and simple	
Other reasons		

11. The “Check Program” feedback helps me to correct and improve the quality of my program.  
Strongly disagree   Disagree   Undecided   Agree   Strongly Agree  
Please give reasons for your answer by checking the appropriate box(es)

<input type="checkbox"/> Have not used it	<input type="checkbox"/> Do not need to use it	<input type="checkbox"/> The function does not work properly (encounter exception)
<input type="checkbox"/> Too vague	<input type="checkbox"/> Ok but will be better if have more explanation	<input type="checkbox"/> Good, points out what and where errors are
<input type="checkbox"/> Other reasons		

12. Please add any other feedback on ELP and the “Check Program” function.

**Figure 71: Third Evaluation Questionnaire**

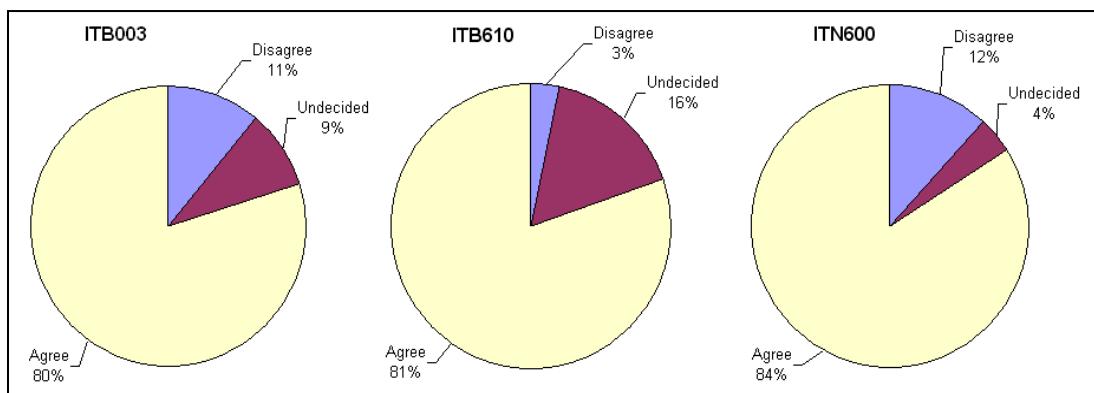
In the semester when the evaluation was conducted, there were 278 students enrolled in the ITB003 subject, 132 students in ITB610 and 45 students in ITN600. A total number of 194 students in three subjects participated in the evaluation. These consisted of 108 ITB003 students, 61 ITB610 students and 25 ITN600 students. Table 30 details the number of students in each subject who participated in the evaluation. In the remainder of this section, evaluation results for each stated objective are reported.

Subjects	Number of Participants
ITB003	108
ITB610	61
ITN600	25

**Table 30: Third Evaluation Participants**

**Objective 1: Do “fill in the gap” style exercises make learning to program easier?**

With reference to Table 29, this objective is addressed by Question 5 in the questionnaire form. All participants completed the question and 156 students (80%) of them agreed that this type of programming exercise makes learning to program easier, while 21 students (11%) of them were undecided and only 17 students disagreed. The detailed results are shown in Figure 72.



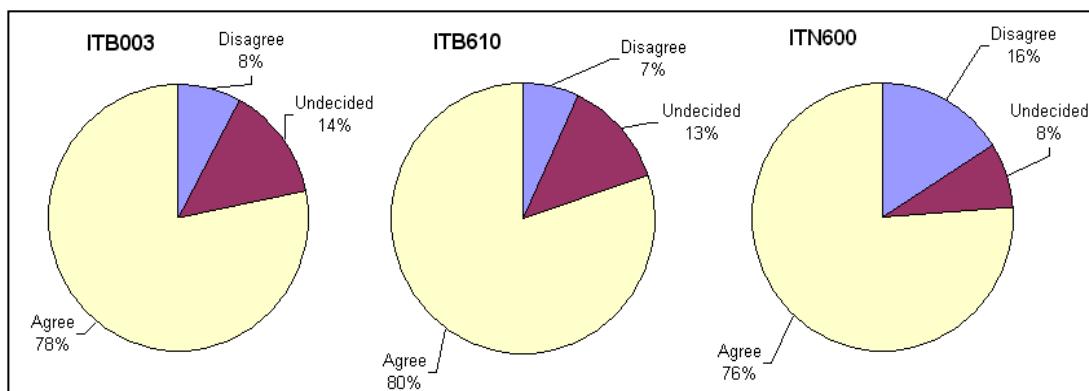
**Figure 72: Student Feedback on the Effectiveness of “Fill in the Gap” Exercises**

As seen from Figure 72, a similar number of students in each subject agreed that “fill in the gap” style exercises make learning to program easier. There were three main reasons stated by students who did not agree that “fill in the gap” style exercises make learning to program easier in Question 5 based on their responses in

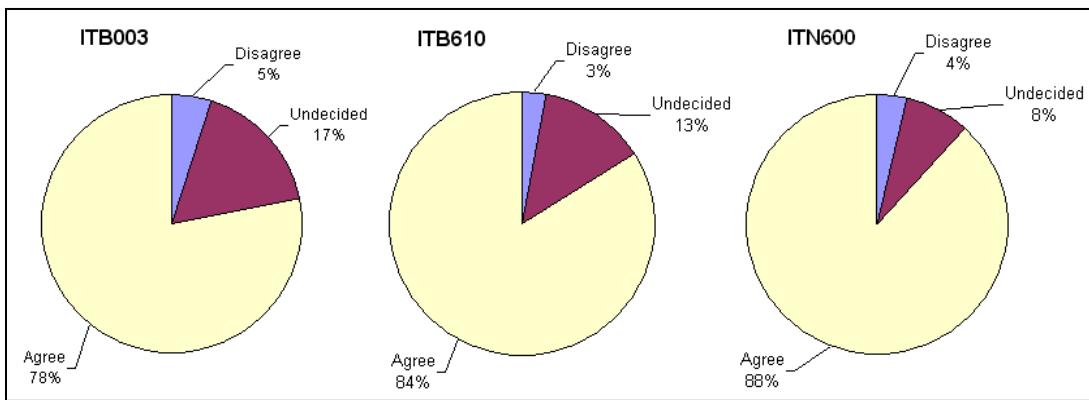
Question 4. First, more advanced students felt that they would learn more by writing a complete program than by only filling in gaps. The second reason was due to the lack of instruction from teaching staff on what should be filled in gaps for exercises; hence, students were confused and felt ‘lost’. Third, students felt that it took more time for them to understand the provided code in order to complete an exercise rather than write a complete program from scratch. Even though the time consuming nature of this aspect of the task is one of the reasons respondents gave for not agreeing that the “fill in the gap” exercises make learning to program easier, it is expectable and can be interpreted as a positive comment. As noted in Chapter 3, this type of programming exercise enforces the lower two levels of Bloom’s taxonomy; it is more effective in developing goals and plans mapping in students’ mental models than a completed working example. Therefore, in order to complete an exercise, it is reasonable that it may take time for students to understand the provided code.

### **Objective 2: Does the ELP system eliminate the programming barriers in learning to program?**

In order to evaluate this objective, the students were asked two questions (Question 3 and 4) in the questionnaire. The first question was if the ELP system helped them to start writing programs from ‘day one’. The second question was if the system was useful in helping them learn to program. Students were required to specify reasons for their answer to the second question. The results are shown in Figure 73 and Figure 74.



**Figure 73: Student Agreement on whether the ELP Helps Them Start Programming from Day One**



**Figure 74: Student Agreement on whether the ELP Helps Them Learn to Program**

It is noted, as can be seen in Figure 73 and Figure 74, that the subject ITN600 has the highest percentage of students who did not agree that the ELP helped them to write programs from ‘day one’; however, they strongly believed that the system was useful in helping them learn to program. This is probably because the majority of students in this subject did not have any previous programming experience; therefore, they might encounter other barriers such as trying to understand the language syntax or learning how to use the computer. Almost the same percentages of students agreed on the usefulness of the ELP system in the other two subjects (ITB003 and ITB610).

Overall, student respondents were satisfied with the ELP. The system enables them to compile the programs and run the programs quickly, and gives them confidence and motivation. Students admitted that the system “*makes them learn to program*”, “*get you working with the code*”, “*give practical experience which makes us understand the theory better*”. As one respondent stated, students can “*try different solutions that may or may not work and you can see where the errors are and easily modify the code*”. Another respondent stated that “*Experience, Interaction, Response*” is what the ELP system is all about. From the students’ point of view, the system is beneficial for two purposes: to teach new programming languages to all students and to revise the knowledge for those who have had programming experience but who may have forgotten it. As an illustration of this point, typical responses include:

*“I think the ELP system is an excellent tool to start programming because it helps people who are unfamiliar or don’t understand the*

*syntax well. It's an unhindered chance to learn C# and programming in general.”*

*“The ELP works better than the traditional way of writing complete, increasingly larger programs, as students learn bits of code at a time, as opposed to fiddling around with keywords and syntax that have not (yet) been covered.”*

Based on comments yielded by the survey, there are five major features through which the ELP system helps students learn to program, as follows:

1. The system is hurdle free. Students do not have to spend time installing a compiler or learning how to use a programming editor. They can start programming, in the words of one student respondent “*from day one without having to worry about downloading and installing large pieces of software*”. The system, as described in other survey comments “*takes away the hassles of setting up the environment*” and enables “*easy to compile and run*” programs. In addition, the system has simple IDE that allows students to focus on the programming task – one respondent commented on the “*very simple IDE, nothing to deal with but the business of coding*”.
2. The system supports “fill in the gap” exercises:
  - a. According to student respondents, providing code in an exercise gives a starting point for a programming problem. One student stated that he often “*has no idea what I am doing so seeing something definitely helps*”.
  - b. Gap-filling exercises help to “separate concepts” from programming language syntax, and allows concepts to be introduced gradually. This type of exercise also narrows down the scopes and emphasises the important syntax that students need to know in a week allowing a quick grasp of language syntax. Learners are, as one respondent observed, “*introduced to the focused part of the language in each exercises*”. Another comment pointed out that it “*narrow down the scope for students to understand, emphasizes the crucial area of a lesson*”. According to student respondents, a gap, “*means things are*

*introduced gradually*”, and “*You learn part of it and you can put it together at the end, a kind of progression*”.

- c. This type of exercise also improves students’ program comprehension skills. As noted by one respondent, the ELP “*helps by making you understand the code that you are given and making you extend on that*”.
  - d. Student respondents commented that this type of exercise gives them confidence and motivation. In this regard, survey comments included: that gap programming exercises “*reduce the number of syntax errors*”; writing a small segment of code makes the programming task “*less daunting*”; the exercise allows “*faster learning*”, and enables beginners to “*achieve a complete program quickly to play with*”.
3. The system is web-based. Student respondents noted that they can “*access exercises from anywhere*” and “*work on exercises in their own time with the solutions and hints are available on-line as well*”.
4. The system provides hints and solutions for each exercise. A majority of students in all three subjects found that making exercise solutions available was a great help for them especially when they had difficulties. This was indicated by comments as follows:

*“It's good. It gives good experience for retain problems (provides quick solution).”*

*“ELP helps you think about the problem but also gives solution so that it sticks in your mind easily.”*

5. The system provides immediate feedback for students through the program analysis framework. Students found that being able to obtain immediate feedback on their program was very useful when they did their practice at home. They were motivated when they received comments about code structure which they had not been aware of and when they were able to test the correctness of their programs without the trouble of setting all the test inputs. This benefit is reflected in survey comments such as:

*“It helps one structure code which I didn't know.”*

*“The ELP provides a reasonably intuitive, seemingly self directed learning opportunity. It presents the users with feedback and check mechanisms.”*

*“It quickly allows me to identify proper coding structure and technologies. It is also like learning from doing and gaining real experience. You can try, learn and even observe the correct code, methods of coding for wider learning. Great.”*

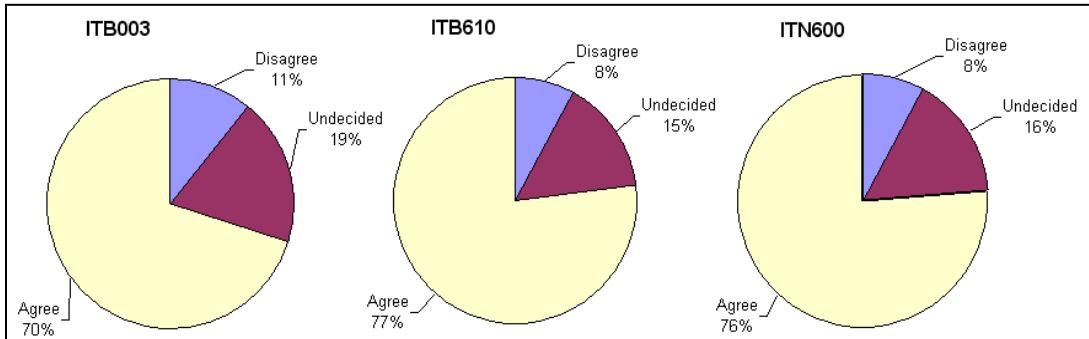
Even though the ELP system has provided substantial assistance to students, there were three additional recommended supports that the system can be further developed to incorporate and make learning to program easier for students. Firstly, students preferred to have assistance on syntax problems. Secondly, additional support materials which explain keywords in a program together with examples of how to use them and how an exercise solution is derived were desirable. Lastly, student respondents indicated that they would like to have a code visualisation tool integrated into the ELP system. Some students also suggested that it would be better if exercise solutions were released after tutorials. This will force them to work on the exercise harder. However, this suggestion can be addressed by lecturers since gaps in exercises are created by them through the ELP authoring tool as discussed in Chapter 3.

### **Objective 3: Is the ELP system easy to use?**

In this objective, the usability of the ELP editor is evaluated by requiring students to comment on the ELP editor applet compared to other IDEs that they have experience with. As shown in Table 29, three questions in the survey (Questions 6, 7 and 8), were used to evaluate this objective. Students were asked if the ELP user interface is user friendly and were required to compare the ELP user interface with other programming editors that they had experience with.

Overall, more than 70% of student respondents agreed that the ELP is easy to use (in Question 6) and stated that Visual Studio, JCreator and Notepad were their commonly used editors. Figure 75 details the percentages of students' agreement on the user-friendly aspect of the ELP system. A majority of student respondents stated that the ELP user interface is much simpler than other IDEs such as Visual Studio,

Eclipse or JCreator but it provides enough functionality for them to learn to program. However, the lack of syntax colouring, IntelliSense and syntax completion features of the ELP editor were identified as its main drawbacks.



**Figure 75: Student Agreement on the Ease of Use of the ELP User Interface**

#### **Objective 4: Does the program analysis feedback support students' learning?**

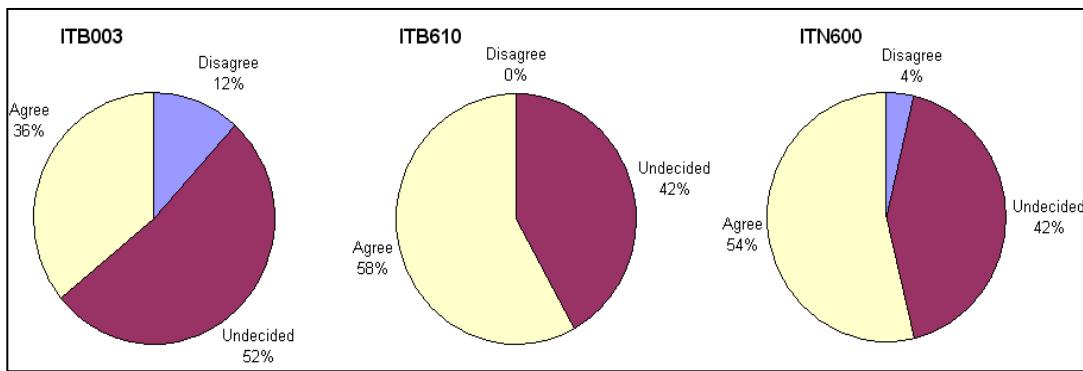
As noted in Table 29, in order to evaluate this objective, students were asked three questions: first, if the “Check Program” function helps them learn to program; second, if the program analysis feedback is understandable; and third, if the feedback based on static and dynamic analyses results helps them to correct and improve the quality of their programs (see Figure 71 for more details of the questions).

Even though 194 students participated in the evaluation, not all students completed Questions 9, 10 and 11 of the questionnaire. Table 31 indicates the number of students who completed those questions.

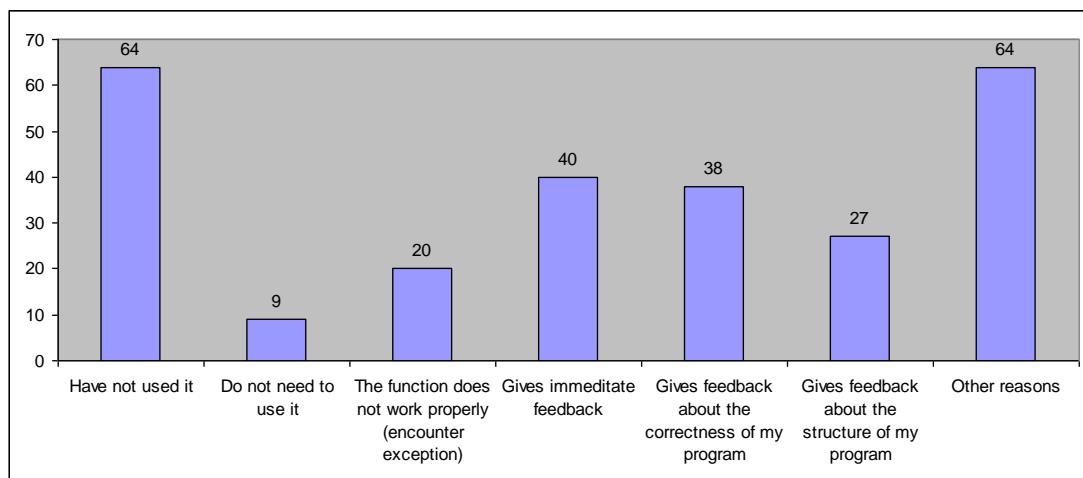
Questions	Number of students who completed
9	183
10	180
11	180

**Table 31: Number of Students who Completed Objective Four Questions**

In Question 9, students were asked if the program analysis framework helps them learn to program together with reasons for their answers. Eighty-one student respondents (or 41%) agreed that the “Check Program” function helps them learn to program; 87 student respondents (or 44%) were undecided and 15% of students disagreed. Figure 76 illustrates the percentage of student agreement in each subject, while Figure 77 shows the number of student distribution for each of the provided reasons on why the program analysis framework helped them learn to program.



