Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

AN INTEGRATED ENVIRONMENT FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT

by
Fadi Pierre Deek

A framework for an integrated problem solving and program development environment that addresses the needs of students learning programming is proposed. Several objectives have been accomplished: defining the tasks required for program development and a literature review to determine the actual difficulties involved in learning those tasks. A comprehensive study of environments and tools developed to support the learning of problem solving and programming was then performed, covering programming environments, debugging aids, intelligent tutoring systems, and intelligent programming environments. This was followed by a careful analysis and critique of these systems, which uncovered the limitations that have prevented them from accomplishing their goals.

Next, an extensive study of problem solving methodologies developed in this century was carried out and a *common model* for problem solving was produced. The tasks of program development were then integrated with the common model for problem solving. Then, the cognitive activities required for problem solving and program development were identified and also integrated with the common model to form a *Dual Common Model for problem Solving and Program Development.* This dual common model was then used to define the functional specifications for a problem solving and

program development environment which was designed, implemented, tested, and integrated into the curriculum.

The development of the new environment for learning problem solving and programming was followed by the planning of a cognitively oriented assessment method and the development of related instruments to evaluate the *process* and the *product* of problem solving. A detailed statistical experiment to study the effect of this environment on students' problem solving and program development skills, including system testing by protocol analysis, and performance evaluation of students based on research hypotheses and questions, was also designed, implemented and the result reported.

AN INTEGRATED ENVIRONMENT FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT

by
Fadi Pierre Deek

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

October 1997

Copyright © 1997 by Fadi Pierre Deek
ALL RIGHTS RESERVED

APPROVAL PAGE

An Integrated Environment for Problem Solving and Program Development

Fadi Pierre Deek

Dr. James A. McHugh, Dissertation Advisor — Date
Professor of Computer and Information Science, NJIT

Dr. Roxanne Hiltz, Committee Member — Date
Distinguished Professor of Computer and Information Science, NJIT

Dr. Michael Hinchey, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Peter A. Ng, Committee Member — Date
Professor and Chair of Computer and Information Science, NJIT

Dr. Wilhelm Rossak, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Murray Turoff, Committee Member — Date
Distinguished Professor of Computer and Information Science, NJIT

Dr. Thomas Marlowe, Committee Member — Date
Visiting Professor of Computer and Information Science, NJIT
and Professor of Mathematics and Computer Science, Seton Hall University

Dr. Howard Kimmel, Committee Member — Date
Professor of Chemistry and
Assistant VP for Academic Affairs / Pre-College Programs, NJIT

Dr. Naomi Rotter, Committee Member — Date
Professor of Management, NJIT

BIOGRAPHICAL SKETCH

Author: Fadi Pierre Deek

Degree: Doctor of Philosophy in Computer and Information Science

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer and Information Science,
  New Jersey Institute of Technology, Newark, NJ, 1997

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1986

- Bachelor of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1985

Major: Computer and Information Science

Related Publications:

"*Tools for Problem Solving and Program Development*", Proceedings of the 14th International Conference on Technology and Education, *in press*, 1997. (With J. McHugh, and M. Hinchey).

"*On the Evaluation of a Problem Solving and Program Development Environment*", IEEE Proceedings of the Frontiers in Education Conference, *in press*, 1997. (With J. McHugh, R. Hiltz, N. Rotter and H. Kimmel).

"*The Delivery of an Introductory Course in Computer Science Through the Virtual Classroom*", IEEE Proceedings of the Frontiers in Education Conference, pp. 958-961, 1996. (With M. Deek and H. Kimmel).

"*First Things First: Problem Solving Before Programming*", INPUT: A Newsletter for Computer Science Educators, John Wiley & Sons Inc., Number 3, Winter 1996.

"*Instructional Technology: A Tool or a Panacea*", Journal of Science Education and Technology, vol. 4, No. 4, pp. 327-332, 1995. (With H. Kimmel)

"*Enhancing the Delivery of Computer Science Instruction for first Year Engineering Curriculum*", Proceedings of World Conference on Engineering Education, pp. 121-124, 1995. (With H. Kimmel).

"*Teaching for Understanding: Redesigning Introductory Courses to focus on the Learner*", IEEE Proceedings of the Frontiers in Education Conference, pp. 336-341, 1994. (With H. Kimmel).

"*Educational Reform: Integration of Technologies and Methodologies in Content Areas*", IEEE Proceedings of the 3rd International Conference for Systems Integration, pp. 124-133, 1994. (With H. Kimmel).

"*A Computer-Assisted Instruction System for Testing and Learning*", Proceedings of the 10th International Conference on Technology and Education, pp. 1095-1097, 1993. (With C. Wingert).

Other Publications:

"*The Management of Distance Education Using Computerized Tools for Instruction*", Electronic Proceedings of the 18th International Council for Distance Education World Conference, 10 pages, 1997. (With A. Lippel).

"*Technology and Hands-On Strategies for Teaching Science and Mathematics to the Special Education Population*", Journal of Information Technology and Disabilities, vol. 3, No. 2, Article 3, 6 pages, 1996. (with H. Kimmel and L. Frazer).

"*ECHOES: A Proposal for Spatializing On-Line Learning Environments*", Proceedings of the 8th International Conference on Systems Research, Informatics and Cybernetics, pp. 143-153, 1996. (with P. Anders).

"*Technology Support for the Enhancement of Science and Mathematics in the Special Education Population*", Proceedings of the 13th International Conference on Technology and Education, pp. 184-186, 1996. (With H. Kimmel and M. Deek).

"*Managing Technology Integration and School Districts limitations*", Proceedings of the 13th International Conference on Technology and Education, pp. 187-189, 1996. (With H. Kimmel and M. Deek).

"*Using the System for Educational Learning and Feedback (SELF) in a Sociology Telecourse*", in M.T. Keeton, B. Mayo-Wells, J. Porosky, and B.G. Sheckley (Eds.), Efficiency in Adult Higher Education: A Practitioners' Handbook, Chapter 7, Case 7.2, pp. 126-129. 1995. (With A. Lippel)

"*A Distance Learning Model for Advising Using Computerized Conferencing*", Academic Advising (ACAD) Network Electronic Journal, vol. 2, No. 1, article 4, 8 pages, 1995. (With M. Tress).

"*Integration of Simulations with the Internet for Distance Education*", Proceedings of 1st LAAS International Conference on Computer Simulation, pp. 128-135, 1995. (With R. Kushwaha et al.).

"*A Multimedia Laboratory and Training Program to Promote Technology Integration in Elementary Education*", Proceedings of World Conference on Educational Multimedia and Hypermedia, p. 768, 1995. (With H. Kimmel and M. Deek).

"*Facilitating Technology Integration in Kindergarten Through Eighth Grade Curriculum*", Proceedings of the 12th International Conference on Technology and Education, pp. 314-316, 1995. (With M. Deek and H. Kimmel).

"*An Electronic Model for Advising in a Distance Learning Environment*", Proceedings of the 12th International Conference on Technology and Education, pp.680-682, 1995.

"*The System for Educational Learning and Feedback (SELF) for Distance Learning*", Proceedings of the 11th International Conference on Technology and Education, pp. 873-875, 1994. (With A. Lippel).

"*Changing the Students' Role: From Passive Listeners to Active Participants*", IEEE Proceedings of the Frontiers in Education Conference, pp. 321-325, 1993. (With H. Kimmel).

"*A Modular Design for a Telecourse in Computer Science*", IEEE Proceedings of the Frontiers in Education Conference, p. 870, 1993. (With A. Lippel).

"*Enhancing Televised Distance Learning Using Computerized Technology*", Proceedings of the Second International Symposium on Telecommunications in Education, pp. 277-279, 1993. (With T. Terry).

"*Teaching Assistants: Ensuring A Quality Experience*", IEEE Proceedings of the Frontiers in Education Conference, pp. 765-769, 1992. (With M. Tress).

"*Distance Learning: A Pedagogical Alternative*", IEEE Proceedings of the Frontiers in Education Conference, p. 847, 1992. (With A. Lippel).

"*Advising the Off-Campus Learner Via Computerized Conferencing*", Proceedings of the National Issues in Higher Education Conference Series on Quality in Off-Campus Credit Programs: Bridging the Distance, vol. 40, pp.294-300, 1992. (With M. Tress and M. Maher).

"*Implementing an Effective Freshman Teaching Assistant Program: Selection, Training, and Monitoring*", Proceedings of the 11th annual Freshman Year Experience Conference: Science and Technological Education, pp. 11-12, 1992. (With M. Tress).

"*Improving Freshman Success Rate Through an Early Warning System*", Proceedings of the 11th annual Freshman Year Experience Conference: Science and Technological Education, pp. 13-14, 1992. (With J. Valyo and M. Tress).

"*Maintaining the Momentum of Excellence: The Honors Program at NJIT*", Proceedings of the 11th annual Freshman Year Experience Conference: Science and Technological Education, pp. 15-16, 1992. (With D. Donahue and R. Baker).

"*Computer-Assisted Advising Relief System*", Proceedings of the 8th Annual NACADA Mid-Atlantic Regional Conference, pp. 5-6, 1992. (With M. Tress).

"*Quality Distance Learning for the Quantitative Sciences: A Collaborative Approach*", Proceedings of the 9th International Conference on Technology and Education, pp. 1484-1486, 1992. (With A. Lippel).

Related Presentations:

"*Problem Solving and Program Development by Example.*" Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1997.

"*Problem Solving and Programming Using Asynchronous Learning Networks.*" African Virtual University Inauguration Meeting (Via satellite from NJIT Candid Classroom to education officials of French speaking African countries), Senegal, Africa. April 1997.

"*Introductory Computer Science Sequence: Can We Make it Work?*" New Jersey Community College Computer Consortium (NJCCCC), Annual Fall Meeting, Edison, New Jersey. October 1996.

"*Recommendations on Content for H.S. Computer Science Curriculum and Professional Development.*" Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1996.

"*Teaching Methodologies for Problem Solving and Programming.*" Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1995.

Other Presentations:

"*Academic Advising for the Virtual University*". Sixth Annual New Jersey Advisors Conference, Newark, New Jersey. June 1997.

"*The Emerging Careers and Curricula of Computing*". New Jersey Transfer Counselors' Association (NJTCA), Annual Spring Meeting, Jersey City, New Jersey. May 1997.

"*Asynchronous Learning Networks: An Example for Distance Education*". Serving The Adult Collegian (STAC), Annual Meeting, Trenton, New Jersey. April 1997.

"*Professional Development for Pre-College Computer Science Education*". Third Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1997.

"*From Pascal to C++: Preparing for the Change in Computer Science Advanced Placement Courses*". Third Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1997.

"*On-Line Advising Center*". Fifth Annual New Jersey Advisors Conference, Jersey City, New Jersey. June 1996.

"*A Report on Computer Science Education in the Secondary Schools*". Second Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1996.

"*Technology and Hands-On Strategies for Teaching Science and Mathematics to the Special Education Population*". 11th Annual International Conference on Technology and Persons With Disabilities, Los Angeles, CA. March 1996.

"*Computer Science Education and the ACM Recommendations*". First Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1995.

"*Computer Science as a High School Subject Matter*". First Conference on Computer Science Education in the Secondary Schools, Newark, New Jersey. May 1995.

"*Active Learning: Challenges and Discoveries*". National Science Teachers Association National Convention, Philadelphia, Pennsylvania. March 1995.

"*Technology and Academic Advising*". Third Annual New Jersey Advisors Conference, New Brunswick, New Jersey. June 1994.

"*Applications of The system for Educational Learning and Feedback (SELF)*". Conference on International Distance Education: A Vision for Higher Education, University park, Pennsylvania. June 1994.

"*Distance Learning: Effective Access to College Courses, Enrichment and Professional Development*". Educational Technology Conference, Long Branch, New Jersey. February 1994.

"*A Report on Using the System for Educational Learning and Feedback (SELF) in a Sociology Telecourse*". Research Seminars on Efficiency in Learning, College Park, Maryland. November 1993.

"*The System for Educational Learning and Feedback (SELF): A Computerized Testing and Tutoring Approach*". Technology-Based Engineering Education Consortium Conference, Melbourne, Florida. November 1992.

"*A Telecourse Design Methodology*". Eastern Educational Consortium Conference on Distance Learning, Newark, New Jersey. February 1992.

"*Computerizing the Campus*". Conference on Technology in the Freshman Year: Computers Across the Curriculum, New York, New York. May 1992.

"*Pedagogical Implications of Freshman Microcomputer Programs*". Decision Science Institute Conference - Innovation Track Symposium, Miami, Florida. October 1991.

Dedicated to

Maura Ann

who made this dissertation possible through love, help and understanding

ACKNOWLEDGMENT

I thank God for providing me with the strength, determination and persistence I needed to complete this journey.

I offer my gratitude to Dr. McHugh, my advisor, for his extraordinary tutelage throughout my academic career at NJIT. This began when, as an undergraduate student, I chose him as my senior project advisor. He guided me into graduate school and offered me my first teaching job as a TA. I later worked with him on my masters project while my advisor was on sabbatical leave. Toward the end of my graduate study he offered me my first full-time teaching job as a Special Lecturer. He continuously encouraged me to go on and get this degree, and was there at every step of the way. I thank him for being an excellent advisor, a great mentor, colleague and a friend.

I express my sincere appreciation to my committee members, Dr. Hiltz, Dr. Hinchey, Dr. Kimmel, Dr. Marlowe, Dr. Ng, Dr. Rossak Dr. Rotter and Dr. Turoff for their outstanding guidance during this research. I was fortunate to have such diverse expertise and interest among the committee members. I enjoyed the many hours of discussions I had with Dr. Kimmel about this research while carpooling to and from NJIT. I thank Dr. Ng, the Chairperson of the Computer and Information Science Department, for providing the supporting environment that enabled me to complete this degree. I am grateful for the unwavering support I received, and continue to receive, from Dr. Thomas, NJIT Provost, both at the academic and professional levels. I thank Dr. John Poate, Dean of the College of Science and Liberal Arts, for his support and interest in my work.

My friend and colleague Bill Anderson, Dean of Admissions at NJIT, read the dissertation and offered excellent feedback. I thank him for all the support and encouragement he has given me. I thank Dr. Sol Magzamen of the Academic Foundations Department at Rutgers-Newark, for providing me with excellent pointers to the literature on problem solving. I am grateful to Dr. Rose Dios for her careful explanation of statistical concepts and for her help in interpreting the evaluation results.

Many NJIT students are to be thanked as well. Richard Ordowich for implementing the SOLVEIT prototype and for his friendship; Michael Butrym, Ann Farrow, and Yaro Zajac for their excellent comments on current systems. I also thank fellow doctoral student Raquel Benbunan for sharing research literature and for providing excellent feedback on the evaluation component of this dissertation. My sincere appreciation to the three Teaching Assistants for the course: Edward Maybert, Michael Piccolo, and Tifanie Levy who demonstrated profound dedication to the students and interest in their teaching responsibilities. A special thanks for Tifanie's efforts in statistical analysis. Also, a thank you for all the students that I have taught in the first course on Problem Solving and Programming for over a decade is owed. They were the inspiration.

I thank my friends, the administrators and support staff in the CIS Department, Michael Tress, Barbara Harris, Carole Poth, Rosemarie Giannetta, and Michelle Craddock for their encouragement. Michael also read earlier versions of the dissertation and provided excellent comments. I also thank Ray Bolden, another NJIT friend, for his caring.

Finally, I am certain that without the support of my family I would not be able to claim this accomplishment. This is the result of the love and prayers of my parents Pierre and Therese Deek and the love and encouragement of my brothers Wadih, Bassam, Sami and Roland Deek. My father-in-law and my mother-in-law John and Nora McShane have also been a generous source of support and encouragement. My brother-in-law Lawrence also read the dissertation and made excellent suggestions. My wife Maura, to whom this dissertation is dedicated, is a continual source of love and sacrifice. She has been there from day-one and has given unconditional love, help, and understanding. I have the rest of my life to say *"thank you"*. To our three kids: Matthew, Andrew, and Rebecca I say "*no more going down to the basement to do my homework; at least not every night!*".

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 DIFFICULTIES IN LEARNING PROGRAMMING 4

1.1.1 Deficiencies in Problem Solving Strategies and Tactical Knowledge 4

1.1.2 Ineffective Pedagogy of Programming Instruction 5

1.1.3 Misconceptions About Syntax, Semantics and Pragmatics 5

1.2 CURRENT SYSTEMS FOR SUPPORTING THE TASKS OF PROGRAM DEVELOPMENT 6

1.2.1 Programming Environments 6

1.2.2 Debugging Aids 8

1.2.3 Intelligent Tutoring Systems 8

1.2.4 Intelligent Programming Environments 8

1.3 LIMITATIONS OF CURRENT SOLUTIONS 8

1.4 PROBLEM SOLVING AND PROGRAM DEVELOPMENT 11

1.4.1 Problem Solving 11

1.4.2 Program Development 12

1.5 THE SOFTWARE PROCESS 13

1.5.1 The Software Process and Relationship with Problem Solving Activities 15

1.6 THEORY OF PROBLEM SOLVING, COGNITIVE SCIENCE AND RELATIONSHIP TO SYSTEM SPECIFICATION 16

1.6.1 The Specification Oriented Language in Visual Environment for Instruction Translation 17

1.7 EXPERIMENTAL DESIGN 19

1.8 RESEARCH CONTRIBUTIONS 20

Chapter	Page

1.9 DISSERTATION OUTLINE 22

2 RELATED RESEARCH: A SURVEY OF ENVIRONMENTS AND TOOLS FOR LEARNING PROGRAMMING 24

2.1 PROGRAMMING ENVIRONMENTS 26

2.1.1 Pict 29

2.1.2 PECAN 30

2.1.3 SCHEMACODE 31

2.1.4 DSP 32

2.1.5 AMETHYST 33

2.1.6 University of Washington Illustrating Compiler 34

2.1.7 BACCII 36

2.1.8 ASA 37

2.1.9 SUPPORT 38

2.1.10 STRUEDI 39

2.1.11 Example-Based Programming System 40

2.1.12 Software Design Laboratory 42

2.1.13 MEMO-II 43

2.2 DEBUGGING AIDS 44

2.2.1 LAURA 45

2.2.2 The Debugging Assistant 46

2.2.3 GENIUS 48

2.2.4 VIPS 49

2.2.5 Lens 50

Chapter **Page**

2.3 INTELLIGENT TUTORING SYSTEMS 50

2.3.1 The BASIC Instructional Program 52

2.3.2 The LISP Tutor 53

2.3.3 PROUST 55

2.3.4 The ACT Programming Tutor 56

2.4 INTELLIGENT PROGRAMMING ENVIRONMENTS 57

2.4.1 Bridge 58

2.4.2 Graphical Instruction in LISP 61

2.4.3 Intelligent Tutor, Environment and Manual for Introductory Programming 62

2.4.4 DISCOVER 63

2.4.5 Episodic Learning Model Programming Environment 65

2.4.6 Capra 67

2.4.7 INTELLITUTOR 68

2.5 CONCLUSION 70

3 ANALYSIS AND CRITIQUE OF EXISTING APPROACHES 72

3.1 FUNCTIONAL WEAKNESSES AND PRACTICAL DEFICIENCIES 72

3.2 FUNCTIONAL WEAKNESSES 73

3.2.1 Absence of Problem Solving/Software Engineering Frameworks 74

3.2.2 Overemphasis on Language Syntax 79

3.2.3 Inadequate User Interface 83

3.2.4 Incomplete Rules-and-Errors Knowledge Bases 83

3.2.5 Simplicity of Problem Domain 85

3.3 PRACTICAL DEFICIENCIES 85

Chapter	Page

3.3.1 Limited Classroom Evaluation 86

3.3.2 Failure to Integrate the Tools into the Curriculum 88

3.3.3 Impede Creativity and Development of Higher Order Thinking Skills 89

3.4 SUMMARY 91

4 PROBLEM SOLVING, PROGRAM DEVELOPMENT 95

4.1 PROBLEM SOLVING 95

4.1.1 The Terminology of Problem Solving 97

4.1.2 Categories of Problems 99

4.2 PROBLEM SOLVING METHODS 101

4.2.1 Early Models 102

4.2.2 Enhancements to Earlier Models 103

4.2.3 Recent Methods 106

4.3 A COMMON METHOD FOR PROBLEM SOLVING 109

4.3.1 Formulating the Problem 110

4.3.2 Planning the Solution 111

4.3.3 Implementing the Solution 112

4.3.4 Verifying and Presenting the Results 113

4.4 PROGRAM DEVELOPMENT 114

4.4.1 Program Development Tasks 114

4.5 A COGNITIVE MODEL FOR PROBLEM SOLVING 118

4.6 A DUAL COMMON MODEL FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT 124

4.6.1 Formulating the Problem 125

4.6.2 Planning the Solution 130

Chapter **Page**

4.6.3 Designing the Solution 135

4.6.4 Translation 139

4.6.5 Testing 142

4.6.6 Delivery 145

4.7 MONITORING THE PROBLEM SOLVING PROCESS 145

4.8 CONCLUSION 147

5 AN ENVIRONMENT FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT: SPECIFICATIONS FOR THE DUAL COMMON MODEL 151

5.1 THE SPECIFICATION ORIENTED LANGUAGE IN VISUAL ENVIRONMENT FOR INSTRUCTION TRANSLATION 152

5.1.1 The Process 154

5.1.2 The Tools 156

5.2 SOLVEIT ARCHITECTURE 158

5.2.1 Tools for Problem Formulation 159

5.2.2 Tools for Solution Planning 161

5.2.3 Tools for Solution Design 162

5.2.4 Tools for Solution Delivery 169

5.2.5 Common Tools 169

5.3 A WALK THROUGH SOLVEIT 171

5.3.1 Formulating the Problem 175

5.3.2 Planning the Solution 180

5.3.3 Designing the Solution 183

5.3.4 Translating the Solution 190

5.3.5 Testing the Solution 191

Chapter	Page

5.3.6 Delivering the Solution 192

5.4 IMPLEMENTATION OF SOLVEIT 192

5.5 SUMMARY 193

6 EXPERIMENTAL DESIGN: TESTING AND EVALUATION 195

6.1 INTRODUCTION 195

6.2 SYSTEM TESTING 196

6.2.1 User Testing 197

6.2.2 Protocol Analysis 198

6.3 EVALUATION METHOD 199

6.3.1 Hypotheses and Research Questions 200

6.3.2 Subjects 202

6.3.3 Design 203

6.3.4 Instrumentation 205

6.3.5 Data Collection 207

6.4 ASSESSING THE EFFECT OF THE SOLVEIT ENVIRONMENT ON STUDENTS' PROBLEM SOLVING AND PROGRAM DEVELOPMENT SKILLS 209

6.4.1 Process Measures 210

6.4.3 Subjective Measures 229

7 EXPERIMENTAL RESULTS AND ANALYSIS 235

7.1 DESCRIPTIVE STATISTICS 236

7.1.1 Demographic Information from Fall 1996 Semester 237

7.1.2 A Comparison of Students in the Control and Experimental Groups 238

Chapter **Page**

7.2 RELIABILITY AND VALIDITY OF PERFORMANCE ASSESSMENT INSTRUMENTS 238

7.3 TEST OF PROCESS HYPOTHESES 240

7.3.1 Results from Fall 1996 Semester 241

7.4 TEST OF PRODUCT RESEARCH QUESTIONS 246

7.5 TEST OF SUBJECTIVE RESEARCH QUESTIONS 259

7.5.1 Results from Fall 1996 Semester 261

7.6 PREVIEW OF RESULTS FROM SPRING 1997 SEMESTER 264

7.7 SUMMARY OF HYPOTHESES AND RESEARCH QUESTIONS TEST 265

8 CONCLUDING REMARKS AND FUTURE WORK 268

8.1 SUMMARY OF EVALUATION RESULTS 268

8.1.1 A Closer-Look at the Results 268

8.1.2 Experimental Problems 270

8.1.3 Effects Related to the Experimental Design 271

8.2 ENHANCEMENTS TO SOLVEIT 271

8.2.1 Enhancing Functionality 272

8.2.2 Extending the Approach to Subsequent Courses 273

8.2.3 Restructuring Functionality for Alternative Programming Paradigms 274

8.2.4 Restructuring Functionality for Distance Learning 274

8.3 A PLAN FOR LONG-TERM EVALUATION 275

8.3.1 Learning Outcome Measures 275

8.4 SUMMARY OF CONTRIBUTIONS 280

Chapter **Page**

APPENDIX 1 SURVEY INSTRUMENTS 283

APPENDIX 2 COURSE SYLLABUS AND RELATED MATERIAL 296

APPENDIX 3 SAMPLE ASSIGNMENTS, QUIZZES AND EXAMS 301

REFERENCES 311

LIST OF TABLES

Table Page

1 Program development tasks and their associated activities 13

2 The Software process and associated activities 14

3 Relationship between the software process and Polya's problem solving stages 16

4 Classifications of systems devised to assist with programming 26

5 Examples of visually-based programming environments 28

6 Examples of text-based programming environments 29

7 Examples of debugging aids 45

8 Examples of intelligent tutoring systems 52

9 Examples of intelligent programming environments 58

10 Functional weaknesses 74

11 Focus of problem solving and program development process in visual oriented programming environments 76

12 Focus of problem solving and program development process in text oriented programming environments 77

13 Focus of problem solving and program development process in debugging aids 77

14 Focus of problem solving and program development process in intelligent tutoring systems 78

15 Focus of problem solving and program development process in intelligent programming environments 78

16 Practical deficiencies 86

17 Focus of problem solving and program development in SOLVEIT 151

18 The SOLVEIT model and relationship to the software process 156

Table **Page**

19 DATA DICTIONARY ENTRY 166

20 DATA DESCRIPTION 183

21 THE TESTING PLAN 196

22 THE EVALUATION APPROACH 199

23 PROCESS HYPOTHESES 201

24 PRODUCT RESEARCH QUESTIONS 201

25 SUBJECTIVE RESEARCH QUESTIONS 202

26 ASSIGNMENTS OF SECTIONS TO CONDITIONS 203

27 HYPOTHESES' (PROCESS) VARIABLES 204

28 RESEARCH QUESTIONS' (PRODUCT) VARIABLES 204

29 RESEARCH QUESTIONS' (SUBJECTIVE) VARIABLES 205

30 HYPOTHESES' (PROCESS) INSTRUMENTS 206

31 RESEARCH QUESTIONS' (PRODUCT) INSTRUMENTS 207

32 RESEARCH QUESTIONS' (SUBJECTIVE) INSTRUMENTS 207

33 CONSENT/PRE/POST TEST INSTRUMENTS 207

34 DATA COLLECTION FOR HYPOTHESES (PROCESS) 208

35 DATA COLLECTION FOR RESEARCH QUESTIONS (PRODUCT) 208

36 DATA COLLECTION FOR RESEARCH QUESTIONS (SUBJECTIVE) 209

37 DEFINITIONS OF DEPENDENT VARIABLES' CATEGORIES 210

38 PROCESS VARIABLES 211

39 INSTRUMENT FOR ASSESSING STUDENTS' PROBLEM FORMULATION SKILLS 214

40 INSTRUMENT FOR ASSESSING STUDENTS' PLANNING SKILLS 216

41 INSTRUMENT FOR ASSESSING STUDENTS' DESIGN SKILLS 219

42 PRODUCT VARIABLES 220

Table		Page

43 INSTRUMENT FOR ASSESSING SOLUTION EFFICIENCY 223

44 INSTRUMENT FOR ASSESSING SOLUTION RELIABILITY 224

45 INSTRUMENT FOR ASSESSING SOLUTION READABILITY 226

46 INSTRUMENT FOR ASSESSING SOLUTION CORRECTNESS 228

47 INSTRUMENT FOR ASSESSING SHORT QUIZZES AND EXERCISE QUESTIONS 229

48 SUBJECTIVE VARIABLES 230

49 INSTRUMENT FOR STUDENTS' SELF-ASSESSMENT REPORTS 232

50 DISTRIBUTION OF SUBJECTS ACROSS CONDITIONS 236

51 INTER-RATER RELIABILITY FOR THE THREE GRADERS 239

52 ANOVA AND MEANS REPORT FOR PROBLEM FORMULATION 243

53 ANOVA AND MEANS REPORT FOR SOLUTION PLANNING 244

54 ANOVA AND MEANS REPORT FOR SOLUTION DESIGN 245

55 ANOVA AND MEANS REPORT FOR OVERALL PROBLEM SOLVING AND PROGRAM DEVELOPMENT SKILLS 246

56 ANOVA AND MEANS REPORT FOR MIDTERM QUESTION 1.1 249

57 ANOVA AND MEANS REPORT FOR MIDTERM QUESTION 1.2 250

58 ANOVA AND MEANS REPORT FOR MIDTERM QUESTION 1.3 251

59 ANOVA AND MEANS REPORT FOR MIDTERM QUESTION 1.4 252

60 ANOVA AND MEANS REPORT FOR FINAL QUESTION 1.1 253

61 ANOVA AND MEANS REPORT FOR FINAL QUESTION 1.2 254

62 ANOVA AND MEANS REPORT FOR FINAL QUESTION 1.3 255

63 ANOVA AND MEANS REPORT FOR SECTION ONE OF THE MIDTERM 256

64 ANOVA AND MEANS REPORT FOR SECTION ONE OF THE FINAL 257

65 ANOVA AND MEANS REPORT FOR OVERALL FINAL GRADE 258

Table **Page**

66 LETTER GRADE DISTRIBUTION FOR FALL 96 SEMESTER 259

67 ANOVA FOR QUESTION ON POST-TEST QUESTIONNAIRE DEALING WITH VERBALIZATION 262

68 ANOVA FOR QUESTION DEALING WITH DISCUSSING PROBLEM SOLVING AND PROGRAMMING WITH STUDENTS OUTSIDE OF CLASS 263

69 ANOVA AND MEANS REPORT FOR OVERALL TOTAL GRADE FOR SPRING 97 SEMESTER 264

70 LETTER GRADE DISTRIBUTION FOR SPRING 97 SEMESTER 265

71 SUMMARY OF HYPOTHESES AND RESEARCH QUESTIONS TEST 266

LIST OF FIGURES

Figure **Page**

1 A PROGRAMMING SESSION IN TURBO PASCAL 7

2 THE BEGINNING OF A CODING SESSION FOR FUNCTION 'CREATE-LIST' USING THE LISP TUTOR (REDRAWN FROM ANDERSON, CORBETT, KOEDINGER, & PELLETIER, 1995)....80

3 AN IN-PROGRESS PROLOG PROGRAM, USING MEMO II, TO FIND WHETHER A NATURAL NUMBER IS EVEN OR ODD (REDRAWN FROM FORCHERI AND MOLFINO, 1994) 82

4 TRANSFORMATION OF A PROBLEM STATEMENT INTO A SOLUTION 96

5 THE DUAL COMMON MODEL FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT 125

6 THE INITIAL TASK OF PROBLEM FORMULATION 127

7 EFFECT OF VERBALIZATION ON PROBLEM FORMULATION 128

8 COGNITIVE SYSTEM OF PROBLEM FORMULATION STAGE 130

9 REFINEMENT OF PROBLEM INTO SUBPROBLEMS 133

10 COGNITIVE SYSTEM OF THE SOLUTION PLANNING STAGE 135

11 ORGANIZATION AND SEQUENCING OF SUBPROBLEMS 137

12 COGNITIVE SYSTEM OF THE SOLUTION DESIGN STAGE 139

13 COGNITIVE SYSTEM OF THE TRANSLATION STAGE 142

14 COGNITIVE SYSTEM OF THE TESTING STAGE 144

15 AN EXTENDED-LEARNING ENVIRONMENT 150

16 THE SOLVEIT MODEL 155

17 THE TOOLS IN TRADITIONAL PROGRAMMING ENVIRONMENTS 157

18 THE TOOLS IN SOLVEIT 157

19 ARCHITECTURE OF THE SOLVEIT ENVIRONMENT 159

20 INITIAL REPRESENTATION OF THE STRUCTURE CHART 163

Figure		Page

21 REFINEMENT OF THE STRUCTURE CHART 164

22 A TEXT VIEW OF THE STRUCTURE CHART 165

23 STRUCTURE CHART AND DATA FLOW IN TEXT VIEW 167

24 MODULE LOGIC DEVELOPMENT 168

25 SOLVEIT: AN INTEGRATED PROBLEM SOLVING PROGRAM DEVELOPMENT ENVIRONMENT 171

26 WORKFLOW IN PROBLEM SOLVING STAGES 172

27 WORKFLOW IN PROGRAM DEVELOPMENT STAGES 173

28 IDENTIFYING INFORMATION 175

29 OUTCOME OF PROBLEM FORMULATION 176

30 PROBLEM DESCRIPTION 177

31 VERBALIZATION IN PROBLEM FORMULATION 178

32 INFORMATION ELICITATION 179

33 OUTCOME OF SOLUTION PLANNING 181

34 SOLUTION STRATEGY 181

35 GOAL DECOMPOSITION 182

36 OUTCOME OF SOLUTION DESIGN 184

37 STRUCTURE CHART - FIRST LEVEL REFINEMENT 185

38 STRUCTURE CHART - SECOND-LEVEL REFINEMENT 185

39 STRUCTURE CHART AND DATA FLOW 186

40 MODULE SPECIFICATION FOR INPUT SUBPROBLEM 188

41 MODULE SPECIFICATION FOR COMPUTATION SUBPROBLEM 188

42 MODULE SPECIFICATION FOR OUPUT SUBPROBLEM 189

43 ALGORITHMIC LOGIC 189

Figure **Page**

44 OUTCOME OF SOLUTION TRANSLATION 190

45 OUTCOME OF SOLUTION TESTING 191

46 DELIVERY OF COMPLETE SOLUTION 192

47 LEARNING OUTCOME VARIABLES 276

CHAPTER 1

INTRODUCTION

The question of how best to facilitate the teaching and learning of programming has been addressed by educators and researchers for some time, and continues to be addressed with growing interest (Weinberg, 1971; Papert, 1980; Shneiderman, 1980; Soloway, Ehrlich, Bonar, & Greenspan, 1982; Mayer, 1988; Shackelford & Badre, 1993; Ebrahimi, 1994). It is now widely agreed that the ability to write programs, and the difficulties encountered therein, extend far beyond learning the syntax of a specific language (Linn & Dalbey, 1985; Perkins, Schwartz, & Simmons 1988; Scholtz & Weidenbeck, 1992, 1993; Weidenbeck, Fix, & Scholtz, 1993; Ennis, 1994). Similarly, developments in programming environments, tools, and languages to support learning problem solving and programming, carried out over the last two decades, have placed increasing emphasis on investigating such questions as: How can programming and problem solving be effectively taught to students? What are the tools and environments necessary to facilitate the teaching and learning of programming? What are the underlying cognitive skills required in programming and what are the cognitive effects of learning programming (Weinberg, 1971; Shneiderman, 1980; Mayer, 1988; Hoc, Green, Samurcay, & Gilmore, 1990; Lemut, du Boulay, & Dettori, 1993).

Given the trend of the research, it is surprising to note that introductory courses on programming still often focus on language syntax, a role which students, and frequently teachers, perceive as appropriate for the first course in computer science (Levy, 1995). This approach is fostered by programming textbooks which present the subject from a

language construct view, ignoring the fundamentals not only of design methodology but also of problem solving concepts. The recurrent debate over the appropriate choice of a programming language for the first course (Noon, 1994) also promotes this emphasis on syntax, an issue that should be incidental to the first course. This misemphasis persists despite a series of computing curricula recommendations by the Association for Computing Machinery (ACM) and the Computer Society of the Institute for Electrical and Electronics Engineers (IEEE-CS) (ACM Curriculum 1968, 1978; ACM/IEEE-CS Computing Curricula, 1991), and the work of noted computer scientists (Denning et al., 1989; Tucker, 1994). For example, the ACM/IEEE-CS 1991 Joint Curriculum Committee Report underscores that "programming is understood to denote the entire collection of activities that surround the description, development, and effective implementation of algorithmic solutions to well-specified problems." While not minimizing the importance of syntactical issues (Shneiderman, 1980; Rogalski & Samurcay, 1990, 1993), *research clearly indicates that the most fundamental obstacles to learning programming are related to its problem solving character* (Mayer, 1981; Perkins & Martin, 1986; Perkins, Hancock, Hobbs, Martin, & Simmons, 1986; Johnson, 1990; Navarat & Rozinajova, 1993; Ebrahimi, 1994; Ennis, 1994). Programming languages should be merely the vehicles by which problem solving is realized, and the common focus on syntax should be replaced by an emphasis on problem solving and program development.