**Figure 76: Student Agreement on the Helpfulness of the “Check Program” Function**



**Figure 77: Students Opinions on the “Check Program” Function**

As can be seen in Figure 77, approximately 72% of student respondents agreed that the framework helped them learn to program because the framework gives immediate feedback about the structure and correctness of student programs. Students indicated that receiving immediate feedback about their programs increased their motivation. This was expressed through comments such as: *“I always feel happy to get feedback of my program”*, and *“It tells if you are on the right track”*. Students also agreed that the feedback from the function is friendly and easy to understand. One student commented that the function has *“a user-friendly explanation show me what’s wrong and where the problem is located”*. Most importantly, students agreed that through the feedback, they can learn from their previous mistakes, as expressed in comments as follows:

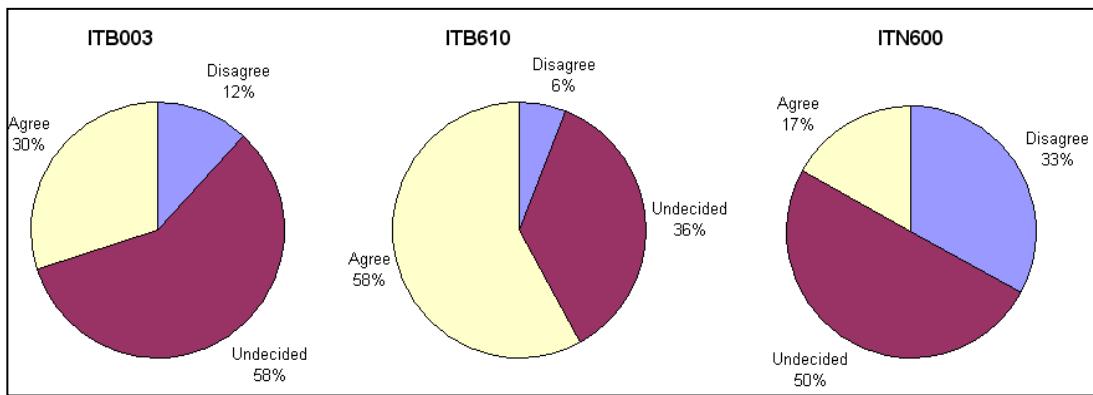
*“It points out areas of poorly constructed or failing code and that I can learn from my mistakes.”*

*“It is an easy way to get comparison against working programs and to debug my own code.”*

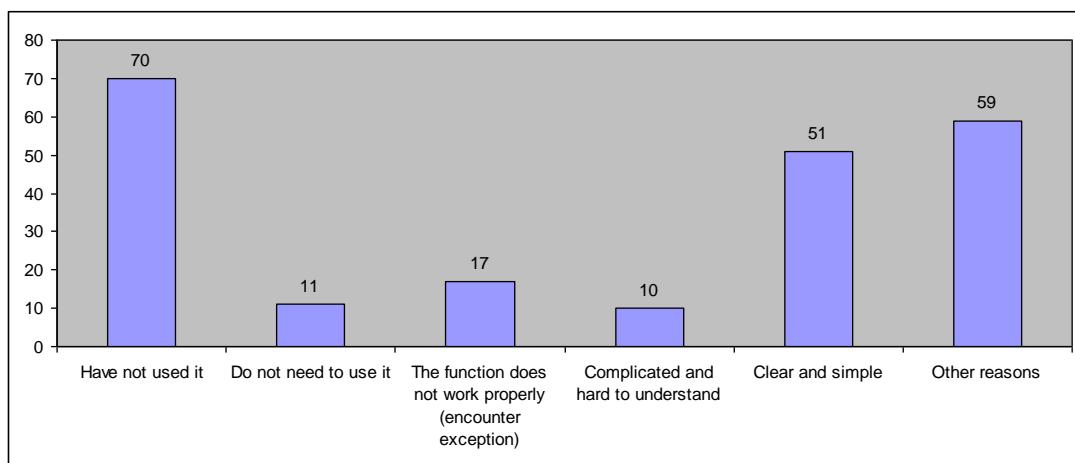
*“Give excellent feedback makes me want to use it as my default PFE.”*

A majority of students who had not used or did not need to use the function did not specify any reason for their answers. Surprisingly, one respondent thought that program quality is not important at the current stage of his/her study when the survey was conducted because he/she was “*not at level to be able to write good programs*” and therefore he/she decided not to try the “Check Program” function. Few students who reported that they encountered an exception when using the function did not describe in which situation exception occurred. One student mentioned that the testing process took a long time when his/her program had infinite loop; this might be because the test time out of the testing process was set to be quite long. There were two main explanations for why some students did not think the function is useful. Firstly, students would like to have assistance in getting a program to compile rather than getting feedback after their programs are successfully compiled. Secondly, some students could not understand the feedback because they did not understand what each analysis does. This is because they did not take notice of the available explanation of each analysis.

In Question 10, generated feedback from the framework was evaluated. Students were asked if they can understand the feedback, and 180 students completed the question. Figure 78 details the percentage of student agreement in each subject, while Figure 79 shows the number of student distribution for each of the provided reasons.



**Figure 78: Student Agreements on the Understandability Level of Program Analysis Feedback**

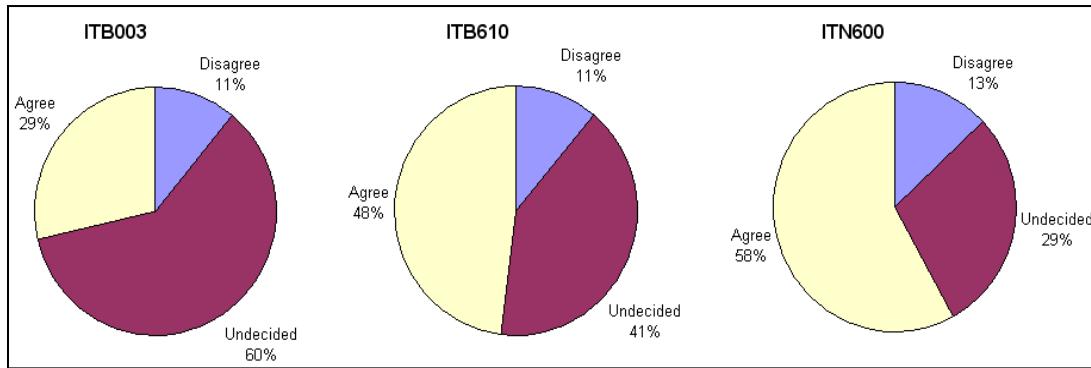


**Figure 79: Student Opinions on the Understandability Level of Program Analysis Feedback**

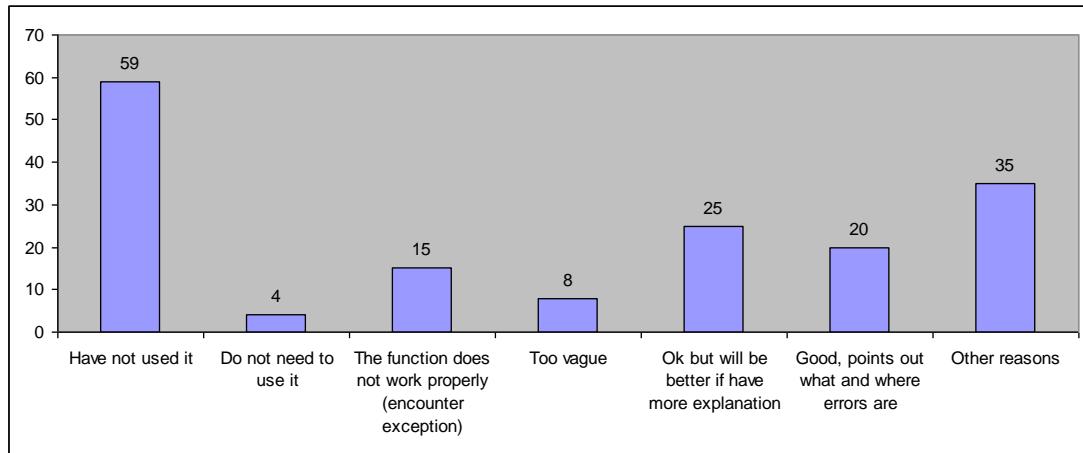
Even though there were many students who did not use the “Check Program” function, the remaining students who used it were satisfied with the provided feedback. Comments included: “*Give excellent feedback*”; “*This is because feedback is provided in the same format. It is easy to understand*”; “[It uses] *clear and simple English*”; “*It [is] written clearly and concisely in a friendly way*”. However, some respondents mentioned that they were not quite sure what to do next after receiving the feedback. A group of students felt that feedback from the framework is quite restricted and a little vague or too cryptic. Some students stated that they could only understand the feedback sometimes, depending on how familiar they were with the problem.

In Question 11, students were asked if the feedback from the function helps them to correct and improve the quality of their programs. The same number of students completed this question as for Questions 9 and 10. Among those students, 69 of them (38%) either agreed or strongly agreed that the feedback helped them to

correct their programs. There was still a large portion of students who had not yet tried it, therefore, they could not decide if the function is helpful; 91 (51%) of the student respondents were undecided; and 11% of students either disagreed or strongly disagreed.



**Figure 80: Student Opinions on the Usefulness of the “Check Program” Function Feedback**



**Figure 81: Student Distribution for Reasons if the Provided Feedback is Useful**

Overall, students agreed that the framework has helped them to improve the quality and correctness of their programs. However, students commented that the static analysis sometimes restricts and the analysis feedback sometimes is a little vague. This is because student programs are only compared with one version of the model solution instead of a set of model solutions. This shortcoming was addressed in a later released version of the system.

#### **Additional Comments regarding the ELP and the “Check Program” Function**

In the last question of the questionnaire, students were asked to provide additional suggestions or comments on the ELP system and the program analysis framework.

From these comments it is apparent that there were three common reasons why students were satisfied with the system. Firstly, the system can be accessed anywhere at anytime. Secondly, they can do hands-on exercises without the trouble of setting up the compiler. Thirdly, they can receive immediate feedback about the quality and correctness of their programs. This conclusion was drawn from a range of comments such as:

*“ELP is a very good system that helps me to start study C#. It makes it easier to learn how to use C#.”*

*“[It is] Good to have immediate feedback rather than having to research for hours sometimes to find where you went wrong.”*

*“The ELP system has been a great help for me. Thank you very much for all the chance to use such a great tool.”*

In addition, there were three improvements to the ELP system recommended by student respondents including: improve the ELP editor to support syntax highlighting, provide blank exercise templates for students to try their code, and allow students to reset gaps in an exercise so that they can re-do the exercise again for revising purposes or when they are completely lost.

In summary, this section has reported on students' comments about the ELP system and the program analysis framework collected through three surveys. Both qualitative and quantitative feedback was obtained and, overall, students found the ELP system and the program analysis framework very helpful. In the next section, focus group results will be reported.

### **7.3.2 Qualitative Evaluations among Students**

Two focus groups were held at the end of stage 3 of the project in which both static and dynamic analyses had been implemented. One focus group was held among ITB003 students and the other was held among ITB610 students.

The first focus group took place at week six of a 13 week semester in one randomly selected ITB003 PASS session. As mentioned earlier, this was the second ‘introduction to programming’ subject offered for first year undergraduate students and C# was used as the teaching language. Fifteen students participated in the focus

group. As noted earlier, PASS students are those who either have academic learning difficulties or who would like to have additional help with their studies. The students were placed into two groups: students who had not tried the “Check Program” function and those who had been using it. There were seven students in the first group and eight students in the second group. Each group was then sub-divided into a group of two or three students. Students worked on the questionnaire, shown in Figure 82, for approximately 15 minutes in groups before starting the discussion.

1. What do you think about the ELP system?
  - 1.1 Do fill-in the gap programming exercises make it easier to start writing programs?
  - 1.2 Does the ELP system help you learn to program?
  - 1.3 Have you encountered any difficulties or problems in using the ELP
  - 1.4 Other comments
2. What do you think about the “Check Program” function of the ELP system?
  - 2.1 Does the function help you learn to program?
  - 2.2 Is the feedback from the function easy to understand and helpful?
  - 2.3 Have you encountered any difficulties or problems in using the function?
  - 2.4 Other comments
3. Please give some suggestions for improving the ELP and the “Check Program” function.
  - 3.1 User interface
  - 3.2 Feedback from the “Check Program” function
  - 3.3 Additional functionality
  - 3.4 Other comments

**Figure 82: Focus Group Questionnaire**

The second focus group was held in week eight of a 13 week semester among nine volunteer ITB610 students. All participants had been using the “Check Program” function. Similar to the first focus group, students were also placed into groups of two or three students and worked on the questionnaire first before starting the discussion. In total, there were three groups and each group had three students.

Overall, students in both subjects found the ELP system and the “Check Program” function were useful.

ITB003 students agreed that the ELP system is extremely useful. It helps them in learning to program, especially when learning a new programming language. The system gives students a starting point with “fill in the gap” exercises allowing them to concentrate on important sections of an exercise. One student stated that this type of exercises helps them to develop their problem-solving skills. However, ELP exercises in this subject were designed with quite limited instructions for gaps, meaning that some students could not understand the purpose of the entire exercise

and felt lost. Some students felt that they were quite restricted since they were only able to edit the gap code in exercises. Additionally, students would like to have more help and more explanation about the solutions of exercises and how they were achieved.

Regarding the “Check Program” function, students who tried it found that the function is helpful. Students who did not use the function were asked for reasons why. Some students did not know what the function does because they did not attend lab sessions, while others reported that they encountered an exception the first time they tried it and therefore they did not try it again. One student mentioned that it required too much effort to save the test driver on his machine.

There were two accessibility problems that students encountered including: ELP applet does not load on the Safari browser on a Mac operating system even though JRE is installed and the ELP applet takes a long time to load through a dial-up connection.

In terms of functionality, students recommended that the ELP editor applet should support colour coding; the ELP system should provide function to reset gaps in exercises and allow students to work with each other collaboratively on an exercise. Further, students would like to have additional help in compiling a program.

Unlike the ITB003 participants, all ITB610 students who took part in the focus group used the “Check Program” function. The students said they have had a great experience in using the ELP system. One student said, “*The ELP system is awesome, it helps me learn a lot*”. Only one student among those participants has had Visual Studio installed on his machine while the remainder had encountered difficulties when installing it. The students thought the ELP system is useful especially with multiple classes exercises because they do not have to spend time to set up all classes in the exercise in the right location and the system CLASSPATH in order to compile them.

Students in both focus groups also agreed that “fill in the gap” exercises is a good way to start learning a new programming language, especially when they are having difficulties in setting up Visual Studio on their own machines. However, students would like to have larger gap exercises and they also would like to have assignments set up in the ELP system. Exercises in the ELP system can be used for assessment purposes and this is a matter for teaching staff.

Students provided positive feedback about the “Check Program” function and the analysis feedback. They all agreed that the feedback is good and easy to understand. One student mentioned that she used it with all the exercises. A student reported that he missed one-third of the feedback because he did not notice that he has to scroll the scroll bar down in the feedback panel. Additionally, students indicated that they would like to have more information about the testing process such as what are the test inputs, what are the expected outputs, and what are their program outputs.

Similar to ITB003 students, ITB610 students also would like to have coloured coding in the ELP editor applet, more feedback about program structure, the ability to reset gaps in exercises, and links to additional documentation such as APIs.

### **7.3.3 Qualitative Evaluations among Staff**

Throughout the development of the project, the system had been continually evaluated with teaching staff verbally and their feedback used to improve the system. Questionnaires were handed out to teaching staff and tutors when the ELP system and both static and dynamic analyses were available. The questionnaire is shown in Figure 83, and Table 32 shows the relationship between questions in the questionnaire form and the objectives which were stated earlier. As shown in Figure 83, the majority of questions not only require teaching staff to state their opinions but also invite them to elaborate on their responses. This style of question provides both quantitative and rich qualitative information. Five staff participated in this evaluation namely: lecturers of the ITB610, ITB003 and ITN600 subjects together with one ITB610 tutor and one ITB003 tutor. In ITB600, the class size was relatively small in the semester in which the evaluation was conducted; hence, the lecturer was also the sole tutor for the subject.

Objectives	Associated Questions
Identify if the ELP system is easy to use	3
Is it time consuming to set up exercises on the ELP system	6
The benefits of ELP and the program analysis for teaching staff	1,2, 5,8,11
Does ELP support constructivist learning through exercises	4
The benefits of ELP and the program analysis framework for students from teaching staff point of view	9,10

**Table 32: Relationship between Teaching Staff Evaluation Objectives and Questions in the Questionnaire Form**

## **ELP**

1. In your experience, does the ELP system help you to teach students how to program?  
 Yes       No
2. Have you used fill in the gap exercises in your subject before the ELP system was developed.  
 Yes       No  
If so is the ELP an improvement over your previous process for doing this?  
Please specify the reason for your answer
3. The ELP system is easy to use.  

Strongly disagree	Disagree	Undecided	Agree	Strongly Agree
4. ELP supports constructive learning through programming exercises.  

Strongly disagree	Disagree	Undecided	Agree	Strongly Agree

  
Please specify the reason for your answer
5. The ELP system has increased the number of students doing hands-on exercises in your subjects prior the system being used.  

Strongly disagree	Disagree	Undecided	Agree	Strongly Agree
6. It is quick and easy to set up exercises on the ELP system.  

Strongly disagree	Disagree	Undecided	Agree	Strongly Agree

  
Please specify the reason for your answer
7. Do you plan on using ELP again in your subjects?  
 Yes       No  
Please specify the reason for your answer

## **“Check Program” Function**

8. The “Check Program” function is useful for you.  

Strongly disagree	Disagree	Undecided	Agree	Strongly Agree

  
Please specify the reason for your answer
9. The feedback from the “Check Program” function is easy to understand and helpful for students.  