This thesis addresses the critical interdependence between the problem solving process and program development. A primary goal is to make the process of solving problems and writing programs simpler, more organized, and more coherent for the

beginning student by providing a *theoretically well-founded* system which supports the entire process. The requirements of this system can be specified only after several objectives have been accomplished. First of all, one must carefully determine the actual difficulties involved in learning programming, since this identifies the needs that have to be addressed. Then, one must review and critique existing environments, tutoring and other systems with respect to how well they meet these needs. This critique will be based on difficulties in problem solving and programming, the research on which is introduced in this chapter. One must also critically survey the existing problem solving methodologies, and synthesize from these methods a unified model for problem solving, which can then be adapted to the particular requirements of program development. Finally, one must identify the cognitive skills pertinent at each stage of the problem solving and program development process, and define the proposed problem solving environment's functionality in a way that is consistent with the needed skills. Subsequent sections of this chapter: identify the difficulties involved in learning programming; overview the kinds of support systems that have been developed to date, and their limitations; briefly introduce problem solving methodology; present an analysis of the software process and its relationship to problem solving; preview the problem solving and cognitive justifications for the system specification; briefly describe the functionality of the proposed system, SOLVEIT; and preview the experimental design that will be used to evaluate the performance of the system.

1.1 Difficulties in Learning Programming

The teaching and learning of programming have been addressed by educators and researchers (Soloway, Ehrlich, Bonar, & Greenspan, 1982; Mayer, 1988; Shackelford & Badre, 1993). Students experience obstacles when learning programming, especially the first language (Mayer, 1981; Linn & Dalbey, 1985; Bereiter & Scardamalia, 1985; Perkins & Martin, 1986; Johnson, 1990; Weidenbeck, Fix, & Scholtz, 1993).

To understand the role of current systems and their inadequacy in responding to the needs of students, it is necessary to first understand the basis of students' difficulties with programming. There are three kinds of challenges students face when learning the tasks of program development: deficiencies in problem solving strategies and tactical knowledge; ineffective pedagogy of programming instruction, and misconceptions about syntax, semantics, and pragmatics of language constructs.

1.1.1 Deficiencies in Problem Solving Strategies and Tactical Knowledge

The ability to solve a problem requires aptitude beyond the syntax and semantics of a programming language (Linn and Dalbey, 1985; Perkins, Schwartz, & Simmons 1988; Weidenbeck, Fix, & Scholtz, 1993). The lack of basic problem solving competence, thinking skills, and transfer of knowledge to other domains is a prominent problem with novice programmers (Mayer, 1981; Perkins, Hancock, Hobbs, Martin, & Simmons, 1986). Errors in students' programs are commonly related to deficiencies in problem solving strategies and insufficient planning, not syntax (Scholtz & Weidenbeck, 1992, 1993; Anjaneyulu, 1994). Even those studies that uncovered novice difficulties with syntax concluded that more emphasis is needed in teaching planning and design strategies

(Soloway, Ehrlich, Bonar, & Greenspan, 1982) along with software engineering principles (Shackelford & Badre, 1993).

1.1.2 Ineffective Pedagogy of Programming Instruction

Students in introductory programming courses have difficulties solving problems. Even *simple* problems are the cause for major errors, often due to inadequate or misdirected teaching. Students are not taught the necessary skills and "have 'no choice' but to blur the distinction between design and implementation. If we take the basic principles of software engineering seriously, we should not be introducing students to programming in this way." (Shackelford & Badre, 1993). In addition, the curriculum should prepare the students to deal with the demands of program development by seeking the connection outside the programming domain (Perkins, Schwartz, & Simmons, 1988; Ennis, 1994).

1.1.3 Misconceptions About Syntax, Semantics and Pragmatics

Learning the elements of a programming language and its constructs is a significant task for programmers. Problem solving competence and understanding of the syntax, semantics, and pragmatics of a programming language constitute the foundation skills required to compose, comprehend, test and debug, document and modify programs. Novice students appear to lack understanding of the purpose, structure and use of the programming language constructs they study (Soloway, Ehrlich, Bonar, & Greenspan, 1982; Bereiter & Scardamalia, 1985; Perkins & Martin, 1986; Johnson, 1990; Ebrahimi, 1994).

1.2 Current Systems for Supporting the Tasks of Program Development

Both systems and methodologies have been developed to improve the learning and practice of programming. The development of tools to enhance the learning of programming began in the 70's with the initial introduction of Computer Aided Instruction (CAI) (Carbonell, 1970; Brown, Burgon & Bell, 1974; Barr, Beard & Atkinson, 1975). Methodologies were devised to aid in activities of program development with the introduction of structured programming and top-down design (Wirth, 1971; Dijkstra, 1976). Developments in the area of programming languages in general, in methodologies (Weinberg, 1971; Shneiderman, 1980; Mayer, 1988), and in tools (Hoc, Green, Samurcay, & Gilmore, 1990; Lemut, du Boulay, & Dettori, 1993), aimed at supporting the learning of programming and enhancing the efficacy of programmers, have since evolved considerably. This is illustrated in the subsequent development of intelligent systems (Anderson & Reiser, 1985; Johnson & Soloway, 1985; Corbett & Anderson, 1993), and programming environments for teaching (Brusilovsky, 1991; Ramadhan & du Boulay, 1993; Hohmann, Guzdial, & Soloway, 1992; Forcheri & Molfino, 1994).

The following breifly describes the four common classifications of these systems: programming environments, debugging aids, intelligent tutoring systems and intelligent programming environments.

1.2.1 Programming Environments

Traditional programming environments, such as Pascal and C compilers, consist of tools that can be used in program construction, compilation, testing and debugging: editors,

language compilers, pre-coded function libraries, linking loaders, parsers, tracers and debuggers. More current systems also include syntax-directed editors, code indentation, graphical user interface tools, dynamic compilation capabilities, powerful library systems and possibly code generators. Programming environments developed specifically for teaching may include additional facilities that allow the student to experiment with specific features of a programming language and some offer limited tutorial functions.

Students learning programming, using current tools, are accustomed to the entry point in any programming language environment being through the compiler's editor. As a result, when presented with a problem, students tend to reach for the keyboard and start coding. This creates an impression that the formulation of the solution to the problem starts by writing the code, a habit which must of course be altered. A typical tool bar is shown in *Figure* 1.

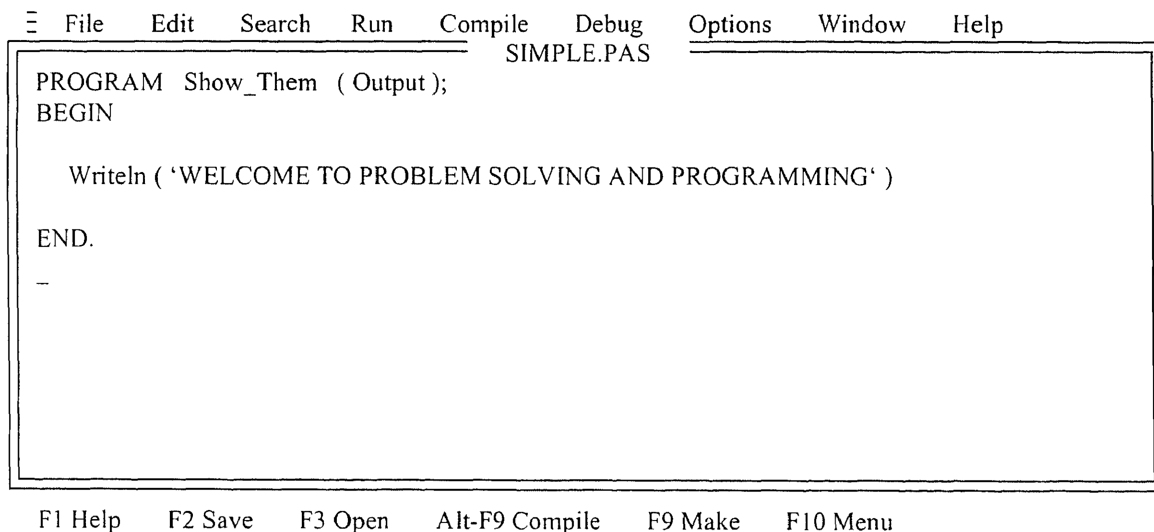


Figure 1 A programming session in Turbo Pascal

1.2.2 Debugging Aids

Debugging aids are used by programmers to test programs, observe program behavior during execution, detect and correct errors. Code watchers, tracers, flags, and visualization and animation utilities are common tools. External debugging utilities can also be used in conjunction with programming environments.

1.2.3 Intelligent Tutoring Systems

Intelligent tutoring systems allow access to tutoring and testing material on language syntax, offer adaptive instruction to individual learner needs. They are also able to analyze student responses and determine correctness; guide and interact with students; and provide feedback and advice. This is typically done by presenting the student with problems to be solved and checking student responses against rules or stored solutions.

1.2.4 Intelligent Programming Environments

Intelligent programming environments combine the features of intelligent tutoring systems such as adaptive instruction, monitoring and assessment of students' progress, and feedback and advice with tools that are used in the program development process. They also provide access to traditional programming environments utilities such as syntax editors, compilers and debuggers.

1.3 Limitations of Current Solutions

Reports from research studies point to students' difficulties with programming beyond the scope of language syntax to include lack of problem solving skills and curricular

shortcomings. Despite this, research and development on the teaching and learning of programming has devoted disproportionate attention to syntax-related activities, with little attention given to the earlier tasks of problem definition, requirement, and specification.

A review of existing systems (described in detail in Chapter 2) and an in-depth analysis of their role in responding to the needs of students learning programming (presented in Chapter 3) reveal a gap between the problems identified and the capabilities and character of current systems. This research indicates these systems have not met their goals as support tools for students learning programming and are ill-suited for the type of obstacles encountered by students.

While it would appear that the systems developed to help students learning programming have answered the problems they addressed, on closer examination, concerns arise regarding the appropriateness of these systems as support mechanisms for learning programming. These concerns fall into two categories, functional weaknesses and practical deficiencies, described below. A complete discussion of each is included in Chapter 3.

Functional weaknesses that apply to intelligent tutoring systems, intelligent programming environments, programming environments and debugging aids include:

- ***Absence of problem solving/software engineering frameworks***. Most systems do not provide the essential facilities to assist the student in performing the tasks of problem formulation, planning and design.
- ***Overemphasis on language syntax***. Separating the coding activity from the rest of the problem solving tasks create an overemphasis on syntax - an attention unjustified

by the findings of research on teaching and learning programming and the challenges encountered by the students.

- ***Inadequate user interface.*** Students were notably dissatisfied with system user interfaces; it is not apparent that either pre-deployment testing or protocol analysis was performed by the prospective users.

Additional problems occur with systems that exhibit artificial-intelligent-like behavior:

- ***Incomplete rules-and-errors knowledge base.*** A series of rules, a collection of error types and an enumeration of these errors are used represent a bug catalog or an ideal solution model. Students can make "*undocumented*" mistakes that will be misinterpreted or erroneously labeled, rendering the knowledge-base incomplete.
- ***Simplicity of the problem domain.*** The need for intelligent tools to behave as experts restricts the systems to the use of simple examples, reducing the systems' ability to provide challenging problems.

Practical deficiencies that apply to intelligent tutoring systems, intelligent programming environments, programming environments and debugging aids include:

- ***Limited classroom evaluation.*** To understand the tools' impact on the learning process, well designed experimental studies must be performed. Reports on evaluation results have been insufficient and inconclusive.
- ***Failure to integrate the tools into the curriculum.*** These tools have not received widespread acceptance, and their impact on introductory courses and curriculum is not clear.

- ***Impeded creativity and development of higher order thinking skills.*** Student creativity and the development of higher order thinking skills are hindered by the rigid problem solving approach of intelligent systems.

1.4 Problem Solving and Program Development

Programming is a kind of problem solving that requires representing solutions to problems in a coded manner suitable for processing. The relationship between problem solving and programming is apparent: an algorithm is a precise step-by-step outline to solve a well-defined problem; a program is a sequence of syntactically and semantically correct instructions forming a solution for a problem.

1.4.1 Problem Solving

Many methods of problem solving have been developed and are reviewed in Chapter 4. We shall justify the use of an enhanced method based on the approach proposed by the mathematician George Polya (1945), the author of a widely recognized series of books on problem solving . Polya defines the following four stages for solving a problem:

- Understanding the problem;
- Devising a plan;
- Carrying out the plan; and
- Looking back.

Each focuses on a unique aspect of the problem. The first step is concerned with understanding the problem's question and requirements. The comprehension of the problem requires the identification of the goal, the givens, the unknowns, the conditions,

the constraints, and their relationship. Devising the plan is the outline and refinement of a potential solution to the problem. Carrying out the plan is the transformation of the plan into a concrete reality and producing a solution to the problem. Finally, looking back is the confirmation of the result and the assessment of correctness of the solution.

Algorithmic problem solving uses techniques, such as structured decomposition and stepwise refinement (Wirth, 1971; Dijkstra, 1976), and facts, such as givens and unknowns, to outline steps leading to a problem's solution.

1.4.2 Program Development

Program development refers to the activities involved in designing and implementing programs. Depending on the complexity of the problem to be solved, these activities can be limited to simple data representation, algorithm design, development and coding. In some cases, such as those of large program developments even more activities requiring additional skills and knowledge may be required (Dijkstra, 1976; Boehm, 1976). A broad set of issues related to programming, or software development (discussed in section 1.5), has evolved into the field of software engineering (Pressman, 1987; Ng & Yeh, 1990).

The collection of cognitive activities involved in programming, referred to as program development tasks, begins with learning the syntax of the language and developing the skills for writing and understanding programs, testing and debugging solutions, and documenting and modifying programs. Problem solving skills: understanding the problem and its requirements, devising a solution, as well as a practical command of programming language constructs are required before a program can be written and its solution tested. Thus, by teaching programming in conjunction with

problem solving skills, planning, and design strategies, the student is provided with an understanding of the overall aspects of problem solving and program development. *Table* 1 enumerates the tasks and their associated activities.

Table 1 Program development tasks and their associated activities

Tasks	Activities
Learning the Language	Learning and understanding syntax, semantics and pragmatics of programming language.
Composing Programs	Representing solution into specific programming language syntax.
Comprehending Programs	Understanding of a given program, its functionality, and design approaches and techniques.
Testing and Debugging Programs	Performing exhaustive testing to verify program correctness.
Documenting Programs	Describing approaches and techniques used to solve problem.
Modifying Programs	Adding, removing, enhancing functionality, or adapting previously written code to solve new problems.

1.5 The Software Process

The software process consists of a series of stages encompassing all the tasks and activities performed in software production, beginning with the point at which a problem is recognized and defined, and extending through requirements analysis, design specifications, implementation and testing, integration and deployment, maintenance and, ultimately, retirement (Page-Jones, 1988; Schach, 1993).

Various models for software production are described in the literature (Blum, 1982; Boehm, 1988). Despite the fact that varying terminologies are employed to refer to the stages of software development, the role of the different stages are very similar. The

following is a generic list of the typical stages of a software process model. *Table* 2 depicts the software process and the activities of each stage.

Table 2 The Software process and associated activities

Phases of software process	**Activities**	
Problem recognition	Awareness of problems or deficiencies in current system	
Feasibility study	Identify scope of problem, objectives, cost/benefits, plan of action, etc.	
Analysis & requirements	Functional & data requirements, physical & conversion requirements, etc.	initial testing begins here
Design & specification	Initial design: decomposition, detailed design: refinement, technical specification, etc.	
Implementation	Translation to code	done in parallel
Integration	Integration of components and units	
Testing	Comprehensive testing: unit testing, integration testing, user testing, etc.	
Delivery	Turn over to user	
Maintenance	Corrective changes & enhancements	
Retirement	Replacement of system	

The problem recognition stage occurs with the awareness of the existence of certain deficiencies in the current operation of a system, either manual or automated. Further investigation of the nature and extent of the problem identified and the implications of undertaking a corrective action is referred to as the feasibility study stage. The requirements analysis stage conducts a more detailed study of needs and produces a clear representation of the problem and its scope. In the design specification stage, needs and requirements undergo a sequence of refinements yielding a detailed technical presentation of all aspects of the design, functionality, documentation, and description of

the potential product. The next three stages are closely related and often overlap. The outcomes of prior stages are translated into code in the implementation stage which are then integrated and tested as a coherent unit. This culminates in the delivery of a product that may undergo additional changes during its operational life until it is eventually removed from service.

1.5.1 The Software Process and Relationship with Problem Solving Activities

The four stages of problem solving defined by Polya (1945), (understanding the problem, devising a plan, carrying out the plan, and looking back), fall within the core of software process stages. Problem recognition, feasibility study, and requirements analysis, are initial stages in the software process, and contribute mainly to understanding the problem, its needs, and its scope. Design specification constitutes planning. Implementation and integration are equivalent to carrying out the plan. Testing is performed by looking back at what was produced and done to solve the problem. *Table* 3 shows the relationship between the software process and problem solving stages.

Table 3 Relationship between the software process and Polya's problem solving stages

Stages of Software Process	**Polya's Problem Solving Stages**
Problem Recognition Feasibility Study	Understanding the problem
Analysis & Requirements Design & Specification	Devising a plan
Implementation Integration	Carrying out the plan
Testing Deployment Maintenance Retirement	Looking back

1.6 Theory of Problem Solving, Cognitive Science and Relationship to System Specification

A critical review of the various classic problem solving methods (presented in Chapter 4) enables one to synthesize a *common method* which incorporates the essential features of the classic methods. The review also justifies the choice of Polya's (1945) method as the basic frame of reference for our work. By identifying the problem solving tasks specific to program development, one can adapt or enhance the general problem solving method to the area of program development. By scrutinizing each task of the common method for the appropriate cognitive techniques and skills it requires, we can then define a *Dual Common Model* which integrates problem solving methodology and program development tasks with the cognitive techniques needed at each step of the process. This Dual Common Model can then serve as the basis for the specification of functionality of

the proposed problem solving and program development environment presented in Chapter 5.

Learning tools should promote the development of cognitive skills, so it is important to explicitly identify those skills and design systems which encourage their development. In addition to identifying the cognitive skills required in problem solving, which are prerequisite to specifying the cognitive functions required for an environment that facilitates problem solving, the cognitive analysis presented in Chapter 4 also serves to define the experimental design for system evaluation.

1.6.1 The Specification Oriented Language in Visual Environment for Instruction Translation

An integrated environment supporting the problem solving and program development approach starting with the initial activity of understanding the problem and continuing through program implementation is proposed (described in Chapter 5). The Specification Oriented Language in Visual Environment for Instruction Translation (SOLVEIT) is a model for a problem solving and program development environment that combines the methodology for problem solving and program development and the supporting tools to perform the tasks. The system takes into consideration the cognitive skills that must be gained by students and the tasks performed in problem solving and program development.

This environment is based on the Dual Common Model for Problem Solving and Program Development produced in Chapter 4. Facilities to assist the student in learning these skills and accomplishing these tasks are provided for each stage of the model. The system was designed, implemented, deployed and evaluated.

SOLVEIT is intended as a problem solving and program development environment that takes into consideration the problem solving skills required by students learning how to program, as well as the specific knowledge related to the use of a programming language. The system provides tools that the students use in *formulating* the problem; *planning* and *designing* the solution; and *monitoring* and *evaluating* the solution's progress. SOLVEIT encourages students to *understand* the problem and its requirements and to *think* about possible solutions before engaging in implementation details.

The SOLVEIT environment combines the process and the tools to support the functionality of a traditional programming environment with a workbench facility and a battery of utilities used in problem solving and program development. SOLVEIT is designed to be used by beginning students working to solve programming problems. The system surmounts the problems associated with current environments by:

- Taking into consideration the skills that must be gained by students and the tasks required for problem solving and program development.
- Removing the emphasis on language syntax.
- Providing the framework and facilities that allow the student to deal with the common difficulties related to problem solving.
- Providing a state-of-the-art user interface.
- Integrating the tool into the learning environment.
- Evaluating the impact of the tool on the learning process.

Specific features of this environment are:

- Students are able to describe the problem in written form, refine it, and update it as required.
- Problem facts are identified through a formal interaction and elicitation process.
- Planning and design are aided with automation.
- Required code is translated into programming language syntax after problem is solved.
- The system focuses on a meaningful subset of language constructs.
- An electronic project notebook and a complete transcript/playback recording is provided.

1.7 Experimental Design

We have developed and implemented a plan for testing and evaluating SOLVEIT and for studying its impact on students' problem solving and program development abilities, their cognitive skills, knowledge, perception, and attitudes and motivation.

The aim of this research is facilitating the development of students' problem solving skills and to enhance the learning of programming. The specific goals identified for using SOLVEIT in the classroom are:

1. To facilitate students' development of problem solving and cognitive skills.
2. To enhance students' acquisition of knowledge necessary for program development.
3. To promote students' development of metacognitive abilities.
4. To encourage students' favorable perception, attitude and motivation toward the learning of problem solving and programming.

5. As a long term goal, to enhance the retention and transfer of such skills, knowledge and abilities to other situations.

This study sought to verify the claims made regarding the cognitive model of SOLVEIT and to investigate the impact resulting from using the SOLVEIT environment as a support tool for problem solving and program development.

The experiment to evaluate the effectiveness of SOLVEIT on students taking the first course on problem solving and programming was conducted over two semesters. The impact of the new methodology and tools was measured by testing a collection of hypotheses and research questions. Data was collected from two main sources: (1) the pre/post-questionnaire and (2) students' performance on course requirements. Questionnaires data was related to identifying background, experience and general information as well as information about the course, problem solving and programming. A collection of statistical procedures were used to analyze the results (presented in Chapter 7).

1.8 Research Contributions

This research addresses the obvious interdependence between the problem solving process and program development and the lack of satisfactory solutions that take into consideration the difficulties encountered by students learning programming. A theoretically well-founded system to support the process of problem solving and program development has been produced and evaluated.

For this to be accomplished, the following has taken place:

- Reviewed the literature to determine the actual difficulties involved in learning the tasks of program development;
- Performed an extensive study of environments and tools, covering artificial intelligent systems and traditional systems, developed to support the learning of problem solving and programming;
- Critiqued existing environments and tools and identified the limitations that have prevented them from accomplishing their stated goals;
- Performed an extensive review of problem solving methodologies developed in this century, and synthesized from these methods a *common model* for problem solving;
- Presented a synthetic view of the tasks required for program development which were integrated with the previously identified common model for problem solving and created the *Dual Common Model for Problem Solving and Program Development*;
- Defined a *cognitive model* that identify the cognitive processes, the cognitive structures that support these processes, and the cognitive outcome of the problem solving and program development process;
- Defined the specifications for the problem solving and program development environment's functionality, based on the Dual Common Model and its underlying cognitive theory, in a way that is consistent with the needs;
- Designed, implemented, tested, and deployed the Specification Oriented Language in Visual Environment for Instruction Translation (SOLVEIT);
- Integrated the new environment into the curriculum;

- Designed a cognitively oriented assessment method and related instruments to evaluate the process and the products of problem solving; and
- Designed a detailed statistical experiment to evaluate the effect of this environment on students' problem solving and program development skills, including system testing by protocol analysis, and performance evaluation of students based on hypotheses and research questions.

1.9 Dissertation Outline

The organization of the rest of this thesis is as follows: Chapter 2, presents the results of a literature review on the development of methodologies and tools to support the teaching and learning of programming, and a brief opinion on each is includes.

In Chapter 3, the related research is analyzed and a thorough critique of the tools and their role in responding to the problems in learning programming and problem solving is provided. Weaknesses and deficiencies of current systems are enumerated and carefully discussed.

In Chapter 4, a comparison of a number of problem solving methods which are synthesized into a *common method*, is presented, the program development tasks are reviewed, the cognitive science and learning theory relevant to problem solving are reviewed, a *cognitive model* of problem solving is defined, and the appropriate cognitive techniques and skills required for each task of the common method is identified. The chapter concludes by presenting a *Dual Common Model* which integrates problem solving methodology and program development tasks with the cognitive techniques needed at each step of the process.

In Chapter 5, the Dual Common Model is used as the basis for the specification of SOLVEIT, an integrated environment that encapsulates the process and tools necessary to support problem solving and program development.

In Chapter 6, the plan for integrating this new environment into the curriculum, testing and evaluating SOLVEIT, and for studying its impact on students' problem solving and program development abilities, their cognitive skills, knowledge, perception, and attitudes and motivation are presented.

In Chapter 7, the results of the experiment conducted to test the hypotheses and to answer the research questions of the study are presented.

Finally, Chapter 8 concludes with a summary of the research results and provides an outline for future work.

CHAPTER 2

RELATED RESEARCH: A SURVEY OF ENVIRONMENTS AND TOOLS FOR LEARNING PROGRAMMING

Developments in the area of programming languages, environments and tools aimed at supporting the learning of problem solving and programming have been carried out for some time and are still evolving. Computer scientists and cognitive scientists (Carroll & Thomas, 1982) have worked on this subject for over two decades, and have produced some encouraging results. An increasing emphasis has been placed on investigating and answering relevant questions: How can problem solving and programming be effectively taught to students? What are the cognitive components and their relationships required for programming? What are the cognitive effects of learning programming on students? What are the tools and environments necessary to facilitate the teaching and learning of programming (Weinberg, 1971; Shneiderman, 1980; Mayer, 1988; Hoc, Green, Samurcay, & Gilmore, 1990; Lemut, du Boulay, & Dettori, 1993)? Some of these questions have been answered. Others remain open for further investigation.

Methodologies and systems were devised to assist in teaching and learning programming and to aid both novices and experts in stages of the software process such as analysis, design, implementation, and debugging. This began with the initial introduction of CAI tools (Carbonell, 1970; Brown, Burgon & Bell, 1974; Barr, Beard & Atkinson, 1975), the subsequent development of intelligent systems for teaching (Anderson & Reiser, 1985; Johnson & Soloway, 1985; Corbett & Anderson, 1993), and programming environments (Brusilovsky, 1991; Ramadhan & du Boulay, 1993;

Hohmann, Guzdial, & Soloway, 1992; Forcheri & Molfino, 1994). This chapter provides a survey of systems in four common classifications:

- Programming environments
- Debugging aids
- Intelligent tutoring systems
- Intelligent programming environments

Each classification, as shown in *Table* 4, contains a wide range of tools, with some classifications providing features that combine overlapping aspects from more than one category. For example, systems that embrace features of both programming environments and intelligent tutoring are known as intelligent programming environments. Tools belonging to each of the four classifications have been designed to include some common features such as on-line help systems, feedback mechanisms, data and memory visualization, and algorithm animation capabilities. Traditional tutoring systems are not commonly used to teach programming and were not reviewed.

Cognition theory played a role in the development of some of these systems. Systems are considered cognitively based if they are designed to employ a certain learning theory by which student knowledge of programming and problem solving skills are analyzed. The ACT* (Adaptive Control of Thought) theory of learning and problem solving, involving acquisition of cognitive skills, is one example (Anderson, 1983). ELM (Episodic Learning Model) is another, based on a case-based learning model that stores knowledge in terms of a collection of episodes (Weber, 1988).

A number of tools in each classification have been designed, prototyped or implemented for students use in the acquisition of programming skills. *Although it is*

hard to classify these systems, due to overlapping characteristics, an attempt is made to associate each with a category that closely matches its primary functions and stated goal. Clearly, some can be associated with more than one category.

The following presents the results of a literature review of this area and includes a brief opinion on each system reviewed. Further analysis and critique of the tools and development efforts in this area is provided in Chapter 3.

Table 4 Classifications of systems devised to assist with programming

Classifications	Functions
Programming Environments	Allow student to experiment with specific features of programming language and are used in program construction, compilation, testing and debugging.
Debugging Aids	Used by programmers to test programs, observe program behavior during execution, detect and correct errors. Some may exhibit intelligent behavior.
Intelligent Tutoring Systems	Allow access to tutoring and testing material, offer adaptive instruction, analyze student responses and determine correctness, and provide feedback and advice based on stored expert knowledge.
Intelligent Programming Environments	Combine features of intelligent tutoring systems with tools used in problem solving and program development process.

2.1 Programming Environments

Programming environments provide systems used by programmers to develop and test programs. They are used by novices and experts, individuals and teams of programmers, and provide a range of functionalities. Traditional programming environments, such as

Pascal and C compilers, consist of tools that can be used in program construction, compilation, testing and debugging: editors, language compilers, pre-coded function libraries, linking loaders, parsers, tracers and debuggers. More current systems also include syntax-directed editors (that understand the language and provide syntax verification), code indentation, graphical user interface tools, dynamic compilation capabilities, powerful library systems and possibly code generators. This section examines only those environment developed primarily for teaching programming and therefore traditional programming environments are not considered..

Programming environments developed specifically for teaching may include additional facilities that allow the student to experiment with specific features of a programming language and some offer limited tutorial functions. In addition to text-based systems, many programming environments use visual tools to demonstrate concepts and provide pictorial explanation of algorithmic logic and data structures. Some use graphical images, or *icons*, to represent language control and data structure constructs, allowing programmers to create their programs by combining a collection of these icons.

Tables 5 and 6 offer a concise description of, both visual and text, programming environments reviewed. A thorough explanation of each system follows. The first eight systems described are visually-based systems.

Table 5 Examples of visually-based programming environments

Programming Environments	Description
Pict	Programs are developed **graphically** by selecting pre-written functions represented as **icons**.
PECAN	Provides the student with multiple **views** of a program in the form of **visual** representations of abstract syntax trees.
SCHEMACODE	Addresses documentation of source code and understanding of flow control. Coding is done using **pseudocode** which is translated to language syntax.
DSP	Constructing a program is a **visual** and **graphical** manipulation task that is done by combining **icons**.
AMETHYST	Provides data **visualization** and **graphical** displays to help the student understand program functions and data structures.
UWPI	Uses code **visualization** to help students learn basic programming concepts.
BACCII	Algorithms are developed using **icons**, independently of syntax details. Iconic code is translated to language syntax.
ASA	Algorithms are developed using **icons** and represented using **flowcharts**. Both pseudocode and language syntax are used.

Table 6 Examples of text-based programming environments

Programming Environments	Description
SUPPORT	Programs are created using **function keys**, **menus**, or parsed text. **Trace** and **debug** facilities are provided.
STRUEDI	Students select **predefined language constructs** from a menu in order to build the program.
EBPS	Uses a template-approach to programming and encourages **software reuse**. Also provides **examples** to aid the programmer.
SODA	Helps in identifying problem modularization during **design** process and in integration of modules to form a solution.
MEMO-II	Helps beginning programmers build **problem solving abstractions** which can be implemented with different programming paradigms.

2.1.1 Pict

Pict (Glinert & Tanimoto, 1984) was developed at the University of Washington, as an aid to program implementation and alternative to algorithm design. While they view the human mind as a "multidimensional, visual, and dynamic" problem solving machine, Glinert and Tanimoto contend that traditional programming methods force humans to think in ways that are "one dimensional, textual, and static," and therefore advocate a graphical and visual method of programming. The system is intended for Pascal programs that be represented graphically.

Pict is a *purely* iconic graphical environment and consists primarily of an editor where programming is done by selecting atoms, which are pre-written functions such as

input, output, looping control, etc. that are represented/displayed as icons. The atoms are selected with a joystick and arranged in a flow-chart-like manner on the screen: with Pict, the keyboard is not used. The program syntax representation can be formed, and the user can observe the execution. The system detects syntactical errors, but does not assist in planning the solution.

Pict is limited in what it can accomplish - no text manipulation, no real numbers, and it is restricted to small programs because of lack of screen size. The graphical aspect is arguably overdone: even numbers must be entered from a "softpad" by clicking on them with the joystick. Pict was evaluated on about 60 graduate and undergraduate students in a Pascal programming course at the University of Washington. They were given a 30-minute explanation of the Pict system and two programming tasks. Beginning students said they were able to write a program without knowing the language syntax, but advanced students already familiar with other textual languages found Pict's graphical system to be confusing and difficult.

2.1.2 PECAN

PECAN (Reiss, 1985) represents a family of program development systems created at Brown University. This programming environment runs on UNIX-based APOLLO workstations and requires powerful processing and high resolution capabilities. PECAN, a language independent system, provides the student with multiple views of a program including representations of the program, its semantics, and its execution.

PECAN consists of three main components: (1) a basic level support module that includes the command manager which provides the user interface, a general purpose

parser, and an incremental compiler that consists of a control module and several special purpose modules which interpret semantic actions; (2) a mid-level service module that includes PLUM, a data structure support that provides data manipulation and database management, event management, and data structure monitoring and ASPEN, which provides program representation via abstract syntax trees; and (3) a higher level module that includes the various program views. PECAN provides templates to help the user build programs, but also allows the user to type text as well.

Program views are various visual representations of the abstract syntax trees that include: the syntax directed editor, a pretty-printing view with multiple fonts; Nassi-Shneiderman structured flow graph view showing nesting of program blocks; and module interconnection diagrams. Semantic views are provided in the form of symbol table, expression, data type, and data flow. PECAN also supports execution views of graphical representation for data structures and statement level execution.

The system focuses on the implementation and testing activities and is, in its current form, only comprehensive in its support for program construction. There was no student evaluation reported.

2.1.3 SCHEMACODE

SCHEMACODE (Robillard, 1986) was developed with the goal of improving the use of programming languages as opposed to improving a programming language itself. It was developed at the Center for Development Technologies, Ecole Polytechnique de Montreal. SCHEMACODE addresses the issues of documenting source code and understanding flow control. The problem with documentation is that it is not systematic

because it is not part of the programming language, and it is usually done after the program is completed (bottom up instead of top down). Robillard notes that the understanding of flow control is another area that can be improved as a result of enhancing documentation.

SCHEMACODE consists of two main components: (1) a schematic editor, where the programmer constructs the problem solution in Schematic Pseudo Code (SPC) and (2) code generation module, where the system generates the program source code in either FORTRAN IV, FORTRAN 77, PASCAL, dBase III, COBOL, or C. The SPC specifications are automatically incorporated as comments into the generated code. Robillard indicates that many universities have used SCHEMACODE in software development courses and fundamental programming courses, but no details are provided.

This system is also concerned with the implementation aspect of program development. It is evident that SPC is of no help in planning and designing solutions to a problem. The system will, however, help the student with automatic code generation based on schematic specification, although the schema language must be learned. No information on classroom evaluation was reported.

2.1.4 DSP

The DSP system (Olsen, 1988) was designed at Molde College in Norway to support the teaching of programming in introductory courses with an emphasis on program development, maintenance, and code reuse. It is intended to encourage algorithmic thinking and to support modular programming. The DSP system is highly visual and allows for selecting, moving or copying predefined templates to avoid syntactical errors.

The DSP system is designed to combine the advantages of application generators and traditional programming languages. Programs can be generated in Ada, Modula II, or Pascal.

The DSP system consists of five main components: (1) a visual high level language, including a set of templates; (2) a language sensitive editor where program construction is done by combining icons; (3) a module base encompassing a predefined set of generic definitions and routines represented in target language syntax, which can be extended by the teacher; (4) a generator for translating DSP code to the target programming language; and (5) a visual runtime system, which executes DSP programs and displays content of data structures and variables. A text editor is also provided, though it is seldom used since constructing a program in DSP is mostly a visual and graphical manipulation task.