Strongly disagree	Disagree	Undecided	Agree	Strongly Agree

  
Please specify the reason for your answer
10. Please specify any other benefits for students from the ELP and the “Check Program” function.
11. What do you think the key benefits of the ELP and the “Check Program” function for you are?

**Figure 83: Staff Questionnaire Form**

Overall, all teaching staff had a positive experience with the ELP system and the “Check Program” function. They all agreed that the system increases the

students' learning experience and helps them in the teaching task. They all strongly confirmed that they would use the system again in the future because there had been an increasing number of students undertaking practical exercises in their subjects. The system provides a platform for students to experiment with a programming problem. All teaching staff agreed that the system is easy to use and that it is easy to set up exercises through the web-authoring tool. More importantly, teaching staff can upload existing exercises to the system and only have to create gaps in the exercises. The remainder of this section will report on the evaluation results for each of the objectives set out in Section 7.2.

### **7.3.3.1 The Benefits of ELP and the Program Analysis Framework for Teaching Staff**

Overall, all lecturers and tutors agreed that the ELP system and the program analysis framework were beneficial for them. This is because the system provides a platform for instructors to give students "fill in the gap" programming exercises. This type of exercise had been previously used in the subjects in pen and paper or electronically, but teaching staff did not have a way to prevent students from editing the provided code. In addition, storing all students' attempts for an exercise in a centralised server enables teaching staff to monitor students' learning progress and perform analyses on the common mistakes and programming practices.

The program analysis framework reduced the workload for instructors because students can work on exercises in their own time and receive analysis feedback. Teaching staff consultation time had not been crowded with students asking straightforward questions. However, for the ITN600 subject, the lecturer could not decide if the function was useful for his class because most of the students were too novice. They encountered many difficulties in achieving a compilable program. Therefore, the "Check Program" function was too advanced for the majority of his students to use extensively, since the framework only analyses a compilable program.

Teaching staff in ITB610 and ITN600 confirmed that the system increased the number of students doing hands-on exercises in their subjects. With ITB003 subjects, however, due to the new teaching model adopted in the subject in which normal classroom tutorials were replaced by practical classes, the number of students who did hands-on exercises increased regardless of which IDEs are used.

### **7.3.3.2 Does ELP Support Constructivist Learning Through Exercises?**

All teaching staff agreed that the ELP system supports constructivist learning through programming exercises for three reasons. Firstly, the ELP system supports “fill in the gap” exercises allowing students to concentrate on parts of the program that they are learning about. Therefore, students can delve into learning about simple concepts much quicker and leave the more advanced concepts for later. Students often have an enormous disinterest in programming and ignore any difficulties that they have to face; hence, small exercises that allow students to interact are good ways to engage and motivate them. Secondly, it is easy to control the gap areas to gradually introduce new knowledge embedded in non-editable regions which contain material the students are already familiar with. Thirdly, the system provides a standard environment in which students can edit, compile, run and test their programs.

### **7.3.3.3 The Benefits of ELP and the Program Analysis Framework for Students from Teaching Staff Point of View**

All teaching staff agreed that the feedback from the “Check Program” function is easy to understand and beneficial for the student learning process. It shows the students exactly what can be fixed and students can receive instant feedback without having to wait for a tutorial or consultation time. More importantly, it makes students aware that a program which compiles and runs is not necessarily a good and correct one. However, all teaching staff suggested that if the feedback from the framework can be improved to reveal more testing information, such as test cases and test classes, rather than just the test case name, this would encourage students to think more on a formal test plan and therefore increase their awareness of the testing process. This recommendation has been addressed in a later release of the system however it was yet to be evaluated at the time the thesis is written.

## **7.4 Discussion of Results and Summary**

In summary, this chapter has discussed the evaluation results of the ELP system and the “Check Program” function from both the perspective of students and teachers. Three surveys and two focus groups were conducted with a total of 295 students participating. The surveys were conducted at various stages. The first survey was

held when the ELP system was first developed. The second survey was organised at the completion of the static analysis development. The third survey and the two focus groups were conducted when both static and dynamic analyses were implemented. Throughout the development process, the system had been continuously evaluated and improved according to the research team's communications with teaching staff. One formal survey was conducted with five teaching staff at the end of stage three of the project. Table 33 summarises the timeline and the number of participants of all surveys and focus groups conducted to evaluate the ELP and the program analysis framework discussed in the chapter.

Evaluations Students/staff	Timeline			Number of participants			
	Stage 1	Stage 2	Stage 3	ITB410	ITN600	ITB003	ITB610
First	✓			30			
Second		✓					46
Third			✓		25	108	61
Students Focus group			✓			15	10
Staff			✓		1	2	2

**Table 33: Number of Students and Staff Participating in the Evaluations**

Overall, the results show that this research achieves all of its objectives. For students, the ELP system makes learning to program easier with gap-filling type exercises and allows students to successfully write programs at the earliest stage of their learning. The system is easy to use. It facilitates constructivist learning and provides a one-on-one learning environment through exercises and various feedback mechanisms. The feedback includes customised compilation error messages and program analysis feedback. The feedback is formative, individual, constructive, and helpful for students to correct mistakes and improve the quality of their programs. Teaching staff find it easy to set up exercises and analyses for the ELP. The system has been used in several units in the Faculty of IT at QUT for the last four years and continues to be used at the time of writing.

Through the evaluation results, the following points can be concluded:

**From the students' perspective:**

1. “Fill in the gap” programming exercises have been of great benefit to students who are new to programming or those who are learning a new programming language.
2. The ELP system allows students to write programs from ‘day one’ by eliminating all the unnecessary hurdles, such as installing a compiler or learning how to use a programming environment. Exercises are successfully given to students in the first week of the semester. In addition, the system increases students’ motivation.
3. The ELP system is embedded in a web browser which allows students to access and do their practical exercises anywhere, at anytime.
4. The “Check Program” function helps students learn to program by making the ELP system more interactive and providing immediate friendly feedback for students about the quality and correctness of their program.

**From the teaching staff perspective:**

Through qualitative analysis among teaching staff, it was ascertained that they all agreed that the ELP system and the program analysis framework have been of great benefit to students because:

1. The system facilitates constructivist learning by supporting “fill in the gap” exercises. This type of programming exercise allows teaching staff to gradually introduce new concepts to students from the early stage of their learning process.
2. The system provides a standard environment for students to experiment with programming exercises.
3. It increases the number of students doing their practical exercises.
4. It provides immediate feedback for students about the quality and correctness of their programs. This enables students to focus more on the quality of their programs, rather than only achieve compilable or correct programs.
5. The system is easy to use as teachers can easily set up exercises and analysis. In order to set up an exercise, instructors can either create a new exercise through the ELP authoring tool (as discussed in Chapter 3), or upload existing

exercises and create gaps. With the program analysis framework, teaching staff only have to provide all test cases and their associated test inputs.

The evaluation results reveal three potential improvements to the ELP system, as suggested by students and teaching staff. Firstly, the ELP editor applet should support syntax colouring and allow cut-and-paste from other programming editors to the applet. Secondly, a function to reset gaps in an exercise should be provided. Thirdly, better support is desirable for browsers in different platforms such as FireFox on Mac machines.

# **Chapter 8 - Conclusions and Further Work**

This chapter summarises the contributions of the thesis and discusses areas of the research suitable for further investigation.

## **8.1 Summary of Contributions**

There are three contributions of the thesis. First is the web-based learning environment, the ELP, which supports “fill in the gap” programming exercises to help students program successfully at the early stage of the learning process. Second is a program analysis framework (integrated with the ELP) which allows students to undertake programming exercises and receive immediate feedback on quality, structure and correctness of their programs. The key novel aspects of the framework are its capability of analysing “fill in the gap” programming exercises and conducting both quantitative and qualitative analyses. Third is an automated feedback engine which gives customised compilation error messages and constructive feedback on the quality and correctness of students’ programs based on the program analysis. It also provides information for instructors to monitor students’ progress.

Overall, the thesis was divided into six parts.

**Part 1:** Chapter 2 overviewed the scaffolding teaching approach, constructivism theory and Bloom’s taxonomy; followed by reviewing the current state of the art and future directions of research and applications in computer science education designed to help students and instructors in teaching and learning to program. The chapter also detailed the design and implementation of existing tools which have great impact on the research.

**Part 2:** Chapter 3 presented the innovative web-based “fill in the gap” programming environment, the ELP, designed to teach beginning students to program in Java and C#. The chapter discussed common novice programmer barriers in learning to program in general and high-level OO languages. This discussion was followed by presenting the environment and elaborating upon how it addresses novices’ common difficulties.

**Part 3:** Chapter 4 presented the static analysis component of the program analysis framework designed to assess the quality of students’ “fill in the gap” Java and C# programming exercises. The chapter reviewed static analysis techniques and

cyclomatic complexity metrics which are commonly used in computer science education applications; followed by a discussion on the design and implementation of the static analysis.

**Part 4:** Chapter 5 presented the dynamic analysis component of the program analysis framework designed to verify the correctness of students' "fill in the gap" Java and C# programming exercises. The chapter reviewed black box and white box testing strategies, as well as discussing how these techniques are applied to analyse students' "fill in the gap" programs.

**Part 5:** Chapter 6 introduced the automated feedback engine which provides customised compilation error messages with timely, formative and constructive feedback on the quality and correctness of students' programs based on the analyses results. The feedback also provides useful information for teaching staff about students' performance and their common mistakes. The chapter overviewed existing methods on providing effective feedback in computer-based instruction before discussing the design and implementation of the feedback engine.

**Part 6:** Chapter 7 reported on results from the qualitative and quantitative evaluation of the research. Surveys and focus groups were conducted among students to evaluate four objectives which reflected the research aims including:

1. Do "fill in the gap" style exercises make learning to program easier?
2. Does the ELP system eliminate novices' common barriers in setting up a programming environment and installing the compiler?
3. Is the ELP system easy to use?
4. Does the analysis feedback support students' learning?

A survey was also conducted among teaching staff to:

1. Identify if the ELP system is easy to use from their perspective
2. Identify if it is time consuming to set up exercises on the system
3. Investigate the benefit of the ELP system and the program analysis framework
4. Identify if the ELP supports constructivist learning through exercises
5. Reveal benefits of the ELP and the program analysis framework for students from the teaching staff point of view.

Details of the questionnaires together with results are discussed in the chapter.

The remainder of this section is divided into four sub-sections; each section highlights the novel aspects and the key design for each of the research contributions.

### **8.1.1 The Environment for Learning to Program**

The ELP system is designed to help novices in learning to program using Java and C# and progressively develop three attributes of experts, including knowledge, strategies and models. The system has four key characteristics: (1) it supports “fill in the gap” exercises, (2) it is web-based, (3) it provides timely and quality feedback, and (4) it is exercise centric. The system has three views: ‘student view’, designed to support students’ learning; ‘lecturer view’, designed to allow teaching staff to manage exercises and monitor students’ progress; and ‘tutor view’ to enable tutors to communicate and monitor students’ progress in their tutorial classes.

Through the four key characteristics, the system facilitates constructivist learning. Students’ learning is scaffolded through the use of “fill in the gap” programming exercises which are accessible from the web. Teaching staff can make use of Bloom’s taxonomy to design their exercises so that new programming concepts are introduced incrementally. All learning materials can also be made available through the web to enable student access anytime and anywhere.

“Fill in the gap” style exercises can help to develop students’ program comprehension, code generation and problem-solving skills. This type of exercise also reduces the complexity of a programming task; hence, students are motivated and engaged in the learning process more actively.

The web-based user interface makes the system easier to use. This interface allows students to start programming from ‘day one’ without the need for setting up and learning how to use a programming environment. This interface also increases the level of accessibility and flexibility of the system; thus, allowing smooth integration between learning materials and practical exercises.

Students are provided with customised compilation error messages and timely feedback on the quality and correctness of their programs. Therefore they will not feel lost and de-motivated when encountering difficulties. Customised compilation error messages make the errors more apparent and more easily fixed. Students can learn from and reflect on their mistakes using the immediate feedback on quality and correctness of their programs. The system is exercise centric as each exercise

comprises all of the necessary learning notes, additional resources, and practical exercises. The system also supports a customisable editor for teaching staff to author the exercises and analyse students' performance for each exercise in the system.

### 8.1.2 The Static Analysis

The static analysis is introduced to assess the quality of students' "fill in the gap" Java and C# programs. It aims to address software quality issues among novice programmers by providing immediate feedback on the quality and structure of their programs while they do their practice. This will reinforce students' mistakes in the context of their current exercise so that they would not make those mistakes again in subsequent exercises.

The novelty aspects of the static analysis are its capability to analyse "fill in the gap" programming exercises; conduct both quantitative and qualitative analysis; and its extensibility and configurability. The analysis consists of two complementary groups of analyses: SE metrics (quantitative analyses) and structural similarity (qualitative analysis). SE metrics analyses are designed to quantitatively evaluate the complexity of a program and identify poor programming practices. There are eight functions available in this group, including: 1) *program statistics*, 2) *check cyclomatic complexity*, 3) *check unused parameters*, 4) *check unused variables*, 5) *redundant logic expression check*, 6) *check magic numbers*, 7) *check methods and variable access modifiers*; and 8) *check switch statements*. Structural similarity analysis verifies the results of SE metrics analyses by identifying high complexity and poor code areas in the program. It is done by comparing the high-level algorithmic structure of a student solution with that of the model solution.

Only students' gap codes of compilable programs are analysed by the static analysis. Even though gaps can be any segments of a program in an ELP exercise, gaps need to be well-formed in order to be analysed by the analysis. There are seven well-formed gap types: expression, block, method(s), class, import(s), declaration(s) and properties. All well-formed gap types are supported by the current implementation of the analysis apart from expression gaps.

The static analysis is flexible, configurable and extensible. All analyses are provided as functions so that additional analyses can be added in easily. Analyses in the framework are fully configurable by teaching staff. They can be set at gap level or class level depending on the objective of an exercise.

### **8.1.3 The Dynamic Analysis**

The dynamic analysis is designed to verify the correctness of students’ “fill in the gap” Java and C# programming exercises. The analysis provides individualised immediate feedback to students and assists teaching staff in the marking task.

Both black box and white box testing are carried out. The testing process is carried out on students’ machines or any third party machine while the process of analysing outputs is conducted on the ELP server. This architecture prevents the execution of malicious code and reduces the load on the ELP server. Black box testing is carried out by executing students’ programs through a set of test data. The program outputs are captured and sent back to the server to check for correctness. In black box tests, print markers are inserted at the start and end of each gap to separate gap outputs from the provided code outputs. With white box testing, gap codes are tested separately one at a time. Each undergone testing gap is inserted into the exercise template while gap solutions are occupied by the remaining gaps to create the mixed program. The outputs of tested gaps are compared with the outputs produced by the corresponding gap solution to verify correctness.

Black box and white box testing results are subordinate to each other. Black box testing identifies which gap does not produce the correct output. However, due to the dependency among gaps, incorrect outputs might be caused by another gap. Therefore, testing one gap at a time in white box testing helps to identify gaps which might have logical errors.

Similar to the static analysis, the key characteristics of the dynamic analysis are its capability to only test gap codes which are syntactically correct, well-formed, configurable and extensible. Teaching staff can configure the output matching process in black box testing through a set of provided filters and normalisers. These filters and normalisers are dynamically loaded at run time; hence additional filters and normalisers can be added in easily.

The analysis currently supports console and OO programming exercises. Extension of the framework to check the correctness of GUI and FileIO programs is discussed in Section 8.2.3.

### **8.1.4 The Feedback Engine**

The automated feedback engine is designed to provide various types of feedback for students and teaching staff. The feedback for students consists of both *verification*

and *elaboration*. The feedback is response-specific and individual to each student's needs. For students, the engine provides customised compilation error messages, as well as feedback on quality, structure and correctness based on the static and dynamic analysis results. Feedback on quality and correctness is comprised of: 1) comments on students' programming habits; 2) students' program structure based on the static analysis framework; 3) all conducted tests in the dynamic analysis together with their results; and 4) any generated runtime errors. Feedback points out gaps that might have high complexity areas, poor programming practices, incorrect structure, or runtime errors together with hints on how to improve the solution.

Using the feedback, teaching staff can monitor students' compilation attempts for an exercise, their programming practices and common logic errors so that additional help can be provided for students to more quickly achieve a compilable program. Based on the information obtained about students' programming practice and their common logic errors, teaching staff can adjust their teaching material to address those issues.

In summary, the research presented the constructive web-based programming environment, the ELP, which supports: 1) Java and C# "fill in the gap" programming exercises; 2) an integrated program analysis framework comprised of static and dynamic analysis components which assesses students' programs for quality and correctness; and 3) an automated feedback engine which provides immediate feedback for students about their programs and for teaching staff on students' performance. The next section will discuss some avenues for further work arising from the research.

## **8.2 Further Work**

This section discusses areas for further research and development based on the contributions of the thesis. The section is structured into four sub-sections. Section 8.2.1 discusses further extensions on the ELP system while Sections 8.2.2, 8.2.3 and 8.2.4 recommend further work on the static analysis, dynamic analysis and the feedback engine respectively.

### **8.2.1 The ELP System**

Collaborative learning and adaptive exercises are two important extensions for the ELP system.

### **8.2.1.1 Collaborative Learning**

Recent research has identified the benefits of collaborative learning especially in introductory programming courses (DeClue, 2003; GILD, 2005; McDowell et al., 2003; Nagappan et al., 2003; Preston, 2005; Wilson et al., 1993). When students are working together on a programming problem, they more often produce higher quality programs in less time. Students have a better understanding of the programming process and enjoy themselves more. Research also has shown that students perform better on exams and course completion rates are increased when they work together during semester (McDowell et al., 2002; McDowell et al., 2003; Nagappan et al., 2003).

Overall, tools for collaborative programming can be placed into two groups: synchronous and asynchronous. With synchronous tools, students can do their programming exercise and discuss with each other in real time while with asynchronous tools, activities are not carried out in real time (e.g. email).

There is great potential for the ELP system to support asynchronous or synchronous activities among students. This can be done by integrating a communication component into the system so that students can discuss with other students who are currently working on the same program. Communication among students for collaboration can be synchronous, such as in real time chatting, or asynchronous, such as using email or forums. Additionally, a collaborative programming editor can be developed to support students to code on the same exercises.

The key challenge in supporting synchronous communication for collaboration is to allow students to converse in the IDE in which they are working, rather than in a different application (Cubranic and Storey, 2005). For example, it is quite distracting when students discuss changes to a Word document in a chat window while at the same time trying to view the actual document in a separate application (Cubranic and Storey, 2005). Therefore, it is important to choose the appropriate collaborative IDE to support the desired communication mode among collaborators. Most collaborative editors today are developed based on the concurrency control algorithm introduced by Ellis and Gibbs (1989) or one of its many variants. Some collaborative tools which have been developed are: GREWPTool (Taneva et al., 2003), NetEdit (Zafer, 2001), GILD (Cubranic and Storey, 2005; GILD, 2005) and Jazz (Cheng et al., 2003).

### **8.2.1.2 Adaptive Exercises**

Adaptive learning tools are systems which are capable of monitoring students' activities, interpreting those activities to understand students' preferences and requirements, and using that information to facilitate learning (Paramythis and Loidl-Reisinger, 2004). Adaptive learning environments have promised a big improvement to students' learning since they can be individualised. There are four types of adaptive learning environments: adaptive interaction, adaptive course delivery, content discovery and assembly, and adaptive collaboration support (Paramythis and Loidl-Reisinger, 2004); Table 34 shows each category definition as stated in (Paramythis and Loidl-Reisinger, 2004).

<b>Categories</b>	<b>Description</b>
Adaptive Interaction	Take place at system's user interface level to make user interaction with the system more convenient, learning content unchanged
Adaptive Course Delivery	Courses are tailored to individual learner's characteristics or requirements
Content Discovery and Assembly	Monitoring users' progress together with using an adaptive model to provide adaptation
Adaptive Collaboration Support	Capture adaptive support in learning processes that involve communications between multiple persons and potentially collaboration towards common objectives

**Table 34: Adaptive Learning Environment Categories**

Adaptive interaction and adaptive course delivery can be easily integrated into the ELP system. Adaptive interaction can be built into the system by supporting a configurable editor through which customised user settings are remembered. Adaptive course delivery can be implemented by classifying exercises on the system based on topics or their difficulty levels. Fischer (2001) presented an approach to sequencing course material using IEEE learning object metadata. Electronic book is the common form of adaptive course delivery which makes use of metadata. Martínez-Unanue and his colleagues (2002) reviewed a set of recent electronic books for programming education and predicted the future research in this area.

### **8.2.2 The Static Analysis**

With the static analysis, there are two main areas for further research and investigation. The first improvement is the matching process between students' solutions and model solutions. Secondly, the analysis can be extended to check

students' coding style such as good variables names, comments in their code and program indentation.

The first improvement can be implemented by supporting multiple model solutions and allowing teaching staff to configure the matching and the normalisation process. Supporting multiple model solutions will provide greater flexibility in algorithm variations. This will require the use of various semantic different algorithms to compare the structure of students' solutions and a model solution. Examples of some semantically different algorithms are set out in Horwitz (1990). With a configurable matching process, teaching staff can judge how closely the structure of students' programs compares with the model solution while configurable normalisation allow teaching staff to enforce the invocation of particular method in gaps.

Style and good naming convention checks of programs can be developed and integrated into the framework by implementing the common StaticAnalysis interface. This extension will ensure students' programs are more readable.

### **8.2.3 The Dynamic Analysis**

As discussed in Chapter 5, there are four categories of programming exercises in introductory computer science courses: console IO, OO, File IO and GUI programs. The dynamic analysis described in this thesis currently supports testing console and OO programs - two major types of exercise that novices are given in the course. In this section, methods which can be used to extend the dynamic analysis to test file IO and GUI exercises are discussed.

#### **8.2.3.1 Testing File Input Output Programming Exercises**

There are three possible operations that a file IO exercise may perform: read only operation, write only operation or both read and write operations. For each operation, there are two possible cases. The first case is teaching staff having some information about the source or destination; this information can be either the file names or the file names and their locations. The second case is where both source and destination files are known by teaching staff.

One of three approaches can be used in the framework to test file IO programming exercises. These approaches are: source code rewriting, binary rewriting and using a customised file IO library package.

In the first approach, locations of the source and/or destination files in the program source are changed to read and write to sources and destination files known by teaching staff. With the binary rewriting technique, the binary code of a class is modified instead of the program source. This would require a third party library such as Java Programming Assistant (Javassist, 2005) or Byte Code Engineering Library (BCL, 2003) for Java programming exercises and Program Executable-Reader/Writer-Application Programming Interface (PERWAPI, 2005) for C# exercises.

The third approach involves developing customised file IO streams used by students in their file IO programming exercises. The customised library is implemented to allow teaching staff to specify the location of source and destination for testing.

### 8.2.3.2 Testing Graphical User Interface Exercises

Testing GUI function can be built into the analysis using reflection or customising an existing open source GUI testing. Table 35 shows some open source GUI testing tools in both Java and C#, as appeared in Tejas Software Consulting (2003) that can be investigated for integration into the framework.

Tools	Platform	Notes
Abbot (Wall, 2002)	Java	GUI test library, with some object-based recording capabilities
AutoIt (AutoIt, 1999)	Windows	Scripting language for analogue GUI automation, also available as an ActiveX control and a DLL
Jacareto (Spannagel, 2005)	Java	Capture/replay tool
Jemmy (NetBeans, 2002)	Java	GUI test library for Java Swing/AWT
jfcUnit (Caswell et al., 2003)	Java	Unit extension for testing Java Swing applications, with event recording
Pounder (Pekar, 2002)	Java	Capture/replay tool
SAFS (Nagle, 2002)	Windows	Keyword-driven test framework for Rational Robot and WinRunner
TestNow (Automation Junkies, 2002)	Windows	Add-on library for Visual Test 4.0
Tester (MSDN Magazine, 2003)	Windows	GUI test DLL with analog capture/replay

**Table 35: Open Source GUI Testing Tools**

In this research, there are three possible scenarios for a GUI exercise. In the first scenario, the program GUI is provided and students are only required to implement the logic behind it. In the second scenario, teaching staff specify all required GUI components in a program for students to implement the logic behind them. In this scenario, additional GUI components can be added but the main foci are those required GUI components. In the last scenario, students are required to design the interface and implement the underlying logic.

In the first and second cases, all required GUI components are known by testers. Reflection can be used to obtain the value of each GUI component at runtime and compare it with the values of those in the model solution. With the third case, object-based GUI testing such as Jemmy (NetBeans,2002) is suggested.

#### **8.2.4 Improvement in Feedback**

The analysis results from static and dynamic analyses can be fine-tuned to provide further feedback for students and teaching staff. This section discusses some possible improvements on the automated feedback engine.

Structural feedback for students can be improved to identify exact areas where the structure of a student program does not match with the structure of the model solution, rather than leaving students to identify the differences themselves. This improvement is crucial for large gap exercises. In addition, suggestions for alternative solutions together with an explanation of why one solution is better than the other should be incorporated. This will help students to think about alternative solutions for a programming problem. Program correctness feedback can be enhanced to provide students with the test data, the expected model solution outputs, together with possible reasons for why a particular test failed.

Currently, teaching staff can monitor students' compilation attempts for an exercise. Further analysis on students' saved attempts on the server can be carried out to provide other useful information such as common syntax and programming structures.

In summary, this section has overviewed major areas of further work which can be extended on the ELP system, and the static and dynamic analyses. With the ELP system, supporting collaboration for students to do their programming practices and providing adaptive exercises are two areas which can be extended. With the static analysis, further investigation is required to support multiple model solutions

that allow teaching staff to customise the matching and normalisation process. With dynamic analysis, testing file IO and GUI programming exercises are two areas for further extension of the framework. With the automated feedback engine, feedback can be improved to point out to students the exact high complexity and poor code structure areas in their programs rather than leaving students to identify the differences themselves; provide alternative solutions and explanations; and reveal to students the actual test inputs and the expected model solution outputs. Additionally, an explanation for why a particular test failed, together with hints on how to fix it, can also be added to the feedback.

# Appendix A

## A.1 ELP Java Exercises

This section gives some examples of beginning students ELP Java programming exercises with gaps are shaded.

### HelloMe

```
import TerminalIO.*;  
  
public class HelloMe {  
    static ScreenWriter writer = new ScreenWriter();  
    static KeyboardReader reader = new KeyboardReader();  
  
    public static void main (String [] args)  
    {  
  
        // edit this line to change the output  
        writer.println( "Hello World!" );  
    }  
}
```

### GetName

```
import TerminalIO.*;  
  
public class GetName {  
    static ScreenWriter writer = new ScreenWriter();  
    static KeyboardReader reader = new KeyboardReader();  
  
    public static void main (String [] args ){  
        String firstName, lastName;  
  
        writer.println("What is your first name?");  
        firstName = reader.readLine();  
  
        // Output a request for the user's last name  
        writer.XXXX("What is your last name?");  
  
        // Assign the user input to the variable lastName  
        lastName = reader.XXXX;  
  
        writer.println("Hello " + firstName + " " + lastName + ", welcome to Java!");  
    }  
}
```

## Results

```
/* Results
 *
 * Program to convert marks to grade
 * Demonstrates use of IF/ELSE
 *
 */
import TerminalIO.*;
public class Results {
    static KeyboardReader reader = new KeyboardReader();
    static ScreenWriter writer = new ScreenWriter();
    public static void main (String [] args)
    {
        // Variable to hold final grade (1 - 7)
        int finalGrade;
        // Request and store input of accumulated marks for the unit (out of 100)
        writer.print("Enter your total marks for this unit: ");
        int totalMarks = reader.readInt();

        // Use IF/ELSE blocks to calculate the final grade
        // according to the marks received
        // 7 = 85 - 100
        // 6 = 75 - 84
        // 5 = 65 - 74
        // 4 = 50 - 64
        // 3 = 47 - 49
        // 2 = 25 - 46
        // 1 = 0 - 24
        if (totalMarks >= 85)
            finalGrade = 7;
        else if (totalMarks >= 75)
            finalGrade = 6;
        else if (totalMarks >= 65)
            finalGrade = 5;
        else if (totalMarks >= 50)
            finalGrade = 4;
        else if (totalMarks >= 47)
            finalGrade = 3;
        else if (totalMarks >= 25)
            finalGrade = 2;
        else
            finalGrade = 1;
        // Print the final grade
        writer.println("Your final grade for this unit is: " + finalGrade);
    }
}
```

## Root

```
/* Root
 *
 * Program to calculate the square root and square of a number
 * Demonstrates simple method implementation
 */
import TerminalIO.*;

public class Root {

    static KeyboardReader reader = new KeyboardReader();
    static ScreenWriter writer = new ScreenWriter();

    public static void main (String [] args) {

        int numb = reader.readInt("Enter an integer: ");

        // Set up a menu system
        writer.println("Please choose from the menu below:");
        writer.println("    Option a - calculate the square root");
        writer.println("    Option b - calculate the square");

        // Print a message, read and store the input
        char ans = reader.readChar("Please enter a or b: ");

        // Using IF/ELSE blocks, determine the method to call
        // and call that method
        if (ans == 'a' || ans == 'A')
            writer.println("The square root of " + numb + " is " + findRoot(numb));
        else if (ans == 'b' || ans == 'B')
            writer.println("The square of " + numb + " is " + findSquare(numb));
        else
            writer.println("Error! Invalid selection.");

    }

    /* Method to calculate the square root of an integer
     * pre: number is a valid integer
     * post: the square root of number is returned
     */
    public static double findRoot(int number)
    {
        return Math.sqrt(number);
    }

    /* Method to calculate the square of an integer
     * pre: number is a valid integer
     * post: the square of number is returned
     */
    public static long findSquare(int number)
    {
        return (number*number);
    }
}
```

## Coin

### Coin

```
public class Coin
{
    private final int HEADS = 0;
    private final int TAILS = 1;

    private int face;

    //-----
    // Sets up the coin by flipping it initially.
    //-----
    public Coin ()
    {
        flip();
    }

    //-----
    // Flips the coin by randomly choosing a face value.
    //-----
    public void flip ()
    {
        face = (int) (Math.random() * 2);
    }

    //-----
    // Returns true if the current face of the coin is
    // heads.
    //-----
    public boolean isHeads ()
    {
        return (face == HEADS);
    }

    //-----
    // Returns the current face of the coin as a string.
    //-----
    public String toString()
    {
        String faceName;

        if ( isHead())
            faceName = "Heads";
        else
            faceName = "Tails";

        return faceName;
    }
}
```

## CountFlips

```
public class CountFlips
{
    //-----
    // Flips a coin multiple times and counts the number
    // of heads and tails that result.
    //-----

    public void run(){
        final int NUM_FLIPS = 1000;
        int heads = 0, tails = 0;

        Coin myCoin = new Coin();

        for (int count=1; count <= NUM_FLIPS; count++)
        {
            myCoin.flip();           // flip the coin

            if (myCoin.isHeads())
                heads++;
            else
                tails++;
        }

        // output the results
        System.out.println ("The number flips: " + NUM_FLIPS);
        System.out.println ("The number of heads: " + heads);
        System.out.println ("The number of tails: " + tails);
    }

    public static void main (String[] args){
        CountFlips tpo = new CountFlips();
        tpo.run();
    }
}
```



## A.2 ELP C# Exercises

### SumTwo

```
// Calculates the sum of two integers input by user
// Pre: Input must be two integers separated by a space/tab
// Demonstrates Simple I/O and splitting a string.

using System;

class SumTwo {

    public static void Main() {
        string[] numbers;
        int v1, v2;
        v1 = v2 = 0;

        Console.Write("\nEnter two numbers, separated by spaces: ");
        numbers = Console.ReadLine().Split();

        v1 = int.Parse(numbers[0]);
        v2 = int.Parse(numbers[1]);

        Console.WriteLine("The sum of the two numbers is " + (v1+v2));
    }
}
```

## RomanToArabic

```
// Program to convert Roman number to Arabic form
using System;

public class RomanToArabic {

    public static void Main() {
        string roman;
        int arabic = 0;

        // Loop while there is input from the user
        do {
            Console.WriteLine("\nEnter Roman number (or empty line to exit): ");
            roman = Console.ReadLine();

            // remove all leading and trailing whitespace from the input
            roman = roman.Trim();

            if (roman.Length != 0) {
                if (ConvertToArabic(roman, ref arabic)) {
                    Console.WriteLine("{0} is the Roman notation for {1}", roman, arabic);
                }
            }
        } // End the loop if input is an empty line
        while (roman.Length != 0);
    } // end Main

    public static bool ConvertToArabic(string romanNum, ref int arabicNum) {
        int units, tens, hundreds, thousands;
        units = tens = hundreds = thousands = 0;
        bool validRoman = true;
        // convert the roman number to uppercase
        romanNum = romanNum.ToUpper();
        for (int i = 0; i < romanNum.Length; i++) {
            char romanChar = romanNum[i];
            switch(romanChar) {
                // Complete the switch statement by adjusting the values of
                // units, tens, hundreds and thousands as appropriate
                // for each roman char encountered
                case 'I':
                    units++;
                    break;
                case 'V':
                    units = 5 - units;
                    break;
                case 'X':
                    tens += 10 - units;
                    units = 0;
                    break;
                case 'L':
                    tens = 50 - tens - units;
                    units = 0;
                    break;
                case 'C':
                    hundreds += 100 - tens - units;
                    tens = units = 0;
                    break;
                case 'D':
                    hundreds = 500 - hundreds - tens - units;
                    tens = units = 0;
                    break;
                case 'M':
                    thousands += 1000 - hundreds - tens - units;
                    hundreds = tens = units = 0;
                    break;
                default:
                    Console.WriteLine("Error: {0} is not a Roman digit", romanChar);
                    validRoman = false;
                    break;
            }
        }
        arabicNum = thousands+hundreds+tens+units;
        return validRoman;
    } // end Arabic
}
```

## CircleArea2

```
using System;

public class CircleArea2 {

    public static void Main() {
        double r; string data;

        Console.WriteLine("\nEnter radius: ");
        data = Console.ReadLine();
        r = double.Parse(data);

        // Express the area as a float with 2 decimal places
        Console.WriteLine("Area is {0:F2}", area(r));

        // Express the area as a fixed point exponential with 3 decimal places
        Console.WriteLine("Area is {0:e3}", area(r));

        // Express the area in e or f-format (whichever is shorter)
        Console.WriteLine("Area is {0:g3}", area(r));

        // Express the area as a float with 5 decimal places, occupying 9 spaces
        Console.WriteLine("Area is {0,9:F3}", area(r));

        // Express the area as a float with 5 decimal places, occupying 9 spaces
        // Justify left
        Console.WriteLine("Area is {0,-9:F3}", area(r));
    }

    public static double area(double rad) {
        const double pi = 3.14159265358979323846;
        return (pi * rad * rad);
    }
}
```

## Creating and Using Money Class

### MoneyTest

```
using System;
using MoneyADT;

public class MoneyTest {
    public static void Main (String args) {
        Money myPay = new Money(1000, 99);
        Console.WriteLine("My pay is: \t\t{0}", myPay);

        Money yourPay = new Money(0, 1);
        Console.WriteLine("Your pay is: \t\t{0}", yourPay);

        // demonstrate use of the copy constructor
        Console.WriteLine("Copy myPay:");
        Money newPay = new Money(myPay);
        Console.WriteLine("New pay is: \t\t{0}", newPay);

        // demonstrate use of + operator
        newPay = myPay + yourPay;
        Console.WriteLine("\nAdd my pay and your pay:");
        Console.WriteLine("My pay is: \t\t{0}", myPay);
        Console.WriteLine("Your pay is: \t\t{0}", yourPay);
        Console.WriteLine("New pay is now: \t\t{0}", newPay);

        // Test CompareTo and IsEqual
        if (Money.AreEqual(newPay, myPay))
            Console.WriteLine("{0} is equal to {1}", newPay, myPay);
        else {
            String str = (Money.CompareTo(newPay, myPay)==1 ?
                "is greater than" : "is less than");
            Console.WriteLine("newPay {0} myPay", str);

        // The following lines are added for the purpose of ELP execution only
        Console.Write("\nPress ENTER to continue");
        Console.Read();
    }
}
```