There are some advantages to the DSP system. For one, the system allows the user to work on various parts of the problem. The template method eliminates the problem of frustrating syntactical mistakes. The system relies on the "understanding by seeing" concept. However, there is no support for the tasks of problem solving and program development beyond the implementation stage.

The DSP system was developed as a prototype and no information on evaluation was provided.

2.1.5 AMETHYST

Amethyst (Myers, Chandhok, & Sareen, 1988) is part of a family of programming environments designed at Carnegie-Mellon University for novice programmers. It was

developed on an Apple Macintosh, written in object Pascal, and stands for A MacGNOME Environment That Helps You See Types.

The goal is to teach introductory programming through visualization. Amethyst provides data visualization and graphical displays to help the student understand program functions and conceptualize the data structure.

Amethyst consists primarily of a sophisticated editor that can represent the syntax for a specified program in different views such as outline, tree decomposition, and linear.

Each basic data type is represented by its own differently shaped box. Structured types such as arrays are represented by rows or columns of boxes. There are also two basic object types. Graphics generators use mapping functions to create the visual objects based on the appropriate data type to be displayed. Amethyst recursively draws composite types such as records and arrays.

Amethyst also has a custom display feature. For example, an instructor may want to illustrate certain complex data types such as linked lists, stacks, and queues using special displays.

System functions focus strictly on the data and control; structures aspect of programming. Support for other programming tasks is not provided. There are no evaluation reports available.

2.1.6 University of Washington Illustrating Compiler

The University of Washington Illustrating Compiler (UWPI) (Henry, Whaley, & Forstall, 1990) was developed at the University of Washington. The system's main purpose is to use program illustration or visualization to help students learn basic programming

concepts, such as those taught in first-year computer science courses and to help them debug programs. UWPI illustrates the data structures for simple program written in a subset of Pascal. A program illustrator works by watching for *events* through *hookpoints* which are inserted into the program. UWPI inserts these hookpoints automatically.

UWPI consists of two main components: (1) a conventional compiler/interpreter including an analyzer, interpreter and runtime-state parts and (2) an extended compiler core including an inferencer, two illustrators, layout tools and a renderer. The inferencer searches for the abstract data types within the code, chooses a layout plan, and passes the plan to the data illustrator. The inferencer is supported by a data flow analyzer which gathers definition information, a statement pattern matcher which looks for intra- and inter-statement idioms, a subrange inferencer which determines the range of values for each variable, and a concrete data type to abstract data type converter which finds which possible ADTs are inferred by a variable.

The data illustrator creates and updates the illustration, and the interpreter-driven source illustrator maps each node in the intermediate representation with the corresponding part in the source code.

UWPI does not understand the illustrated program. Instead it gathers information and accesses knowledge base of rules to construct and display the program view.

The system provides tools that are useful for activities of program implementation and debugging. Earlier stages of program development are not supported. The authors do not report on evaluation but note that although UWPI performs well despite small bugs, larger bugs can cause a break down in the system.

2.1.7 BACCII

BACCII (Calloni & Bagert, 1994) was developed at Texas Tech University. The system's premise is that the best way to learn programming is by developing algorithms using icons, independently of syntax details. BACCII runs under Microsoft Windows environment and is used to teach beginning procedural programming. Students develop algorithmic flowcharts using icons selected from a menu without any regard to the programming language syntax. ASCII format code is automatically generated for Pascal, C, FORTRAN and BASIC. The current version enforces top-down design methodology and an object oriented version is being considered.

BACCII consists of two main components: (1) the iconic algorithm development module, which allows the students create solution logic by combining icons and (2) the code generation system allowing the students to convert the iconic algorithm into language syntax.

As with other traditional programming environments, the system supports activities closely related to the implementation stage - namely flowchart construction and code generation only. Other problem solving tasks cannot be performed using the system. BACCII was evaluated in a five section Pascal course using two groups. One group (three sections) used VAX Pascal and the other group (two sections) used both BACCII and VAX Pascal. Calloni and Bagert report that the results were successful and state that the BACCII group earned significantly higher grades.

2.1.8 ASA

ASA (Guimaraes, de Lucena, & Cavalcanti, 1994) was created at SENAC, a Brazilian organization that promotes technical education. ASA is an environment designed to teach algorithms at the introductory level. It supports code execution, animated tutorials, and a program construction facility.

ASA consists of two main components: (1) the lessons model, which presents information in tutorial form with animation of concepts and algorithms and (2) the constructor, which consists of a flowchart editor and an interpreter allowing for algorithms to be formed using menus and icons, and to be represented graphically using flowcharts. The student can also visualize the algorithm in pseudocode, Pascal, C, or Clipper.

While ASA provides instruction in a tutoring fashion, it does not monitor students' performance, and no records of students progress are maintained. ASA was evaluated in two steps using eight students in the first step and 20 in the second step. Guimaraes, de Lucena, and Cavalcanti reported, with no specific details, that the main advantages were an increased learning rate for experimental students versus control students, as well as improved student motivation for learning in the experimental group. They also noted two drawbacks: students tended to create correct algorithms without understanding their purpose, and the teacher could lose control of the students due to lack of knowledge about their strengths and weaknesses.

2.1.9 SUPPORT

SUPPORT (Zelkowitz, Kowalchack, Itkin, & Herman, 1987) is a development environment that was designed at the University of Maryland and has been in use since 1986. This PC-based system is intended for programming using a subset of Pascal. The goal is to create a self-sufficient system which does not require the student to deal with the operating system or even be exposed to language features that are beyond the scope of the first course.

SUPPORT consists of three main components: (1) a syntax-directed editor to build the program; (2) a Pascal interpreter to execute the programs; and (3) a user interface module, which uses windows to communicate with the user.

The student communicates with the system through the Program Text window and the syntax-directed editor that only understands the features of the Pascal subset used. Programs are created using function keys, menus, or parsed text from the internal parser. The code is always syntactically correct since it is stored as a parse tree and will not allow for incorrect input. Menu buttons are used for constructs such as statements and procedures, and the parser is used for constructs such as expressions. This eliminates the frustration of novices trying to build a syntactically correct program. If the text is syntactically incorrect, then the Internal Character Oriented Editor (COED) is invoked. COED consists of simple cursor commands. It is the major tool for making modifications. Changes must keep the "syntactic balance" of the program syntax tree. SUPPORT allows to import or export programs and fragments to and from other applications.

SUPPORT includes a logging function to simplify generation of screen output. All write and writeln statements are automatically output to a file. The system also includes an interface between the text window and execution. The interpreter can be halted at any point during execution via the keyboard. Program trace and debug windows are also available at this time.

The utilities provided by support are valuable for the beginning programmer. But, the system's functionality is limited to the implementation and testing stages.

SUPPORT was used in a computer science course in Spring of 1986 with an estimated 240 students registered. A limited evaluation consisting of two student surveys was performed, one during the third week and one at the end of the semester.

2.1.10 STRUEDI

STRUEDI (Kohne & Weber, 1987) is a LISP STRUcture EDItor for novice programmers developed at the Institute of Psychology, Technische Universitat Braunschweig in Germany. The system was designed to reduce syntax difficulties common with novice programmers.

The basic idea behind STRUEDI is to allow the student to select predefined language constructs from a menu in order to build the program. STRUEDI maintains control of syntax, provides for understandable presentation of code, and helps the student understand syntax and semantics by offering explanations. The number of predefined constructs is limited to those taught in a first year LISP course. However, the number can be modified by the instructor.

STRUEDI consists of primarily of a sophisticated syntax-directed editor and a collection of predefined language-constructs that are selected from a menu.

STRUEDI works by copying constructs from a menu to the work space. The student begins by copying template constructs, such as selective or iterative statements. The constructs contain empty slots that need to be completed by the student such as specific operators, variable or constant names. Pre-implementation activities are not considered in STRUEDI. All facilities are actually used to build the program syntax.

A prototype of STRUEDI was evaluated by ten students with varying degrees of knowledge. Subjects were divided into two groups. One group wrote programs using STRUEDI and the other used a LISP environment that support a common display editor. The students participated in six sessions of programming using recursive and iterative LISP functions. They were given two tests -- one before the programming sessions and one after. The reported results showed that the syntactical skills of STRUEDI users increased by 14% and only by 5% for students using the LISP display editor. Semantic skills increased by 26% for STRUEDI users and 12% with LISP editor users. The authors concluded that STRUEDI enables students to learn syntax and semantics better than they would with an ordinary LISP display editor. However, the increase in algorithmic and problem solving skills was the same, at 5%. Such result is consistent with the fact that the system aimed to support coding only.

2.1.11 Example-Based Programming System

Example-based Programming System (EBPS) (Neal, 1989), developed at Harvard University, combines the concepts of syntax-directed editors and software reuse. It was

observed that novice programmers had problems with the terminology used in currently available syntax-directed editors. It was also noticed that programmers use previously written code, either from textbooks or their own code, when writing programs. The idea for this system is to use pre-written examples to aid the programmer.

There were two goals for this system: to use sample language constructs to enhance the template approach to programming, and to provide a tool that displays examples within the programming environment, and thus develop a new approach to programming. The system prototype was developed on a MacIntosh computer for the Pascal language.

EBPS consists of two main components: (1) a syntax-directed editor and (2) an example library. The syntax-directed editor is supported with an editing and an example windows. Example programs can be viewed, edited, or copied into the editing window.

The system provides very limited capabilities to the student and does not extend its functionality beyond the implementation stage.

EBPS was tested on 22 undergraduate and graduate students with various degrees of programming expertise. They were asked to write a program which converts an integer value into change (dimes, nickels, etc.). Only six of the students did not use the example facility at all. Half of the other 16 used the examples during program development, while the others used it only while editing. Most of the students used the example library for syntactical purposes, such as when to use "div" vs. "/", or "write" vs. "writeln". Only seven of the 16 subjects copied all or part of an example. The author states that overall response to the system was reported as useful and positive.

2.1.12 Software Design Laboratory

Software Design Laboratory (Hohmann, Guzdial, & Soloway, 1992) (SODA) is a design support environment for introductory programming in Pascal. It was created at the University of Michigan, Ann Arbor. SODA is based on a software design model, and provides a "workspace" for each stage of the process. The system was developed to help students overcome serious difficulties in identifying problem modularization during the design process, and to integrate those modules in the proper order to form a solution to the problem. Its primary goal is to support the software design process.

SODA consists of three main components: (1) a problem decomposition module, which provides reference to existing program solution techniques; (2) a composition module, which support the students in integrating solution parts; and (3) a debugging module to facilitate solution verification.

SODA has a graphical interface consisting of multiple windows for the Code View, the Goal-Plan View, and the Hierarchy View. The system is worksheet-based and self-paced. Students complete worksheets (programs, output, etc.) that are maintained in a notebook and are used for grading.

Although SODA provides a more realistic support for the problem solving process than other systems, it nevertheless ignores support for earlier stages of the process and begins with problem decomposition in the design stage. The system also maintains a tight control over the design process which must be followed in a sequential fashion. The student must first identify explicit goals, then specify plans, and then assemble the pieces into a working solution. Though a good idea, it is possible that certain activities can be

performed in parallel or through evolving iterations. Otherwise, it may so restrictive that it hinders student discovery.

The system was used at a high school in 1991, but does it not appear that a controlled evaluation was performed. However, a predecessor to SODA, called the GPCeditor, was used and evaluated by having the students write an end-of-semester program, using LightSpeed Pascal, without the use of GPCeditor. The evaluation was based on the length of programs written by the students. Students in the experiment group performed better than those in the control group. Hohmann, Guzdial, and Soloway state they feel confident discussing the "expected outcome of SODA in terms of actual outcomes from the GPCeditor."

2.1.13 MEMO-II

MEMO-II (Forcheri & Molfino, 1994) was developed at the Istituto per la Matematica Applica del Consiglio Nazionale delle Richerche, Genoa, Italy. Forcheri and Molfino address what they perceive as shortcomings in earlier tutoring systems developed specifically for learning particular languages and paradigms, and systems that promote learning programming methodologies, but do not aid students in solving problems. They classify these systems as being too "cognitively oriented." Because of the variety and complexity of the cognitive processes involved in programming, they observe that these systems can only apply to some of them. On the other hand, Forcheri and Molfino state that traditional programming environments are too "software engineering oriented," and are design tools based on systems for software development. They note that the software tools developed fall short of solving the problems with learning programming. They

propose to surmount these limitations by designing a system which combines aspects of the two categories into a single tool.

MEMO II consists of three major components: (1) a specification acquisition module used to build solution specifications using an editor and a verifier; (2) a reasoning mechanism, which proves the specification base; and (3) a direct implementation module that allow students to automatically translate specifications into code.

MEMO II is a realistic teaching and learning environment that falls between two extremes (cognition based and software engineering based) and is designed to help beginning programmers build proper problem solving abstractions that can be implemented with different programming paradigms. It allows students to deal with problems in the following areas: problem representation, specification validation, implementation, and execution of the program.

The system takes an approach consistent with the process of problem solving. The activities performed by the students are essential for understanding and solving problems. However, the system may not be suited for novice students. For example, problem representation is performed using a syntactically complex language that consists of a set of predefined operators. The students must learn the syntax of this language to define the problem. This seems a inconsistent with the goals of MEMO II. In addition, it appears that MEMO II was not evaluated in the classroom.

2.2 Debugging Aids

Debugging aids are used by programmers to test programs, observe program behavior during execution, detect and correct errors. Some debugging aids may include limited

program development tools in the same way that traditional programming environments include facilities designed specifically for debugging. Code watchers, tracers, flags, and visualization and animation utilities are common tools. External debugging utilities can also be used in conjunction with programming environments.

Table 7 offers a concise description of debugging aids reviewed. A thorough explanation of each system follows.

Table 7 Examples of debugging aids

Debugging Aids	Description
LAURA	Uses information on what program is intending to achieve in order to **debug** it. That information is presented in form of **program model**.
DA	If the student's program does not **match** the library plan, it compares the plan to its own **stored plan** and suggests corrections.
GENIUS	Uses **natural language interface** approach to program **debugging**. Presented as an alternative to knowledge-based systems.
VIPS	Provides control flow **visualization** and uses graphical interface that shows **changes** in programs' data during execution.
Lens	Uses a **debugger** of the C language to help user find logical mistakes by **graphically** executing code.

2.2.1 LAURA

LAURA (Adam & Laurent, 1980) was created at the University of Caen, France. The purpose of this debugger is not to find syntactic errors in programs, but to locate and possibly correct (or suggest corrections for) semantic errors in a program. Adam and Laurent note that much work has been done on using the computer to teach students how

to write correct programs. LAURA, on the other hand, is used to debug programs that have already been written.

LAURA consists of two main components: (1) a knowledge base that contains program model and (2) a debugger that translates programs into graphs and then compared to solution representations in the knowledge base to provide diagnostics.

LAURA runs independently after a program has been written, and therefore will not interact with the student except to produce the final outcome. The system requires information about what the program is intended to achieve in order to debug it properly. That information is presented in the form of a "*program model.*" The model is a syntactically and semantically correct version of the program to be tested. By recreating a representation of the program using graphs, the variation in syntax is removed and LAURA is able to identify or localize the semantic inconsistencies in the program.

In order for the debugging to be effective, it is essential for the system to receive correct information about the program. The method used is to supply LAURA with the correct program which is in turn compared against the student's solution. LAURA was evaluated with 100 programs written to solve 8 different problems. The system was able to identify correct versions and find errors in others, regardless of the structure of the programs.

2.2.2 The Debugging Assistant

The Debugging Assistant (DA) (Laubsch & Eisenstadt, 1981) is an aid designed at Open University in England. It was developed in MacLISP for use by novice programmers with the programming language SOLO.

DA consists of four main components: (1) a translation module; (2) a plan recognition module; (3) a symbolic evaluation module; and (4) an effect description matching module. In the translation step, the program is translated into a "plan diagram" notation which is language independent. It contains the program's control flow, data flow, and a description of the program's overall effect. It detects the following instances of "irrational code": unreached code, unbound variables, and useless code.

The plan recognition step recognizes sequences of code as manifestations of different "standard library plans". It refers to a plan library and recognizes the plan and its effect description.

The symbolic evaluation step is when all possible paths are analyzed, the debugger generates an "effect description" which accounts for any changes in the database achieved by the program.

The effect description matching step matches the previously generated effect description with an "ideal" effect description to find mismatches. If the student's program does not match the library plan, the domain specific plan library (DSPL) compares the plan to its own "*stored plan*" and suggests corrections. It checks the bindings of the variables, checks to see if a step insertion is needed, looks at the branches to see if a change occurred at the wrong branch, and finally looks for a branch where a disagreement in the effect exists, despite correct conditions. No reporting on student evaluation was provided.

2.2.3 GENIUS

The GENIUS prototype (McCalla, 1991) was designed at the University of Saskatchewan in Canada as an experiment in "ignorance-based reasoning" using a "psychiatric" approach to program debugging. It was written in C, runs on UNIX-based PDP 11/70, and is intended for the novice PL/C programmer. McCalla contends that knowledge-based systems require great intelligence and much knowledge and that current systems do not provide complete solutions to the problems of novice programmers. Such systems, McCalla states, concentrate on only a subset of the many problems facing students learning programming and it will be some time before a general and flexible knowledge-based support tools are available. Thus, he proposed GENIUS, a program debugging assistant.

GENIUS was designed to give the student the illusion that the system knows a lot about programming when in fact it does not (hence the ignorance-based system). It was created based on the idea that if a student spends enough time examining the program closely, the errors will eventually be found and resolved.

GENIUS consists of two main components: (1) a natural language module that finds "*granules*" in the students input, such as "yes", "no" or "don't know" and their variations and (2) a domain knowledge module which is made up of nodes containing hints and advice.

When a student first calls on GENIUS, the system tries to determine if the error is a syntax error or a logic error. The system is not equipped to handle logic problems since it would need more information on the type of program the student is working on. The syntax error handler asks the student to provide the error number given in the program

listing, then attempts to find the cause of the error by repeatedly asking questions and then provides a general advice on that type of error.

Several evaluations were performed. The GENIUS system fell short in several areas. For example, 60% of the requests were unfulfilled, which means GENIUS could not find the reason for the error. Also, the GENIUS system is not equipped to deal well with logic errors. As the complexity of the problems increased, the usefulness of the ignorance-based system decreased.

2.2.4 VIPS

VIPS, a visual debugger (Isoda, Shimomura, & Ono, 1987; Shimomura & Isoda, 1991) was created at the Electrical Communications Laboratories, NTT. It was developed to overcome weaknesses in existing visual debuggers, such as inadequate program flow visualization, the lack of dynamic code execution, and character-based representation of the debugging data (as opposed to a graphical representation).

VIPS consist primarily of a visual debugger for Ada, with a graphical interface that shows the user the changes in the programs' data during execution.

An advantage of the system is that it allows the user to determine the graphical method used to represent the various data types. On the other hand the user must learn Figure Description Language (FDL) before doing so. VIPS is useful only as a debugging aid for Ada, and no details on system evaluation are provided.

2.2.5 Lens

Lens (Mukherjea & Stasko, 1994) was developed at Georgia Institute of Technology. Lens provides programmers with a tool for viewing the code as a series of animation without having to learn a graphics paradigm or writing additional code. Lens is not intended to catch syntax errors. Instead, its purpose is to help the user see logical mistakes resulting from incorrect coding. By graphically executing the code, the user can see what the program is actually doing. The system is suited for learning, especially in introductory courses and courses that focus on software development.

Lens is primarily of an algorithm animation with an interface to a traditional source-level debugger.

Currently, the system uses the dbx debugger of the C language. It is not capable of creating animations of conditional executions. Also, mapping certain algorithmic principles, such as recursion, can prove to be difficult since Lens does not support direct manipulation of graphical items. Without this option, the user may actually have to work to create the animation themselves. Stasko and Mukherjea have evaluated their system and reported that all of its operations are functional, but they do not provide information on the methodology for their evaluation.

2.3 Intelligent Tutoring Systems

Artificial intelligence techniques are employed in some tutoring systems to provide more sophisticated support for students learning programming. Such systems are referred to as intelligent tutoring systems and allow access to tutoring and testing material on language syntax. Intelligent tutoring systems offer adaptive instruction to individual learner needs

and are able to analyze student responses and determine correctness; guide and interact with students; and provide feedback and advice (Wenger, 1987; Snow & Swanson, 1992). This is typically done by presenting the student with problems to be solved and checking student responses against rules or stored solutions. Such systems are used to help novice students learn a specific programming language, develop cognitive models, form mental representations of problem domains, and gain testing and debugging skills. Intelligent tutoring systems include three main components:

- The *domain knowledge base* contains representations of solutions, errors and rules constructed by compiling expert knowledge and skills on domain-specific problems. This knowledge base is used to evaluate student progress toward the solution, compare and verify results to form basis for error detection and correction.
- A *student model* used to construct a representation of the student learning progress and knowledge acquisition process. This information is crucial for the tutoring component as it determines what to teach, when to teach it, and how to teach it.
- A *tutoring agent* concerned with delivery of organized instructional modules to students. The selection of teaching material is based on the overall learning goals of a tutoring session, and feedback from the student model formed as a result of monitoring student interaction with the system.

These components may vary depending on the specific system architecture. However, most intelligent tutoring systems, as well as intelligent programming environments and debugging aids with intelligent behavior, share many commonalties. *Table* 8 offers a concise description of intelligent tutoring systems reviewed. A thorough explanation of each system follows.

Table 8 Examples of intelligent tutoring systems

Intelligent Tutoring Systems	Description
BIP	Determines program correctness by comparing student answers to its **optimal solution**. If program does not match, student starts coding process again.
LISP Tutor	Monitors for divergence from **expert behavior**. If student deviates from correct path, tutor "guides" student back toward solution.
PROUST	Examines student's code using **stored solutions** that are already in its library and returns feedback on this basis.
APT	Assumes that programming knowledge can be modeled around a series of **production rules** forming an **ideal student model**.

2.3.1 The BASIC Instructional Program

The BASIC Instructional Program (BIP) (Barr, Beard & Atkinson, 1976) was developed at the Institute for Mathematical Studies in the Social Sciences of Stanford University, and was designed to study the feasibility and effectiveness of tutorial methods of computer aided instruction. BIP is an independent, self-contained instructional course for learning the BASIC programming language, geared toward students at the high school or college level as an informal introduction to problem solving in BASIC. BIP was intended for students with no prior computer knowledge as a tutor or a supplement for learning computer literacy.

BIP consists of five main components: (1) an enhanced BASIC interpreter which also collects knowledge about student performance and presents errors in a more readable

fashion; (2) a database of 100 programming problems of varying difficulty; (3) a HINT feature which provides text help and text-based "graphical" problem solving hints; (4) the Curriculum Information Network (CIN), which through a series of error counters and self-reported student ability information will select problems for the student based upon skill set; and (5) a BASIC language student manual.

BIP determines a correct program by comparing student answers to its "*optimal solution*". This is done by looking for key syntactical program elements. If the student's program does not match with the model solution, the student is considered to have failed to acquire a particular skill, and is expected to start the process again. The system presents another lesson dealing with that skill, but it teaches mainly through hints and examples. Error counters and self-reported information about the users' understanding of the lesson are the primary methods of determining user ability.

One evaluation involving 42 students, split into control and an experiment groups, was completed. Each student used the system for ten hours solving the same problems, but using two different strategies for task selection: the control group followed a predetermined strategy for task selection and the experiment group used BIP's task selection algorithm. No significant difference was found in students post-test scores. However, a significant difference was reported in throughput with the experiment group solving more problems and encountering few difficulties.

2.3.2 The LISP Tutor

Goal-Restricted Environment for Tutoring and Educational Research on Programming (GREATERP), also known as the LISP Tutor (Anderson & Reiser, 1985), was developed

at Carnegie-Mellon University to provide students with assistance as they work on exercises using the LISP language. The LISP Tutor combines a psychological theory of skill acquisition with artificial intelligence to create a teaching device for the LISP language. It was developed on the assumption that private tutoring is much more effective than classroom training (Bloom, 1984). According to the Anderson and Reiser studies involving learning LISP, students with private tutors need only 11 hours to learn the same amount as students who had 43 hours of classroom training. They also note that humans learn best with immediate feedback about their errors. The tutor is intended for beginning LISP programmers in introductory courses, is written in LISP and runs on VAX systems.

The LISP Tutor consists of three main components: (1) the domain expert, which writes LISP functions from problem specifications; (2) the bug catalog, which holds possible divergence from the ideal "*expert behavior*"; and (3) the tutoring module, which consists of an expert system used for instruction.

The LISP Tutor provides immediate feedback at all stages of learning. If a student follows the path of a correct solution, the tutor remains in the background and monitors the student's progress. As soon as the student deviates from that path, the tutor steps in to "*guide*" the student back toward the solution. When an error is encountered, the tutor offers an explanation of the correct answer based on templates stored with the LISP production rules.

Two evaluations of the tutor were reported. The system was compared to experienced tutors and self-learning conditions. The results of the evaluations of the LISP tutor versus a human tutor and "on your own" conditions for learning six lessons

are not surprising. In one study, students with human tutors took 11.4 hours, students with the LISP Tutor took 15 hours, and students learning on their own took 26.5 hours. The system has an effective rate of catching and correcting student errors of between 45 and 80 percent, depending on the complexity of the problem being solved. Students complained, however, that the system uses too many menus.

2.3.3 PROUST

PROUST, a tutoring system for Pascal programs (Johnson & Soloway, 1985), was developed at Yale University with a two-fold goal: to provide students with their own "programming expert", and to create a "pedagogical expert" that could interact effectively with students.

PROUST consists of three main components: (1) a module to address the location and content of the bugs in a program; (2) a module to determine what the student intends to do with the code; and (3) a module, the most important, to identify misconceptions a student may have that explain the presence of bugs in the program.

On the highest level, PROUST uses an explicit system of goal decomposition to break down a program. This system creates a tree-like structure that takes into account the algorithms for solving a problem. After this occurs, PROUST examines the student's code using "*stored solutions*" contained in its library. During debugging, PROUST looks for matches between the student's code and its own. If it cannot reconcile differences between the two, a series of transformation rules are invoked. These rules make it possible to break the program up again by examining individual functions. It prints out messages that describe the error, its location, and offers help for users if they request it.

PROUST examines the student's code using solutions from its library and provides feedback on this basis. The system was evaluated on 206 novice solutions to a simple program that computed average rainfall in a day. Seventy nine percent (of 161 programs) were completely analyzed. Ninety four percent of the bugs were recognized, though not accurately (there were 55 instances of either misinterpretation or erroneous labeling). Among the 35 programs receiving partial analyses, 191 bugs were found. Seventy one of the bugs were recognized correctly. Seventy were deleted from the analysis and 50 were not recognized. PROUST was unable to analyze four percent (or nine) of the programs.

Johnson and Soloway state that these inaccuracies in bug recognition are likely to occur in programs that contain undocumented errors, or in cases where the students use novel plans or ambiguous cases requiring interaction with the tutor. This is a drawback for a system designed to teach programming. Novice students will invariably write programs that contain unusual errors or even use some creative ways to solve a problem.

2.3.4 The ACT Programming Tutor

The ACT Programming Tutor (APT) (Corbett & Anderson, 1993), a cognition based tutor, was created at the Psychology Department of Carnegie-Mellon University. The APT tutor is designed as a programming environment to help students complete short programming assignments, and based on an *ideal student model* using production rules as its knowledge base. The knowledge base currently supports LISP and Prolog syntax.

APT consists of two main components: (1) the tutor interface, which presents the exercises for the students to solve and provides feedback and (2) the cognitive model,

which consists of a set of rules for writing LISP code. APT is a graphical environment that uses several windows, each with a different function. Some of these include the 'Problem Statement', 'Exercise', 'Hint', 'Menu' and 'Skill Meter' windows. The functions of these windows are self explanatory, except the 'Skill' window, which shows as a bar graph the tutor's model of the student -- the probability that the student has learned the skills being presented in that exercise. This process is called "knowledge tracing".

The APT tutor was created to test viability of the ACT* theory of skill acquisition (Anderson, 1983). The ACT* theory assumes that programming knowledge can be modeled around a series of "if-then" production rules forming an "*ideal student model.*" In the case of the APT tutor, it is several hundred production rules.

APT is designed to help students write short programming exercises. As such, a set of production rules can be used to create an ideal student model. However, as the complexity of problems increase, so does the complexity of the rules. Moderate success is reported on system evaluation, but the number of students used in that testing may have been inadequate. APT was evaluated in one semester using 41 students.

2.4 Intelligent Programming Environments

Intelligent programming environments combine the features of intelligent tutoring systems such as adaptive instruction, monitoring and assessment of students' progress, and feedback and advice with tools that are used in the program development process. In addition to the domain knowledge base, the student model, and the tutoring agent,

intelligent programming environments provide access to traditional programming environments utilities such as syntax editors, compilers and debuggers.

Table 9 offers a concise description of intelligent programming environments reviewed. A thorough explanation of each system follows.

Table 9 Examples of intelligent programming environments

Intelligent Programming Environments	Description
Bridge	Relies on pre-stored **ideal solutions** to verify student's program.
GIL	Uses a knowledge base of **reasoning rules and plans** and responds to errors by comparing the student's steps with its own reasoning.
ITEM/IP	Compares solutions against **model programs**. If any inconsistencies are detected, system *explains* **wrong behavior** of student's program.
DISCOVER	Uses a pre-stored **reference solution** against which it matches user's program.
ELM-PE	Determinations of correctness is made by the use of **rules knowledge base** for good, suboptimal, and buggy code.
Capra	Uses stored **model solutions** to verify correctness of student's work.
INTELLITUTOR	Evaluates student's intentions based on **algorithmic structure** knowledge base that contains possible solutions.

2.4.1 Bridge

Bridge (Bonar & Cunningham, 1985) was developed at the University of Pennsylvania. Bridge is an intelligent tutor that provides a "complete tutorial environment" for the beginning programmer. In addition to finding student errors, it also understands partially

completed programs and student intentions for their code. Bonar and Cunningham also wanted to *bridge* the gap between the syntactic approach to learning a language and the cognitive processes necessary for problem solution. The tutor is able to understand natural language specifications to problems and the syntactical solution.

Bridge consists of three main components: (1) solution specification module, which allows the students to formulate their ideas in English; (2) plan specification module, which teaches the students programming skills and allows them to translate from informal specifications to plan specifications; and (3) the syntax module, where students build the programming language code for the solution.

One benefit of Bridge is its use of the natural language to allow the students to describe a problem solution, build it into a programming plan, and receive guidance through the conversion into Pascal syntax. However, the student is restricted to using the natural language vocabulary selection such as 'Sum', 'Add', 'Keep doing steps', etc. The sentence is then completed by selecting choices from a menu. There are three main steps for building a program: informal specification of the solution, translation from the informal specification to the programming plan (more specific than an informal specification - also using terms like 'Read in', 'Count', etc.), and translation of the programming plan to program code. This provides some flexibility and encourages students to think about problem solutions in simple language, an important step in problem solving as it encourages the student to be creative. However, support for design and problem decomposition is lacking. Bridge does not support functions and procedural abstractions.

To build the English language solution, in phase 1, the student selects choices from the "Natural language selections menu". A second menu selection is used to complete the sentence, which is then placed into the plan window.

Phase 2 involves matching representations of the prior phase with menu selections of formal programming plans. For example, 'Add integer to running total' produces a 'Plan to: Keep a running total' which is translated to 'INITIALIZE', 'UPDATE', and 'VALUE'. In phase 3, the student uses a menu to select Pascal constructs and matches them to programming plan of phase 2.

Although Bridge allows the student to build a natural language plan specification of the problem, transform it into plans by making choices from a menu, and consider possible solutions to a problem, it remain restrictive due to its limited vocabulary and syntax. Problems used in Bridge are of a very simple nature and emphasis is on algorithmic and syntactic implementation throughout the process of problem solving. The system also relies on a pre-stored "*ideal solution*" to verify the student's program. The student solves the problem by first using a "well defined" language to form the English language solution, then converts it to a plan using coded terms, and finally translate the plan to Pascal syntax.

Bonar and Cunningham compare Bridge with PROUST, which they consider unable to have meaningful interaction with the user, and the LISP tutor, which forces the student into a highly directed process, does not allow informal ideas and provides no provision for intermediate components during problem solving. The evaluation of the system took place with 10 students who were able to work through the lessons with little or no human intervention. However, the sample is too small to be able to draw

conclusions. Students reported difficulties with the visual display and inadequate textual materials presented.

2.4.2 Graphical Instruction in LISP

Graphical Instruction in LISP (GIL) (Reiser, Ranney, Lovett, & Kimberg, 1989) was developed at Princeton University. The goal of GIL is to construct explanations based on its knowledge of the problem and the solution and to use visual representations to aid the student in writing simple LISP programs.

GIL consists of four main components: (1) a problem solver, which uses a knowledge base of reasoning rules and plans; (2) an explainer, which follows the problem solver's logic and explains its own reasoning; (3) a response manager, which responds to program errors, errors which are legal LISP expressions, but are not useful to the program, non-confirming strategies, and specific hint requests; and (4) a graphical interface, which allows students to build programs by connecting objects representing different program constructs into a graph instead of using LISP's traditional text form.

The user selects functions from a menu and specifies its input or output. The complete program is a graph representation of functions that transform input data into the desired output. GIL allows the user to plan in several directions, for example: forward, from the data toward the goal, or backward, from the goal back toward the data. In GIL, there is no formal planning and design stage. Students begin solving the problem by implementing the solution logic. GIL compares the student's steps with its own reasoning and suggests steps to take based on the student's method of problem solving.

GIL was tested on nine undergraduate students, who had no prior programming knowledge. The students learned basic LISP functions from a textbook. They were assigned 14 to 15 programming problems dealing with lists. The solutions required three to seven steps. After the ten minute demonstration, none of the subjects asked for assistance with the interface. The subjects completed the assignment in less than two hours which is half the time spent by non-tutored students. GIL students solved the problems faster than other students (an average of 15 minutes for GIL students vs. 58 minutes). The subjects made only .4 errors per problem and requested .3 hints per problem. 95% used forward reasoning. Backward reasoning was rarely used and was not used more than once per problem. More legal errors were made (83%) than strategic (17%), with most occurring when the operators were misunderstood. Subjects were able to fix approximately 50% of the errors. Even though the subjects did not regularly request hints, they often followed the hints (74%) immediately and were successful 69% of the time. The sample of students used for the evaluations is too small to draw accurate conclusions.

2.4.3 Intelligent Tutor, Environment and Manual for Introductory Programming

Intelligent Tutor, Environment and Manual for Introductory Programming (ITEM/IP) (Brusilovsky, 1992) was developed at the International Center for Scientific and Technical Information in Moscow. ITEM/IP is an integrated intelligent tutor an programming environment for teaching the mini-language Turingal (Brusilovsky, 1991), designed for this system's use. ITEM/IP was developed as an example of a system that

provides functionality by integrating an electronic manual, a tutor, and a programming/learning environment.

ITEM/IP consists of three main components: (1) the programming laboratory as an integrated editor/debugger; (2) the information kernel, which contains the student history); and (3) the pedagogical module, which controls teaching operations such as when to move on or repeat a lesson.

The system presents a task for the student to solve, guides the student through the process, and then evaluates the outcome. The solution is compared against the "*model program*". If any inconsistencies are detected, the system "*explains*" its solution by demonstrating the "*wrong behavior*" of the student's program.

The intended user is a high school or beginning college student. The system was evaluated over a year period with 45 students (three groups of 15 each) from 14-year-olds to Moscow State University freshman. The students in the experimental group learned Turingal using ITEM/IP, while students in the control group learned Pascal via blackboard lessons. Brusilovsky reports, "without any special measurement", an increase in interest among the ITEM/IP students, and a decrease in the number of students deemed "weak". Brusilovsky notes that the design of such an environment for a "real" language is a difficult task.