## MoneyADT

```
namespace MoneyADT {

    using System;

    public class Money {
        // * copy constructor
        // * overloaded operator +
        // * overloaded ToString method with formatting
        // * CompareTo and IsEqual methods

        // default constructor
        public Money() {
            dollars = 0;
            cents = 0;
        }

        // copy constructor
        public Money(Money tempMoney) {
            this.dollars = tempMoney.dollars;
            this.cents = tempMoney.cents;
        }

        // another constructor
        public Money(int dollars, int cents) {
            this.dollars = dollars;
            this.cents = cents;
        }

        // properties
        public int Dollars {
            get { return dollars; }
            set { dollars = value; }
        }

        public int Cents {
            get { return cents; }
            set { cents = value; }
        }

        public override string ToString() {
            return String.Format ("${0,10:N0}.{1,2:D2}", dollars, cents);
        }

        // static method to compare moneys
        public static int CompareTo(Money a, Money b) {
            int num = (a.dollars*100 + a.cents)-(b.dollars*100 + b.cents);
            return (num > 0 ? 1 : (num == 0 ? 0 : -1));
        }

        // static method to test equality of moneys
        public static bool IsEqual(Money a, Money b) {
            return (CompareTo(a, b) == 0);
        }

        public static Money operator + (Money m1, Money m2) {
            Money tempMoney = new Money(m1);
            tempMoney.dollars += m2.dollars;
            tempMoney.cents += m2.cents;

            if (tempMoney.cents > 99) {
                tempMoney.dollars += (tempMoney.cents / 100);
                tempMoney.cents = (tempMoney.cents % 100);
            }
            return tempMoney;
        }

        private int dollars;
        private int cents;
    }
}
```

Exercise	Gap ID	Gap Type						Gap Behaviour	
		Expression	Block	Method(s)	Class	Declaration	Import	Properties	
HelloMe	1	✓							✓
GetName	1	✓							✓
	2	✓							✓
Results	1		✓						✓
Root	1			✓					✓
	2			✓					✓
Coin	1				✓				✓
	2		✓						✓
SumTwo	1		✓						✓
	2		✓						✓
	3		✓						✓
RomanToArabic	1		✓					✓	✓
	2		✓						✓
	3		✓						✓
CircleArea2	1		✓					✓	
	2		✓					✓	
	3		✓					✓	
	4		✓					✓	
Money	1			✓					✓
	2			✓					✓
	3			✓					✓
	4						✓		✓
	5			✓					✓
	6			✓					✓
	7			✓					✓

Table 36: Gaps Types and Their Behaviours

## Appendix B

### B.1 XML Serialized Format of Primitive, Array and User Defined Types

#### Primitive type

```
<primitive-int name="sum.so"><! [CDATA[128]]></primitive-int>
```

#### Array type

```
<array name="newArray.so" type="System.Int32">
<members>
    <! [CDATA[7]]><! [CDATA[14]]>
    <! [CDATA[21]]><! [CDATA[28]]>
    <! [CDATA[35]]><! [CDATA[42]]>
</members></array>
```

#### User defined type

```
<struct name="bancroft.so">
<fields>
    <field type="String" name="name">
        <value>Peter Bancroft</value>
        <modifiers><Private /></modifiers></field>
    <field type="Int32" name="room">
        <value>834</value>
        <modifiers><Private /></modifiers>
    </field>
    <field type="Int64" name="phone">
        <value>38644571</value>
        <modifiers><Private /></modifiers>
    </field>
    <field type="String" name="email">
        <value>p.bancroft@qut.edu.au</value>
        <modifiers><Private /></modifiers>
    </field>
</fields>
<constructors>
    <constructor><name>.ctor</name>
        <parameters>
            <parameter name="name" type="System.String" />
            <parameter name="room" type="System.Int32" />
            <parameter name="phone" type="System.Int64" />
            <parameter name="email" type="System.String" />
        </parameters>
    </constructor>
</constructors>
<methods>
    <method><name>ToString</name>
        <modifiers><IsPublic /><IsVirtual /></modifiers>
        <parameters />
        <return-type>System.String</return-type>
    </method>
</methods>
<properties />
</struct>
```



## References

- Adam, A. and J. Laurent. 1980. LAURA, a system to debug student programs. *Artificial Intelligence*, 15 (1): 75-122.
- Affleck, G. and T. Smith. 1999. Identifying a need for web-based course support. In *Proceedings of the 16th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education*, Brisbane, Australia.
- Ahmazadeh, M., D. Elliman and C. Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, 84-88. Lisbon, Portugal: ACM Press.
- Ala-Mutka, K. 2005. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15 (2): 83-102.
- Ala-Mutka, K. and H. M., Jarvinen. 2004. Assessment Process for Programming Assignments. In *International Conference on Advanced Learning Technologies*, 181-185: IEEE.
- Al-Ayyoub, A.-E. 2004. *A literature survey on ICT-Based learning*. <http://cisin.metu.edu.tr/ayyoub.php> (accessed July 24, 2006).
- Alice. 1999. Carnegie Mellon University. <http://www.alice.org/> (accessed May 28, 2006).
- Allen, E., R. Cartwright and B. Stoler. 2002. DrJava: a lightweight pedagogic environment for Java. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, 137-141. Cincinnati, Kentucky: ACM Press.
- Alm, J., R. Baber, S. Eggers, C. O'Toole and A. Shahab. 2002. You'd better set down for this!: creating a set type for CS1 & CS2 in C#. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education*, 14-18. Aarhus, Denmark: ACM Press.
- Almstrum, V. L., N. Dale, D. M. Miller, A. Berglund, M. Petre, M. Granger, P. Schragger, J. Little Currie and F. Springsteel. 1996. Evaluation: turning technology from toy to tool - report of the working group on evaluation. In *Proceedings of the 1st conference on Integrating Technology into Computer Science Education*, 201-217. Barcelona, Spain: ACM Press.
- Apert, S. R., M. K. Singley and J. M. Carroll. 1995. Multiple multimodal mentors, delivering computer-based instruction via specialized anthropomorphic advisors. *Behaviour and Information Technology*, 14 (2): 69-79.
- Andelson, B. and E. Soloway. 1985. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11 (11): 1351 - 1360.

Anderson, J. R., A. T. Corbett, K. R. Koedinger and R. Pelletier. 1995. Cognitive Tutors: Lessons learned. *Journal of the Learning Sciences*, 4: 167 - 207.

Anderson, J. R. and B. J. Reiser. 1985. The LISP tutor: it approaches the effectiveness of a human tutor. *BYTE*, 10 (4): 159-175.

Andreae, P., R. Biddle, G. Dobbie, A. Gale, L. Miller and E. Tempero. 1998. Surprises in teaching CS1 with Java. Victoria University of Wellington, CS-TR98/9.

Arnow, D. and O. Barshay. 1999. WebToTeach: an interactive focused programming exercise system. In *Proceedings of the 29th Annual Frontiers in Education Conference*, 12A9/39-12A9/44. San Juan, Puerto Rico: IEEE.

AutoIt. 1999. <http://www.autoitscript.com/> (accessed May 15, 2006).

Automation Junkies. 2002. Test Now. <http://www.automationjunkies.com/tools/vt.shtml> (accessed May 17, 2006).

Bailey, M. W. 2005. IRONCODE: think-twice, code-once programming. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 181-185. St. Louis, Missouri, USA: ACM Press.

Ball, T. 2001. *Improving Java programming language code using static analysis*. Sun Microsystems, Inc. <http://java.sun.com/javaone/javaone2001/pdfs/1012.pdf> (accessed May, 2002).

Bancroft, P. and P. Roe. 2006. *Program Annotations: Feedback for Students Learning to Program*. In Australasian conference on Computing Education, 19-23. Hobart, Australia: ACM Press.

Bandura, A. 1986. *Social foundations of thought and action*. 1st ed. Englewood Cliffs, N.J.: Prentice Hall.

Bangert-Drowns, R. L., C.-L. C. Kulik, J. A. Kulik and M. T. Morgan. 1991. The instructional effect of feedback in test-like events. *Review of Educational Research*, 61 (2): 213-238.

Barnes, D. J. and M. Kölling. 1999. *BlueJ*. <http://www.bluej.org/> (accessed March, 2002).

BCL. 2003. *Byte Code Engineering Library*. Apache. <http://jakarta.apache.org/bcel/index.html> (accessed September 30, 2005).

Beck, J., M. Stern and E. Haugsjaa. 1996. *Applications of AI in Education*. <http://www.acm.org/crossroads/xrds3-1/aied.html> (accessed April 21, 2005).

Beizer, B. 1995. *Black Box Testing - Techniques for Functional Testing of Software and System*. New York, Chichester, Brisbane, Toronto, Singapore: John Wiley & Sons, Inc.

Ben-Ari, M. 1998. Constructivism in computer science education. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, 257-261. Atlanta, Georgia, United States: ACM Press.

Ben-Ari, M. 2001. Constructivism in Computer Science Education. *Journal of Computers in Mathematics & Science Teaching*, 20 (1): 24-73. <http://stwww.weizmann.ac.il/g-cs/benari/CSE.HTM>.

Benbunan, R. 1997. *Effects of computer-mediated communication systems on learning, performance and satisfaction: a comparison of groups and individuals solving ethical case scenarios*. PhD Thesis, Rutgers University, Newark NJ.

Bergin, J., K. Brodie, M. Patino-Martinez, M. McNally, T. Naps, S. Rodger, J. Wilson, M. Goldweber, S. Khuri, and R. Jimenez-Peris. 1996. An overview of visualization: its use and design: report of the working group in visualization. In *Proceedings of the 1st conference on Integrating technology into computer science education*, 192-200. Barcelona, Spain: ACM Press.

Berry, R. E. and B. A. E. Meekings. 1985. A style analysis of C programs. *Communications of the ACM*, 28 (1): 80-88.

Bettini, L., P. Crescenzi, G. Innocenti, M. Loreti and L. Cecchi. 2004. An environment for self-assessing Java programming skills in first programming courses. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT'04*, 161-165. Joensuu, Finland: IEEE Computer Society, Washington, DC, USA.

Biddle, R. L. and E. Tempero 1998. *Java Pitfalls for Beginners*. ACM SIGCSE Bulletin, 30 (2): 48-52.

Bitzer, D.L. and R. L. Johnson. 1971. PLATO: A computer-based system used in the engineering of education. *Proceedings of the IEEE*, 59 (6): 960-968.

Blackboard. 1997. <http://www.blackboard.com/us/index.aspx> (accessed June 20, 2002).

Blackboard. 2000. *Educational Benefits of Online Learning*. [http://resources.blackboard.com/scholar/general/pages/ictraining/Online\\_Learning\\_Benefits.pdf](http://resources.blackboard.com/scholar/general/pages/ictraining/Online_Learning_Benefits.pdf) (accessed April 8, 2002).

Blank, G., S. Parvez, F. Wei and S. Moritz. 2005. A Web-based ITS for OO Design. In *Proceedings of workshop on Adaptive Systems for Web-based Education at the 12th International Conference on Artificial Intelligence in Education*, 59-64. Amsterdam, the Netherlands: ISO Press.

Bloom, B. S. 1956. *Taxonomy of Educational Objectives, Handbook 1: The Cognitive Domain*. White Plains, N.Y: Longman.

Bloom, B. S. 1984. The 2 Sigma Problem: The Search for methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher*, 13 (6): 4-15.

Blumenstein, M., S. Green, A. Nguyen and V. Muthukkumarasamy. 2004a. An experimental analysis of GAME: A Generic Automated Marking Environment. In *Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education*, 67-71. Leeds, UK: ACM Press.

Blumenstein, M., S. Green, A. Nguyen and V. Muthukkumarasamy. 2004b. GAME: A Generic Automated Marking Environment for programming assessment. In *Proceedings of Information Technology: Coding and Computing*, 212-216. Las Vegas, USA: IEEE Computer Society Press.

Bodington. 1997. Leeds University. <http://bodington.org/index.php> (accessed November 15, 2004).

Booth, S. 2001. Learning computer science and engineering in context. *Computer Science Education*, 11 (3): 168-188.

Boroni, C. M., F. W. Goosey, M. T. Grinder and R. J. Ross. 1998. A paradigm shift! The Internet, the Web, browsers, Java and the future of Computer Science Education. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 145-152. Atlanta, GA, US: ACM Press.

Box, I. 2004. Object-oriented analysis, criterion referencing, and Bloom. In *Proceedings of the 6th conference on Australian computing education*, 1-8. Dunedin, New Zealand: ACM Press.

Boyle, T. 2000. Constructivism: A Suitable Pedagogy for Information and Computing Sciences?

Brown, M. H. 1988. *Algorithm animation*: MIT Press.

Bruner, J. 2001. *Constructivist Theory*. <http://tip.psychology.org/bruner.html> (accessed May 13, 2006).

Bruner, J. S. 1966. *Toward a Theory of Instruction*. Cambridge, MA: Harvard University Press.

Brusilovsky, P., E. Schwartz and G. Weber. 1996. ELM-ART: An intelligent tutoring system on World Wide Web. In *Proceedings of the 3rd International Conference on Intelligent Tutoring System, Lecture Notes in Computer Science*, no 1086, pp. 261-269. Berlin: Springer Verlag.

Bryson, B. 2003. *Bridging the gap between black box and white box testing*. IBM. <http://www-128.ibm.com/developerworks/rational/library/1147.html> (accessed N.A, 2002).

- Buck, D. and D. J. Stucki. 2001. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, 16-20. Charlotte, North Carolina, United States: ACM Press.
- Bull, J. and C. McKenna. 2004. *Blueprint for computer assisted assessment*. 1st ed. US and Canada: Routledge Falmer.
- Burton, R.R. and J. S. Brown. 1982. An Investigation of Computer Coaching for Informal Learning Activities. In *Intelligent Tutoring Systems*, eds. D. Sleeman and J. S. Brown, pp. 79-98. New York: Academic Press.
- CAA Centre. 1998. University of Luton. <http://www.caacentre.ac.uk/index.shtml> (accessed March 28, 2003).
- Cain, A. 2000. JMetric: School of Information Technology at Swinburne University of Technology.
- Campbell, A. E. R., G. L. Catto and E. E. Hansen. 2003. Language-independent interactive data visualization. In *Proceedings of the 34th SIGCSE technical symposium on Computer Science Education*, 215 - 19. Reno, Nevada, USA: ACM Press.
- Carter, J., J. English, K. Ala-Mutka, M. Dick, W. Fone, U. Fuller and J. Sheard. 2003. How shall we assess this? *ACM SIGCSE Bulletin*, 35 (4): 107-123.
- Carver, R. 1996. Computer assisted instruction for a first course in computer science. In *Proceedings of the 26th Annual Conference in Frontiers in Education*, 721-724. Salt Lake City, UT, USA: IEEE.
- Caswell, M., V. Aravamudhan and K. Wilson. 2003. jfcUnit. <http://jfcunit.sourceforge.net/> (accessed May 30, 2006).
- Cattaneo, G., P. Faruolo, U. F. Petrillo and G. F. Italiano. 2004. JIVE: Java Interactive Software Visualization Environment. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing*, 41-43. Rome, Italy: IEEE.
- Cecchi, L., P. Crescenzi, and G. Innocenti. 2003. C: C++ = JavaMM: Java. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, 75-78. Kilkenny City, Ireland: ACM Press.
- Cenqua. Clover. <http://www.cenqua.com/clover/> (accessed May 17, 2006).
- Chang, K.-E., Y.-T. Sung and I.-D. Chen. 2002. The effect of concept mapping to enhance text comprehension and summarization. *Journal of Experimental Education*, 71 (1): 5-19.

Chase, J. D. and E. G. Okie. 2000. Combining cooperative learning and peer instruction in introductory computer science. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 372 - 376. Austin, Texas, United States: ACM Press.

Chatley, R. 2001. *Kenya*. Master, Department of Computing, Imperial College, London.

Cheang, B., A. Kurnia, A. Lim and W. C. Oon. 2003. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41: 121-131.

CheckStyle. <http://checkstyle.sourceforge.net/> (accessed May 17, 2006).

Chee, Y. S. and S. Xu. 1997. SIPLeS: Supporting Intermediate Smalltalk Programming through Goal-based Learning Scenarios. In *Proceedings of AI-ED 97: 8th World Conference on Artificial Intelligence in Education*, 95-102. Kobe, Japan: IOS Press.

Cheng, L.-T., C. R. B. Souza, S. Hupfer, J. Patterson and S. Ross. 2003. Building collaboration into IDEs. *ACM Queue*, 1 (9): 40-50. <http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=104>.

Cherniak, B. 1976. Introductory programming reconsidered - a user-oriented approach. In *Proceedings of the ACM SIGCSE-SIGCUE technical symposium on Computer science and education*, 65 - 68: ACM Press.

Clancy, M. J and M. C. Linn. 1999. Patterns and Pedagogy. In *Proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, 37 - 42. New Orleans, Louisiana, United States: ACM Press.

Clariana, R. B. 1990. A comparison of answer-until-correct feedback and knowledge-of-correct-response feedback under two conditions of contextualization. *Journal of Computer-Based Instruction*, 17 (4): 125-129.

Clariana, R., S. Ross, M. and G. Morrison. 1991. The effects of different feedback strategies using computer-administered multiple-choice questions as instruction. *Educational Technology Research and Development*, 39 (2): 5-17.

Clark, D., C. MacNish, C. and G. F. Royle. 1998. Java as a Teaching Language - opportunities, pitfalls and solutions. In *Proceedings of the 3rd Australasian conference on Computer science education*, 173-179. The University of Queensland, Australia: ACM Press.

Clemens, L. 1997. JavaNCSS. <http://www.kclee.com/clemens/java/javancss/> (accessed September 20, 2002).

Cliburn, D. C. 2003. Experiences with pair programming at a small college. *Journal of Computing Sciences in Colleges*, 19 (1): 20-29.

- Cole, O. 2000. White-box testing should check every line of code. *Dr. Dobb's journal of software tools for the professional programmer* (March).
- Conte, S. D., H. E. Dunsmore and V. Y. Shen. 1986. *Software Engineering Metrics and Models*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc.
- Costelloe, E. 2004. Teaching Programming The State of the Art. Institute of Technology Tallaght.
- CourseMaster. 2000. School of Computer Science & IT, The University of Nottingham, UK. [http://www.cs.nott.ac.uk/CourseMaster/cm\\_com/index.html](http://www.cs.nott.ac.uk/CourseMaster/cm_com/index.html) (accessed May 15, 2002).
- Cross II, J. 1999. jGRASP. Auburn University. <http://www.eng.auburn.edu/department/cse/research/grasp/> (accessed May, 2006).
- Cubranic, D. and M. A. D. Storey. 2005. Collaboration support for novice team programming. In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, 136-139. Sanibel Island, Florida, USA: ACM Press.
- Culwin, F. 1995. *Algorithms, metrics, testing and production*. <http://www.scism.sbu.ac.uk/law/Section5/contents.html> (accessed N/A, 2003).
- Curtis, R. 2000. Computer science education past and radical changes for future. In *Computer Science Education in the 21st Century*, ed. T. Greening, pp. 19-26. NY, Berlin, Heidelberg: Springer.
- Czejdo, B. and M. Rusinkiewicz. 1985. Program transformation and their application in teaching procedural and nonprocedural languages. *ACM SIGCSE Bulletin*, 17 (1): 202-210.
- Dalziel, J. and S. Gazzard. 1999. Next generation computer assisted assessment software: The design and implementation of WebMCQ. In *Proceedings of the 3rd Computer Assisted Assessment Conference*, 61--71. Loughborough, UK: Loughborough University.
- Davies, S. P. 1991. The role of notation and knowledge representation in the determination of programming strategy: A framework for integrating models of programming behaviour. *Cognitive Science*, 15 (4): 547 - 572. <http://www.cogsci.rpi.edu/CSJarchive/1991v15/i04/p0547p0572/MAIN.PDF>.
- de Raadt, M., R. Watson, and M. Toleman. 2003. Introductory programming languages at Australian universities at the beginning of the twenty first century. *Journal of Research and Practice in Information Technology*, 35 (3): 163-167.
- DeClue, T. H. 2003. Pair programming and pair trading: effects on learning and motivation in a CS2 course. *Journal of Computing Sciences in Colleges*, 18 (5): 49-56.

Deek, F. P. and J. A. McHugh. 1998. A survey and critical analysis of tools for learning programming. *Journal of Computer Science Education*, 8 (2): 130-178.

DeMillo, R. A., M. W. McCracken, R. J. Martin and J. F. Passafiume. 1987. *Software Testing and Evaluation*. Redwood City, CA, USA: Benjamin/Cummings.

DePasquale, P., J. A. N. Lee and M. A. Pérez-Quiñones. 2004. Evaluation of sub setting programming language elements in a novice's programming environment. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education SIGCSE '04*, 260 - 264. Norfolk, Virginia, USA: ACM Press.

Dershem, H. L., R. L. McFall and N. Uti. 2002. Animation of Java linked lists. In *Proceedings of the 33rd SIGCSE technical symposium on Computer Science Education*, 53 -57. Cincinnati, Kentucky: ACM Press.

DEST. 2006. *Australia's Future Using Education Technology: Report*. [http://www.dest.gov.au/sectors/school\\_education/policy\\_initiatives\\_reviews/reviews/australias\\_future\\_using\\_educational\\_technology/report.htm](http://www.dest.gov.au/sectors/school_education/policy_initiatives_reviews/reviews/australias_future_using_educational_technology/report.htm).

Dewey, J. (1933/1997). How we think. Mineola, New York: Dover Publications, INC.

Dihoff, R. E. 2004. *Provision of feedback during preparation for academic testing: learning is enhanced by immediate but not delayed feedback*. Copyright Psychological Record Spring 2004. [http://www.findarticles.com/p/articles/mi\\_qa3645/is\\_200404/ai\\_n9395185/print](http://www.findarticles.com/p/articles/mi_qa3645/is_200404/ai_n9395185/print).

Dingle, A. and C. Zander. 2000. Assessing the ripple effect of CS1 language choice. In *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, 85-93. Oregon Graduate Institute, Beaverton, Oregon, United States.

Dougiama, M. 1998. Moodle. Curtin University of Technology. <http://moodle.com/> (accessed August 29, 2005).

du Boulay, B. 1986. Some difficulties of learning to program. *Journal Educational Computing Research*, 2 (1): 57-73.

Ducasse, M. and A.-M. Emde. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software Engineering*, 162-171. Singapore: IEEE Computer Society Press.

Edwards, S. H. 2004. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 26-30. Norfolk, Virginia, USA: ACM Press.

- Efopoulos, V., V. Dagdilelis, G. Evangelidis and M. Satratzemi. 2005. WIPE: A Programming Environment for Novices. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, 113-117. Monte de Caparica, Portugal: ACM Press.
- Ehrlich, K. and E. Soloway. 1984. An empirical investigation of the tacit plan knowledge in programming. In *Human factors in computer systems*, eds. J. C. Thomas and M. L. Schneider, pp. 113-133. Norwood, New Jersey: Ablex Publishing.
- Eitelman, S. M. (2006). Computer Tutoring for Programming Education. In *Proceedings of the 44th annual Southeast regional conference*, 607-610. Melbourne, Florida: ACM Press.
- Ejiogu, L. O. 1985. A simple measure of software complexity. *ACM SIGMETRICS-Performance and Evaluation Review*, 13 (2): 33-34.
- Elenbogen, B. S., B. R. Maxim and C. McDonald. 2000. Yet, more Web exercises for learning C++. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 290 - 294. Austin, Texas, United States: ACM Press.
- Ellis, C. A. and J. S. Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, 399 - 407. Portland, Oregon, United States: ACM Press.
- Ellsworth, C. C., J. B. Fenwick and B. L. Kurtz. 2004. The Quiver System. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 205-209. Norfolk, Virginia, United States: ACM Press.
- Emory, D. and R. Tamassia. 2002. JERPA: A Distance-Learning Environment for Introductory Java Programming Courses. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, 307-311. Cincinnati, Kentucky: ACM Press.
- Endres, A. 1975. An analysis of errors and their causes in system programs. *SIGPLAN Notices*, 10 (6): 327-336.
- Entin, E. B. 1984. Using the cloze procedure to assess program reading comprehension. In *Proceedings of the fifteenth SIGCSE technical symposium on Computer science education*, 44-50: ACM Press.
- ESC/Java. 1990. *Extended Static Checking for Java*. Compaq. <http://research.compaq.com/SRC/esc/> (accessed November 11, 2002).
- Esteves, M. and A. Mendes. 2003. OOP-Anim, a system to support learning of basic object oriented programming concepts. In *Proceedings of the 4th international conference on Computer systems and technologies: e-Learning*, 573-579. Rousse, Bulgaria: ACM Press.

Etheredge, J. 2004. CMeRun: program logic debugging courseware for CS1/CS2 students. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 2 - 25. Norfolk, Virginia.

Fenton, N. E. 1991. *Software Metrics A Rigorous Approach*. London: Chapman & Hall.

Fenton, N. E. and M. Neil. 2000. Software metrics: roadmap. In *International Conference on Software Engineering*, 357-370. Limerick, Ireland: ACM Press.

Fischer, S. 2001. Course and exercise sequencing using metadata in adaptive hypermedia learning systems. *Journal on Educational Resources in Computing (JERIC)*, 1 (1es): Articles 5, 21 pages.

Fix, V., S. Wiedenbeck and J. Scholtz. 1993. Mental representations of programs by novices and experts. In *Conference on Human Factors in Computing Systems*, 74-79. Amsterdam, the Netherlands: ACM Press.

Forsythe, G. E. 1964. Automatic machine grading programs. In *Proceedings of the 1964 19th national conference*, 141.401: ACM Press.

Foulkes, A. and P. Thomas. 2001. *Web-based assessment: A new conceptual framework for supporting student progression and achievement*. CAA Centre, University of Luton. <http://www.glow.ac.uk/resources/part2/soar2.pdf> (accessed July, 2006).

Fowler, L., J. Armarego and M. Allen. 2001. CASE Tools: Constructivism and its application to learning and usability of software of engineering tools. *Computer Science Education*, 11 (3): 261-272.

Foxley, E. 1999. *Ceilidh on the World Wide Web*. <http://www.cs.nott.ac.uk/~ceilidh/> (accessed January 1, 2002).

Fritze, P. 2003. *Evaluating e-learning environments*. The University of Melbourne. <http://mettleweb.unimelb.edu.au/guide/evaluate4.html> (accessed May 21, 2006).

Frosini, G., B. Lazzerini and F. Marcelloni. 1999. Performing automatic exams. *Computers & Education*, 31 (3): 281-300.

Gamma, E and K. Beck. 2000. JUnit. <http://www.junit.org>. (accessed February 20, 2003).

Garner, S., P. Haden and A. Robins. 2005. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian conference on Computing education*, 173-180. Newcastle, Australia: ACM Press.

Garner, S. 2001. A tool to support the use of part-complete solutions in the learning of programming. In *Proceedings of Informing Science 2001*, 222-228. Krakow, Poland.

Glaserfeld, E. Radical Constructivism. A Way of Knowing and Learning. Routledge Falmer, London, 1995.

George, C. E. 2000. EROSI - visualising recursion and discovering new errors. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 305-309. Austin, Texas, United States: ACM Press.

German, A. 2003. Software Static Code Analysis Lessons Learned. *Journal of Defence Software Engineering*. <http://www.stsc.hill.af.mil/crosstalk/2003/11/0311German.html> (accessed May 14, 2003).

GILD. 2005. *GILD: Groupware enabled Integrated Learning and Development*. University of Victoria, Canada. <http://gild.cs.uvic.ca/> (accessed February 5, 2006).

Gilman, D. A. 1969. Comparison of several feedback methods for correcting errors by computer assisted instruction. *Journal of Educational Psychology*, 60 (6): 503-508.

Glassman, M. 2001. Dewey and Vygotsky: Society experience and inquiry in education practice. *Educational Researcher*, 30 (4): 3-14.

Goldstein, I. P. 1982. The genetic graph: a representation for the evolution of procedural knowledge. In *Intelligent Tutoring Systems*, pp. 51-77. London: Academic Press.

Gray, K. E. and M. Flatt. 2003. ProfessorJ: a gradual introduction to Java through language levels. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 170-177. Anaheim, CA, USA: ACM Press.

Gross, P. and K. Power. 2005. A meta study of software tools for introductory programming. In *Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference*, 12-13. Indianapolis, IN: IEEE Computer Society Press.

Gugery, L. and G. M. Olson. 1986. Debugging by skilled and novice programmers. *ACM SIGCHI Bulletin*, 17 (4): 171-174.

Hadjerrouit, S. 1998. A constructivist framework for integrating the Java paradigm into the undergraduate curriculum. *ACM SIGCSE Bulletin*, 30 (3): 105-107.

Hadjerrouit, S. 2005. Constructivism as guiding philosophy for software engineering education. *inroads - The SIGCSE Bulletin*, 37 (4): 45-49.

Hall, W. E. and S. H. Zweben. 1986. The cloze procedure and software comprehensibility measurement. *IEEE Transaction on Software Engineering*, 12 (5): 608-623.

Halstead, M. H. 1977. *Elements of software science*: Elsevier, New York.

Hattori, N. and N. Ishii. 1996. A method to remove variations in source codes. *Information and Software Technology*, 38: 25-36.

Heaney, D. and C. Daly. 2004. Mass production of individual feedback. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, 117 - 121. Leeds, UK: ACM Press.

Heiser, J. E. 1997. An overview of software testing. In *Proceedings of the 1997 IEEE Automated Testing Conference*, 204-211. Anaheim, CA USA.

Henry, S. and D. Kafura. 1981. Software metrics based on information flow. *IEEE Transactions on Software engineering*, 5 (7): 510-518.

Higgins, C., P. Symeonidis, and A. Tsintsifas. 2002. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, 46-50. Aarhus, Denmark: ACM Press.

Hitz, M. and S. Kogeler. 1997. Teaching C++ on the www. In *Proceedings of the 2nd conference on Integrating Technology into Computer Science Education*, 11-13. Uppsala, Sweden: ACM Press.

Hobgood, B., M. Thibault and D. Walbert. *Kinetic connections: Bloom's taxonomy in action addressing higher-level thinking with the World Wide Web*. The University of North Carolina at Chapel Hill School of Education. <http://www.learnnc.org/articles/bloom0405-1> (accessed September 15, 2006).

Hollingsworth, J. 1960. Automatic graders for programming classes. *Communications of the ACM*, 3 (10): 528-529.

Horwitz, S. 1990. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 234-245. White Plains, New York, United States: ACM Press.

Howden, W. E. 1981. A survey of static analysis methods. In *Tutorial: Software Testing and Validation Techniques*, eds. E. Miller and W. E. Howden, pp. 209--231: IEEE Computer Society Press.

Howden, W. E. 1982. Validation of scientific programs. *Computing Surveys*, 14 (2): 193-227.

Howles, T. 2003. Fostering the growth of a software quality culture. *inroads - The SIGCSE Bulletin*, 35 (2): 45-47.

Hristova, M., A. Misra, M. Rutter and R. Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, 153-156. Reno, Nevada, USA: ACM Press.

- Hundhausen, C. D. and J. L. Brown. 2005. *An experimental study of the impact of feedback self-selection on novice programming*. Journal of Visual Languages and Computing. <http://www.eecs.wsu.edu/~veopl/pub/index.htm> (accessed April 27, 2006).
- Hung, S.-L., I.-F. Kwok and R. Chan. 1992. Automatic programming assessment. *Computers & Education*, 20 (2): 183-190.
- Ibrahim, B. and S. D. Franklin. 1995. Advanced educational uses of the World-Wide Web. In *Proceedings of the Third International World-Wide Web conference on Technology, tools and applications*, 871 - 877. Darmstadt, Germany: Elsevier North-Holland, Inc.
- Jackson, D. 1991. Using software tools to automate the assessment of student programs. *Computers & Education*, 17 (2): 133-143.
- Jackson, D. and M. Usher. 1997. Grading student programs using ASSYST. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, 335-339. San Jose, California, United States: ACM Press.
- Jackson, J., M. Cobb and C. A. Carver. 2004. Identifying top Java errors for novice programmers. In *Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference*, T4C-24-T4C-27. Indianapolis: IEEE Computer Society Press.
- Jadud, M. C. 2005. A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15 (1): 25-40. <http://www.jadud.com/people/mcj/files/2004-PPIG-flcbBlueJ.pdf>.
- James, R., C. McInnis and M. Devlin. 2002. *Assessing learning in Australian universities*. Centre for the Study of Higher Education. <http://www.cshe.unimelb.edu.au/assessinglearning/docs/AssessingLearning.pdf>.
- Javassist. 2005. *Java Programming Assistant*. <http://www.csg.is.titech.ac.jp/~chiba/javassist/> (accessed September 30, 2005).
- Jeffries, R. A. 1982. Comparison of debugging behaviour of novice and expert programmers. Department of Psychology, Carnegie Mellon University.
- Jiménez-Peris, R., M. Patiño-Martínez, and J. Pacios-Martínez. 1999. VisMod: a beginner-friendly programming environment. In *Proceedings of the 1999 ACM symposium on applied computing*, 115 -120. San Antonio, Texas, United States: ACM Press.
- Johnson, L. W. and E. Soloway. 1985. PROUST: An automatic debugger for Pascal Program. *Byte*: 179-190.
- Jones, E. L. 2001a. Grading student programs - a software testing approach. 16 (2): 187-194.

Jones, E. L. 2001b. Integrating testing into the curriculum -- arsenic in small doses. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, 337-341. Charlotte, North Carolina, United States: ACM Press.

Joni, SaJ-Nicole A. and E. Soloway. 1986. But my program runs! Discourse rules for novice programmers. *Educational Computing Research*, 2 (1): 95-125.

Joy, M., P.-S. Chan and M. Luck. 2000. Networked submission and assessment. In *Proceedings of the 1st Annual Conference of the LTSN Centre for Information and Computer Science*, 41-45. Heriot-Watt, Edinburgh.

Joy, M. and M. Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42 (2): 129-133.

Kearney, J. P., R. L. Sedlmeyer, W. B. Thompson, M. A. Gray and M. A. Adler. 1986. Software complexity measurement. *Communications of the ACM*, 29 (11): 1044-1050.

Kelleher, C. and R. Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37 (2): 83-137.

Kessler, C. M. and J. R. Anderson. 1986. Learning flow of control: recursive and iterative procedures. *Human Computer Interaction*, 2: 135-166.

Khoshgoftaar, T. M. and J. C. Munson. 1990. The lines of code metric as a predictor of program faults: a critical analysis. In *Proceedings of the Fourteenth Annual International Conference in Computer Software and Applications*, 408-413: IEEE Computer Society Press.

King, M., B. Maegaard, J. Schutz, L. Tombe, A. Bech, A. Neville, A. Aeppe, et al. 1995. Evaluation of natural language processing systems. ISSCO, University of Geneva, Switzerland, EAG-EWG-PR.2.

Kokol, P., J. Brest and V. Zumer. 1996. Software complexity - an alternative view. *Systems, Man, and Cybernetics*, 4: 2862 - 2867.

Kolikant, B.-D. Y. 2001. Gardeners and cinema tickets: high school students' preconceptions of concurrency. *Computer Science Education*, 11 (1): 221 - 245. <http://education.huji.ac.il/yifatk/papers/gardener/gardeners.html> (accessed March 15, 2004).

Kolling, M. and J. Rosenberg. 1996. Blue - a language for teaching object oriented programming. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, 190-194. Philadelphia, Pennsylvania, United States: ACM Press.

Koundinya. 2003. *Black box testing, its advantages and disadvantages*. The Code Project. [http://www.codeproject.com/Purgatory/Black\\_Box\\_Testing.asp](http://www.codeproject.com/Purgatory/Black_Box_Testing.asp) (accessed N/A, 2004).

Krakatau. 2000. Power Software. <http://www.powersoftware.com/> (accessed September 1, 2002).

Kulhavy, R. W. and W. A. Stock. 1989. Feedback in written instruction: The place of response certitude. *Educational Psychology Review*, 1 (4): 279-308.

Kulik, J. A., and C.-L. C. Kulik. 1988. Timing of Feedback and Verbal Learning. *Review of Educational Research*, 58 (1): 79-97.

Kurland, L. C., R. D. Pea, C. Clement and R. Mawby. 1989. A study of the development of programming ability and thinking skills in high school students. In *Studying the novice programmer*, pp. 83-112. Hillsdale, NJ: Lawrence Erlbaum.