2.4.4 DISCOVER

DISCOVER (Ramadhan, 1992; Ramadhan & du Boulay, 1993) is an intelligent tutor and programming environment designed to teach beginning programming. It was developed at the School of Cognitive and Computing Sciences, University of Sussex, United

Kingdom. Ramadhan and du Boulay (1993) acknowledged the results of intelligent tutoring systems have been questioned, and designed DISCOVER as their response. It combines visualization and traditional programming environment features with intelligent tutoring.

The system consists of two main components: (1) the free phase module, which allows the students to construct programs on their own with no feedback from the system other than memory visualization and (2) the guided phase module, where students learn from a set of problems under the guidance of an "*intelligent programming expert.*"

DISCOVER's free phase allows students to construct programs without the intervention of the system, and the guided phase offers problems to the student and monitors progress toward a solution. All programs must be written in a pseudocode language designed for DISCOVER. In the guided phase, the system selects the problem and presents the specification to the student to find a solution by combining programming concepts from a menu (similar to Bridge). It uses a pre-stored "*reference solution*" which it matches against the user's program. Its ability to analyze partial code segments is an asset.

The language of DISCOVER is, however, very limited. It supports input, output, selection (using an "If" statement), and repetition (using a "While" statement). The use of pre-stored problem statements and solutions as a means of verification in the guided phase is restrictive and limits the number of solutions that the system will recognize as correct. The user builds answers by selecting "concepts" from a menu. These concepts are then translated into program statements. An apparently effective way to teach language constructs and syntax. DISCOVER was evaluated using eight students, four in

the experimental group and four in the control. The authors' experiment showed that the students using the system solved problems quicker and with greater accuracy than those who did not use the system. They commented that the application of these findings is limited due to the small number of participants in the experiment.

2.4.5 Episodic Learning Model Programming Environment

Episodic Learning Model Programming Environment (ELM-PE) (Weber, 1993), a cognition based intelligent programming environment designed for teaching the LISP language, was developed at the Department of Psychology, University of Trier, Germany. Weber notes that novices usually need explicit examples and reminders of previous problems when they are faced with new problems to solve. Thus, ELM-PE is an example-based system designed to support students learning LISP through the use of analogies.

ELM-PE consists of five main components: (1) a syntax-based structure editor, designed to reduce syntax errors by filling in LISP statement slots with appropriate insertions, allowing only valid LISP syntax to be constructed; (2) example-based programming tutor to teach LISP whose code may be modified and copied into the student's own programs; (3) a stepper module allowing the student to visualize the flow of data during the execution of the program, which can be stepped through line by line and stopped anywhere to allow the student to see mistakes made; (4) error messages delivery and explanation module; and (5) a cognitive diagnostic module based on the ELM theory.

An important feature of ELM is the fact that the code produced by the student undergoes a cognitive diagnosis. This diagnosis serves as the impetus for offering hints to the student on incorrect code, which plans should be followed, and partial solutions to the problem. Student solutions are categorized as good (correct), suboptimal (correct but could be better), and buggy (incorrect). The cognitive diagnosis makes such determinations by the use of a rules knowledge base (similar to the LISP tutor) for good, suboptimal, and buggy examples.

The system offers the student reminders and analogies when errors occur or when the student's code is determined to be suboptimal. Also, the student can ask for help while writing the code for a problem and the system responds by offering similar examples. This is possible because ELM stores resulting explanation structures of pre-analyzed examples in the case-based learner model.

Novices do have difficulties recalling analogies between old and new problems (Gentner & Landers, 1985). This system is based on the premise that, in an example-based learning environment, a tutor's ability to show and explain relevant examples can ameliorate the problem. An important factor would then be the comprehensive state of the problem domain case-base. The same is true regarding the completeness of the buggy rules which are stored in an error library. ELM-PE was tested with a total of 20 students, ten novices and ten advanced; clearly too small a sample to demonstrate the system's ability to promote creative problem solving and understanding.

2.4.6 Capra

Capra (Verdejo, Fernandez, & Urretavizcaya, 1993) is an intelligent programming environment developed at the Ciudad Universitaria and Universidad del Pais Vasco, Spain. It was designed to overcome shortcomings of similar existing environments. The authors say many such systems focus on the drill and practice, through exercises, of a particular programming language constructs, imparting knowledge of the syntax and semantics only. The goal of the Capra system is to teach program design at an elementary level. Its methodology is based on the use of plans or schemas for reasoning about the construction of a program. Capra supports planning and implementation and combines features of programming environments and tutoring systems. The system helps the student understand and write elementary programs and provides tutoring based on students' knowledge. There are three steps to a programming solution: problem abstraction, relationship to a class of solutions, and refinement to produce a final answer as designed and stored in the system's knowledge base.

Capra consists of three main components: (1) the tutor module, which presents the students with exercises to check concept comprehension; (2) the knowledge based debugger, which monitors students problem-solving activities; and (3) the interface module, which manages the system's multi-window interface.

The tutor's role is defined as "a process where two agents cooperate to attain some instructional/learning objective." The first agent facilitates a Socratic style dialog which is followed by a decision facilitated by a second agent as an instructional plan for the student presented in the form of lessons. The debugger uses stored "*model solutions*" to verify the correctness of the student's work. The diagnosis of a student's work

involves matching the version of the code written by the student to the correct one stored in the knowledge base. The interface consists of a graphical user interface through which the student communicates with the system.

Capra relies on a knowledge base of previously defined solutions to verify the student answers. The system uses classes of stored problems to teach problem solving. A problem may arise in the case where a student may come up with a unique solution that is correct, but does not exist in the knowledge base. This method restricts the student's thought process. The student is expected to apply the system's logic in solving certain types of problems. It has not been reported whether the system was evaluated.

2.4.7 INTELLITUTOR

INTELLITUTOR (Ueno, 1994) is another example of an integrated intelligent programming environment. The system was developed at the Tokyo Denki University, Saitama-ken, Japan.

INTELLITUTOR consists of three main components: (1) GUIDE, (2) ALPUS, and (3) TUTOR. GUIDE is a knowledge-based editor which assists the student in writing programs with built-in syntax knowledge. ALPUS is the portion of the system which attempts to "understand" the student's intentions based on an "*algorithmic structure*" knowledge base that contains possible solutions. It analyzes buggy statements and detects logical errors. It then "guesses" the intentions of the student and offers advice for fixing errors. The TUTOR subsystem receives information from GUIDE and ALPUS to build a model of the current user's abilities. TUTOR can then present the appropriate

knowledge to the student to facilitate learning. The system does not, however, consider the student's knowledge level.

The system is not intended for the novice; it was developed for students with a basic knowledge of Pascal. According to Ueno, it is best used in an intermediate programming course in which students are learning algorithms and practicing programming skills. INTELLITUTOR's program understanding capability has been evaluated with experiments on intermediate level programmers using sorting algorithms at the Department of Systems Engineering of Tokyo Denki University. Students were first taught the Quicksort and Straightsort algorithms in a class lecture. They were then asked to code the algorithms. The results were entered into the system and examined by ALPUS. In the Quicksort programs 68.6% were correctly evaluated by ALPUS. For Straightsort, 77.5% were correctly understood by ALPUS.

There are several limitations to the INTELLITUTOR system. In terms of its teaching capability, intentions are inferred by evaluating buggy code and not by evaluating the student's individual knowledge level. The ability of the ALPUS module to "comprehend" a program is limited. One reason for this is that the "algorithmic structure" knowledge base does not include all possible solutions - a serious problem since the system was specifically designed for intermediate students. Such students are unlikely to write only "simple" programs. Take as an example the 68% success with the common Quicksort routine coded by the students. Ueno states there are two reasons for this: the knowledge base does not contain every possible solution pattern and therefore it cannot recognize everything coded by the students; and the pattern matching routines are not powerful enough to understand complex bugs, only relatively simple ones. It does

not appear that a comprehensive evaluation testing of INTELLITUTOR was done. The two sorting algorithms can not be considered general enough to test the effectiveness of the system.

2.5 Conclusion

The efforts in teaching and learning programming and the development of computing systems to assist novice programmers underscore the necessity and importance of enhancing the experiences of learning a programming language. Computer Science is now a well defined field. College students can earn degrees in many areas of computing-related specialization. Computer science, or computer education as it is known at the pre-college level, is also studied in the secondary schools as a required or elective subject. More students in secondary schools and colleges are taking computer science courses - in fact, a course on programming and problem solving is a standard requirement in virtually all university engineering and science curricula. Many other curricula - business, liberal arts, and education - also include a required course in computer science, though not necessarily in programming. Additionally, applications software ranging from drawing and engineering applications to database systems, spreadsheets, and word processors provide programming facilities for the end user. Problem solving skills are essential to understanding the fundamentals of computing, and should be learned while studying programming. Clearly, recent development efforts have enabled more people to take advantage of technology.

As evidenced by the literature review, a notable emphasis has been placed on meeting the needs of students learning programming. A large number of tools and

environments have been designed and developed to improve the learning and teaching of programming skills and, consequently, the productivity of programmers. However, there are questions that must still be answered. These are: Did the systems achieve their defined goals? Was the developments of these tools grounded by the needs of the classroom? Were they consistent with the difficulties in problem solving and program development - namely, errors and misconceptions in understanding of syntax, semantics, and use of programming languages, deficiencies in problem solving strategies, tactical knowledge, and ineffective pedagogy of programming instruction? The following chapter presents a thorough analysis and critiques of the surveyed systems and their role in enhancing the acquisition of programming, and provides some answers the these questions.

CHAPTER 3

ANALYSIS AND CRITIQUE OF EXISTING APPROACHES

Problem formulation, planning and design are essential prerequisite tasks to coding and testing because any difficulties or errors at these earlier stages, due to deficiencies in problem solving skills or inadequate domain knowledge, lead to errors in the final stages (Wirth, 1971; Walker, 1994). Reports from research studies point to students' difficulties with programming beyond the scope of language syntax to include lack of problem solving skills and curricular shortcomings. Despite this, research and development on the teaching and learning of programming has devoted disproportionate attention to syntax-related activities, with little attention given to the earlier tasks of problem definition, requirement, and specification. However, language syntax should be learned in conjunction with problem solving skills and analysis and design methodologies to provide students with an overall understanding of effective program development.

This chapter provides a thorough analysis of the problems and critiques the systems designed to address students' difficulties in learning programming.

3.1 Functional Weaknesses and Practical Deficiencies

It would appear that the systems developed to help students learning programming have answered the problems they addressed. But, on closer examination, concerns arise regarding the appropriateness of these systems as support mechanisms for learning programming. These concerns fall into two categories, functional weaknesses and practical deficiencies, described in sections 3.2 and 3.3.

Some of the problems apparent in current research and development are: the absence of problem solving and software engineering frameworks; an overemphasis on syntax; a lack of integration of the technology and pedagogy in the classroom; insufficient testing and evaluation (Rosenberg, 1987; Ramadhan, 1992); and problems with the user interface. For AI systems, there are problems with the domain and rules knowledge bases; the bug catalog; the limited scope of examples; and rigid teaching paradigms (Sleeman & Brown, 1982; Snow & Swanson, 1992; Eisenstadt, Price, & Domingue, 1993; Ramadhan & du Boulay, 1993).

3.2 Functional Weaknesses

Is the focus on programming too narrow? Is this focus warranted? Have these overly ambitious systems, especially intelligent tutors, accomplished their defined goals? Are the expense and complexity of these systems justified when dealing with simple programs that are often about one page in length? Researchers themselves have asked these questions (Johnson & Soloway, 1985) and some are reassessing their research directions. For example, the developers of the LISP tutor have totally abandoned the original idea of emulating a human tutor (Anderson, Corbett, Koedinger, & Pelletier, 1995). We address answers to these questions from a learning viewpoint. *Table* 10 summarizes the functional weakness in programming environments, debugging aids, intelligent tutoring systems and intelligent programming environments.

Table 10 Functional weaknesses

Functional Weaknesses	Reasons
Absence of Problem Solving/Software Engineering Frameworks.	Lack of facilities to assist student in performing problem formulation, planning and design of solution.
Overemphasis on Language Syntax.	Intense emphasis on implementation aspect of software process.
Inadequacy of User Interface.	Intricate systems and complex user interface add to students frustration.
Incomplete Rules-and-Errors Knowledge Base.	Rule-based systems are "incomplete systems". It is impossible to assume that all knowledge is represented.
Simplicity of Problem Domain.	Necessity for intelligent tools to behave as experts and anticipates all options requires use of simple examples.

3.2.1 Absence of Problem Solving/Software Engineering Frameworks

A gap exists between the functionality of present intelligent systems and programming environments and the actual needs of students using these systems to learn problem solving and programming. Problem solving skills must be an integral part of the body of knowledge a student acquires when first learning programming. However, most current tools are basically environments for coding and testing. This may be reasonable in a software development environment, given the expertise and specialized role of programmers. But, in a learning environment, the student, a novice programmer, is responsible for the development of a complete solution and can be overpowered by the need to simultaneously understand the problem, apply design methodologies, and implement details. *Table* 11 through *Table* 15 provide a comparative view of tools'

conformity with the problem solving and program development process for the four categories of systems described in Chapter 2.

Some of the tools support a broader, but still incomplete, aspect of the problem solving and program development process. Examples are the intelligent programming environment Bridge (Bonar & Cunningham, 1985), the design support environment SODA (Hohmann, Guzdial, & Soloway, 1992), and the programming environment MEMO-II (Forcheri & Molfino, 1994).

Bridge allows the student to build an English language specification for the solution by making "task" choices from a well-defined "natural language selections menu." To refine the solution, a second selections menu is used where the student can convert the specification to plans. Finally, the program syntax is built, using once again menu selections. However, Bridge has no explicit planning stage or a design stage where the solution is developed and improved by successive refinements. The solution evolves from natural language specifications, which is then converted to an algorithm, and then to syntax.

With SODA, on the other hand, problem solving starts with the design stage, bypassing problem definition and planning stages. Specialized tools for defining goals, plans and data objects are provided. The process only consists of problem decomposition, solution composition and debugging.

The activities performed in MEMO-II are problem representation, specification, implementation and execution, but it omits solution planning and design stages. Furthermore, under problem representation, the student uses a syntactically complex language which employs predefined operators to model problem specification, obligating

the student to deal with another form of language syntax in the very first stage of the process.

Table 11 Focus of problem solving and program development process in visual oriented programming environments

Programming Environments	Understanding Problem	Planning Solution	Designing Solution	Implementing Solution
Pict				X
PECAN				X
SCHEMACODE				X
DSP				X
AMETHYST				X
UWPI				X
BACCII				X
ASA				X

Table 12 Focus of problem solving and program development process in text oriented programming environments

Programming Environments	Understanding Problem	Planning Solution	Designing Solution	Implementing Solution
SUPPORT				X
STRUEDI				X
EBPS				X
SODA			X	X
MEMO-II	X			X

Table 13 Focus of problem solving and program development process in debugging aids

Debugging Aids	Understanding Problem	Planning Solution	Designing Solution	Implementing Solution
LAURA				X
DA				X
GENIUS				X
VIPS				X
Lens				X

Table 14 Focus of problem solving and program development process in intelligent tutoring systems

Intelligent Tutoring Systems	Understanding Problem	Planning Solution	Designing Solution	Implementing Solution
BIP				X
LISP Tutor				X
PROUST				X
APT				X

Table 15 Focus of problem solving and program development process in intelligent programming environments

Intelligent Programming Environments	Understanding Problem	Planning Solution	Designing Solution	Implementing Solution
Bridge		X		X
GIL				X
ITEM/IP				X
DISCOVER		X		X
ELM-PE				X
Capra		X		X
INTELLITUTOR				X

3.2.2 Overemphasis on Language Syntax

A major shortcoming of present tools is an intense emphasis on syntax. The student is led directly into the implementation stage, without any allusion to the problem solving activities. Research results (Soloway, Ehrlich, Bonar, & Greenspan, 1982; Perkins, Schwartz, & Simmons, 1988; Shackelford & Badre, 1993) have consistently uncovered student difficulties with programming beyond the scope of syntax. Despite this, the overwhelming majority of the tools continue to solely address the implementation stage of the software process. Even the tools that do not solely focus on the coding task devote a disproportionate amount of attention to this activity compared to other software process activities (Bonar & Cunningham, 1985; Hohmann, Guzdial, & Soloway, 1992; Forcheri & Molfino, 1994).

Moreover, current systems are language specific. Both intelligent programming environments and intelligent tutoring systems include features which require the students to solve problems by writing simple code segments or short programs using a programming language such as Pascal, LISP or Prolog. These systems provide useful facilities such as syntax editors, partial code execution, data structures and memory animation, and an interface designed to help the student with the syntax. But they lack facilities to assist students in focusing on the problem solving tasks. The LISP tutor, an extensively referenced system, is used as an example to illustrate this point. *Figure* 2 shows the beginning of a coding session for function '`create-list`' using the LISP tutor and is redrawn from Anderson, Corbett, Koedinger, & Pelletier (1995). There are a few problems with this example:

1. The student is immediately engaged in syntax construction.

2. The student does not have to do any investigation of problem requirements. For example, the wording of problem description includes phrases like "*accepts one argument*".

3. The student is not given the opportunity to think about possible solutions, rather; the student is told "*You should count down in this function*", or "*...just insert each new number into the front of the result variable.*"

Define a function called "create-list" that accepts one argument, which must be a positive integer. This function returns a list of all the integers between 1 and the value of the argument, in ascending order. For example.

(create-list 8) returns (1 2 3 4 5 6 7 8).

You should count down in this function, so that you can just insert each new number into the front of the result variable.

CODE for create-list

```
( defun create-list <parameters>
        <process>)
```

Figure 2 The beginning of a coding session for function '`create-list`' using the LISP tutor (redrawn from Anderson, Corbett, Koedinger, & Pelletier, 1995)

Even where facilities for problem solving are provided, such as Bridge (Bonar & Cunningham, 1985) and MEMO II (Forcheri & Molfino, 1994), these systems appear to be rigid and complex to use. For example, Bridge allows the student to build a natural language specification for the problem and the solution by choosing keywords from a menu. But the students must continue to deal with a restrictive syntax and a limited vocabulary.

It is even more complex to use MEMO II, where the initial activity of problem representation is performed using the computational interpretation of algebraic specifications, a syntactically complex language that consists of a set of predefined operators. This basically requires the students to learn the syntax of another language, one using terminology that approaches the complexity of any other language, to define the problem.

Given that the system is designed for novice programmers, this seems a contradiction of the stated goals. The system would be better suited for intermediate students. *Figure* 3 shows an in-progress Prolog program to find whether a natural number is even or odd, redrawn from Forcheri and Molfino (1994).

<table>
<tr><th>DIALOG</th><th colspan="3">INFO</th></tr>
<tr><td>FILE NAME: natbool
LANGUAGE: prolog</td><td colspan="3">NO ERRORS ON SPEC</td></tr>
<tr><th>PROBLEM</th><th colspan="3">WORKING</th></tr>
<tr><td rowspan="4">OBJECT natbool IS
SORT nat
CONSTRUCTOR
0: -> nat;
succ: nat -> nat;
SORT bool
CONSTRUCTOR
true: -> bool;
false: -> bool;
OPERATION even: nat -> bool is AXIOM
even (0) = true;
even (succ(X))=odd(X).
OPERATION odd: nat -> bool is AXIOM
odd(0)=false;
odd(succ(X))=even (X).</td><td colspan="3">even (0,true):-!
even (X,Y):- Z is X-1, odd(Z,Y).
odd(0,false):-!.
odd(X,Y):-Z is X-1, even (Z,Y).</td></tr>
<tr><th colspan="3">OPERATION</th></tr>
<tr><td>Design</td><td>Coding</td><td>Utilities</td></tr>
<tr><td>Edit
Verify
Prove</td><td>Translate
Include
Run</td><td>Save
Quit
Help</td></tr>
</table>

Figure 3 An in-progress Prolog program, using MEMO II, to find whether a natural number is even or odd (redrawn from Forcheri and Molfino, 1994)

3.2.3 Inadequate User Interface

The inadequacy of the user interface has also been criticized. Students must typically spend considerable time learning how to use the system, creating the need for tutorial support to overcome basic operational difficulties. With these systems, student dissatisfaction with the user interface is frequent (Eisenstadt, Price, & Domingue, 1993).

Problems with the user interface were reported with many of the graphical systems. For example, students who used Bridge (Bonar & Cunningham, 1985) complained of difficulties with the system's visual display. Advanced programmers familiar with text based languages found Pict (Glinert & Tanimoto, 1984) to be confusing and difficult. Students who used the LISP tutor (Anderson & Reiser, 1985) complained about the system's menus.

Protocol analysis is a technique developed in order to examine the thought processes involved in problem solving (Newell & Simon, 1972), but it is commonly utilized as an important measure of information systems' ease of use (Turoff & Hiltz, 1997). It is not apparent if either pre-deployment testing or protocol analysis were performed for these systems. For example, there is no report in the literature of protocol analysis performed by prospective users before the systems were used in the classroom. Such testing and analysis is vital given the inexperience of the users, the intricacy of the tools, and the complexity of the user interface.

3.2.4 Incomplete Rules-and-Errors Knowledge Bases

Intelligent systems evaluate student programs based on comparison with stored "*ideal solutions.*" A knowledge base of solutions, errors and rules is maintained for comparison

and verification. Such a knowledge base is intrinsically incomplete because it is impossible to include a pattern for every solution and a complete list of errors (Haga & Kojima, 1993). Because the solution knowledge base is incomplete, students will inevitably propose correct solutions deemed invalid by the system (Sleeman & Brown, 1982). The error knowledge base or bug catalog, typically organized as a collection of errors categorized by type, is also ineffective because of its incompleteness (Rosenberg, 1987; Eisenstadt, Price, & Domingue, 1993). Thus, student responses that reflect factors not represented in a model solution may be incorrectly categorized as errors (Snow & Swanson, 1992), or students may make "undocumented" mistakes that are placed under the wrong error category (Wenger, 1987; Snow & Swanson, 1992), as happened with PROUST. Johnson and Soloway (1985) characterize three kind of programs that will *always* cause PROUST problems, due to knowledge base incompetence, and require the aid of a teacher. Those are:

1) unusual bugs;

2) programs containing novel plans; and

3) ambiguous cases requiring interaction with the student.

It is unrealistic to expect such cases to occur infrequently. It is not unusual to encounter programs containing one, a combination, or all of these characteristics. Novice students will likely write programs that contain unusual errors or use a creative way to solve a problem.

The incompleteness of a rule-based system is normally ameliorated by having a domain-specific expert maintain and extend the knowledge base over time (Sleeman & Brown, 1982; Snow & Swanson, 1992). This was not done in any of the tools we have

considered, and in any case would have to be done by a programmer or an experienced user, complicating the application of the system.

3.2.5 Simplicity of Problem Domain

The simplicity of their examples is another concern regarding the problem solving approach of intelligent systems. As observed, this is inevitable given the limitations intrinsic to the rules-and-errors knowledge base essential to an intelligent tutoring system (Haga & Kojima, 1993). The tools actually address this incompleteness problem by the expedient of using only highly specified and simple problems for which reasonably complete knowledge bases can be developed. Thus, the very "problem simplicity" that enables the knowledge based system to behave as an expert simultaneously reduces its usefulness once the student is ready to move beyond the basics of the subject. The reliance on a limited number of teaching paradigms and simple examples makes these systems too restrictive to suit student needs, reducing their ability to provide significant experiences (Eisenstadt, Price, & Domingue, 1993).

3.3 Practical Deficiencies

Do current tools solve the problems of learning programming? Are the characteristics of these tool driven by actual classroom needs? And what are the effects of these systems on students? An attempt to answer these questions is handicapped because many of these systems remain prototypes (Rosenberg, 1987; Snow & Swanson, 1992) and there are few reports on evaluation and integration into the curriculum. *Table* 16 summarizes the

practical deficiencies in programming environments, debugging aids, intelligent tutoring systems and intelligent programming environments.

Table 16 Practical deficiencies

Practical Deficiencies	Reasons
Limited Classroom Evaluation.	Reports on evaluation results are insufficient and inconclusive.
Failure to Integrate Tools into Curriculum.	Tools remain isolated and have no direct relationship to learning taking place in classroom.
Impede Creativity and Development of Higher Order Thinking Skills.	Imposing constraints on structure of program as student constructs it to assure that certain solution is found.

3.3.1 Limited Classroom Evaluation

It is important to understand the impact of these tools on student/teacher dynamics and the effect this might have on computer science education. Evaluation of post-system deployment is an important assessment technique; however, little such evaluation has been reported. Those experiments that were done, almost invariably report success and improvement in student performance, but offer little supporting detail. An examination of the reported evaluations reveal a variety of problems with experimental design, including frequency of experiments, selection of course sections, number of subjects, the type of evaluation performed and the conclusions reached by the authors.

For example, two evaluations were conducted in the case of the LISP tutor (Anderson & Reiser, 1985 & Anderson, Corbett, Koedinger, & Pelletier, 1995). Information on the number of students involved and a clear numerical discussion of the outcome is not provided. The first study stated that "generally, students are happy with the tutor and rate it better than learning experiences they had in other introductory

programming courses." However, it is not clear what kind of students were involved in this evaluation, what introductory courses are referred to, and under what conditions these courses were taken. The LISP tutor was compared to experienced human tutors and self-learning conditions. But, the tutors' level of experience was not specified, nor were the conditions of self-learning. The first study concluded that "the human tutor is still the best, the computer tutor not far behind (and constantly improving), and the traditional on-your-own (self-learning) condition much worse". The second evaluation reported similar results.

PROUST (Johnson & Soloway, 1985) was evaluated using a simple programming example that computed the average rainfall in a day. It was pronounced successful, although the authors identified three kind of common program errors that cause the tutor problems.

SODA (Hohmann, Guzdial, & Soloway, 1992) was used at a high school but not evaluated. A predecessor to SODA, called the GPCeditor, was used and evaluated, but no details were reported. Despite this, the authors still discussed the expected outcome of SODA in terms of actual outcomes from the GPCeditor (Hohmann, Guzdial, & Soloway, 1992).

The lack of adequate evaluation is common to many of the systems. The following are some additional examples. The evaluation of Bridge (Bonar & Cunningham, 1985) was done with 10 students. APT (Corbett & Anderson, 1993) was evaluated in one semester using 41 students. DISCOVER (Ramadhan & du Boulay, 1993) was evaluated using eight students, four in the experimental group and four in the control. ELM-PE (Weber, 1993) was evaluated with a total of 20 students, 10 novices

and 10 advanced. After one semester of evaluation, ITEM/IP (Brusilovsky, 1992) was said to have fulfilled its research mission. ASA (Guimaraes, de Lucena, & Cavalcanti, 1994) was evaluated in two steps, first using eight students and then 20 students. The program understanding ability of INTELLITUTOR (Ueno, 1994) was evaluated using two sorting algorithms. Lens (Mukherjea & Stasko, 1994) was evaluated but the result, other than that the system was fully operational, was not provided. No evaluation was reported on Capra (Verdejo, Fernandez, & Urretavizcaya, 1993), MEMO II (Forcheri & Molfino, 1994), SCHEMACODE (Robillard, 1986), or VIPS (Isoda, Shimomura, & Ono, 1987; Shimomura & Isoda, 1991).

3.3.2 Failure to Integrate the Tools into the Curriculum

To measure the impact of intelligent systems, programming environments and debugging aids on computer science education and the delivery of introductory courses, they must receive widespread acceptance. However, it is questionable whether these systems have been integrated into the curriculum. There is no evidence in the literature that true integration has taken place; by *integration*, we mean a system has become an important and integral part of the learning process and is regularly used by students and teachers to enhance the learning environment.

The literature lacks any clear description of how these systems are being used in the curriculum beyond the reported research studies and limited evaluations. Indeed, it appears that the tools largely remain prototypes or, exceptionally, are isolated and with no direct relationship to classroom learning (Rosenberg, 1987; Snow & Swanson, 1992).

Researchers admit a number of factors had not been addressed (Anderson, Corbett, Koedinger, & Pelletier, 1995), including:

- No attempt to consider the curriculum content.
- Ignoring overall curriculum objectives in favor of immediate results, such as performance on a specific exam. Students' needs beyond the tutor were not addressed.
- The inflexibility of the tutors hinder teachers' ability to reconcile their instructional methodology with the tutoring system.
- Lack of support for teachers once the tools were deployed in the classroom.

Another reason for these systems lack of widespread acceptance is hardware requirements. AI systems in general tend to be large, complex, and expensive; educational software is not much different. These tools require considerable computational power. Many were designed in research laboratory settings to run on mainframe computers that can be accessed from school facilities, creating time and place restrictions that hindered students from taking full advantage of systems designed to be self-paced learning tools.

3.3.3 Impede Creativity and Development of Higher Order Thinking Skills

Intelligent systems present the student with a simple problem containing a clear definition, specifications and constraints. The student is then *led* into finding the "*ideal solution.*" The systems monitor student activity very closely and adapt to their responses, but never relinquish control (Marco & Colina, 1992; Snow & Swanson, 1992). Therefore, students often become dependent on the system's ability to *lead* them in

solving the problem. Imposition of such barriers to creativity and the acquisition of higher order thinking skills undermines student cognitive development. This is a serious drawback, considering that these are the very skills the intelligent systems propose to teach. For example, it is often the case that a problem can be solved in a variety of ways. The problem solver may also explore alternative solutions to a particular problem. Students may identify correct solutions that are judged erroneous by the system because the solution is not within its domain. Such is the case with the LISP tutor, which is designed to prevent students from diverging off the optimal solution path (Marco & Colina, 1992; Snow & Swanson, 1992). Indeed, the rule in these systems seems to be imposing additional and artificial constraints on the structure of programs as students construct them in order to assure certain solutions. This inhibits student creativity, unnecessarily confining and restricting the student to the systems' pre-defined boundaries.

The effects of these systems on the development of students' higher order thinking skills must be addressed. Therefore, we ask: Is the development of higher order thinking skills an important factor to be considered when designing tools to support students in learning problem solving and program development? Higher order thinking skills include an understanding of problem solving methodologies, creative and critical thinking skills, logical and reasoning skills, analysis, synthesis and evaluation abilities, and cognitive strategies. The following (Resnick, 1987) are key characteristics of higher order thinking skills:

- The total solution path is not visible, conceptually, from any single vantage point.

- Higher order thinking yields multiple solutions, each with distinct costs and benefits, rather than unique solutions.
- Higher order thinking involves the application of multiple, conflicting criteria.
- In contrast to guided learning, higher order thinking involves self-regulation of the thinking process.

We question the wisdom of merely obtaining a solution, rather than focusing on the entire problem solving process. While under the "*ideal*" circumstances of "*guided learning*", it may not be harmful to jump right into the development and implementation of a solution, the students may not appreciate the consequences of their mistakes. Intelligent systems continually "*correct and guide*" the student back within the boundaries prescribed by the "*expert behavior*" for an "*optimal solution,*" eventually "*leading*" the student to a solution, though not necessarily one of the student's own devising.

3.4 Summary

The reported evaluations indicate that intelligent systems have proved partially effective as support tools for the novice programmer, but, in spite of their complexity and expense, have provided only modest results (Rosenberg, 1987 & Ramadhan, 1992). In any case, many of these systems were not integrated into the curriculum or adequately evaluated. The investment in developing such systems appears not to have paid off (Lippert, 1989). While such systems do show notable success in teaching syntax and language constructs, they do not appear as successful in the comprehensive teaching of problem solving skills.

The type of problems these systems ask the students to solve are simple and well-defined. The solutions to these problems are circumscribed by the knowledge domain of the system. Students delve into the solution immediately and are confined to trying to identify the correct solution under "*coaching*" from the system.

The effect of this methodology is debatable. It is known that students can solve problems without even understanding required concepts (Halloun & Hestenes, 1987). This is even more feasible in the context of the intelligent system. The student may not experience meaningful problem understanding, planning and design tasks using such systems, even though these activities constitute a large and essential part of the problem solving process.

The findings of this research coincide with other research that has challenged the ability of intelligent systems to seriously meet the needs of novice programmers (Rosenberg, 1987; Lippert, 1989; Ramadhan, 1992; Snow & Swanson, 1992; Eisenstadt, Price, & Domingue, 1993). Indeed some researchers have questioned whether educational need or merely available technology is the motivation behind these tools (Rosenberg, 1987). Others concede there have been no clear indications of success for intelligent programming tutoring and debugging systems (Ramadhan, 1992; Snow & Swanson, 1992).

Programming environments, including those with development facilities, and debugging aids suffer from similar problems: (1) they focus on the programming aspect of the process, and (2) they provide a totally unstructured facility for problem solving. Only the functionalities required to construct, test and debug code are provided.

Traditional programming environments, of course, provide even less support for problem solving. They are tools to support only the programming aspect of the software process. A language compiler is the centerpiece of such facilities. Normally an editor, a programmer's workbench, and various additional tools are the components used to carry out the tasks of coding, testing and debugging. Predefined libraries and utilities are also provided (e.g. the Turbo Pascal Environment). Apart from the editing and debugging tools, traditional programming environments offer no support for the software process. The problem definition, planning and design stages are totally neglected by these tools. To a large extent, the software process with these systems remains a methodology disconnected from the language and the tool (Yeh, 1990). This may perhaps be viewed as a natural result of using a tool (the compliler) designed for one purpose - the non-interactive translation of high-level language programs into executable code - for another - teaching programming skills.

In large development environments, the entire software process may be aided by tools. Progress toward software development systems encompassing the language, the methodology, and technology has been made (Yeh, 1990). For example, a Computer Aided Software Engineering (CASE) tool may be used in various stages of the software process, most commonly in analysis, design and implementation (Mimno, 1990). Some programming environments now include tools used for the design of the user interface, data modeling, code generation, and integration (e.g. the Visual C++ Environment). Programming environments designed for students must similarly offer functions beyond mere specialized components (Papert 1980, Brusilovsky, 1993). Chapter 4 presents the

theoretical foundations for such a system and Chapter 5 includes the design specifications for a system to be used by novice students learning problem solving and programming.

CHAPTER 4

PROBLEM SOLVING, PROGRAM DEVELOPMENT AND COGNITION

This chapter introduces the concepts and terminology of problem solving, carefully compares a number of classic problem solving methods, and then synthesizes a *common method* incorporating the essential features of the classic methods. We then review the problem solving tasks specific to program development, identifying how to adapt or enhance the general common method to the area of program development. We next examine the cognitive science and learning theory relevant to problem solving, and define a *cognitive model* of problem solving. Finally, we identify for each task of the common method the appropriate cognitive techniques and skills required, thus defining a *Dual Common Model* which integrates problem solving methodology and program development tasks with the cognitive techniques needed at each step of the process. This Dual Common Model serves as the basis for the specification of SOLVEIT presented in Chapter 5.

4.1 Problem Solving

Problem solving is often described using state transition terminology. A problem solver goes through a sequence of subjective mental states or processes (or operations on information in the subject's memory) which progress toward the goal of solving a problem (Mayer 1983). Objectively, the problem solver creates a sequence of problem transformations, which transform the given problem from an *initial state* to a *goal state,* and which taken together define a path to the *solution* (Simon, 1978; Mayer, 1983).

Figure 4 illustrates the correspondence between the subjective mental processes of problem-solving and the corresponding objective transformations of a problem statement into a solution.

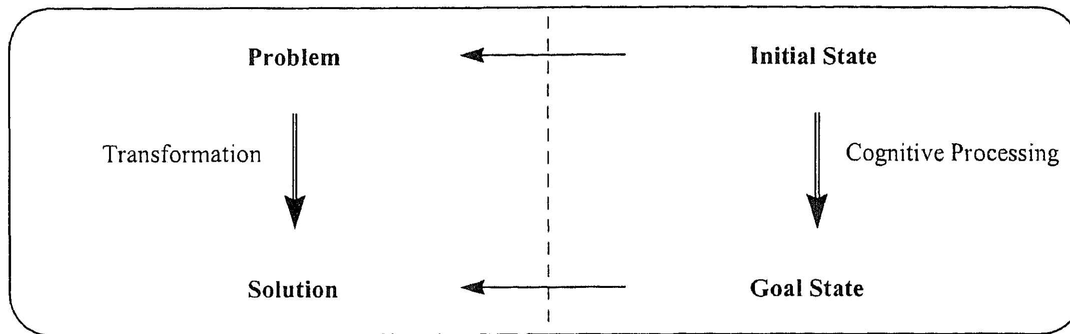


Figure 4 Transformation of a problem statement into a solution

A well-defined problem statement contains three principal parts*: goal, givens, and unknowns*, which are shaped by the process of problem solving into a solution. A problem may also contain additional important elements that must be recognized and defined, such as *conditions, constraints, and operators*. Identifying the problem's goal, givens, unknowns, conditions and constraints, based on the current representation of the problem, is the first step in problem solving. The process starts with formulation of a general representation of a solution, and progresses to a more highly specified solution through a series of *reformulations* (Polya, 1945 & 1962; Schoenfeld, 1979). One way to accomplish such transformations is to redefine the problem into subproblems and restate the goal in terms of *subgoals* (Duncker, 1945; Newell & Simon 1972; Wickelgren, 1974; Rubinstein, 1975; Mayer 1983). In order to find the *unknown*, one applies a sequence of *operators* on the *givens* of the problem, changing the *problem state* from an initial

representation of the problem, into a final representation of a solution (Newell & Simon, 1972).

4.1.1 The Terminology of Problem Solving

Terminology for the various elements of problem solving has developed over time. The most common terms are defined below, following (Polya, 1945, Duncker, 1945; & 1962; Newell & Simon 1972; Wickelgren, 1974; Rubinstein, 1975; Mayer, 1983).

4.1.1.1 Goal, Givens and Unknowns

A *goal* is what one wants to accomplish. It must be extracted from the problem statement by the problem solver and then appropriately represented. The goal is accomplished by a sequence of problem transformations. A well-defined problem begins with a representation of its specific facts that must be identified prior to solving the problem, and which are known as the problem *givens*. A problem begins with a single goal, but may have many unknowns. The goal tends to be a general statement of the problem objective, while the unknowns are the detailed, particular things that have to be found out in order to accomplish this goal. As one example of this terminology, consider the problem of sorting a list. The given is the unsorted list. The goal is to rearrange the list so it is sorted. The unknown is the sorted list. In this case, the goal and the unknown are almost synonymous. In a more complex problem, such as where the goal is to design a payroll system, there may be many unknowns, such as: gross pay, net pay, tax deduction, etc., that have to be determined in order to accomplish the general goal.

4.1.1.2 Conditions and Constraints

Conditions and *constraints* are qualifying factors that must be taken into consideration when solving a problem. Conditions tend to be logical restrictions, while constraints tend to be quantitative restrictions. Conditions are similar to givens since both are existing factors imposed in developing the solution. Constraints include, for example, restrictions on the types of operators that can be used, their frequency, the conditions under which they can be used, or the sequence they must follow (Wickelgren, 1974; Mayer 1983). Any solution path from the initial to the goal state must satisfy the problem's conditions and constraints. In particular, performing an operation that changes the problem state from one state to another must either satisfy all or part of a condition or constraint.

4.1.1.3 Subgoals and Tasks

Subgoals are identified by restating the problem goal in terms of subproblem goals. This refinement of the goal into subgoals makes it easier to devise a solution (Duncker, 1945; Newell & Simon, 1972; Wickelgren, 1974; Rubinstein, 1975; Mayer 1983). Divide-and-conquer with step-wise refinement is one common method for identifying and integrating subgoals (Wirth, 1971). The work required to achieve a subgoal is called a *task*. The integration of these tasks forms a complete solution.

4.1.1.4 States and Operators

The *states* that a problem takes on during problem solving can be distinguished into the initial, problem and goal states. However, the problem itself may be ill-defined, so it is the problem solver's first responsibility to create a well-defined initial problem

representation, which constitutes the problem statement. This may be a written, verbal, or visual description of the problem's elements. The *initial state* is the situation represented by this well-defined problem statement. In general, a *problem state* is a description of the elements of the problem at any stage of the problem solving process. The *goal state* is reached when a solution is found and the goal is met.

An *operator* is a function that accomplishes a task, moving the process closer to the solution. Operators transform problems from one state to the next (Ernst & Newell, 1969; Simon, 1978; Mayer, 1983). Operators can be represented as sequences of steps or algorithms that take into consideration the conditions and constraints of the problem. Successive problem states are produced by applying *operators* that transform one problem state to the next.

4.1.2 Categories of Problems

Two main kinds of problems are recognized in problem solving research (Rubinstein, 1975): problems of analysis, also known as transformational problems, and problems of synthesis, which are the kind of problems frequently faced by engineers. Problems may have characteristics of both analysis and synthesis.

Rubinstein (1975) defined a *problem of analysis* as one in which the solution consists of a series of transformations, or changes in the representation, of a given problem statement into a final solution. Greeno (1978) referred to these as *problems of transformation* (p. 245) where, given an initial state and a desired goal, a set of operators are defined and applied to produce the solution. Polya (1945) suggests using regressive reasoning or solution backwards (p. 142) to solve such problems. In this technique, one

works backwards from what is required, developing and executing a plan which always keeps in mind the goal or result sought, until one eventually arrives back at the problem givens.

In contrast, a *problem of synthesis* is one in which the major effort is in selecting and integrating known components to achieve a desired goal. While the problem as a whole may be new, the individual steps needed to solve the problem are not (Rubinstein, 1975). Greeno (1978) referred to these as *problems of arrangement* (p. 255): given some components, the task is to find a combination of components which meet the solution criteria. Polya (1945) refers to these as *problems of construction* (p. 23) and suggests solving them using progressive reasoning, which starts with what is already known, such as the plan of action. As an example, integrating the known kinds of components of a computer into a desired architecture requires selecting components with appropriate characteristics, for each type of component category, in such a way that the selected components can be integrated into a system which meets the requirements of the problem.

Computer scientists address both types of problems, but most of the problems in a first course on programming are of the transformation type. In such programming problems, the initial conditions and the desired result are known, but the tasks required to solve the problem must be established by the problem solver. Problem solving methods and systems that support the solution of synthesis problems require a very different set of capabilities. They tend to be highly domain specific and knowledge dependent. For example, in designing a house (a synthesis problem), the known components: walls, doors, windows, etc., come in a variety of forms. A variety of basic plans are also available to the overall architecture of the house. The solution entails first selecting a

plan to meet specified characteristics from a (knowledge) base of plans, then selecting appropriate components that can be integrated within the plan. The solution is both knowledge-based, for the overall plan and the individual components, and highly domain-specific: a system that supported "solving" a house would be totally different from the system that supported defining a computer architecture. Since this dissertation is about problem solving in the context of programming, we focus on problems of analysis.

4.2 Problem Solving Methods

This section describes twelve different models of problem solving from Dewey (1910) to Hartman (1996). The Polya (1945) model, already described in Chapter 1 and used as the basis for the problem solving and program development model proposed later in this chapter, is reconsidered and related to these other models of problem solving. This review legitimizes the choice of Polya's as the basic frame of reference for our model. Where possible, the authors' original terminology has been retained in the following discussion.

Interest in problem solving is not new, with major developments that still influence current problem solving methods reaching far back into history. The work of Rene Descartes (1596-1650) on geometry was an important milestone in problem solving (Rubinstein, 1975; Grabiner, 1995). Descartes (1637) in his *Discourse on Method* observed that the problem solver must go about things in the right way and must use the right *method* to arrive at a solution, otherwise nothing will be discovered. Further back still, there was the work of Alkowarazmi (A.D. 825) from whose name the very word *algorithm* is derived (Rosen, 1995). Euclid's (300 B.C.) *Elements* was of fundamental

significance to the whole systematic enterprise of science (Rosen, 1995), and even provided Descartes with the first problem on which he applied his new "method" (Grabiner, 1990). However, since our work is concerned less with historical developments in problem solving, than with the *process* of problem solving itself, we restrict our review to research starting early in this century.

4.2.1 Early Models

Two of the earliest methods for problem solving were given by Dewey (1910) and Wallas (1926), and represent opposite approaches. Dewey's approach essentially articulates the scientific method for problem solving, while Wallas' approach represents the non-systematic, creative view of problem solving.

4.2.1.1 Dewey's Model

Dewey (1910) divided the problem solving process into four stages: (1) Define problem - document exact nature and requirements of problem; (2) Suggest possible solutions - identify possible alternative solutions; (3) Reason about the solutions - assess, select, plan, and implement solution; (4) Test and prove - evaluate and verify results through experimentation and validation.

Dewey's model resembles the classic "scientific method," following an orderly process of observation, analysis, hypothesis and experimental validation.

4.2.1.2 Wallas's Model

Wallas (1926) identified four stages in problem solving: (1) Preparation - formulate problem and gather information about the problem; (2) Incubation - unconscious thinking about problem while engaged in other activities; (3) Illumination - gain insight into problem and discover solution; (4) Verification - inspect solution for accuracy.

This model resembles the "sudden solution" or "Eureka" method described by Hadamard (1945) and the "creative method" of Poincare (1913); see also Couger (1995). It describes the introspective accounts and personal observation of prominent scientists regarding their own *creative* process when they solved problems. The phenomenon was recorded by Descartes who described a vision he had that led to his discovery of Cartesian coordinates, establishing the link between algebra and geometry (Rubinstein, 1975); similarly, there is the famous anecdote of the French mathematician Poincare who described how he solved an important mathematical problem while traveling by bus, even though he was not (consciously) thinking about the problem.

4.2.2 Enhancements to Earlier Models

Subsequent models combined elements of both the scientific and the creative approaches. Principal among these is Polya's famous work on problem solving. The Polya model (1945 & 1962) elaborately specifies a problem solving method supported with examples and documented in a series of books. Independently, Johnson's model (1955) refers to Wallas, while Kingsley and Garry's model (1957) elaborated on Dewey. A separate, but similar, model was presented by Osborn (1953) and Parnes (1967).

Neither Johnson nor Kingsley and Garry introduced significant development over their predecessors. Despite the independence of these three methods, they are basically consistent in their approach, an important indication of the stability of the methodology over time. A different approach was introduced by Simon (1960) who viewed the process as a collection of four cognitive abilities: intelligence, design, choices and implementation.

4.2.2.1 Polya's Model

Polya (1945 & 1962), a prominent mathematician, wrote a series of books on problem solving that are considered an outstanding contribution to the study of problem solving. In two of his works, *How to Solve It* and *Mathematical Discovery*, he presented a general *method* and applied it to solve many types of problems. Polya's model is among the most widely used and referenced framework for a problem solving methodology. This model, based on Polya's classroom experimentation and his own teaching observations, comprises four stages: (1) Understand the problem - state the question, identify goal, givens, unknowns, conditions, and their relationship. The student may draw a figure and introduce suitable notation; (2) Devise a plan - outline a potential solution. Look at a similar problem, restate the problem differently and break into parts; (3) Carry out plan - refine and transform plan into a solution to the problem, decompose tasks and relate to givens and unknowns, check validity of each step, and define each in relation to whole problem; (4) Look back - confirm results and arguments, assess effectiveness of solution, accuracy of results and usefulness of solution for other problems.

4.2.2.2 Johnson's Model

Johnson (1955) presented a variation on Wallas' creative method and divided problem solving into three stages: (1) Preparation - understand the problem, gather relevant information, and plan possible solutions; (2) Production - construct solution and produce results; (3) Judgment - examine solution and results to determine accuracy, effectiveness, and suitability of solution.

4.2.2.3 Kingsley's and Garry's Model

Kingsley and Garry (1957) presented a variation on the Dewey's scientific method that includes five stages: (1) Clarify and represent problem - investigate and define problem requirements; (2) Search for clues - identify and learn about alternatives for solving the problem; (3) Evaluate alternatives - review and compare alternatives to select most suitable solution; (4) Accept an alternative - select the best alternative and refine it into final solution; (5) Test solution - validate results through experimental observations.

4.2.2.4 Osborn's and Parnes' Model

Osborn (1953) initially presented a three-stage model that included: fact finding, idea finding and solution finding. Later Parnes (1967) revised it to include two additional stages as follows: (1) Find facts - identify and analyze information; (2) Find problem - produce representation of the problem; (3) Find idea - examine and develop alternatives for a solution; (4) Find solution - evaluate alternatives, select and refine a solution, and formulate a plan for implementation; (5) Find acceptance - carry out the plan and produce results.

4.2.2.5 Simon's Model

Simon (1960) offered a four component model: (1) Intelligence - ability to recognize a problem, gather pertinent information, and produce an accurate definition; (2) Design - plan and generate possible solutions; (3) Choice - select and implement a suitable solution from available alternatives; (4) Implementation - put choice into effect and produce the solution. Each of Simon's components encompasses a set of skills comparable to the ones required in other methods, such as those suggested by Dewey and Polya.

4.2.3 Recent Methods

More recent methods were developed to provide mathematics, science and engineering students with an explicit method for problem solving. Generally, these models divided the problem solving process into a more finely specified process than the earlier methods. Notable among these models is the work of Rubinstein (1975), who introduces an element of reservation. One such reservation is at the problem understanding stage where he looks at possible solutions before finalizing the problem statement; there is a similar withholding of commitment at the final problem solution. Otherwise his method represents the standard view. Similar observation were made by Halloun and Hestenes (1987). Other popular methods are Etter's (1995) and Hartman's (1996), who presented models that basically follow the Polya model without any radical changes.

4.2.3.1 Rubinstein's Model

Rubinstein (1975) presented a six stage model (taught to engineering students at UCLA) described in his book *Patterns of Problem Solving*: (1) Get total picture - examine problem elements until a pattern emerges. Defer working with details; (2) Withhold judgment - search for a number of possible solutions without committing to any course of action; (3) Model - verbalize and communicate; write down statement of problem using mathematical and pictorial symbols, if needed; (4) Change representation - devise a plan for solution and apply transformations that simplify the solution process; (5) Ask questions - change the frame of reference while searching for information and solution patterns; (6) Doubt results- accept premises as tentative until proven. In other words, verify the outcome.

4.2.3.2 Stepien's, Gallagher's and Workman's Model

Stepien, Gallagher, and Workman (1993) offered a standard , now familiar view of the methodology: (1) Analyze problem - understand problem. Discuss with others, if possible; (2) List what is known - write down information known about the problem; (3) Develop problem statement - describe what is to be solved and produced from what is known about the problem; (4) List what is needed - write down questions to be answered, information to be found, and concepts or principles to be learned; (5) List possible actions - identify tasks to be completed and consider potential solutions; (6) Analyze information - inspect information gathered and implement appropriate solution; (7) Present findings - verify outcome and report results.

4.2.3.3 Etter's Model

Etter (1995) presented a general problem solving model, another close variation of Polya's, used by students to solve engineering and science problems both manually and by computer. The model includes five steps: (1) Define problem - state problem clearly; (2) Gather information - describe problem input and output; (3) Generate and evaluate potential solutions - find and assess possible solutions; (4) Refine and implement solution - select, develop and produce solution; (5) Verify and test solution - evaluate solution method and result.

4.2.3.4 Meier - Hovde's Model

Meier, Hovde & Meier (1996) introduced a recent instance of the standard model of problem solving as a method for teaching mathematics and science problem solving. The model also includes five steps: (1) Define problem - ask questions, collect preliminary data, and understand problem; (2) Assess situation - collect information, establish hypothesis, and begin investigation; (3) Plan strategy - establish strategy for solving the problem; (4) Implement plan - carry out plan, modifying it as need arises; (5) Communicate results - analyze and evaluate the outcome; share results.

4.2.3.5 Hartman's Model

Hartman (1996) describes an explicit model, similar to Polya's, to help students improve their thinking and problem solving skills: (1) Identify and define problem - describe problem, find givens, unknowns, and identify relevant/irrelevant information; (2) Diagram problem - draw a sketch of problem and arrange in relationship to each other;

(3) Recall content - search for required concepts, definition, and rules needed to solve problem; (4) Explore alternative strategies - find and plan most efficient way to solve problem. Break problem into parts and examine difficulties; (5) Apply content and strategies - combine knowledge and skills to carry out the solution; (6) Monitor work-in-progress - review approach and progress toward solution; (7) Assess solution product and process - look back at problem statement and answer. Check results for accuracy and completeness. Evaluate solution method and experience.

4.3 A Common Method for Problem Solving

The purpose of this section is to identify a common integrated model for problem-solving based on the models just reviewed. However, we first make some general observations. *Problem solving methodologies have stabilized over time, become more explicit, and are demonstrably natural.* The fact that the methods have settled down to a fairly well agreed-upon and detailed form indicates that they provide a reliable theoretical framework for the present work. The naturalness of these methods, in the sense that they are psychologically spontaneous, has been established by the work of (Duncker, 1945; Newell & Simon, 1972; Chi, Glaser, & Rees, 1982) using *thinking-aloud* verbalization, protocol analysis, and related experimental techniques. Thus, Newell and Simon carefully monitored how students thought about problems while solving them. The process identified was similar to the methods reviewed and consisted of a series of stages. Problem solvers began by trying to understand what was expected of them and by gathering and organizing information. Facts about the problem were then used to examine and plan possible solutions. The plan was then refined, executed, and tested. If

the putative solution was not confirmed, it was modified or new solutions were generated and the process was repeated.

Although the general form of the methodology is clear from the review, it will be beneficial to carefully synthesize these methods into a coherent, comprehensive model for problem-solving. Polya's (1945 & 1962) method captures the essential features of these problem-solving approaches, and so provides an established, recognized (Grabiner 1995) framework which can serve as the basis for a problem solving method. Polya defined a four-stage process that required: formulating the problem, developing a plan for solving the problem, implementing the plan to produce a solution, and verifying and presenting the results. A synthetic view of the detailed tasks involved by these steps and based on all the models follows. It represents a *common model* for problem solving. A more comprehensive model that addresses the cognitive and program development aspects of the process will be defined later in this chapter.

4.3.1 Formulating the Problem

If we identify all the significant recommendations by the different methods regarding the problem definition/understanding phase, then that stage includes the following. The key ingredient was captured by Polya: *State* the question, and *identify* the *goal*, *givens*, *unknowns*, and *relations*. Kingsley-Garry and Osborn-Parnes emphasize producing a *representation* of the problem. Polya's method accomplishes one such representation, though others are possible. Simon highlights the ability to *recognize that there is a problem* in the first place; however, our emphasis tends to be on problems that are given. Rubinstein's exhortation to *defer details* is implicitly addressed by any method, since a

method, by definition, enforces caution and clarification, constraining the impulse to charge blindly ahead. Nonetheless, Rubinstein's recommendation is a good guideline to keep in mind throughout the whole process of problem solving. Hartman recommends *diagrammatic* aids and an initial search for relevant *concepts*. Stepien, Gallagher, and Workman recommend *collaboration*, that is, discussing the problem with others.

Our approach includes all these elements, thus representing an inquisitive (Socratic) approach requiring: understanding the problem through verbalization, asking and answering questions, gathering information, restating the problem, introducing notations and drawing diagrams to visualize the problem and its solution: the goal being to identify the pertinent facts about the problem, ignoring inessentials. The initial problem state produced in this way is a description of the problem and an organized representation of all relevant information: the goal, givens, unknowns, conditions and constraints. All of this is subject to revision as problem understanding develops.

4.3.2 Planning the Solution

A review of the methods for this stage reveals two key recommendations: *identify alternative solutions* and *devise a plan*. Almost all the methods explicitly emphasize the necessity of *generating alternative solutions*, which are then *evaluated*, and from which *one is selected.* Polya, in contrast, recommends *examining similar* and/or *simpler problems* and *restating the problem*. Though apparently different, this is in fact just a *more fundamental recommendation* than "finding an alternative solution," because it provides an actual technique for generating solutions by examining simpler or alternative problems, which one may be able to solve, and whose solutions can then be adapted to

the current problem. This provides a technique, for example, for accomplishing what Wallas only recommends: gain insight into the problem and discover solution, or into Rubinstein's recommendation to: change the frame of reference and search for solution patterns. Once a solution is selected, Polya again provides the most inclusive recommendation; namely, *devise a plan*, by *outlining a potential solution* and *breaking the problem into parts*. The outline or plan for a solution is just a high level view of the solution. This high level view serves several purposes. It helps ensure the coherence of the implemented solution and its fidelity to the objective of the original problem, by deferring premature and distracting immersion in the details of implementation. Once such a high level view is defined, the next logical step is to refine the plan by breaking the plan/problem/solution into parts.

In summary, possible alternatives are assessed and a strategy for solving the problem is devised. The solution is more manageable when the problem is reformulated into a set of smaller subproblems. Therefore, the goal is refined into subgoals that are more easily achieved, the tasks to accomplish each subgoal are defined, and the relationship between the problem's givens and unknowns is established.

4.3.3 Implementing the Solution

If we identify all the significant recommendations by the different methods regarding the implementation of the solution, then that stage includes the following. Most of the methods explicitly emphasize the necessity to *select* a solution from generated *alternatives*, which is then *refined* and *produced*. The essential tasks were clearly stated by Polya in his carry-out-plan stage: *Refine* and *transform* the plan into a solution, and

decompose tasks. Others also call for refinements, transformations and decomposition. For example, Kingsley-Garry and Osborn-Parnes emphasize *refining* the solution, Rubinstein calls for *transformations* to simplify the process and Hartman recommends *breaking* the problem into parts.

In summary, the plan devised in the earlier stage must be implemented in order to produce the desired outcome. This is done by refining and transforming the plan into a solution to the problem. A transformation from a high level solution outline to a precise solution may require further decomposition of tasks, reorganization and specification of an explicitly stated algorithm.

4.3.4 Verifying and Presenting the Results

This is the last stage of the problem solving process and is the most similar among all reviewed methods. The standard recommendation of the different methods is *verify the solution*. All of the methods explicitly emphasize the necessity for *verifying solutions*. This verification procedure includes effectiveness of solution and accuracy of results. Many of the methods also emphasize the *evaluation* of solution suitability for other problems and, naturally, *sharing* and *reporting* results.

In summary, the main purpose of this stage is to produce an answer consistent with the goal of the problem. Therefore, the problem solver has to look back and evaluate the process and verify the correctness of the solution. This is determined by testing the solution using data and examining the results. An equally important purpose is to learn from the problem solving experience itself, acquiring knowledge and skills that

can be transferred to other problem solving situations. Finally, the solution and the results are presented in a readable and organized manner.

4.4 Program Development

Programming, as distinguished from the specific task of coding, refers to the activities involved in both designing and implementing programs in order to solve problems (Wirth, 1971). These activities may be limited to simple data representation, algorithm design, development and coding, but often, when dealing with large program development, more complex activities requiring additional skills and knowledge are required (Dijkstra, 1976; Boehm, 1976; Pressman, 1987; Page-Jones, 1988; Ng & Yeh, 1990). This section presents a synthetic view of the tasks required for program development. These tasks will later be integrated with the previously identified common problem solving methodology to define a Dual Common Model. These programming development tasks will also be useful later on when specifying the evaluation plan for the system.

4.4.1 Program Development Tasks

Programmers must develop skills which include: learning the language, composing new and comprehending existing programs, testing and debugging solutions, and documenting and modifying the programs they write. These are cognitive tasks related to language and require knowledge of the syntax and semantics of the programming language (Shneiderman, 1980; Rogalski & Samurcay, 1990, 1993). Other cognitive tasks, related to problem solving, such as problem understanding, analysis, and design of the solution,

require domain, strategic and tactical knowledge, as well as practical knowledge of the programming language (Wirth, 1971; Pennington & Grabowski, 1990). All of the following skills are required for program development.

4.4.1.1 Learning the Language

Acquisition of a programming language is the first important task (Hoc & Nguyen-Xuan, 1990). Students must learn and understand the syntax, semantics and pragmatics of language constructs and become familiar with the tools and utilities of the programming language environment they use. A programming language has three aspects: *syntactical*, *semantic*, and *pragmatic*. Syntactical knowledge refers to the ability to construct grammatically correct instructions in order to write a program. This task requires both accurate comprehension and detailed knowledge of the language rules, control structures, and data structures. Syntax includes the grammar of a programming language, such as the precise form for declaring a variable, constructing a repetition statement, forming a selection condition or referencing a specific location in a list of values. Semantic knowledge, in contrast, refers to functional understanding of the programming language and the meaning of its instructions, such as the behavior of the language's control and data structures. For example, given integer variables A and B, the syntax of the relational expression: $A \geq B$ refers to its numeric variables and relational operator, while the semantics of the expression indicates that $A \geq B$ is *true* whenever the value of A is larger than or equal to the value of B, and *false* otherwise. Pragmatic knowledge refers to an understanding of the context and use of language features, such as under what

circumstances recursion should be used as opposed to iteration, or when post-test looping is more appropriate than pre-test looping.

A knowledge of the syntax, semantics, and pragmatics of a programming language, combined with a knowledge of problem solving methodology, together constitute the foundation skills required to compose, comprehend, test and debug, document and modify programs.

4.4.1.2 Composing Programs

Program composition involves the representation of a solution for a problem in a specific programming language, which is usually referred to as *coding*. The detailed design is translated into instructions suitable for execution by a computer. Program composition is a principal task of program development, requiring close attention to implementation details and knowledge of language syntax, semantics, and pragmatics.

4.4.1.3 Comprehending Programs

Program comprehension involves understanding the code from data/control structure and design views, thus making it an especially inclusive task of program development. Program comprehension is not merely problem understanding, which relies on a rather different set of skills. Students must develop the ability to read a program in such a way that they understand its functionality and design. This requires comprehending the data representations, logic and data flows, the purpose of individual instructions and subprogram references, and the collective function of the program as a whole (Pennington

& Grabowski, 1990). It thus includes domain and strategic knowledge, as well as a knowledge of language syntax and semantics.

4.4.1.4 Testing and Debugging Programs

A program must meet its specification requirements and errors must be identified and fixed. Programs are tested for correctness at various stages of their development, both as a whole and in parts. Students should learn to perform code testing, to develop and use test data suitable for verifying program correctness, and to correct identified errors. Testing and debugging are complex tasks and require a thorough grasp both of domain knowledge and logic tracing skills not easily mastered. In addition to errors related to the syntax and semantics of language constructs, programs may also contain errors that result from mistakes arising at the problem definition and analysis phase, or errors that arise at the design stage of software development.

4.4.1.5 Documenting Programs

Program documentation is essential for both comprehension and modification of programs. Documentation may be *internal*, with comments and explanations embedded in the code describing the approach and techniques used in solving the problem. *External* documentation includes documentation developed prior to writing the code, such as algorithms, charts, data modeling, and end-user documentation.

4.4.1.6 Modifying Programs

Testing is successful if it finds errors. Thus the result of testing will typically entail changes to a program that may affect its logic, language constructs, or data representations. Programmers must also be able to modify programs in order to alter their functionality or adapt previously written code to solve new problems. The ability to modify a program, especially after deployment, depends on the availability of documentation, as well as the program comprehension and composition skills of the programmer doing the modification. Knowledge of language syntax, semantics, and pragmatics are paramount here too.

4.5 A Cognitive Model for Problem Solving

Cognitive psychology is relevant to this dissertation for several reasons. First of all, the cognitive skills required in problem solving must be identified before we can specify the cognitive functions required for a computerized environment that facilitates problem solving. Secondly, learning tools should promote the development of cognitive skills, so it is important to explicitly identify those skills and design systems which encourage their development. Finally, explicit recognition of the relevant cognitive skills will help in framing suitable hypotheses for system evaluation. In order to clearly address these issues, it will be useful to have a framework that explicitly identifies the cognitive elements of problem solving (Gabel, 1989), a so-called *cognitive model* for problem solving (Schoenfeld, 1985). The subsequent section will integrate this cognitive model with the previously developed program development and problem solving views. We will define the cognitive model after the following brief historical review.

Historically, cognitive psychology became a factor in research on problem solving and program development in the late 1960's and early 1970's with such work as (Sackman, 1970; Weinberg, 1971) on the psychology of programming, and has continued to be an active area of research (Shneiderman, 1980; Mayer, 1981 & 1988; Hoc, Green, Samurcay, & Gilmore, 1990; Lemut, du Boulay, & Dettori, 1993). The objective of cognitive psychology is to provide a more precise and detailed understanding of human cognition, which in turn should lead to improvements in problem solving, teaching programming, and ultimately to more effective software environments (Shneiderman, 1980). Some areas addressed in cognitive psychology include: comprehension and mental models, knowledge acquisition and processing, knowledge organization and management; and knowledge retention and transfer (Ormerod, 1990; Rogalski & Samurcay, 1990, 1993; Shih & Alessi, 1993, 1994; Bertels, 1994; Greeno, Collins, & Resnick, 1996). The related area of learning theory investigates learning, defined as the acquisition of knowledge and understanding of information, concepts, and strategies. Learning is a fundamental element of problem solving and an important element of such cognitive processes as memory, perception, and thinking (Lachman, Lachman, & Butterfield, 1979; Mayer, 1983). Problem solving itself requires a broad range of cognitive skills, abilities, and knowledge essential to recognizing, understanding, and utilizing facts, as well as planning, designing, and implementing solutions (Polya, 1945 & 1962; Simon, 1980; Mayer, 1983).

A *cognitive model* for problem solving should identify: the cognitive processes that problem solving uses, the (hypothesized) cognitive structures or systems that support these processes, and the cognitive results and affects on cognition of the problem solving

process. Accordingly, the cognitive model that we propose will have three elements: a set of *cognitive processes,* based on Bloom's research (Bloom, 1956); a *cognitive structure,* based on Sternberg's work (Sternberg, 1985); and *cognitive results,* based on Gagne's learning outcomes (Gagne, 1985). Bloom's work is the *most extensively referenced model* for the cognitive processes of thinking. Sternberg's well-known model of the human information-processing system postulates cognitive systems which are presumed to underlie these cognitive processes, thus defining a cognitive structure. Gagne's work identifies the cognitive outcomes and effects of these processes.

Bloom (1956), in his *Taxonomy of Educational Objectives*, identifies a two-level cognitive or thinking framework built on six cognitive processes: a higher-level set of processes where problems are analyzed, synthesized, and evaluated, and a lower-level which support knowledge, comprehension, and application. Bloom's conception of the processes involved in problem solving parallels Polya's work.

The higher level processes Bloom defined are as follows. *Analysis* refers to the strategies applied to problem solving using heuristic methods such as subgoal decomposition to break down a problem into its component parts, means-end analysis, and the use of similar problems or solution by analogy. Analysis includes identifying and hierarchically organizing parts (Polya, 1945; Lindsay & Norman, 1972, Mayer, 1983). *Synthesis* refers to the tactics applied to reintegrate the component parts, rearrange when necessary, establish their relationship, and produce a new and well-organized whole as a viable solution to the problem (Polya, 1945). *Evaluation* refers to judging the quality and correctness of solutions based on established criteria and the problem requirements.

Evaluating the adequacy of a process and its appropriateness for other situations is another element of evaluation (Polya 1945).

Bloom's lower-level processes are as follows: *Knowledge* refers to the ability to bring past problem solving experience to bear and to identify and recall relevant facts. This may include both general and domain knowledge, such as concepts, theories and principles, as well as basic observations and advanced understanding of subjects. Knowledge is demonstrated by awareness of specific facts about a problem available from within a problem statement, or by awareness of information about related problems (Polya, 1945; Lindsay & Norman, 1972; Mayer, 1983). *Comprehension* refers to the ability to interpret and understand the meaning of presented material and relevant information. This may be demonstrated by correctly explaining and answering questions about the problem, restating the problem in a different verbal, written, or visual form, and describing important facts about the problem (Polya, 1945). *Application* refers to the ability to use the knowledge and identified facts and to apply the recalled concepts, theories, or principles to plan a solution to the problem. This may be demonstrated by outlining necessary steps to reach a solution, solving a simpler problem, or drawing charts or graphs to represent the solution (Polya, 1945).

Sternberg's *Beyond IQ: A Triarchic Theory of Human Intelligence* (1985) defines a hypothetical architecture for human thinking based on three *components* or *componential categories*: knowledge acquisition, performance, and metacognition. Sternberg's hypothesized *Knowledge Acquisition Component* includes processes used in acquiring new knowledge, determining what is relevant, and integrating new and previously acquired knowledge to solve problems (Gagne, 1985). Sternberg's

Performance Component executes processes concerned with devising and implementing the problem solving plan. These entail goal decomposition, task selection, task organization and relationships, and task execution (Duncker, 1945; Newell & Simon 1972; Wickelgren, 1974). Sternberg's *Metacognitive Component* guides thinking about thinking. It performs the control processes that monitor all problem solving activities, including knowledge acquisition and performance. These processes guide strategies and tactics, beginning with problem representation through planning, implementing and evaluating the solution (Schoenfeld, 1992; Butler & Winne, 1995).

Gagne's *Essentials of Learning for Instruction* (1985) describes what a good learning environment should develop, its so-called *cognitive results* or *learning outcomes*. Gagne identifies verbal information, intellectual skills, cognitive strategies, and attitudes as the major categories of learning goals (Gagne 1985; Gagne & Driscoll, 1988). These learning outcomes also *demonstrate that one has gone through the various cognitive processes* that Bloom has identified.

Verbal information refers to knowledge acquired by observing and reading, such as stated facts and recalled principles, possession of which *demonstrates* awareness and understanding of concepts. Such verbal information confirms that the problem solver has gone through Bloom's knowledge and comprehension processes, and demonstrates "*knowing that*." Acquiring and organizing such information is an essential requirement for further learning since it is drawn upon as a source of ideas and possible solutions when solving problems (Mayer, 1983; Gagne, 1985).

Intellectual skills are correlative to verbal information. The ability to plan, define concepts, select objects, identify obstacles, and the demonstration of "*knowing how*" are

applied to problems in the form of concepts and knowledge to formulate and express solutions (Gagne, 1985). The application for these skills demonstrates Bloom's application process has been done.

Cognitive strategies refer to the mental processes in solving problems. Tactics and approaches used to transform knowledge and facts in order to generate a solution for a problem include: perception and reasoning (recognition of input stimuli and identification of information), learning and understanding (encoding of information), remembering (retrieval of information), and thinking (manipulation of information) (Lachman, Lachman, & Butterfield, 1979; Mayer, 1983). These demonstrate completion of Bloom's analysis and synthesis processes.

Attitudes are internal states that influence one's actions and preferences toward or away from a situation, a concept or a person. For example, it is possible that students, due to a specific learning environment or experience, will develop attitudes that will positively or negatively affect their outlook on the learning process (Mager, 1968; Rokeach, 1972). Attitude as a learning outcome is not directly related to the actual process of solving problems and writing programs in the same way as verbal information, intellectual skills, or cognitive strategies are, but is certainly an important measure to consider when evaluating students' perspective of learning.

This tripartite view of the structure of problem solving: Bloom's cognitive processes of problem solving, Sternberg's architecture structure of the thinking system, and the associated learning outcomes of Gagne, defines our cognitive model of problem solving and serves as the foundation for the model of problem solving in a program development environment introduced in the following section.