Kurland, M. D. and L. C. Kurland. 1987. Computer Applications in Education: A historical overview. *Annual Reviews Computer Science*, 2: 317-358.

Kwong, Y. I. (2001, April). Motivating Students by Providing Feedback. Centre for Development of Teaching and Learning, Singapore National University. <http://www.cdtl.nus.edu.sg/brief/v4n2/default.htm> (accessed May 27, 2007).

Le, Q and T. Le. 2001. Where does the superhighway lead us? a learners' perspective. In *Proceedings WCCE2001 Australian Topics: Selected Papers from the Seventh World Conference on Computers in Education*, 61-66. Copenhagen, Denmark: Australian Computer Society, Inc.

Leach, R. J. 1995. Using metrics to evaluate student programs. *ACM SIGCSE Bulletin*, 27 (2): 41-43.

Lesgold, A., S. P. Lajoie, M. Bunzo and G. Eggan. 1992. Sherlock: A coached practice environment for an electronics trouble shooting job. In *Computer assisted instruction and intelligent tutoring systems: Shared goals and complementary approaches*, pp. 201-238. Hillsdale, NJ: Lawrence Erlbaum Associates.

Levow, R. 1997. *Programming style for foundations of computer science*. <http://www.cse.fau.edu/~roy/cot3002.97f/style-reqs.html> (accessed May 10, 2006).

Lewandowski, G., A. Gutschow, R. McCartney, K. Sanders and D. Shinners-Kennedy. 2005. What novice programmers don't know. In *Proceedings of the 2005 international workshop on Computing education*, 1-12. Seattle, WA, USA: ACM Press.

Lewis, G. 2004. *Introduction to objective testing and CAA at Warwick*. Centre for Academic Practice the University of Warwick. [http://www2.warwick.ac.uk/services/cap/resources/eguides/printing/e1\\_print.pdf](http://www2.warwick.ac.uk/services/cap/resources/eguides/printing/e1_print.pdf) (accessed February 13, 2006).

Lewis, S. and G. Mulley. 1997. Experiences gained from producing a compiler to guide first year programming students. In *Proceedings of the fifth Annual Conference on Teaching of Computing*, 129-131. Dublin.

Lewis, S. and M. Watkins. 2001. Using Java tools to teach Java, the integration of Bluej and CourseMaster for delivery over the Internet. In *5th Java in the Computing Curriculum Conference (JICC 5)*. South Bank University: LTSN (Support Learning and Teaching).

Li, X. and C. Prasad. 2005. Effectively teaching coding standards in programming. In *Proceedings of the 6th conference on Information technology education*, 239 - 244. Newark, NJ, USA: ACM Press.

Lieberman, H. and C. Fry. 1995. Bridging the gulf between code and behaviour in programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 480 - 486. Denver, Colorado, United States: ACM Press.

Liffick, B. W. and R. Aiken. 1996. A novice programmer's support environment. In *Proceedings of the 1st conference on Integrating technology into computer science education*, 49 - 51. Barcelona, Spain: ACM Press.

Linder, S. P., D. Abbott and M. J. Fromberger. (2006). An instructional scaffolding approach to teaching software design. In *Journal of Computing Sciences in Colleges*, 21 (6): 238 - 250.

Lister, R. 2000. On blooming first year programming, and its blooming assessment. In *Australasian conference on Computing education*, 158-162. Melbourne, Australia: ACM Press.

Lister, R. and J. Leaney. 2003a. First year programming: let all the flowers bloom. In *Proceedings of the fifth Australasian conference on Computing education*, 221 - 230. Adelaide, Australia: ACM Press.

Lister, R. and J. Leaney. 2003b. Introductory programming, criterion-referencing, and bloom. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, 143 - 147. Reno Nevada, USA: ACM Press.

Lister, R., O. Seppälä, B. Simon, L. Thomas, E. S. Adams, S. Fitzgerald, W. Fone, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36 (4): 119-150.

Liquid Krystal. 1999. CodeSaw. <http://www.codesaw.com/> (accessed March 20, 2003).

Luck, M. and M. Joy. 1999. A secure online submission system. *Software - Practice and Experience*, 29 (8): 721-740.

Lui, K. A., R. Kwan, M. Poon and Y. H. Y. Cheung. 2004. Saving weak programming students: applying constructivism in a first programming course. *inroads - The SIGCSE Bulletin*, 36 (2): 72-76.

MacNish, C. 2000a. Evolutionary programming techniques for testing students' code. In *Proceedings of the Australasian conference on Computing education*, 170 - 173. Melbourne, Australia: ACM Press.

MacNish, C. 2000b. Java facilities for automating analysis, feedback and assessment of laboratory work. *Computer Science Education*, 10 (2): 147-163.

Malmi, L., A. Korhonen and R. Saikkonen. 2002. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, 55-59. Aarhus, Denmark: ACM Press.

Manaris, B. and R. McCauley. 2004. Incorporating HCI into the undergraduate curriculum: Bloom's taxonomy meets the CC'01 curricular guidelines. In *Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference*, T2H10-T2H15. Savannah, GA: IEEE Computer Society Press.

Marcelino, M., A. Gomes, N. Dimitrov and A. Mendes. 2004. Using a computer-based interactive system for the development of basic algorithmic and programming skills. In *Proceedings of the 5th international conference on Computer systems and technologies*, 1- 6. Rousse, Bulgaria: ACM Press.

Martínez-Unanue, R., M. Paredes-Velasco, C. Pareja-Flores, J. Urquiza-Fuentes and J. Á Velázquez-Iturbide. 2002. Electronic books for programming education: a review and future prospects. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, 34-38. Aarhus, Denmark: ACM Press.

Maschhoff, B. 2000. *XML: green light, go*. ASTD's Source for E-Learning. <http://www.learningcircuits.org/2000/aug2000/Marschhoff1.htm> (accessed N.A, 2002).

Mason, D. V. and D. M. Woit. 1999. Providing mark-up and feedback to students with online marking. In *Proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, 3-6. New Orleans, Louisiana, United States: ACM Press.

Mayer, R. 1981. The psychology of how novices learn computer programming. *ACM Computing Survey*, 13 (1): 121-141.

McCabe, T. J. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, 2 (4): 308-320.

McCauley, R. 2004. Thinking about our teaching. *inroads - The SIGCSE Bulletin*, 36 (2): 18-19.

McCracken, M. W., V. L. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B. D. Kolikant, C. Laxer, L. Thomas, I. Utting and T. Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, 33 (4): 125-140.

McDowell, C., L. Werner, H. Bullock and J. Fernald. 2002. The effects of pair programming on performance in an introductory programming course. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, 38 - 42. Cincinnati, Kentucky: ACM Press.

McDowell, C., L. Werner, H. Bullock and J. Fernald. 2003. The impact of pair programming on student performance, perception and persistence. In *Proceedings of the 25th International Conference on Software Engineering*, 602 - 607. Portland, Oregon: IEEE Computer Society Press, Washington, DC, USA.

McGettrick, A., R. Boyle, R. Ibbett, J. Lloyd, G. Lovegrove and K. Mander. 2005. Grand challenges in computing education - A summary. *The Computer Journal*, 48 (1): 42-48.

McGill, T. and S. Volet. 1995. An investigation of the relationship between student algorithm quality and program quality. *SIGCSE Bulletin*, 27 (2): 44-48.

McKenzie, J. 1999. Scaffolding for success. *The Educational Technology Journal*, 9 (4): online. <http://www.lincolnparkboe.org/Scaffolding.pdf> (accessed March 15, 2005).

McKeown, J. 2004. The use of a multimedia lesson to increase novice programmers' understanding of programming array concepts. *Journal of Computing Sciences in Colleges*, 19 (4): 39-50.

Mendes, A. J., A. Gomes, M. Esteves, M. J. Marcelino, C. Bravo, and M. A. Redondo. 2005. Using simulation and collaboration in CS1 and CS2. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, 193-197. Caparica, Portugal: ACM Press.

Mengel, S.A. and J. V. Ullans. 1999. A case study of the analysis of novice student programs. In *Proceedings 12th Software Engineering Education and Training*, 40-49. New Orleans, LA, USA: IEEE Computer Society Press.

Mengel, S. and V. Yerramilli. 1999. A case study of the static analysis of the quality of novice student programs. In *Proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, 78-82. New Orleans, Louisiana, United States.

Merrill, D. C., B. J. Reiser, M. Ranney and G. J. Trafton. 1992. Effective Tutoring Techniques: A comparison of Human Tutors and Intelligent Tutoring Systems. *Journal of the Learning Sciences*, 2 (3): 277-305.

Merrill, J. 1987. Levels of questioning and forms of feedback: Instructional factors in courseware design. *Journal of Computer-Based instruction*, 14 (1): 18-22.

Microsoft. *Visual C# Language*. 2001. [http://msdn2.microsoft.com/en-us/library/aa287558\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/aa287558(VS.71).aspx) (accessed September 21, 2005).

Miller, G. A. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 101 (2): 343-352.

Milton, J. 2001. *Feedback to students*. RMIT University. <http://www.lts.rmit.edu.au/renewal/assess/faq3.htm> (accessed January 28, 2005).

Mizoguchi, R. and J. Bourdeau. 2000. Using ontological engineering to overcome common AI-ED problems. *International Journal of Artificial Intelligence in Education*, 11 (2): 107-121.

Mono. [http://www.mono-project.com/Main\\_Page](http://www.mono-project.com/Main_Page). (accessed June 1, 2003)

Morris, D. 2003. Automatic grading of student's programming assignments: an interactive process and suite of programs. In *Proceedings of the thirty third ASEE/IEEE Frontiers in Education Conference*, S2F1-S3F6. Boulder, CO: IEEE Computer Society Press.

Morris, J. 2004. *Algorithms Animators*. [http://www.cs.auckland.ac.nz/software/AlgAnim/alg\\_anim.html](http://www.cs.auckland.ac.nz/software/AlgAnim/alg_anim.html) (accessed March 7, 2006).

Morrison, G. R., S. M. Ross, M. Gopalakrishnan and J. Casey. 1995. The effects of feedback and incentives on achievement in computer-based instruction. *Contemporary Educational Psychology*, 20: 32-50.

Morrison, M. and T. S. Newman. 2001. A study of the impact of student background and preparedness on outcomes in CS I. *ACM SIGCSE Bulletin*, 33 (1): 179-183.

Mosemann, R. and S. Wiedenbeck. 2001. Navigation and comprehension of programs by novice programmers. In *Proceedings of the 9th International Workshop on Program Comprehension*, 79-88. Toronto, Ont.: IEEE Computer Society Press.

Moser, R. 1997. A fantasy adventure game as a learning environment: why learning to program is so difficult and what can be done about it. In *Proceedings of the 2nd conference on integrating technology into computer science education*, 114-116. Uppsala, Sweden: ACM Press, New York, USA.

Moss. 1994. A system for detecting software plagiarism. <http://theory.stanford.edu/~aiken/moss/> (accessed .28 July, 2005).

Motil, J. and D. Epstein. 1998. *JJ: a language designed for beginners (less is more)*. Caltech [http://www.publicstaticvoidmain.com/JJ\\_A\\_Language\\_Designed\\_For\\_Beginners\\_LessIsMore.pdf](http://www.publicstaticvoidmain.com/JJ_A_Language_Designed_For_Beginners_LessIsMore.pdf) (accessed November 20, 2004).

MSDN Magazine. 2003. Tester. <http://msdn.microsoft.com/msdnmag/issues/02/03/Bugslayer/> (accessed May 29, 2006).

Muller, H. 2003. *Teach the children well.* Sun. [http://weblogs.java.net/blog/hansmuller/archive/2003/11/teach\\_the\\_child.html](http://weblogs.java.net/blog/hansmuller/archive/2003/11/teach_the_child.html) (accessed April 28, 2005).

Murphy, L., K. Blaha, T. VanDeGrift, S. Wolfman and C. Zander. 2002. Active and cooperative learning techniques for the computer science classroom. *Journal of Computing Sciences in Colleges*, 18 (2): 92-92.

Murray, G. 1995. WebCT. <http://www.webct.com/>. (accessed April 20, 2003).

Murray, R. 1986. TALUS: Automatic program debugging for intelligent tutoring systems. The University of Texas at Austin, TR 86-32.

Murray, T. and B. Woolf. 1992. Results of encoding knowledge with tutor construction tools. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 17-23. San Jose, CA: AAAI Press.

Murray, W. M. 1988. *Automatic program debugging for intelligent tutoring systems*. Pitman, London: Morgan Kaufmann.

Myers, G. J. 1979. *The art of software testing*. New York, Chichester, Brisbane, Toronto, Singapore: John Wiley & Sons.

Nagappan, N., L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller and S. Balik. 2003. Improving the CS1 experience with pair programming. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, 359-362. Reno, Nevada, USA: ACM Press.

Nagle, C. 2002. Software Automation Framework Support. <http://safsdev.sourceforge.net/> (accessed Jun 1, 2006).

Nanja, M. and C. R. Cook. 1987. An analysis of the online debugging process. In *Empirical Studies of Programmers*, pp. 172-183. Norwood, NJ: Ablex Publishing.

Naps, T. L., L. L. Norton and J. R. Eagan. 2000. JHAVE - An environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 109 - 113. Austin, Texas: ACM Press.

Naps, T. L., G. Robling, V. L. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally and S. Rodger. 2002. Exploring the role of visualization and engagement in computer science education. In *Working group reports from ITiCSE on Innovation and technology in computer science education*. Aarhus, Denmark: ACM Press.

NetBeans. 2002. Jemmy. <http://jemmy.netbeans.org/> (accessed December 1, 2005).

Niemeyer, P. BeanShell. 2000. <http://www.beanshell.org/> (accessed April 15, 2003).

- Norcio, A. F. 1980a. Human memory for comprehending computer programs. In *Proceedings of the 1980 IEEE International Conference on Cybernetics and Society*, 974-977. Hyatt Regency, Cambridge, Massachusetts: IEEE Computer Society Press.
- Norcio, A. F. 1980b. Comprehension aids for computer programs. In *Proceedings of the annual convention of the American Psychological Association*. Montreal.
- Norcio, A. F. 1982a. Chunking and understanding computer programs. Applied Science Department, U.S. Naval Academy. Technical Report A8-2-82.
- Norcio, A. F. 1982b. Indentation, documentation and programmer comprehension. In *Proceedings of the 1982 conference on Human factors in computing systems*, 118-120. Gaithersburg, Maryland, United States: ACM Press.
- Norman, D. A. and J. C. Spohrer. 1996. Learner-centered education. *Communication of the ACM*, 39 (4): 24-27.
- Odekirk-Hash, E. 2001. *Providing automatic feedback to novice programmers*. Master Thesis, Department of Computer Science, The University of Utah, Utah.
- Olson, J. and J. Platt. 2004. *Teaching children and adolescents with special needs*. 4th ed. Upper Saddle River, N.J.: Merrill Prentice Hall.
- O'Neal, M. B. 1993. An empirical study of three common software complexity measures. In *Proceedings of the 1993 ACM/SIGAPP symposium on applied computing: states of the art and practice*, 203-207. Indianapolis, Indiana, United States: ACM Press.
- Ong, J. and S. Ramachandran. 2000. *Intelligent tutoring systems: the what and the how*. <http://www.learningcircuits.org/2000/feb2000/ong.htm> (accessed April 21, 2005).
- Open Text Corporation. 1990. FirstClass. <http://www.softarc.com/index.shtml> (accessed April 9, 2004).
- Osborne, M. and K. Lambert. 2000. *BreezyGUI*. <http://faculty.cs.wvu.edu/martin/Software%20Packages/BreezySwing/Default.htm> (accessed March, 2004).
- Overbaugh, R. C. 1994. Research-based guidelines for computer-based instruction development. *Journal of Research on Computing in Education*, 27 (1): 29-47.
- Owston, R. D. 1997. The World Wide Web: A technology to enhance teaching and learning? *Educational Researcher*, 26 (2): 27 - 33.
- Paas, F. G. W. C. 1992. Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of Educational Psychology*, 84 (4): 429-434.

Panitz, T. 1999. *The case for student centered instruction via collaborative learning paradigms*. (accessed February 15, 2006 from ERIC database).

Paramythis, A. and S. Loidl-Reisinger. 2004. Adaptive learning environments and e-learning standards. *Electronic Journal of e-Learning*, 2 (1): online. <http://www.ejel.org/volume-2/vol2-issue1/issue1-art11.htm>.

Parker, J. R. and K. Becker. 2003. Measuring effectiveness of constructivist and behaviourist assignments in CS102. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, 40-44. Thessaloniki, Greece: ACM Press.

Pekar, M. 2002. Pounder. <http://pounder.sourceforge.net/> (accessed May 20, 2006).

Perkins, D. N., C. Hancock, R. Hobbs, F. Martin and R. Simmons. 1986. Conditions of learning in novice programmers. *Journal Educational Computing Research*, 2 (1): 37-55.

Perkins, D. N. and F. Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *Empirical Studies of Programmers: First Workshop*, 213-229. Washington, DC: Ablex Publishing Corporation.

Perkins, D. N., S. Schwartz and R. Simmons. 1988. Instructional strategies for the problems of novice programmers. In *Teaching and Learning Computer Programming: Multiple Research Perspectives*, ed. R. E. Mayer, pp. 154-178. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Perry, R. 2004. View/Edit/Compile/Run web-based programming environment. In *Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference*, T3H-1-T3H-6. Savannah, GA: IEEE Computer Society Press.

PERWAPI. 2005. *Program Executable-Reader/Writer-Application Programming Interface*. QUT. <http://www.plas.fit.qut.edu.au/perwapi/Default.aspx> (accessed September 30, 2005).

Piaget, J. (1969). *The psychology of the child*. New York: Basic Books.

Pierson, W. and S. H. Rodger. 1988. Web-based Animation of Data Structures Using JAWAA. *ACM SIGCSE Bulletin*, 30 (1): 267-271.

Pillay, N. 2003. Developing intelligent programming tutors for novice programmers. *SIGCSE bulletin*, 35 (2): 78-82.

PMD. <http://pmd.sourceforge.net/> (accessed May 1, 2006).

Pohl, I. 2003. C# Should CS 1 Switch. In *WG2.4 IFIP Conference*. Santa Cruz. <http://www.cs.ucsc.edu/~pohl/Papers/ShouldCS1Switch.doc> (accessed January 15, 2005).