4.6 A Dual Common Model for Problem Solving and Program Development

Previous sections of this chapter provided a review of problem solving methods, a common model summarizing the essence of these methods, the tasks of program development, and a cognitive model for problem solving. In this section, our common model of problem solving, the work of Bloom (1956) on cognition, Gagne (1985) on human learning, and Sternberg (1985) on human information-processing were joined with the program development tasks to create a Dual Common Model for Problem Solving and Program Development (see *Figure* 5) supported by the necessary cognitive skills that must be developed and the expected learning outcomes at each stage of the process. This dual model (called dual because it brings problem solving and program development into one method) forms the basis of the SOLVEIT environment for problem solving and program development described in Chapter 5. The stages of this comprehensive model are considered in details in the following six sections. They are problem formulation, solution planning, solution design, solution translation, solution testing and delivery.

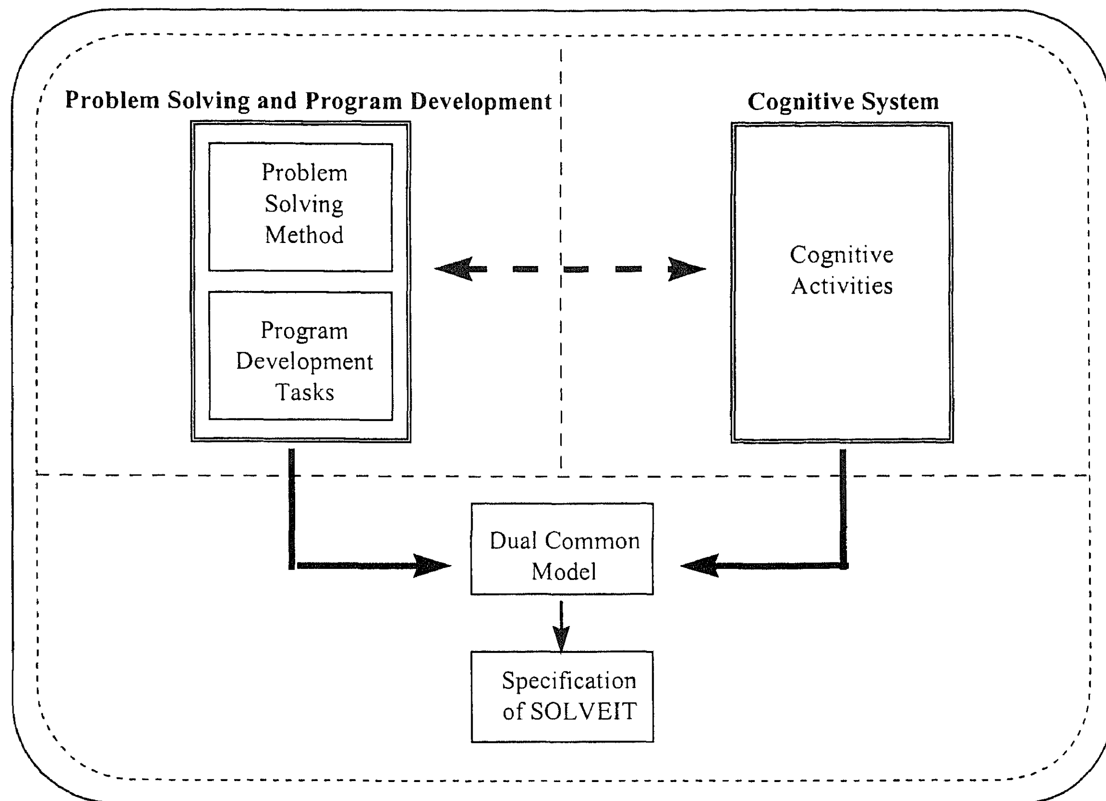


Figure 5 The Dual Common Model for Problem Solving and Program Development

4.6.1 Formulating the Problem

The common model indicates this stage should result in an organized representation of all relevant information: the goal, givens, unknowns, conditions and constraints of the problem. This corresponds to the conclusions of a cognitive analysis of problem solving, according to which: The objective of this stage is to produce a model of the problem and its elements (Mitroff & Turoff, 1973; Benbasat & Taylor, 1982; Rogalski & Samurcay, 1993). This cognitive activity, marking the beginning of problem comprehension,

requires developing a mental representation of the problem before it is solved (Pylyshyn, 1984).

To understand a problem, the student must define initial and goal states of the problem based on facts. Problem description is normally the first task encountered in both the problem solving and the software process (Chestnut, 1967; Sage & Palmer, 1990) and is the initial task of problem formulation. Problem interpretation and understanding (Bloom, 1956) require the construction of a well-defined description through progressive refinement and elaboration of the given problem statement. This process continues by extracting and organizing the relevant information from the problem description (Hayes & Simon, 1976; Espinasse, 1994) and defining initial and goal states for the problem (Greeno, 1978; Simon, 1978; Mayer, 1983). Domain knowledge and problem modeling skills are required to understand the problem and identify its facts.

4.6.1.1 Preliminary Problem Description

The literature describes many representation techniques for defining the problem question (Smith, 1993) and identifying needed information, and proposes language models for defining problems (Mitroff & Turoff, 1973; Volkema, 1988; Smith, 1993). Descriptions may be verbal, written, symbolic, graphic, or a combination (Rubinstein, 1975; Eden, 1988; Huff, 1990; Greeno, Collins, & Resnick, 1996). Rubinstein (1975) suggests writing the problem down in its primitive form, then transforming it to simpler language, translating it to mathematical formulation if necessary, and finally, representing it using diagrams, charts or graphs. Greeno, Collins, and Resnick (1996) also stress the importance of written problem description as a basis for encoding information from text

into meaningful mental representations of letters, word, phrases and sentences conveying coherent information. Describing a problem effectively and then identifying and utilizing its facts compensates for two of the most common difficulties in problem solving: overlooking known information which can be found within the problem statement, and introducing unnecessary constraints which are not part of the problem, yet are included (Rubinstein, 1975; Anderson, 1983). *Figure* 6 shows the initial task of problem formulation.

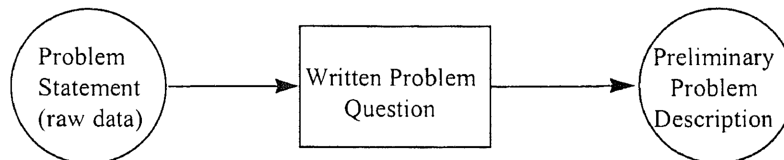


Figure 6 The initial task of problem formulation

4.6.1.2 Preliminary Mental Model

A preliminary mental model of the problem can be constructed using verbalization and inquiry questions. This preliminary model is then used by the subsequent phase where a more structured model of the problem is defined.

Verbalization usually takes place as communication between individuals, or with oneself, for the purpose of understanding a problem, understanding a solution, or explaining an idea. Verbalization is an essential part of problem solving (Whimbey & Lochhead, 1980; Whimbey, 1987) because problem formulation requires the understanding of the question as well as the meaning of the problem's terminology and facts (Charles, Lester, & O'Daffer, 1987). Written notes and diagrams are indicators of

verbalization. Although verbalization may be used throughout the process of problem solving, the most common form of verbal interaction takes place in problem formulation. The effects of verbalization on the subsequent tasks of problem solving beyond the problem description are critical (Tversky & Kahneman, 1981).

Understanding the problem and finding important information within its description require the use of *inquiry questions* (Polya, 1945; Mitroff & Turoff, 1973; Lauer, Peacock & Graesser, 1992), which oblige the problem solver to explicitly identify what is known about the problem, what needs to be discovered, what should be done, and how it should be done (Stepien, Gallagher, & Workman, 1993). Inquiry questions also force the problem solver to perform verbalization. Problem understanding, central to successful problem solving (Lyles & Mitroff, 1980), is the primary beneficiary of this technique. Inquiry questions can be effected by asking questions which provoke the student to examine the problem closely and uncover its goal, givens, unknowns, conditions, constraints or any additional requirements for understanding and solving the problem (Polya, 1945; Rubinstein, 1975). The result is an initial mental model of the problem to be solved. These problem understanding activities have a permanent effect on the rest of the problem solving process (Volkema, 1983). *Figure* 7 shows the effect of verbalization on problem formulation.

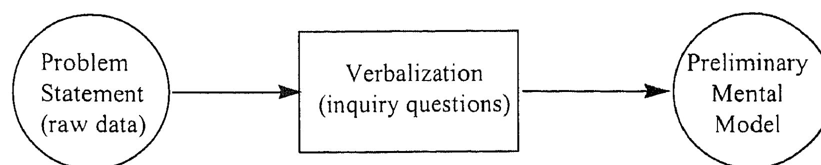


Figure 7 Effect of verbalization on problem formulation

4.6.1.3 Structured Representation of Problem

Identifying and organizing *all* the relevant information in a problem requires the use of a more structured approach. Finding problem *facts* takes place by applying a *formal information elicitation* method to the problem description in *order to extract and organize meaningful information in an appropriately structured model* for use in the next stages of the process (Simon, 1969; Benbasat & Taylor, 1982). The goal, givens, unknowns, conditions and constraints are extracted from the problem description and organized by category. Such formal elicitation and documentation of information is essential for identifying, retrieving, and utilizing information in problem solving (Anderson, 1983; Rubinstein, 1975; Miller, 1956). Students proceed through this task by refining the preliminary problem description, transforming the problem statement into an organized *knowledge base* that will then evolve during the remaining stages of the process. The knowledge base contains only the essential facts of the problem, ignoring irrelevant details. This explicit prompting for re-examination of the problem helps ensure that the student actually attempts to find all relevant information before seeking a solution.

4.6.1.4 Relationship to Cognitive Model

Identification of knowledge through information gathering methods and representation of this knowledge are primary requirements of the problem solving process. From the viewpoint of the cognitive model, the combination of this information with other knowledge such as domain knowledge, leads to comprehension of the problem question, a major objective of this stage (Bloom, 1956). In terms of cognitive structures,

knowledge acquisition processes are used to acquire, recall, and integrate the information and knowledge needed to devise and implement a solution (Sternberg, 1985). In terms of cognitive outcomes, verbal information that confirms problem understanding and identifies facts is an important result of this stage (Gagne, 1985). *Figure* 8 describes the cognitive system of the problem formulation stage.

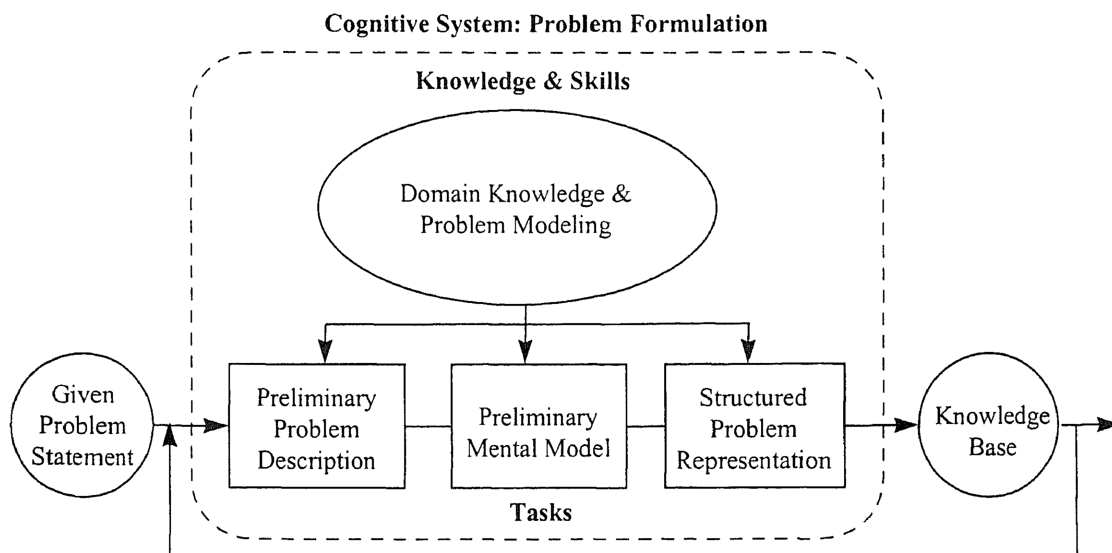


Figure 8 Cognitive system of problem formulation stage

4.6.2 Planning the Solution

The common model indicates that this phase should *identify alternative solutions and devise a plan by outlining the potential solution and breaking the problem into parts.* From the viewpoint of cognitive theory, planning is the cognitive activity in which the student determines, without carrying out the steps, the best course of action for reaching the goal state. Planning requires using general problem solving strategies to assess

solution alternatives and produce a plan for the problem (Greeno, Collins, & Resnick, 1996).

Solution generation and planning require access to relevant, well-organized knowledge, adequate domain-specific knowledge, and strategies or heuristics to solve problems (Polya, 1945; Newell, 1980; Greeno, Collins, & Resnick, 1996). Two prominent strategies are *solution by analogy* and *solution by subgoal decomposition*. In solution by analogy, students draw on prior knowledge and experience to recall similar or simpler problems (Polya, 1945). Correspondences between the current problem and related, previously solved problems are exploited, with similarities and differences between the problems providing hints to solve the current problem. Another powerful strategy is subgoals. The original goal is decomposed into a collection of intermediate subgoals, which are then decomposed into sub-subgoals, and so on (Duncker, 1945; Newell & Simon 1972; Wickelgren, 1974; Rubinstein, 1975; Mayer 1983). This allows one to reach the problem goal by meeting each of its subgoals one at a time, eventually producing a complete solution. Thereafter, implementation of the outlined design strategy can proceed. Domain knowledge and strategic knowledge are required to plan and to further develop the solution plan (Pennington & Grabowski, 1990).

4.6.2.1 Devising a Preliminary Plan

Devising a preliminary plan to solve the problem is necessary before the additional problem transformations that take place during the subsequent design and translation stages. The pre-existing knowledge, beliefs and information about the problem afford an understanding of the problem's requirements, enabling the student to plan a preliminary

course of action and to devise a potential solution for the problem (Butler & Winne, 1995).

4.6.2.2 Refining the Goal into Subgoals

Refining the goal into subgoals and subgoals into smaller subgoals is an effective problem solving strategy. The intent is to break the problem into smaller problems that are more easily solved. This is done by restating the problem in terms of a series of coherent subproblems. Decomposing complex problems into smaller parts is another known difficulty in problem solving (Whimbley & Lockhead, 1980) and a systematic technique is required. *Divide-and-conquer*, using step-wise refinement, is one commonly used technique, based on identifying, organizing and sequencing subgoals (Wirth, 1971). The work required to achieve a subgoal is later defined as an autonomous task. Subgoal decomposition is an intrinsic concept in programming since almost all problems are trivially divided into at least three subproblems: input, computation, and output (Wickelgren, 1974). These subproblems are further subdivided into smaller subproblems. This stage is concerned mainly with the subdivision aspect, while organization and sequencing (and further subdivision, if needed) are done in the next stage. The integrated ("conquered") collection of these ("divided") subgoals forms the complete solution to the problem. *Figure* 9 shows the refinement of problem into subproblems.

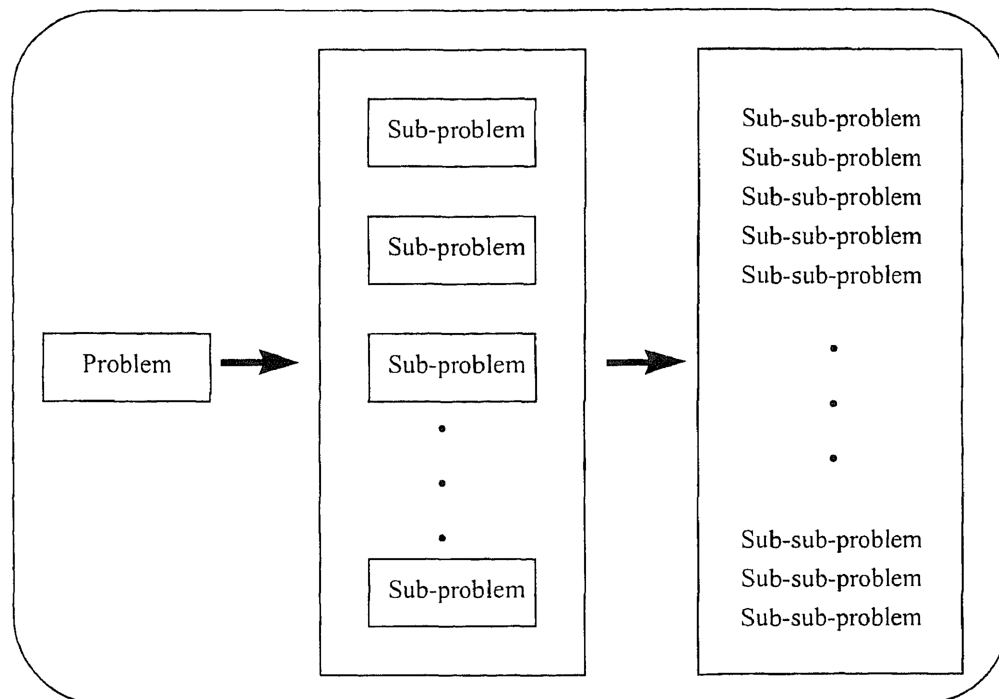


Figure 9 Refinement of problem into subproblems

4.6.2.3 Refining the Data Description

The next step is refining the data description from givens and unknowns, which were already identified through the elicitation technique of problem formulation, into data structures (Wirth, 1971). Having outlined a plan and a strategy to implement it, an accurate organization of information suited for manipulation is the last effort of this stage. Facts acquired during problem formulation may be incomplete or imprecise, but can be used as the basis for a comprehensive analysis and refinement of data requirements. The relationship between the problem's givens, unknowns, and the various solution components will need be to established in the next stage.

4.6.2.4 Relationship to Cognitive Model

From a process viewpoint, the major cognitive activities at this stage are the application of knowledge, and problem analysis and decomposition. According to Moore and Newell (1973), understanding of knowledge is demonstrated by the appropriate application of that knowledge. The use of knowledge, facts, and the application of concepts, theories or principles to plan a solution are in turn demonstrated by outlining the steps necessary to reach a solution by solving simpler, related problem, or by drawing charts and graphs which visually depict a solution. The cognitive processes of analysis and decomposition, which involve breaking the problem into component parts, entail identifying and establishing a hierarchy which organizes the problem into its parts and sub-parts (Bloom, 1956). The most relevant cognitive structure is the performance component which directs the solution planning and the problem decomposition process (Sternberg, 1985). The important cognitive outcomes of this stage include intellectual skills, which demonstrate the ability to apply knowledge and outline a detailed plan for a solution (Gagne, 1985).

Figure 10 describes the cognitive system of the planning stage as the evolution of knowledge gathered from the problem description into a detailed plan for the solution to be designed.

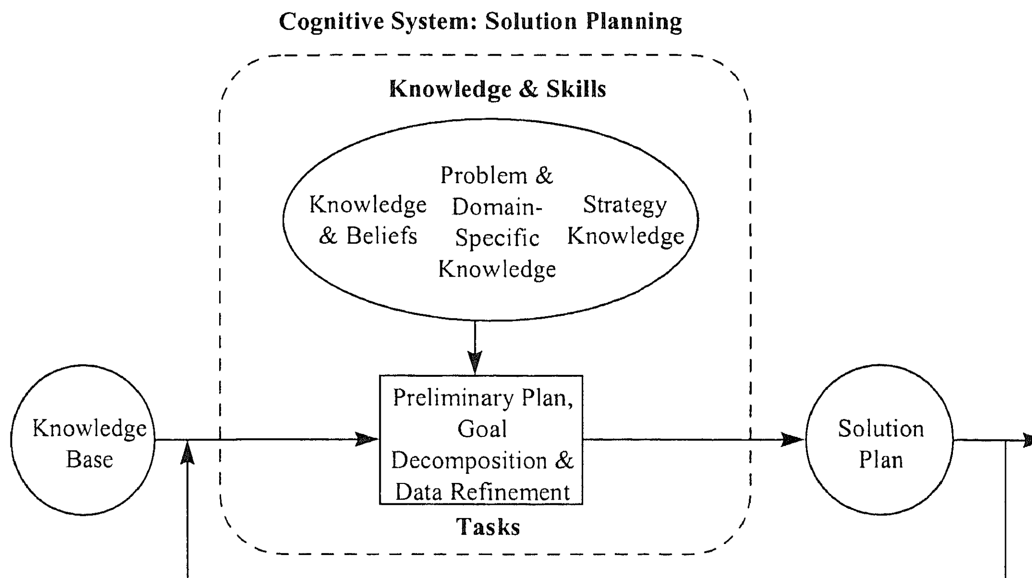


Figure 10 Cognitive system of the solution planning stage

4.6.3 Designing the Solution

The objective of this stage is to use specific problem solving strategies to carry out the solution plan outlined in the previous stage (Polya, 1945). A solution to the problem may be designed using the outcome of the prior two stages. Design is a two-level cognitive activity: (1) the student organizes and refines the components of the solution strategy, and (2) develops, and represents the solution specification algorithmically (Wirth, 1971; Rogalski & Samurcay, 1990, 1993; Bertels, 1994).

Design is a central point of the process simultaneously representing the final stage of problem solving and the beginning stage of program development. The high level phase of design produces the initial framework for the solution to the problem, based on what already started in problem formulation and planning. The solution outline,

described either in text or in visual form, is refined. This involves the sequencing of subgoals, the determination of whether the subgoals require further decomposition, the establishment of relationship among the various solution components, and the association between data and subgoals. The detailed phase transforms each subgoal into corresponding algorithmic specification, and the solution logic is readied to be translated into programming language syntax. As with planning, domain knowledge and strategic knowledge are required to design and carry out the solution plan (Pennington & Grabowski, 1990).

4.6.3.1 Organizing, Sequencing and Further Decomposition

Refinement of subgoals should proceed until each subgoal corresponds to a functionally well defined task. Wirth (1971) viewed the programming activity as a sequence of design decisions for decomposing tasks into subtasks, and maintains that the level of decomposition will effect the ease or difficulty with which a solution will be implemented, adapted or changed. These decisions begin early in the process and continue through this stage. The initial problem state defined in problem formulation was replaced by the decomposition of the planning stage. Now, the hierarchy among the various components of the solution must be examined. Because the decomposition proceeds in a top-down flexible manner, the tasks' inter-relationship may require reorganization and sequencing once refinement is complete. The structure chart, a visual representation of problem decomposition and the hierarchy between subgoals, is used.

This organization and sequencing produces a design sketch, but additional refinements may still be required. The solution components, represented by the structure

chart, are examined and, if necessary, subgoals are decomposed into yet smaller subgoals. This refinement process determines the complexity of code translation, which takes place in the next stage, and must continue until each subgoal is considered well-defined, focused, and easily solvable. A structure chart representing the different subgoals, or modules, of a solution is drawn in *Figure* 11 to show the organization, hierarchy, and decomposition of a problem into subproblems.

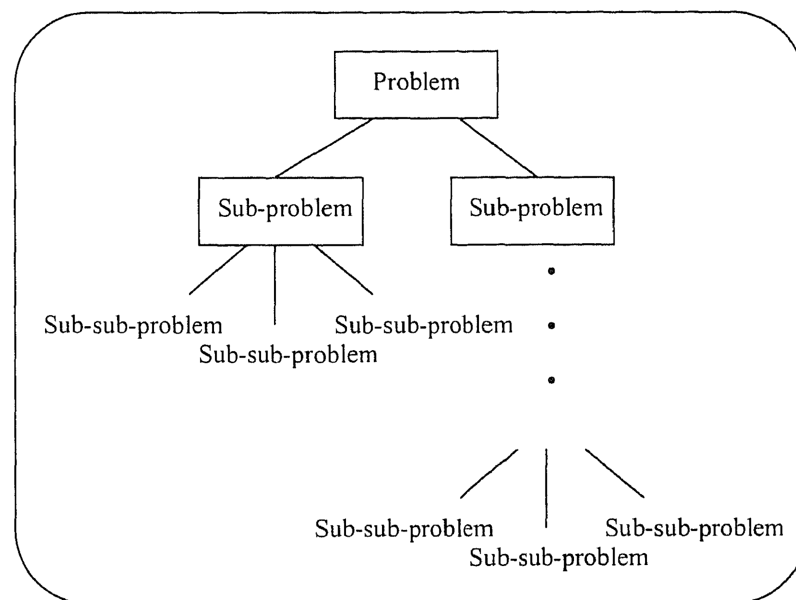


Figure 11 Organization and sequencing of subproblems

4.6.3.2 Communicating between Modules

Module communication is necessary to facilitate the flow of data from one module to another. The information already identified and organized in the previous two stages is related to the various tasks and the role of each is defined. The structure chart, produced earlier, shows the depth of the decomposition, the individual modules, and their hierarchy. Data and control interface among modules is formed next. Wirth (1971 &

1975) suggested that each refinement in the description of subgoals should be accompanied by a refinement of the data description, which is the primary mode of communication between the subgoals. There are two categories of module interconnection: data flow and calling sequence. Data flow refers to the dependency between inter-modules data sharing. Calling sequence refers to the module control flow (transfer/return) process.

4.6.3.3 Specifying Modules' Logic

Module logic is specified based on the information gathered and transformed during the current and previous two stages. Once modular decomposition and data flow are completed, the process proceeds by defining the algorithmic logic needed to accomplish the desired outcome for each module. This is done by describing what each module computes and how it computes it by specifying the required steps. Specifying the design requires the use of a suitable algorithmic language that supports the data structures, control structures, operators, and notations required to describe the modules' operations. Thus, the target language may in fact affect the nature of data structure selection or problem decomposition.

4.6.3.4 Relationship to Cognitive Model

From a process viewpoint, the major cognitive activity at this stage is synthesis, which is concerned with the reintegration of interrelated components into a coherent whole, rearranging when necessary, establishing relationships, and producing a new and well-organized whole as a viable solution to the problem (Bloom, 1956). The most relevant

cognitive structure is the performance component which in addition to directing the decomposition process, it is concerned with the identification and selection of tasks; and the organization, sequencing and execution of these tasks (Sternberg, 1985). *Figure* 12 describes the cognitive system of the solution design stage.

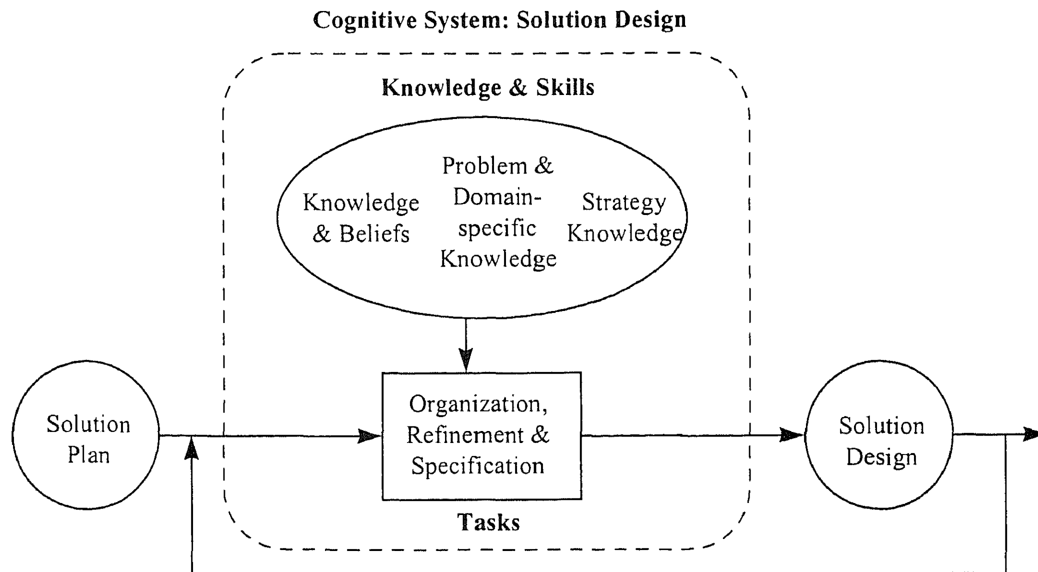


Figure 12 Cognitive system of the solution design stage

4.6.4 Translation

The objective of this stage is to use program development skills to translate the algorithmic solution into programming language code. Translation is the first syntax-related stage, and is a cognitive activity where the student makes the transition into the programming language environment (i.e. Pascal, C, C++ compilers) to produce a program that runs on the computer.

Program implementation, normally carried out in parallel with the next stage of testing, proceeds by carefully selecting an order for module translation and then

converting module specifications from algorithmic logic into language syntax. The program is then compiled, executed, tested, and results are produced. This brings an end to the process of transformation and signifies reaching the goal state - the solution to the problem and the results. Syntax, together with semantic and pragmatic knowledge that includes data structures, control structures, and modularization skills, are required to translate the solution design into language code.

4.6.4.1 Ordering Module's Translation

Module translation is primarily based on the structure chart and the algorithmic solution, and should proceed in conjunction with a testing plan. Although it is feasible to implement program modules in any order, an organized and incremental implementation strategy allows for modules to be tested as they are translated. Implementation strategies must be chosen based on the circumstance of the problem (i.e. type of problem, size, complexity).

4.6.4.2 Translating Module's Specifications

This involves transforming the detailed design into instructions suited for compilation and execution by the computer. Although program implementation can be a complex task, its difficulty is decreased when the design stage is done carefully. Data described in problem formulation stage and refined in the planning stage is transformed into type definition, declaration, and parameter statements. Algorithmic notations are converted to syntax statements specifying operators to manipulate and transform the data to produce the desired results. These tasks are done for each module and include data modeling, and

data and logic flow. The documentation of the role of individual instructions and the module as a whole should be done at the same time.

4.6.4.3 Documenting Modules' Logic

Documenting modules as they are being implemented has significant consequences on the clarity and readability of the whole solution and is essential for comprehending and modifying programs (Tremblay & Bunt, 1989). This covers both program documentation and programming style. In addition to the documentation generated during the earlier stages of problem solving, comments and explanations in the program are important to understand the approaches and techniques used to solve the problem. Maintenance - modification of existing program functionality or addition of new user requirements - would be difficult without adequate documentation. Other forms of documentation, such as help features or user manuals in the case of complex systems, are also essential for understanding system operations. Program style, which varies from one programmer to another, refers to the presentation of code in language syntax. It is impractical to impose a specific style on all programmers, but establishing and adhering to conventions and guidelines is crucial in order to produce a readable solution (Tremblay & Bunt, 1989). *Figure* 13 describes the cognitive system of the solution translation stage.

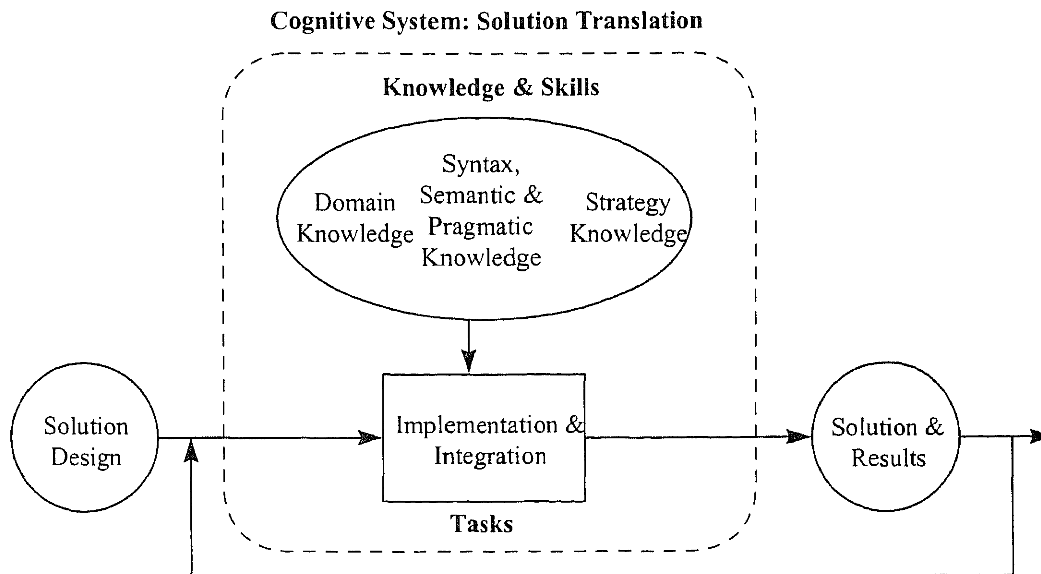


Figure 13 Cognitive system of the translation stage

4.6.5 Testing

The objective of this stage is to also use program development skills, specifically testing and debugging, to validate the solution and assure that the problem requirements' have been met (Yeh, 1990). Testing is a cognitive activity where the student is expected to develop and apply test data to verify program correctness and accuracy of produced results (Tremblay & Bunt, 1989; Graham, 1985).

Ideally, designing the solution and implementing the program would produce the desired results. Realistically this does not always happen, even with experienced programmers. Students need to learn how to find and correct three types of errors: (1) Syntax errors, such as misspelling of reserved words and violating grammatical rules. (2) Run-time errors, such as attempting to assign a character value to a numeric variable or

trying to divide by zero. (3) Logical errors in solution design that produce unexpected results, such as using the wrong formula to calculate the weighted average of a set of grades. The first two types of errors are detected by the programming environment. Syntax errors are found when the program is compiled. Modern programming environments provide sophisticated error reporting and debugging utilities to assist with syntax problems. Run-time errors are detected and reported during program execution. Logical errors, on the other hand, are not detected by the system and may be a challenge for the student to correct. Tracing and code visualization utilities provided by programming environments are helpful, but the work is largely done by the programmer. To guard against logical errors, each module must first be verified independently and then the solution as a whole should also be verified. Syntax, semantics, and domain knowledge are required to ascertain program correctness.

4.6.5.1 Developing Test Data

Developing test data to use as input for the program under verification is the first task in this stage. The objective is to design a testing strategy that uncovers all program errors. Students should develop and use comprehensive test data to verify program correctness, and fix errors when found. This requires that expected program output is determined based on a problem's requirements and its design specifications (Graham, 1985). Although it is difficult or even impossible to develop for programs performing massive calculations, it is easier to develop for the type of problems encountered in the first course on programming and may be verified by hand computations based on design specifications.

4.6.5.2 Performing Code Testing

Ensure that the program functions properly under all test cases to which it is subjected, works for all valid input, and anticipates and responds to all invalid input is the most common type of program verification is essential (Tremblay & Bunt, 1989). The program is considered successful if it contains no errors. In the case where errors are found, the program will require debugging and perhaps modification either in the code or the design. This requires that the location of the error is determined, its cause established, and corrective measures taken. The result of testing may require changes to a program affecting its logic, language constructs used, or data representation. *Figure* 14 describes the cognitive system for the solution testing stage.

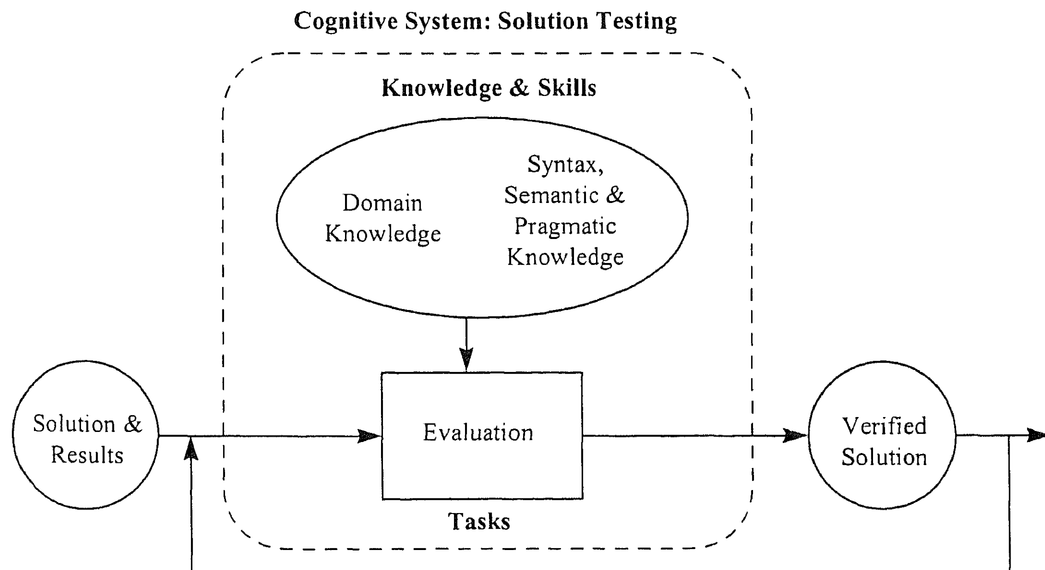


Figure 14 Cognitive system of the testing stage

4.6.6 Delivery

This problem solving and program development process consists of five stages each made up of multiple tasks, beginning with describing the problem through producing the results. Delivery, the least complex stage of all, signifies the end of the process and entails no further problem transformations. The process culminates with the presentation of different solution parts in an organized and comprehensible form.

4.7 Monitoring the Problem Solving Process

As a solution plan is being devised and implemented, either the whole solution or some part of it frequently does not match the purpose intended when the plan was created. It may be necessary to reorganize or retrace the solution path, returning to "planning the solution" or even "formulating the problem". This requires *monitoring* the thinking process and *evaluating* transformations as the solution evolves.

Cognitive strategies, or metacognition, refer to techniques used to monitor thinking. Metacognition guides a person's own knowledge of thought processes and the regulation of these processes during problem solving. Students become effective learners when they are aware of their own thinking processes and develop the ability to monitor their understanding of the tasks they perform. Realizing that there is a need to assess or modify the problem solving strategy requires that students develop effective skills for monitoring and evaluating their thinking and produce progress feedback.

Understanding of the problem deepens as the process of problem solving evolves (McAllister, 1995), peaking as the complete solution is implemented (Smith, 1993). Observations made as the solution evolves, called internal feedback, provide grounds for

reassessing the problem's need and the solution. Internal feedback is an important progress indicator of the problem solving process and is triggered as a result of the problem solver's own comprehension of what has been done, is being done, and remains to be done.

External feedback, such as comments provided by a teacher, classmate, or a problem solving tool, can either confirm or conflict with the student's strategy, thereby also causing reassessment and adjustment (Butler & Winne, 1995). External feedback heightens students awareness and helps to develop metacognitive skills to monitor and evaluate their thinking strategies through the problem solving process.

Students have difficulty in initiating the problem solving process and adhering to a specific methodology. They often do not know where to begin, what to do next, how and where to look for relevant information, how to make sense of the given information, and how to organize the information gathered and later retrieve it for subsequent use. A structure to aid student's thinking process while solving problems is needed (Pea & Sheingold, 1987).

Task flow coordination and outcome visualization help guide students through the process, from the initial stage of problem formulation until the delivery of a solution. Visualization, allowing students to view the result of their work in an organized form at all points in problem solving, has been shown to be beneficial in computer-based learning (du Boulay, 1981; Reiser, Ranney, Lovett, & Kimberg, 1989; Ramadhan, 1992; Ramadhan & du Boulay, 1993). It enables the student to observe and certify the outcome of their work and witness the progress they have made. This is especially important for

the novice since the programming language syntax may obscure the solution behind its coded programming language representation.

4.8 Conclusion

The tasks of problem solving and program development form an interdependent process. Each stage requires specific skills and cognitive abilities. Problem formulation, planning, and design are fundamental skills required for successful program development (Mayer, 1983; Scholtz & Wiedenbeck, 1992; Ebrahimi, 1994). The role of problem solving in programming is extensive, and the language does not have to become a factor until implementation stage later in the software process. Problem formulation allows for the understanding of problem question and the identification of its facts. Domain knowledge and problem modeling skills are necessary for this stage. Planning and design allow for representation of possible solutions; decomposition of problem into subproblems; organization and refinement of solution; and specification of solution logic without profound concern for implementation details. Domain knowledge and strategic knowledge are essential skills to plan and design a solution to a problem. Translation allows for the algorithmic solution to be mapped into programming language code. Syntax, semantic, and pragmatic knowledge are important skills for this stage.

The question of how to integrate this new model into the classroom must be addressed. Programming is usually taught in a lecture/recitation format. In the lecture, language syntax and simple examples are presented to illustrate the concepts. In the recitation, the instructor presents examples, demonstrates the algorithmic solution, and applies the syntax presented in the lecture to the algorithm. This method and sequence

follow the typical textbook approach to the subject, and while students and instructors do interact during class somewhat, the material is essentially presented by the instructor.

An alternative method (Deek & Kimmel, 1993, 1995) is to introduce the problem in the lecture, engage the students in defining the problem, analyze the requirements, and allow the students to seek a possible solution independent of the programming language. Once the problem is solved, the language syntax is presented and the complete solution implemented.

The lecture begins where it would be expected to end: The examination of a problem to be solved (*programming textbooks do this last*). With the introduction of the problem as the first activity of the lecture session, the students are engaged in understanding the problem, analyzing and identifying the needs, and devising possible solutions with total independence of the programming language syntax. The students concentrate on the design of the algorithms, choice of data representation, and outlining testing strategies. Once the algorithmic solution for the problem is constructed, the language syntax (i.e. control structures, data structures) necessary to carry out the solution to its final stages may be presented (*programming textbooks do this first*). Equipped with both the algorithmic solution which the students developed and the language syntax - now more meaningful and appreciated - the complete solution is implemented and tested.

Throughout the process the focus is on problem solving with less emphasis on the language syntax. This emphasis on problem solving becomes possible by delaying the programming activity and by easing the syntax burden on the novice programmer, typically in the last two stages of the process: implementation and testing. Reducing the

concerns for language syntax allows the students to focus on problem solving, and algorithm design and development principles.

A paradigm for teaching and learning programming which integrates the problem solving methods, the language, and the instructional methodology in a comprehensive system is necessary. Such an environment for facilitating the study of programming must *truly* mirror the activities of the actual learning situation and support the entire problem solving and program development process. Thus, we propose a complete environment to support the problem solving and program development approach, starting with the initial activity of describing the problem, through testing the solution. Thus, the entire aspect of the problem solving and program development process, and not only the programming activity, receives adequate attention and support.

The system, intended as an adjunct to classroom instruction, must comprise three components: The problem solving and program development method, the supporting tools, and the learning setting. The instructor continues to be an important part of the learning equation. *Figure* 15 depicts this learning environment. The classroom environment is extended to form a link with students working independently. The result, in essence, is the mapping of the guided learning atmosphere of the class into a student's workspace.

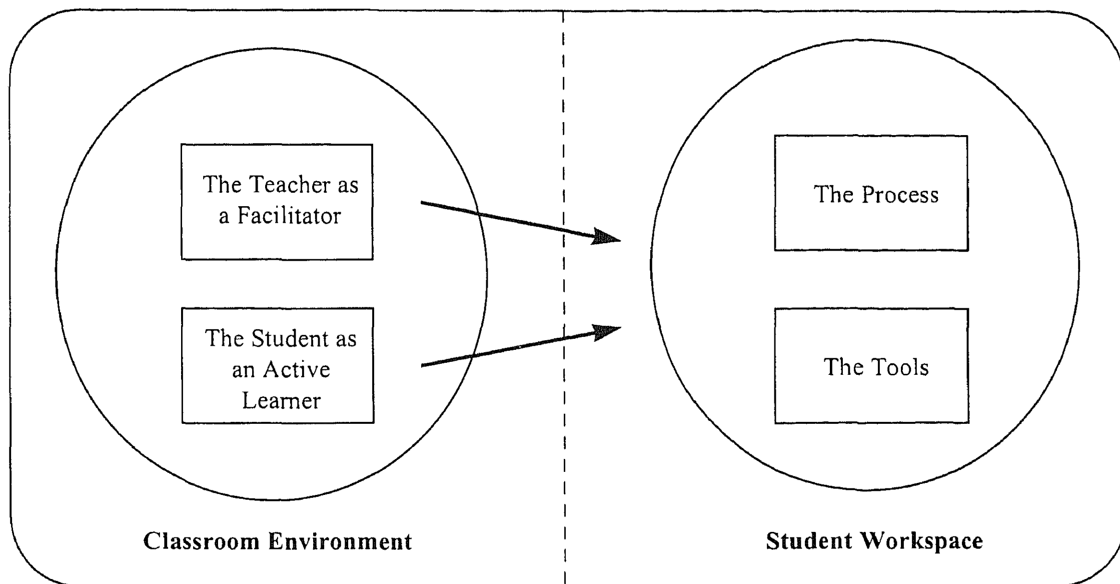


Figure 15 An extended-learning environment

The problems with learning programming, in particular, and the related skills of problem solving, in general, are addressed by a two-fold approach which is both technological and pedagogical. Students learn programming and strengthen their problem solving experiences by using the necessary problem solving strategies, along with tools that help them understand the problem, develop the plan, and the produce the design needed to implement the solution.

CHAPTER 5

AN ENVIRONMENT FOR PROBLEM SOLVING AND PROGRAM DEVELOPMENT: SPECIFICATIONS FOR THE DUAL COMMON MODEL

A framework for an integrated environment to support the students in all problem solving and program development stages including problem formulation, planning, design, translation, testing and delivery is proposed. This environment is based on the Dual Common Model for Problem Solving and Program Development produced in Chapter 4 and takes into consideration the cognitive skills that must be gained by students and the tasks performed in problem solving and program development (see *Table* 17). Facilities to assist the student in learning these skills and accomplishing these tasks are provided for each stage of the model. The system was designed, implemented, deployed and evaluated.

This chapter describes this environment which we call: SOLVEIT, an acronym for Specification Oriented Language in Visual Environment for Instruction Translation.

Table 17 Focus of problem solving and program development in SOLVEIT

Integrated Problem Solving and Programming Environment	Understanding Problem	Planning Solution	Designing Solution	Implementing Solution
SOLVEIT	X	X	X	X

5.1 The Specification Oriented Language in Visual Environment for Instruction Translation

The SOLVEIT environment combines the process and the tools to support the functionality of a traditional programming environment with a workbench facility and a battery of utilities used in problem solving and program development. SOLVEIT is designed to be used by beginning students working to solve programming problems. It provides an intuitive graphical environment complemented by online help and enforces a structured methodology as suggested by software engineering. The system surmounts the problems associated with current environments (as discussed in Chapter 3) by:

- *Taking into consideration the skills that must be gained by students and the tasks required for problem solving and program development.* The essential facilities to assist the student in learning these skills and performing these tasks are provided. Students produce their solutions in an environment that does not restrict creativity or cognitive development.
- *Removing the emphasis on language syntax.* Problem solving and program development in this environment is a progressive task that begins with the problem formulation stage using language-independent tools and continues through subsequent stages of problem solving in a similar manner. Code translation and testing takes place later in the process and is a further refinement of the earlier, language-independent solution.
- *Providing the framework and facilities that allow the student to deal with the common difficulties related to problem solving.* Information gathering, organization and retrieval, solution planning, problem decomposition, and task flow coordination are facilitated by system tools.

- *Providing a state-of-the-art user interface.* The system features a graphical, iconic, window-based interface that was tested in the classroom before it was fully deployed. Protocol analysis by prospective users was performed during and after development.
- *Integrating the tool into the learning environment.* The system was used in conjunction with normal classroom activities and provided a reporting mechanism for teachers who in turn provided feedback and comments to the students.
- *Evaluating the impact of the tool on the learning process.* An experimental study was performed over two semesters in multiple sections of the introductory course on problem solving and programming.

Specific features of this environment are:

- *Students are able to describe the problem in written form, refine it, and update it as required.* This is done freely without the restrictions of a limited dictionary of natural language keywords or the complexity of a problem definition language.
- *Problem facts are identified through a formal interaction and elicitation process.* Goals, givens, unknowns, conditions and constraints are identified and organized in a reference database. They may be refined and restructured as more knowledge is gained.
- *Planning and design are aided with automation.* Goal decomposition, data definition, subgoal hierarchy, data flow, and logic specification are performed using specialized tools.
- *Required code is translated into programming language syntax after the problem is solved.* Code is created by the students based on problem requirements only once

design specifications are developed independent of the specific language syntax. Syntax and modular code templates are provided by the system.

- *The system focuses on a meaningful subset of language constructs.* Control structures and data structures necessary to learn the fundamentals of programming (sequential, selective and repetitive structures; input and output functions; basic data types; and a composite list structure) are provided.
- *An electronic project notebook and a complete transcript/playback recording is provided.* Students' problem solving self-monitoring and feedback is achieved by maintaining progress records that can be examined by both the student and teacher.

5.1.1 The Process

The problem solving and program development model described in Chapter 4 provides the theoretical and cognitive basis for the SOLVEIT environment. A synthesized common method for problem solving (see section 4.?) and the tasks of program development were combined into a Dual Common Model supported by cognition, human information-processing and learning theories to form the stages of this model. They are: Problem formulation, solution planning, solution design, translation, testing and delivery. *Figure* 16 describes the model. A detailed description of the Dual Common Model for Problem Solving and Program Development is provided in section 4.6.

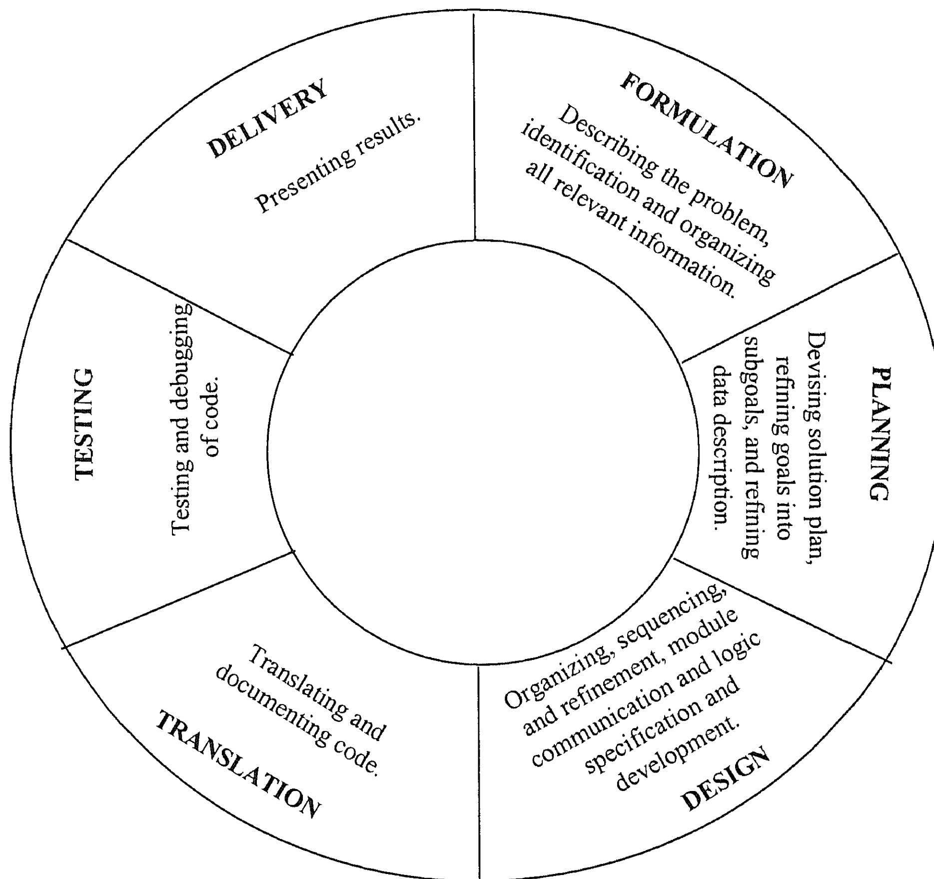


Figure 16 The SOLVEIT model

The stages of the SOLVEIT model are also consistent with the stages of the software process described in Chapter 1. A comparison of this model and the software process is shown in *Table* 18.

Table 18 The SOLVEIT model and relationship to the software process

Stages of Software Process	**Stages of SOLVEIT Process**
Problem recognition Feasibility study	Problem formulation
Analysis & requirements	Planning
Design & specification	Design
Implementation Integration	Translation
Testing	Testing
Deployment	Delivery
Maintenance Retirement	

5.1.2 The Tools

The SOLVEIT environment combines existing tools of traditional programming environments, shown in *Figure* 17, with additional tools to support the process of problem solving in problem formulation, planning, designing, translating, testing and delivering the solution. *Figure* 18 describes the tools in SOLVEIT.

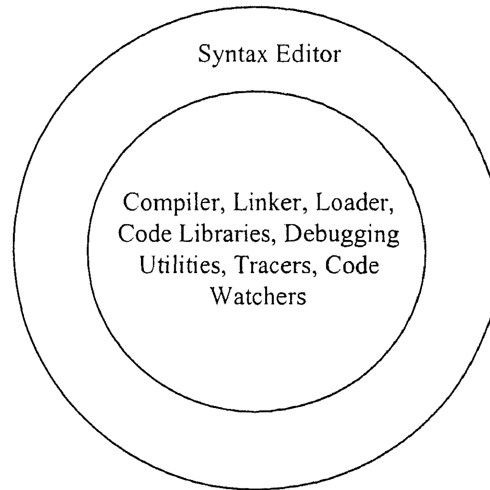


Figure 17 The tools in traditional programming environments

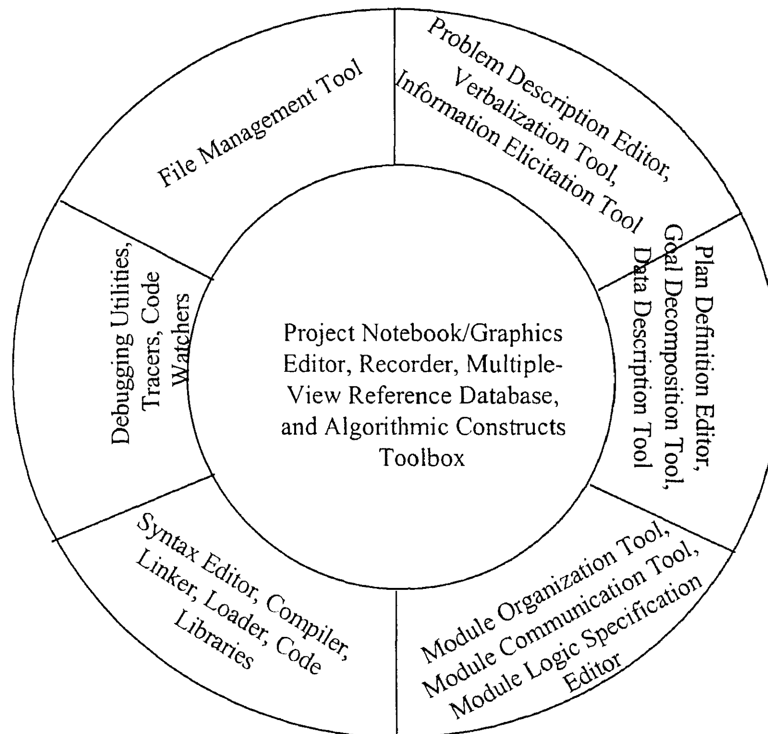


Figure 18 The tools in SOLVEIT

While some of the tools are shared by the entire SOLVEIT environment (inner layer of *Figure* 18), others are associated with specific stages of the process (outer layer of *Figure* 18) and are used to perform the various tasks of problem solving and program development. Compiling and debugging tools shown in this figure are utilities that already exist in traditional programming environments and are used by SOLVEIT. Section 5.2 presents a more detailed description of the tools and their inter-relationship.

5.2 SOLVEIT Architecture

The relationship between the different tools and components are shown in *Figure* 19 representing the system architecture of the SOLVEIT environment. Tools are grouped based on their association with a specific stage. This section provides a functional description of each problem solving and program development tool (upper-level of *Figure* 19) in the SOLVEIT environment. Common tools (mid-level of *Figure* 19) are described in a separate section. stage. Existing tools of traditional programming environments (lower-level of *Figure* 19) are not discussed.

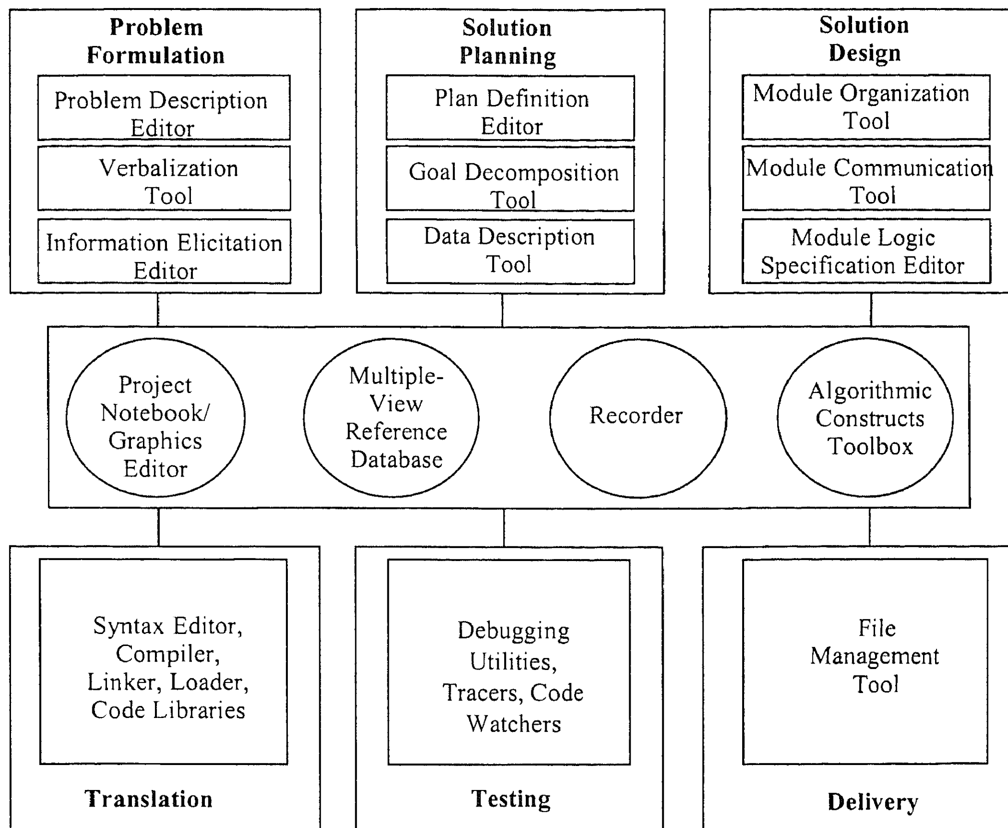


Figure 19 Architecture of the SOLVEIT environment

5.2.1 Tools for Problem Formulation

Three tools are used for problem formulation: (1) the problem description editor (2) the verbalization tool, and (3) the information elicitation tool. They are described below.

5.2.1.1 The Problem Description Editor

The problem description editor is invoked from within the problem formulation stage to enter and save the problem statement into the system. The assignment sheet distributed

by the instructor or the student's own terminology can be used. If the problem is already in electronic form, the file can be opened and read into the editor. The problem description text is stored in the multiple-view reference database.

5.2.1.2 The Verbalization Tool

The verbalization tool is active in all stages of SOLVEIT, but is invoked automatically in problem formulation after the problem description is saved. Questions are presented to the student while the problem statement is still visible in the editor. The student can answer the questions or can skip the entire verbalization session. These questions will provoke the student to re-examine the problem statement which remains displayed throughout the interaction session. The result of this verbalization is saved, along with subsequent verbalization sessions, by the recorder and are available to the student during the entire problem solving process. A complete transcript is created as part of project deliverables. Questioning in SOLVEIT is general, but can be changed to fit a specific class of problems. The teacher or the student can add more questions that might be appropriate to the current problem. In addition to questioning, access to a project notebook/graphics editor is made available to the student to draw and make notes as necessary. The interaction transcript is stored in the project notebook/graphics editor.

5.2.1.3 The Information Elicitation Tool

The information elicitation tool is invoked from within the problem formulation stage. It is used to extract and organize relevant information, found within the problem description, in a structure suited to perform the transformations of subsequent stages and

to carry out the tasks that will lead to the solution. This technique of information elicitation will transform the problem description from its text format into a database of problem facts. Using the information elicitation tool, the student returns to the typed problem and extracts the goal, givens, unknowns, conditions and constraints. All gathered information is stored in the multiple-view reference database and is accessible from within other stages of SOLVEIT.

5.2.2 Tools for Solution Planning

Three tools are used for solution planning: (1) the plan definition editor, (2) the goal decomposition tool, and (3) the data description tool. They are described below.

5.2.2.1 The Plan Definition Editor

The plan Definition editor is invoked from within the planning stage. It is a simple text editor used by the student to describe their approach and the steps required for solving the problem. Although the SOLVEIT environment provides a linear problem solving process, it does not impose a rigid sequence on the student. The student may decide to design a solution before completing all the tasks of problem formulation or planning stages. However, some of the information required for the completion of a certain task may not have been collected and organized. This initial plan will help coordinate student's thoughts and actions in solving the problem. The plan is written to the project notebook/graphics editor.

5.2.2.2 The Goal Decomposition Tool

The goal decomposition tool is invoked from within the planning stage. It allows the student to begin the process of transforming the goal identified in the information elicitation phase into subgoals that need to be completed to solve the problem. The student refines the goal into a collection of subgoals and associates two attributes with each subgoal: an identification name and a description. A goal can have multiple subgoals and subgoals themselves may be decomposed into smaller subgoals. The decomposition tree and the information about all the subgoals are stored in the multiple-view reference database.

5.2.2.3 The Data Description Tool

The data description tool is invoked from within the planning stage. It is used to transform the givens and unknowns identified in the information elicitation phase into data representation. Givens are the problem's input-data or constants. Unknowns are output-data or intermediate data elements. Using the editor, the student associates a series of attributes with givens and unknowns: name, description, origin/source, type and structure. Data description table is stored in the multiple-view reference database.

5.2.3 Tools for Solution Design

Three tools are used for solution design: (1) the module organization tool, (2) the module communication tool, and (3) the module logic specification editor. They are described below.

5.2.3.1 The Module Organization Tool

The module organization tool is invoked from within the design stage. It is used to establish the hierarchy of the subgoals already refined in previous stages and now contained in the reference database. The goal and subgoals are displayed in a graphical representation using a structure chart format as shown in *Figure* 20. SOLVEIT uses a simplified version of the structure chart covering only the module decomposition and data flow.

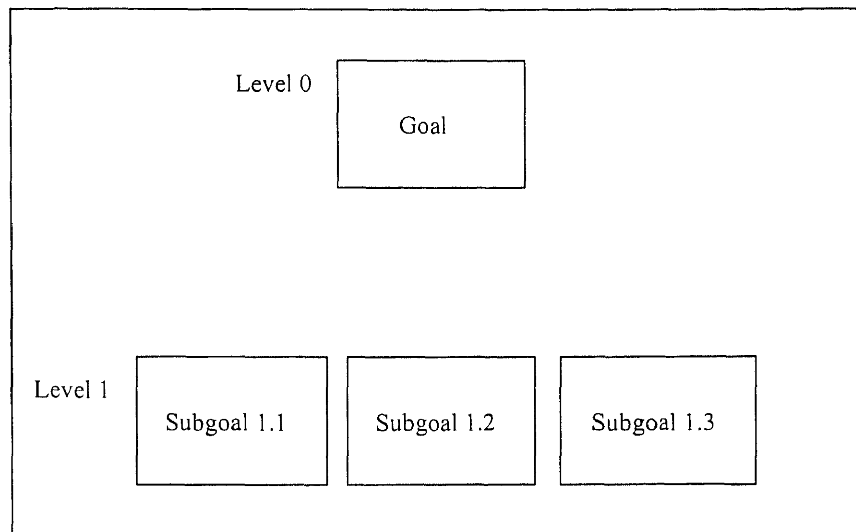


Figure 20 Initial representation of the structure chart

This initial representation may require organization and refinement. The top level of the structure chart is the primary goal selected during the information elicitation task of problem formulation. At the next level are the subgoals refined in goal subdivision task of solution planning. Refinements of any subgoal will create subsequent levels in the structure chart as shown in *Figure* 21.

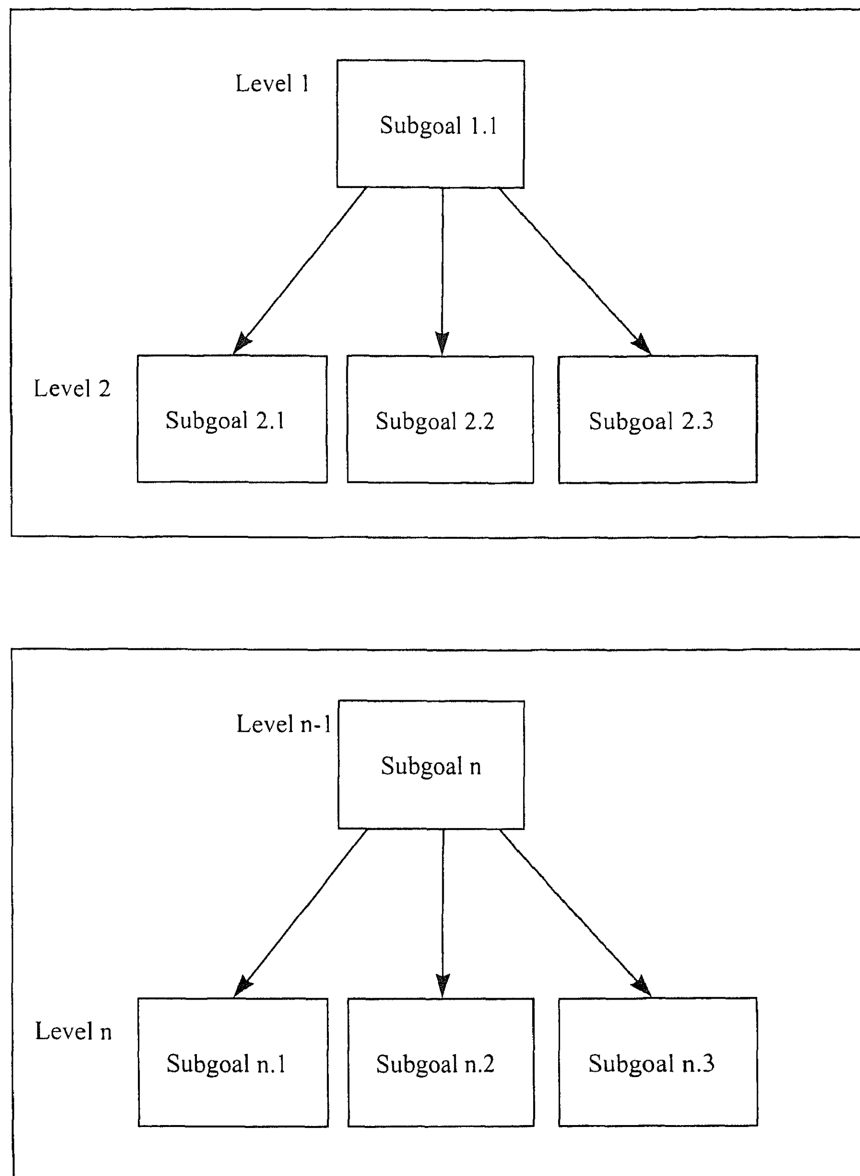


Figure 21 Refinement of the structure chart

There is an implied left to right organization and top to bottom hierarchy of subgoals in the structure chart. To reorganize or change the hierarchy, goals can be

repositioned vertically and horizontally. New subgoals can be added, others can be removed or merged.

Information about the organization/hierarchy of subgoals in the structure chart is also maintained in the reference database and can be viewed as shown in *Figure* 22.

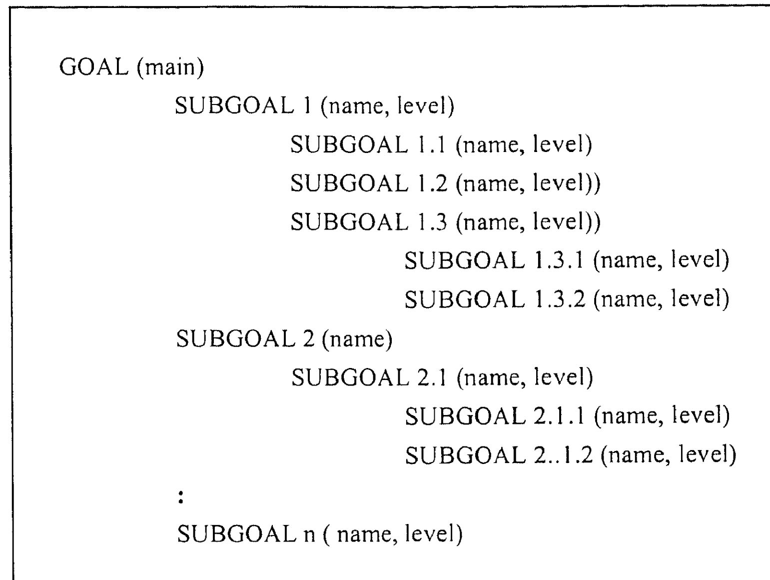


Figure 22 A text view of the structure chart

5.2.3.2 The Module Communication Tool

The module communication tool is invoked from within the design stage. It is used to establish the data flow relationship between modules. Communication between modules is formed using information gathered during the planning and design stages, now stored in the reference database. Additional data may be defined and added to the database as needed. Data elements can be associated with specific modules as a sender or receiver of that element. The result is a data dictionary table, also stored in the reference database,

that includes the data element names, type, description, the associated goal/subgoal and the direction of the data flow as shown in *Table* 19.

Table 19 Data dictionary entry

Data Element Name	Data Type	Data Description	Goal/Subgoal Name	Input/Output

5.2.3.3 The Module Logic Specification Editor

The module logic specification editor is invoked from within the design stage. It is used once the organization and refinement of the structure chart and data flow is completed. This tool enable the student to develop module logic for each box of the structure chart. Module logic is constructed using an algorithmic constructs toolbox. A program shell, based on the structure chart, is displayed as a "table of contents" to facilitate module logic development as shown in *Figure* 23.

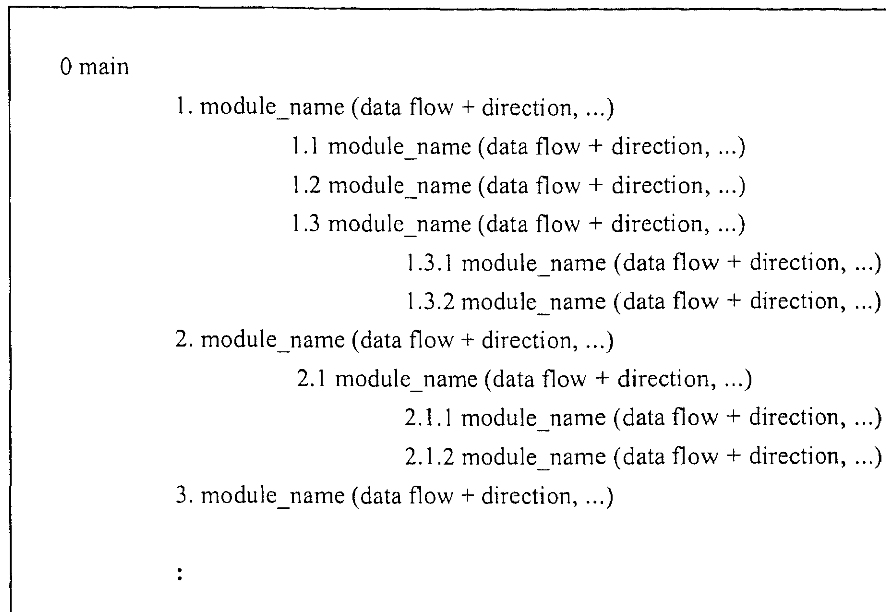


Figure 23 Structure chart and data flow in text view

The "table of contents" shows the name of the main module and all other program modules along with the data flow and the direction of each. Each module and associated entry in the "table of contents" is hypertext sensitive and when selected it invokes the module logic development window as shown in *Figure* 24.