Poole, B. J. and T. S. Meyer. 1996. Implementing a set of guidelines for CS majors in the production of program code. *SIGCSE Bulletin*, 28 (2): 43-48.

Prechelt, L., G. Malpohl and M. Phlippsen. 2000. JPlag: Finding plagiarisms among a set of programs. Technical Report. <http://page.mi.fu-berlin.de/~prechelt/Biblio/jplagTR.pdf> (accessed August, 2003).

Preston, D. 2005. PAIR programming as a model of collaborative learning: a review of the research. *Journal of Computer Sciences in Colleges*, 20 (4): 39-45.

Preston, J. A. and R. Shackelford. 1999. Improving on-line assessment: an investigation of existing marking methodologies. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, 29-32. Cracow, Poland: ACM Press.

Pridemore, D. R. and J. D. Klein. 1991. Control of feedback in computer assisted instruction. *Educational Technology Research and Development*, 39 (4): 27-32.

Pridemore, D. R. and J. D., Klein. 1995. Control of practice and level of feedback in computer-based instruction. *Contemporary Educational Psychology*, 20: 444-450.

Proulx, V. K. and R. Rasala. 2004. Java IO and testing made simple. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 161-165. Norfolk, Virginia, USA: ACM Press.

Pullen, M. 2001. The network workbench and constructivism: learning protocols by programming. *Computer Science Education*, 11 (3): 189-202.

Ramadhan, H. A. 2000. DISCOVER: an intelligent discovery programming system. *Cybernetics & Systems*, 31 (1): 87-114.

Ramalingam, V., D. LaBelle, and S. Wiedenbeck. 2004. Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, 171 - 175. Leeds, United Kingdom: ACM Press.

Ray, E. 2004. *Paintbrush of discovery: using Java applets to enhance mathematics education*.

[http://www.findarticles.com/p/articles/mi\\_qa3997/is\\_200403/ai\\_n9370098/print](http://www.findarticles.com/p/articles/mi_qa3997/is_200403/ai_n9370098/print) (accessed May 5, 2005).

Reek, K. A. 1989. The TRY system -or- how to avoid testing student programs. In *Proceedings of the twentieth SIGCSE technical symposium on Computer science education*, 112-116. Louisville, Kentucky, United States: ACM Press.

Rees, M. J. 1982. Automatic assessment aids for Pascal programs. *ACM SIGPLAN Notices*, 17 (10): 33-42.

Reges, S. 2002. Can C# Replace Java in CS1 and CS2? In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, 4-8. Aarhus, Denmark: ACM Press.

Rich, C. and R. C. Waters. 1990. *The Programmer's Apprentice*, ACM Press Frontier Series. New York: ACM Press.

Rich, C. and L. M., Wills. 1990. Recognizing a program's design: a graph-parsing approach. *IEEE Software*, 7 (1): 82-89.

Rist, R. S. 1986a. Focus and learning in program design. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 371-380. University of Massachusetts, Amherst, MA: Lawrence Erlbaum Associates.

Rist, R. S. 1986b. Plans in programming: definition, demonstration, and development. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, 28-47. Washington, D.C., United States: Ablex Publishing Corp.

Rist, R. S. 1989. Schema creation in programming. *Cognitive Science*, 13 (3): 295-465.

Rist, R. S. 1995. Program structure and design. *Cognitive Science*, 19 (4): 507-562. <http://www-staff.it.uts.edu.au/~rist/docs/program.htm> (accessed May 20, 2003).

Robins, A., J. Rountree and N. Rountree. 2003. Learning and teaching programming: a review and discussion. *Computer Science Education*, 13 (2): 137-172.

Robinson, S. S. and M. L., Soffa. 1980. An instructional aid for student programs. In *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*, 118 - 129. Kansas City, Missouri, United States: ACM Press.

Rogalski, J. and R. Samurcay. 1993. Task analysis and cognitive model as a framework to analyse environments for learning programming. In *Cognitive Models and Intelligent Environments for Learning Programming*, 111: 6-19. Berlin, Heidelberg, NY, London, Paris, Tokyo, HK, Barcelona, Budapest: Springer-Verlag.

Roper, W. J. 1977. Feedback in computer assisted instruction. *Programmed Learning and Educational Technology*, 14 (1): 43-49.

Rosenshine, B. and C. Meister. 1992. The use of scaffolds for teaching higher-level cognitive strategies. *Educational Leadership*, 49 (7): 26-33.

Rountree, J., N. Rountree, A. Robins and R. Hannah. 2005. Observations of student competency in a CS1 course. In *Proceedings of the seventh Australasian Computing Education Conference*, 145-149. Newcastle, Australia: ACM Press.

Rust, C. 2002. The impact of assessment on student learning. *Active Learning in Higher Education*, 3 (2): 145-158.

- Saikkonen, R., L. Malmi and A. Korhonen. 2001. Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science*, 133-136. Canterbury, United Kingdom: ACM Press.
- Sanders, D. and J. Hartman. 1987. Assessing the quality of programs: A topic for the CS2 course. In *Proceedings of the eighteenth SIGCSE technical symposium on Computer science education*, 92-96. St. Louis, Missouri, United States: ACM Press.
- Sanders, I. 1991. AAPT: algorithm animator and programming toolbox. *SIGCSE Bulletin*, 23 (4): 41-47.
- Schank, R. C. 2000. *A vision of education for the 21st century*. [http://thejournal.com/articles/14552\\_1](http://thejournal.com/articles/14552_1) (accessed November 20, 2005).
- Schimmel, B. J. 1988. Providing meaningful feedback in courseware. In *Instructional designs for microcomputer courseware* 183-196. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Schorsch, T. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, 168-172. Nashville, Tennessee, United States: ACM Press.
- Seale, J. 2002. *Using computer assisted assessment to support student learning*. The Higher Education Academy. [http://www.heacademy.ac.uk/resources.asp?process=full\\_record&section=generic&id=38](http://www.heacademy.ac.uk/resources.asp?process=full_record&section=generic&id=38) (accessed May 1, 2004).
- Shaffer, S. C. 2005. Ludwig: an online programming tutoring and assessment system. *inroads - The SIGCSE Bulletin*, 37 (2): 56-60.
- Schon, D. A. 1983. *The reflective practitioner: how professionals think in action*. United States: Basic Books.
- Slavin, R. E. 1995. *Cooperative Learning Theory, Research and Practice*. Boston, United States.
- Slay, J. 1997. The use of the internet in creating an effective learning environment. In *Proceedings of the third Australian World Wide Web Conference*. Southern Cross University, Lismore NSW 2480: online.
- Smith, P. A. and G. I. Webb. 1995. Reinforcing a generic computer model for novice programmers. In *ASCILITE*. Melbourne, Australia.
- Søndergaard, H. and P. Gruba. 2001. A constructivist approach to communication skills instruction in computer science. *Computer Science Education*, 11 (3): 203-209.

Software Quality Institute 1991. *PASS: Program Analysis and Style System for the Ada, C and Java Languages*. <http://www3.sqi.gu.edu.au/pass/> (accessed September 23, 2005).

Soh, L.-K., A. Samal, S. Person, G. Nugent and J. Lang. 2005. Designing, implementing, and analysing a placement test for introductory CS courses. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, 505 - 509. St. Louis, Missouri, USA: ACM Press.

Soloway, E. 1986. Learning to program = learning to construct mechanisms and explanations. *Communication of the ACM*, 29 (9): 850-858.

Soloway, E., K. Ehrlich and J. B. Black. 1983. Beyond Numbers: Don't Ask "How Many"..... Ask "Why". In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 240-246. Boston, Massachusetts, United States: ACM Press.

Song, J.S, S. H. Hahn, Y.K. Tak and J. H. Kim. 1997. An intelligent tutoring system for introductory C language course. *Computer Educations*, 28 (2): 93-102.

Spannagel, C. 2005. Jacareto. <http://jacareto.sourceforge.net/> (accessed May 15, 2006).

Stasko, J. T. 1990. Tango: a framework and system for algorithm animation. *Computer*, 23 (9): 27-39.

Stasko, J. T. 1997. Using student-built algorithm animations as learning aids. *ACM SIGCSE Bulletin*, 29 (1): 25-29.

Stuyf, R. 2002. *Scaffolding as a teaching strategy*. <http://condor.admin.ccny.cuny.edu/~group4/Van%20Der%20Stuyf/Van%20Der%20Stuyf%20Paper.doc> (accessed December 1, 2005).

Sun. 2005. *Scope*. <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/scope.html> (accessed January 1, 2002).

Suthers, D. 1996. Introduction: architectures and methods for designing cost effective and reusable ITS's. In *ITS'96 Workshop on Architectures and Methods for Designing Cost Effective and Reusable ITSs*. Montreal, Canada.

Sykes, E. R. and Franek, K. 2004. A prototype for an intelligent tutoring system for students learning to program in Java™. In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education*, 78 -83. Rhodes, Greece.

Taneva, S., R. Alterman, K. G. Granvill, M. R. Head and T. J. Hickey. 2003. GREWPtool: a system for studying online collaborative learning. Department of Computer Science, Brandeis University, CS-03-239.

Taylor, K. 2005. *Common Java coding errors*. [http://java.about.com/cs/beginningjava/a/comm\\_errs.htm](http://java.about.com/cs/beginningjava/a/comm_errs.htm) (accessed N/A, 2005).

Tejas Software Consulting 2003. *Open Testware Reviews - GUI test driver survey*. <http://tejasconsulting.com/open-testware/feature/gui-test-driver-survey.html> (accessed September 26, 2005).

The Joint Task Force on Computing Curricula. 2004. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, Computing Curricula*: IEEE Computer Society Press.

Thomas, M. and S. H. Zweben. 1986. The effects of program-dependent and program-independent deletions on software cloze tests. In *Proceedings of the first workshop on Empirical Studies of programmers on Empirical Studies of programmers*, 138-`52. Washington, D.C., United States: Ablex Publishing Corp.

Thorburn, G. and G. Rowe. 1997. PASS: an automated system for program assessment. *Computers & Education*, 29 (4): 195-206.

Tobias, S. 1982. When do instructional method make a difference. *Educational Researcher*, 11 (4): 4-9.

Townhidnejad, M. and T. B. Hilburn 2002. Software quality across the curriculum. In *Proceedings of the 15th Conference on Software Engineering Education and Training*, 268-272. Covington, KY, USA: IEEE Computer Society Press.

Tuovinen, J. E. 2000. Optimising student cognitive load in computer education. In *Proceedings of the Australasian conference on Computing education*, 235-241. Melbourne, Australia: ACM Press.

Turingscraft. 2002. CodeLab. <http://www.turingscraft.com/index.php> (accessed March 5, 2004).

Tyma, P. 1998. Why are we using Java again? *Communication of the ACM*, 41 (6): 38-42. [http://www.cs.cornell.edu/ugrad/Tyma\\_Article.htm](http://www.cs.cornell.edu/ugrad/Tyma_Article.htm).

Ueno, H. and T. Inoue. 1999. A shared intelligent programming environment on the internet for learning C programming. *Frontiers in Artificial Intelligence and Applications*, 55: 752-759.

Ulloa, M. 1980. Teaching and learning computer programming: a survey of student problems, teaching methods, and automated instructional tools. *ACM SIGCSE Bulletin*, 12 (2): 48 - 64.

Usdin, T. and T. Graham. 1998. XML: not a silver bullet, but a great pipe wrench. *StandardView*, 6 (3): 125-132.

Van Gorp, M. J. and S. Grissom. 2001. An empirical evaluation of using constructive classroom activities to teach introductory programming. *Computer Science Education*, 11 (3): 247-260.

Van Merriënboer, J. J. (in press). 1990. Strategies for programming instruction in high school: Program completion vs. Program generation. *Journal Educational Computing Research*, 6: 265-285.

Van Merriënboer, J. J. and M. B. De Croock. 1992. Strategies for computer based programming instruction: program completion vs. program generation. *Journal Educational Computing Research*, 8 (3): 365-394.

Van Merriënboer, J. J. and H. Krammer. 1987. Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science*, 16 (3): 251-285.

Van Merriënboer, J. J., H. Krammer and M. R. Maaswinkel. 1994. Automating the planning and construction of programming assignments for teaching introductory computer programming. In *Automating instructional design, development and delivery*, 119: 61-77. Berlin; New York: Springer-Verlag.

Van Merriënboer, J. J. and F. G. Paas. 1990. Automation and schema acquisition in learning elementary computer programming: implications for the design of practice. *Computers in Human Behaviour*, 6: 273-289.

VanDoren, E., K. Sciences and C. Springs. 1997. *Cyclomatic complexity*. Software Engineering Institute [http://www.sei.cmu.edu/str/descriptions/cyclomatic\\_body.html](http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html) (accessed N/A, 2002).

Vazhenin, A. and D. Vazhenin. 2001. Applied programming on web-based environment. In *Topics in Applied and Theoretical Mathematics and Computer Science* 287-292: WSEAS Press.

Vazhenin, A., Y.-H. Wang and D. Vazhenin. 2002. Web-based multimedia platform for programming. In *Proceedings of the 8th International Conference on Distributed Multimedia Systems*, 76-83. San Francisco, California, USA.

Vazhenin, D., A. Vazhenin and Y-H. Wang. 2005. Web-based environment for programming and distance learning. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, 109-112 vol.2: IEEE Computer Society Press.

Virginia Tech. 2003. Web-CAT. <http://web-cat.cs.vt.edu/> (accessed May 1, 2003).

Vizcaino, A., J. Contreras, J. Favela and M. Prieto. 2000. An adaptive, collaborative environment to develop good habits in programming. In *Proceedings of the 5th International Conference on Intelligent Tutoring Systems*, 262-271. Montreal: Springer-Verlag, London, UK.

Vygotsky, L. S. 1978. *Mind and society: The development of higher mental processes*: Cambridge, M.A: Harvard University Press.

Waldrop, P. B., J. E. Justen and T. M. Adams. 1986. A comparison of three types of feedback in a computer assisted instruction task. *Educational Technology*, 26 (11): 43-45.

Wall, T. 2002. Abbot Java GUI Test Framework. <http://abbot.sourceforge.net/doc/overview.shtml> (accessed May 15, 2006).

Watson, A. and T. J. McCabe. 1996. *Structured testing: a testing methodology using the cyclomatic complexity metric*. Computer Systems Laboratory. <http://www.softwaresafety.net/Metrics/nist235r.pdf>.

Weber, G. and P. Brusilovsky. 2001. ELM-ART: an adaptive versatile system for web-based instruction. *International of Artificial Intelligence and Education*, 12 (4): 351-384.

West, T. 2000. *Teaching with Java: the good, the bad and the opportunity*. Holt Software.

[https://cemc.math.uwaterloo.ca/csteachers/Institute2000/Java\\_Good,\\_Bad,\\_Ugly.pdf](https://cemc.math.uwaterloo.ca/csteachers/Institute2000/Java_Good,_Bad,_Ugly.pdf) (accessed August 29, 2004).

Whyte, M. M., D. M. Karolick, M. C. Nielsen, G. D. Elder and T. W. Hawley. 1995. Cognitive styles and feedback in computer assisted instruction. *Journal of Educational Computing Research*, 12 (2): 195-203.

Wie, C. R. 1998. Educational Java applets in solid state materials. *IEEE Transactions on Education*, 41 (4): Online. <http://www.ewh.ieee.org/soc/es/Nov1998/14/begin.htm>.

Wiedenbeck, S., V. Fix and J. Scholtz. 1993. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International of Man-Machine Studies*, 39 (5): 793-812.

Wiedenbeck, S., V. Ramalingam, S. Sarasamma and C. L. Corritore. 1999. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11 (3): 255-282.

Wiggins, M. 1998. An overview of program visualization tools and systems. In *ACM Southeast Regional Conference*, 194-200: ACM Press.

Wilson, J. D., N. Hoskin and J. T. Nosek. 1993. The benefits of collaboration for student programmers. In *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, 160-164. Indianapolis, Indiana, United States: ACM Press.

Winslow, L. E. 1996. Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28 (3): 17-22.

Wittry, D. and D. E. Poll. 2005. *JUnit Auto-Matic Tester*. <http://www.jamtester.com/> (accessed January 20, 2006).

- Wolz, U. and E. Koffman. 1999. simpleIO: a Java package for novice interactive and graphics programming. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, 139 - 142. Cracow, Poland: ACM Press.
- Wood, D., J. S. Bruner and G. Ross. 1976. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, 17(2): 89-100.
- Woods, P. J. and J. R., Warren. 1995. Rapid prototyping of an intelligent tutoring system. In *Proceedings of Twelfth Annual Conference of the Australian Society for Computers in Learning in Tertiary Education (ASCILITE95)*. Melbourne, Australia.
- Woolf, B. P., J. Beck, C. Eliot and M. Stern. 2001. Growth and maturity of intelligent tutoring system: a status report. In *Smart machines in education: the coming revolution in educational technology*, 99-144. Cambridge, MA, USA: MIT Press.
- Wulf, T. 2005. Constructivist approaches for teaching computer programming. In *Proceedings of the 6th conference on Information technology education*, 245 - 248. Newark, New Jersey: ACM Press.
- Xu, S. and Y. S. Chee. 1999. SIPLeS-II: an automatic program diagnosis system for programming learning environments. In *Artificial Intelligence in Education*, 397-404: IOS Press.
- Xu, S. and Y. S. Chee. 2003. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29 (4): 360-384.
- Yang, W. 1991. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21 (7): 739-755.
- Yousoof, M., M. Sapiyan and K. Kamaluddin. 2006. Reducing cognitive load in learning computer programming. *Transactions on engineering, computing and technology*, 12: 259 - 262. <http://www.enformatika.org/data/v12/v12-49.pdf>.
- Zafer, A. 2001. NetEdit: A Collaborative Editor, Master Thesis, Virginia Polytechnic Institute and State University, Blacksburg.
- Zaidman, M. 2004. Teaching defensive programming in Java. *Computing Sciences in Colleges*, 19 (3): 33-43.
- Zin, A. M. and E. Foxley. 1991. Automatic program assessment system. In *National Workshop on CASE Tools*. Sadar Patel University, Vidyanagar, Gujarat India.
- Ziring, N. 2001. *Java Mistakes Page*. <http://users.erols.com/ziring/java-npm.html> (accessed October 20, 2001).