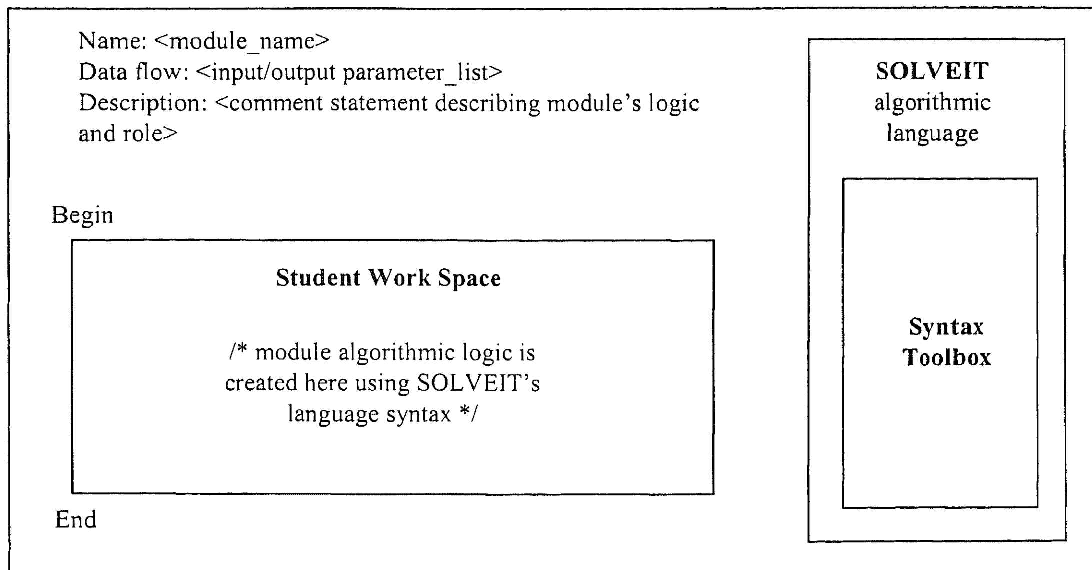


Figure 24 Module logic development

The student gains access to the SOLVEIT algorithmic constructs toolbox which is used to describe the logic for each module. This toolbox contains templates for a SOLVEIT-specific algorithmic language consisting of basic data types, control structures, a list structure and modular abstraction. Syntax templates are selected and copied into the student's work space.

The module logic is saved in the reference database as a text file using SOLVEIT algorithmic language syntax. General purpose routines that are developed in this stage may be stored into a library of functions for reuse in other problems. Similarly, any modules developed in earlier problem solving sessions that are appropriate for other situations may be integrated and reused.

The algorithmic logic created in the specification editor, stored in the reference database, can not be compiled directly. The logic is copied into the syntax editor of the target programming environment (i.e. Pascal, C, C++) and is used by the student as the basis for code translation from algorithmic logic into language syntax.

5.2.4 Tools for Solution Delivery

The delivery stage is the simplest stage of the process and only one file management tool is provided, as described below.

5.2.4.1 The File Management Tool

The file management tool is invoked from within the delivery stage. It is used to organize and produce the complete solution package consisting of the outcome of each stage and, if desired, the content of the reference database and project notebook/graphics editor. The result is a fully documented program maintained in an organized manner and can be printed or saved on a disk.

5.2.5 Common Tools

In addition to the specialized tools in each stage of the process, SOLVEIT provides a host of environment-wide tools that may be invoked from within any stage of SOLVEIT.

5.2.5.1 The Project Notebook/Graphics Editor

The Project notebook/graphics editor is a text and graphics editor used as a scratch pad by the student. The student is prompted to invoke the editor during the problem formulation

stage, but can also be invoked at any time during the problem solving process. Text and graphics capabilities are available for the student to make notes and to draw.

5.2.5.2 The Recorder

The recorder monitors the student's activities within SOLVEIT and acts as an event-capture log. It allows the student and teacher to replay the activities of a problem solving session, at a later time. Each time the student enters a stage within SOLVEIT, the event is recorded, the task performed is also recorded. Upon exiting, a snapshot of the reference database is copied to the recorder. Activities log are listed in the sequence they occurred. The recorder content cannot be edited.

5.2.5.3 The Multiple-View Reference Database

The multiple-view reference database contains the result of all data transformation within SOLVEIT beginning with entering problem description through the development of the algorithmic solution. The database organizes this information and makes it available to the student at different stages of the problem solving process.

5.2.5.4 The Algorithmic Constructs Toolbox

The algorithmic constructs toolbox is a syntax template database represented as icons and is accessible from within the design stage to specify modules' logic.

In summary, the process of the SOLVEIT environment, the tools supporting this process and the existing tools of a traditional programming environment are combined to

form an integrated environment for problem solving and program development as shown in *Figure* 25.

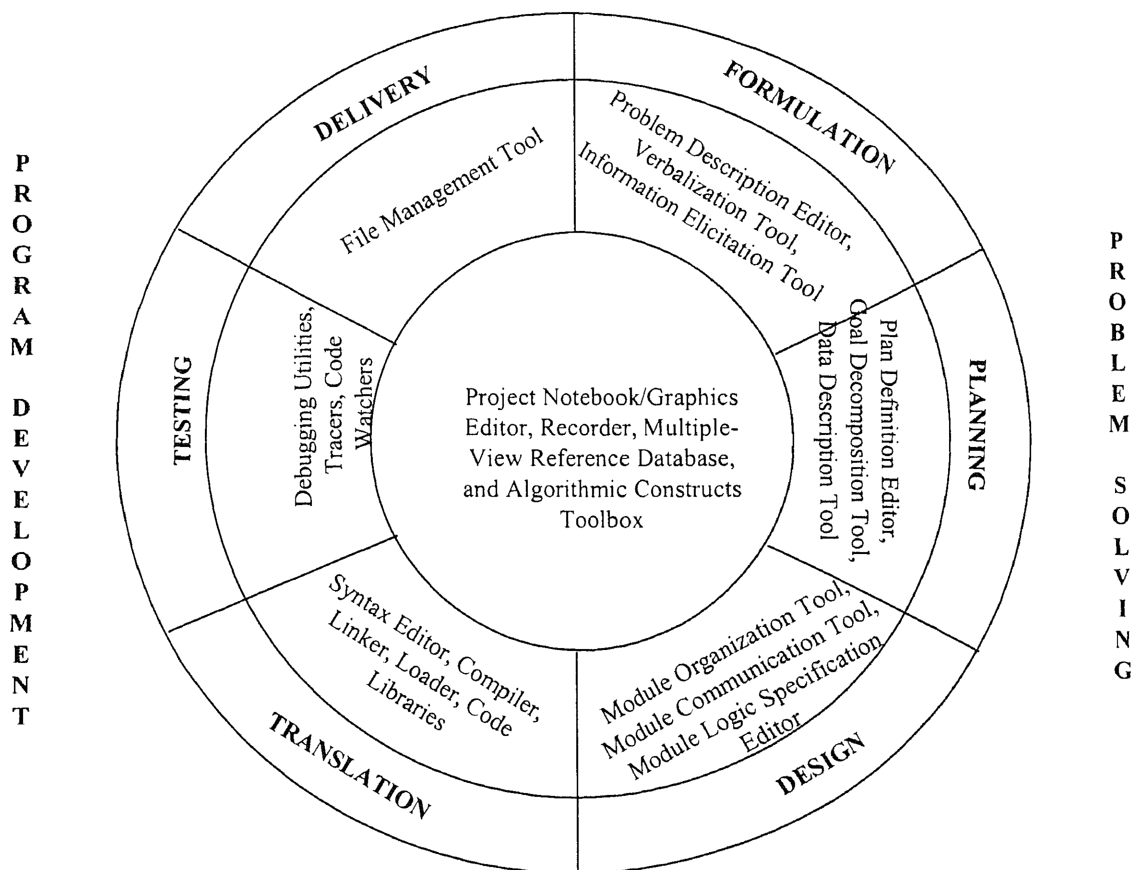


Figure 25 SOLVEIT: An integrated problem solving program development environment

5.3 A Walk Through SOLVEIT

SOLVEIT consist of six stages. The first three stages are problems solving stages and the next three stages are program development stages. SOLVEIT will guide the student through a linear process of problem solving. However, the student can begin working at any stage of the process. *Figures* 26 and 27 depict the student's work flow from problem formulation through delivery of solution using SOLVEIT.

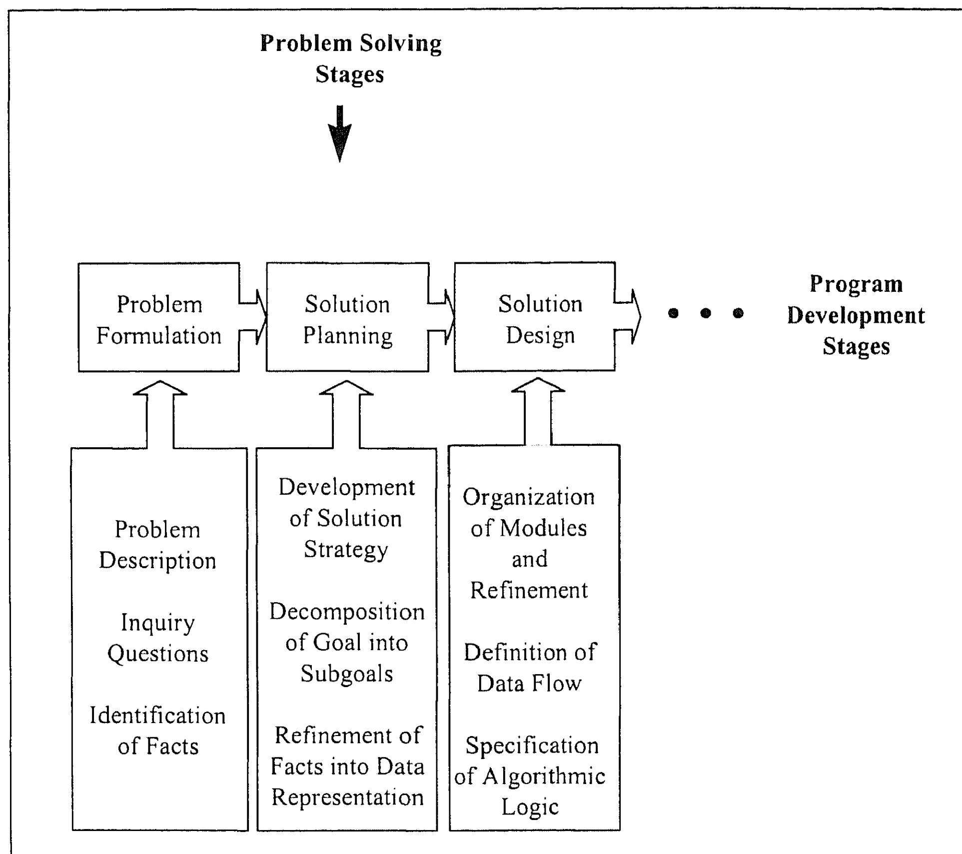


Figure 26 Workflow in problem solving stages

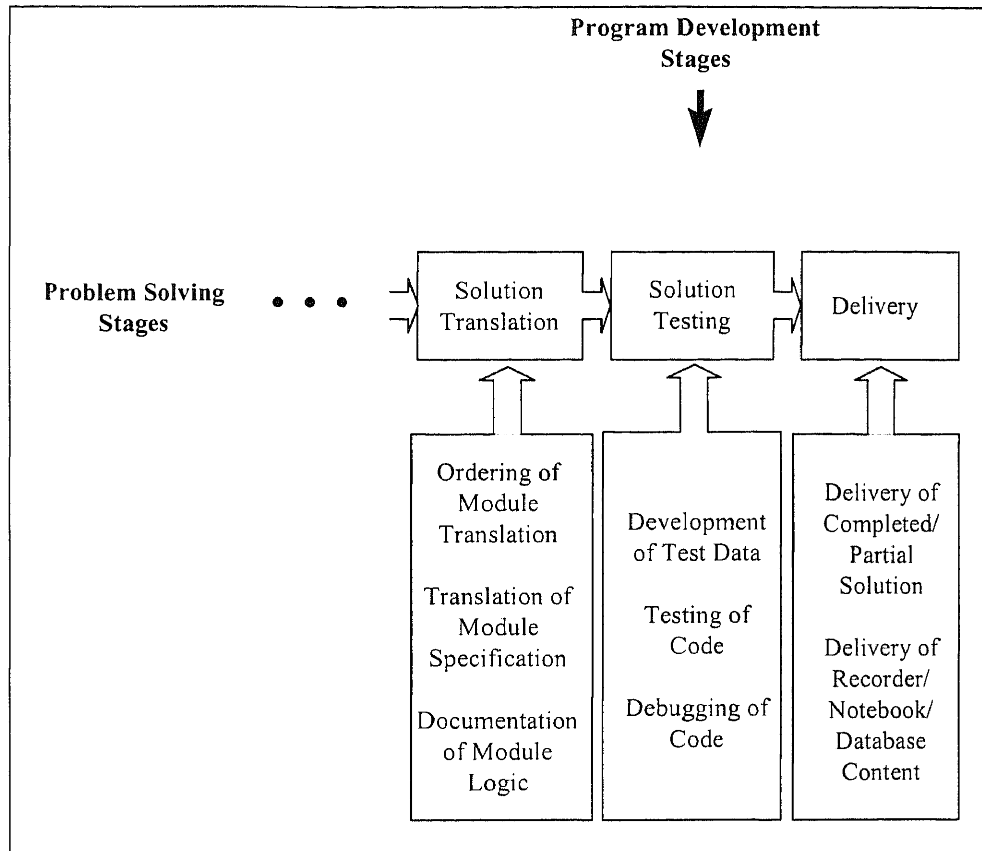


Figure 27 Workflow in program development stages

Each step of every stage is supported by tools to assist the students in solving the problem. SOLVEIT captures the outcome of each stage and stores it in a reference database. Each subsequent stage of SOLVEIT can access the database and use the information relevant to the tasks done in that stage.

The following is an example problem solving and program development session using SOLVEIT. The figures presented in this section show the sequence of transformations beginning with problem description through code testing and the outcome of each stage from problem formulation to solution delivery. Arrows in the figures refer to the output of each step which is stored in the *reference database* and may be refined in subsequent stages. Dotted boxes imply that the data is stored in the *reference database*, but is not transformed any further. An example is also provided showing the progress toward a complete solution for a problem comparable to those solved by students in the earlier part of the first course on problem solving and programming. Only the activities of the problem solving stages performed within SOLVEIT are included.

The student invokes SOLVEIT, encountering an initial screen that displays a project status and a view of all six stages and the tasks carried out in each stage. This screen remains active throughout the problem solving session, allowing the student to have an overall view of project progress. The system displays, within this main window of SOLVEIT, a command bar giving five options to choose from: FILE, EDIT, SOLVEIT, TOOLS, AND HELP. FILE opens a new file or an existing file, saves a file, prints or exits the system. EDIT provides cut, copy, and paste functions. SOLVEIT, the primary feature of the system, provides the student with the choice to formulate the

problem, plan, design, test, translate and deliver the solution. TOOLS provide access to the *project notebook/graphics editor*, *verbalization tool*, *information elicitation editor, reference database and recorder*. A HELP feature is also provided. An iconic representation of functionalities and tools is also displayed.

Starting a problem solving session requires the opening of a new file. For this, the student must enter identifying information including project name, student name, identification number, course, section, instructor name, teaching assistant name and project number. This information, shown in *Figure* 28, is written to the database. To continue with a previously started session, an existing file can be re-opened and the desired tasks can be selected.

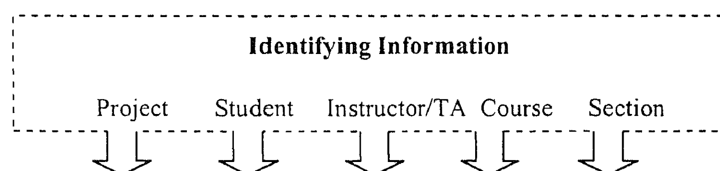


Figure 28 Identifying information

5.3.1 Formulating the Problem

The student begins the process by describing the problem to be solved. Typically, the problem is presented to the student in the form of a handout given in class. Then, the student must carefully examine this description and find the information necessary to solve the problem. By the end of this stage, all relevant facts of the problem are identified and stored in the *reference database*, as shown in *Figure* 29.

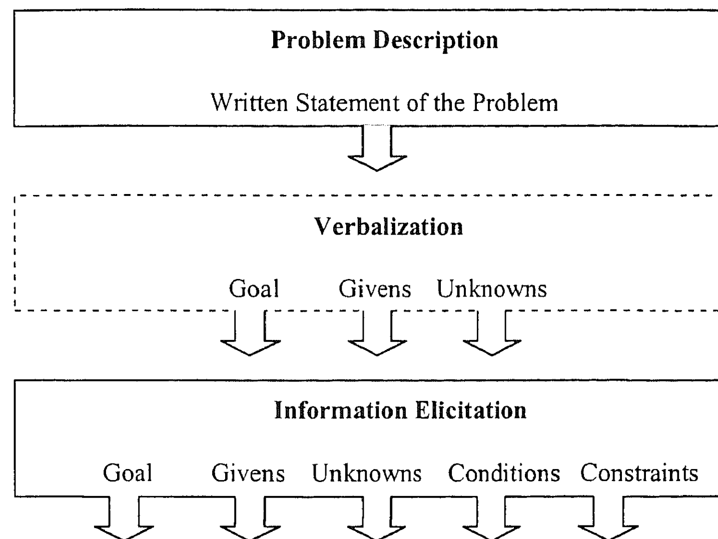


Figure 29 Outcome of problem formulation

5.3.1.1 Describing the Problem

A *text editor* enables the student to type in the problem description using either their own wording or the wording of the assignment sheet. Access to editing functions and to the *notebook* is provided. Once the problem description is entered and saved in the *reference database*, the student invokes the *verbalization tool* that initiates a series of questions. *Figure* 30 shows a problem description.

Formulating the Problem

1. Problem Description

Space travel will soon become a reality. Commercial rockets will be able to take those interested for short trips into space. Flight departures and arrivals will have to be predictable, as you would not want to travel into the far reaches of the unexplored without first considering what time your favorite computer science class is held, and whether you will return on time for the lecture. A program to solve this dilemma should be written.

Given the blastoff-time of a rocket ship represented as 3 numeric values (hours, minutes and seconds) and a 4th numeric value representing the ship's flight-time given in seconds, calculate the return-time of the rocket ship (time-of-day and day). Rocket refueling must be taken into consideration when calculating return-time. That is: for every 6 hours of fight-time a 15 second refuel-time is required.

The program will run (accept keyboard input) as many times as the user requests.

Figure 30 Problem description

5.3.1.2 Verbalization

Questions about the problem are asked using the verbalization tool to make the student revisit the problem description typed-in via the editor. One way for the student to answer these questions is to closely re-examine the various aspects of the problem. The questions are related to problem facts such as goal, givens, unknowns and other requirements. If the student cannot answer these questions using the current description of the problem, this may indicate that the problem was not well formulated by the student, or perhaps was poorly presented in the assignment sheet. Otherwise, the student can then edit the problem description in the editor. The *verbalization tool* can be reinvoked as many times as it is necessary to assure that the description accurately

represents the problem. Repeated answers to the questions are overwritten in the *reference database*, with only results of last session stored, but saved in their entirety into the recorder. The questions are based on Polya's problem solving method and can be modified to suit specific types of problem. *Figure* 31 shows a verbalization session.

Formulating the Problem
2. Verbalization

1. What is the goal?
   To find the return-time and return-day of a rocket ship.

2. What are the givens?
   The blastoff-time of the rocket ship, the ship's flight-time and the refuel time.

3. What are the unknowns?
   The return-time and return-day of the rocket ship.

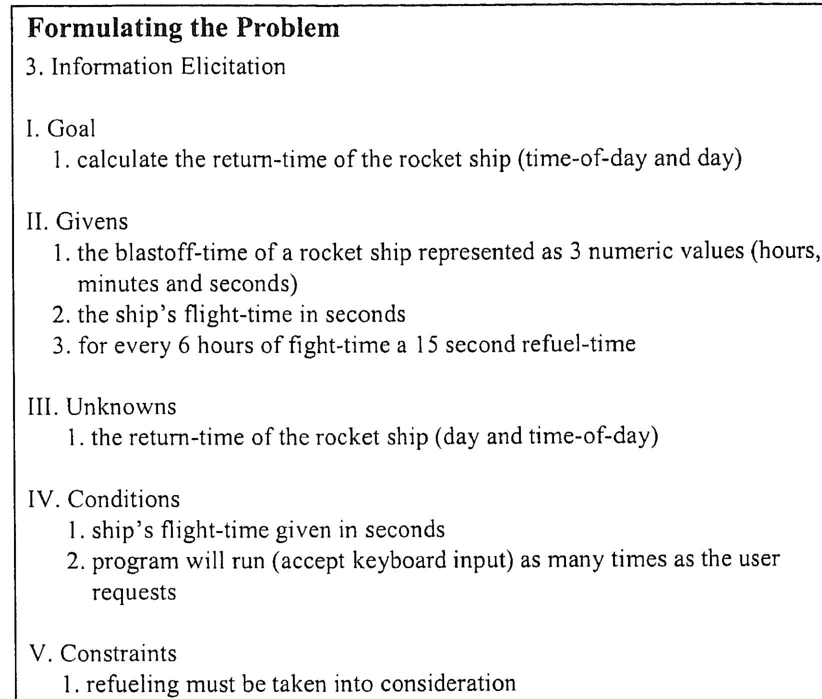
Figure 31 Verbalization in problem formulation

5.3.1.3 Finding Relevant Facts

The student now moves further into problem formulation. The information elicitation option is selected and a new window is displayed. Once again, the system redisplays the problem description. The student re-examines the description and identifies problem's goal, givens, unknowns, conditions, and constraints using the *information elicitation editor*. Information elicitation is done by allowing the student to select the raw data in the problem description by highlighting the relevant text and adding it to appropriate fields, such as goal, givens, unknowns, conditions, and constraints.

All information gathered in this stage and subsequent stages, as well as any changes to the information are stored in the *reference database*. Also, a *project notebook/graphics editor* is made available at the beginning of a problem solving session

for students to use and a project *recorder* is activated to maintain a progress transcript. *Figure* 32 shows the result of information elicitation.



Formulating the Problem

3. Information Elicitation

I. Goal
1. calculate the return-time of the rocket ship (time-of-day and day)

II. Givens
1. the blastoff-time of a rocket ship represented as 3 numeric values (hours, minutes and seconds)
2. the ship's flight-time in seconds
3. for every 6 hours of fight-time a 15 second refuel-time

III. Unknowns
1. the return-time of the rocket ship (day and time-of-day)

IV. Conditions
1. ship's flight-time given in seconds
2. program will run (accept keyboard input) as many times as the user requests

V. Constraints
1. refueling must be taken into consideration

Figure 32 Information elicitation

5.3.1.4 Organizing Relevant Facts

The information produced in all stages of SOLVEIT is logged to the *reference database* where it is maintained for access and transformation throughout the problem solving session.

5.3.1.5 Making Notes and Drawing

The student is given access to a *project notebook/graphics editor* that can be used to write notes, record ideas, test possibilities, draw diagrams, and introduce notations to represent aspects of the solution as the student deems necessary. This *project notebook/graphics*

editor is analogous to the scratch pad usually used by students when solving problems. The notebook and its content remain available during a problem solving session.

5.3.1.6 Project Transcript and Playback

A transcript of all interactions is maintained by the project *recorder*, which is write-protected from the student. In addition, snapshots of the *reference database* are also copied to the *recorder* any time the student advances through the stages of SOLVEIT. This recording can be used by the instructor to evaluate the students thinking process and approach to problem solving, or can be examined by students to retrace their progress toward the solution.

5.3.2 Planning the Solution

The student has identified the goal, givens, unknowns, conditions and constraints in the problem formulation stage. In this stage, an initial plan for solving the problem is outlined and the information gathered earlier is transformed into subgoals and data structures. By the end of this stage, the problem is decomposed into smaller sub-problems and a solution is ready to be designed. Subgoals and data description are stored in the *reference database*, as shown in *Figure* 33.

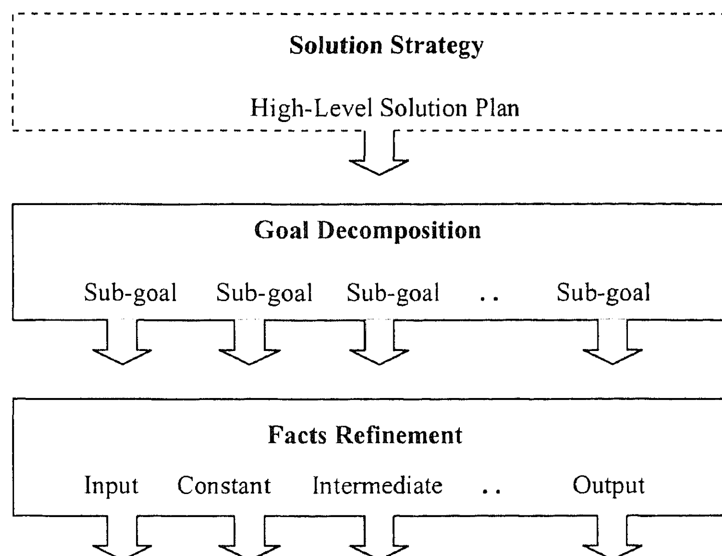


Figure 33 Outcome of solution planning

5.3.2.1 The Project Plan

The student must outlines the steps to be taken to solve the problem. The *Project Plan* is entered into a text editor. This plan is an initial and brief description of the student's strategy to solve the problem. *Figure* 34 shows a high-level solution strategy.

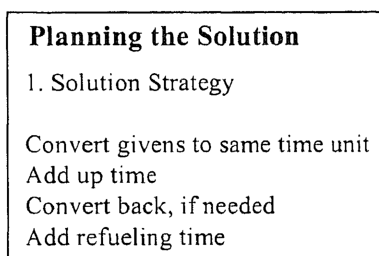


Figure 34 Solution strategy

5.3.2.2 Refining Goals into Subgoals

The goal identified in problem formulation is retrieved from the *reference database* and the student decomposes the goal into subgoals and assigns an identification name and a description to each subgoal. The result is a series of subgoals maintained in the *reference database*. *Figure* 35 shows goal decomposition.

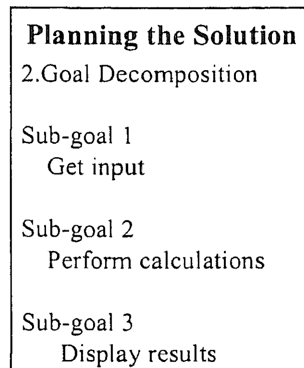


Figure 35 Goal decomposition

5.3.2.3 Refining Givens and Unknowns into Data Structures

Having refined the goal, the student may begin to refine the givens and unknowns into data structures. This information is retrieved from the *reference database* and the student can edit the field to use a syntactically correct or more meaningful name. Object types are selected from a list of valid data types. The origin or source of the data, for example whether the values for variables are to be entered through the keyboard or a file, is specified. Finally the structure of each object is defined as a basic or composite data type. The definition of the output data proceeds in a similar fashion with the contents of the *reference database* retrieved and appropriate fields edited as required. The destination of the output can be the screen, printer, or file. The same procedure is

followed for intermediate data and problem constants. As a result the *reference database* is updated. *Table* 20 shows a list of data description.

Table 20 Data description

Planning the Solution

3. Data Description

I. Input (givens)

Name	**Description**	**Origin**	**Type**	**Structure**
1. Hours	Blastoff hours	Keyboard	int	Basic
2. Minutes	Blastoff minutes	Keyboard	int	Basic
3. Seconds	Blastoff seconds	Keyboard	int	Basic
4. Flight_Time	Length of trip	Keyboard	int	Basic
5. TIME_TO_REFUEL	Tank capacity in hours	Constant	int	Basic
6. REFUEL_TIME	Time required for refueling	Constant	int	Basic

II. Output (unknowns)

Name	**Description**	**Destination**	**Type**	**Structure**
1. Return_Hour	Computed hour of return	Screen	int	Basic
2. Return_Minute	Computed minute of return	Screen	int	Basic
3. Return_Second	Computed second of return	Screen	int	Basic
4. Return_Day	Arrival day	Screen	int	Basic
5. Is_Valid	Data validity	Intermediate	int	Basic
6. Temp	Temporary calculations	Intermediate	long int	Basic
7. Total	Converted time in seconds	Intermediate	long int	Basic

5.3.3 Designing the Solution

The student has refined the goal into subgoals and refined the givens and unknowns into data structures in the solution planning stage. In this stage, subgoals are organized and refined, relationships between them are established, and specifications and algorithmic logic are determined. By the end of this stage, the algorithmic solution is ready to be translated into language code. Design charts, solution specification and algorithmic logic are stored in the *reference database*, as shown in *Figure* 36.

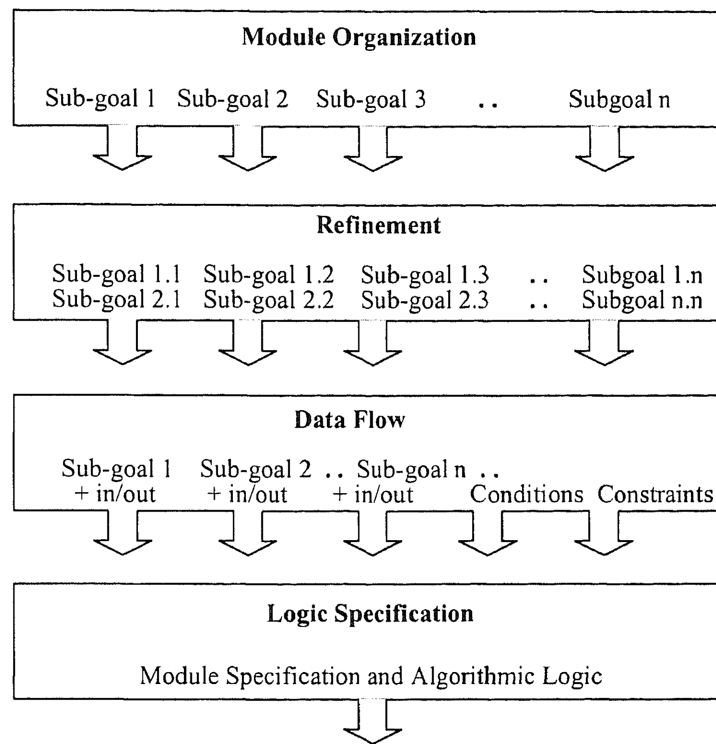


Figure 36 Outcome of solution design

5.3.3.1 Module Organization and Refinement

The first task to be performed in design is construction of the structure chart, a modularized graphical representation of the solution. Though incomplete, the structure chart is already formed with the main goal identified in the problem formulation stage represented as the main module and first-level module refinement consisting of subgoals created during the planning stage, already stored in the *reference database*. This structure chart consists of a collection modules that require organization and refinement. The student is able to rearrange the modules, refine them as necessary and establish their

hierarchy to form the logic of the solution. *Figures* 37 and 38 show a structure chart refinement.

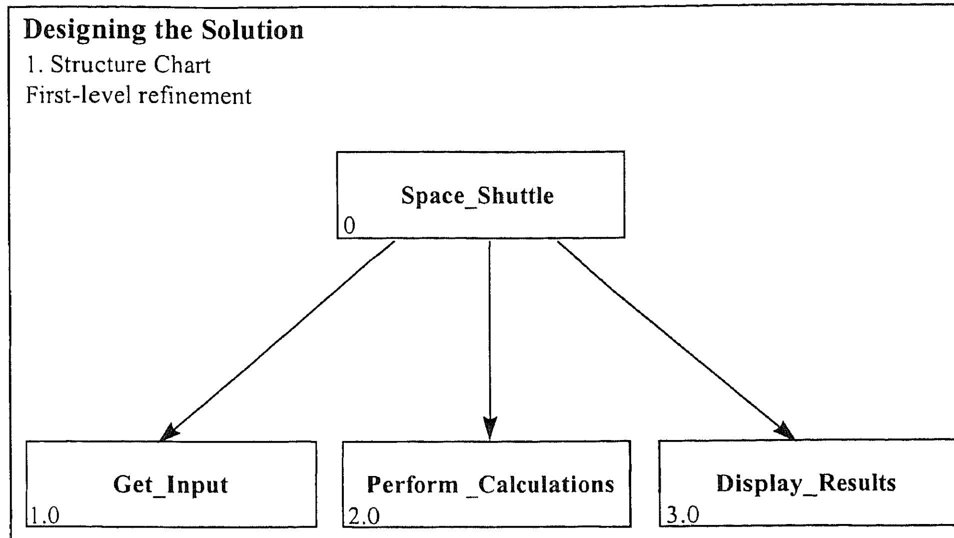


Figure 37 Structure chart - first level refinement

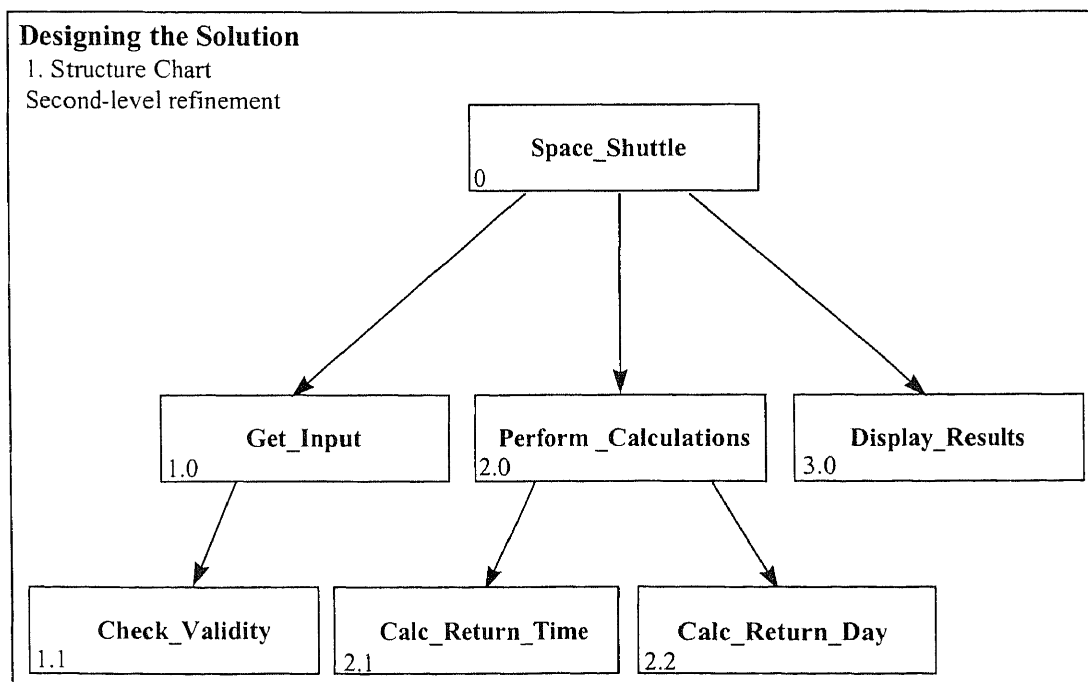


Figure 38 Structure chart - second-level refinement

5.3.3.2 Module Communication

Communication between modules is established as a data flow relationship. A list for all input/output data is retrieved from the *reference database* and displayed side-by-side with the structure chart. The student selects the data element and associates it with a specific module as a sender or receiver of that element. *Figure* 39 shows the data flow between modules.

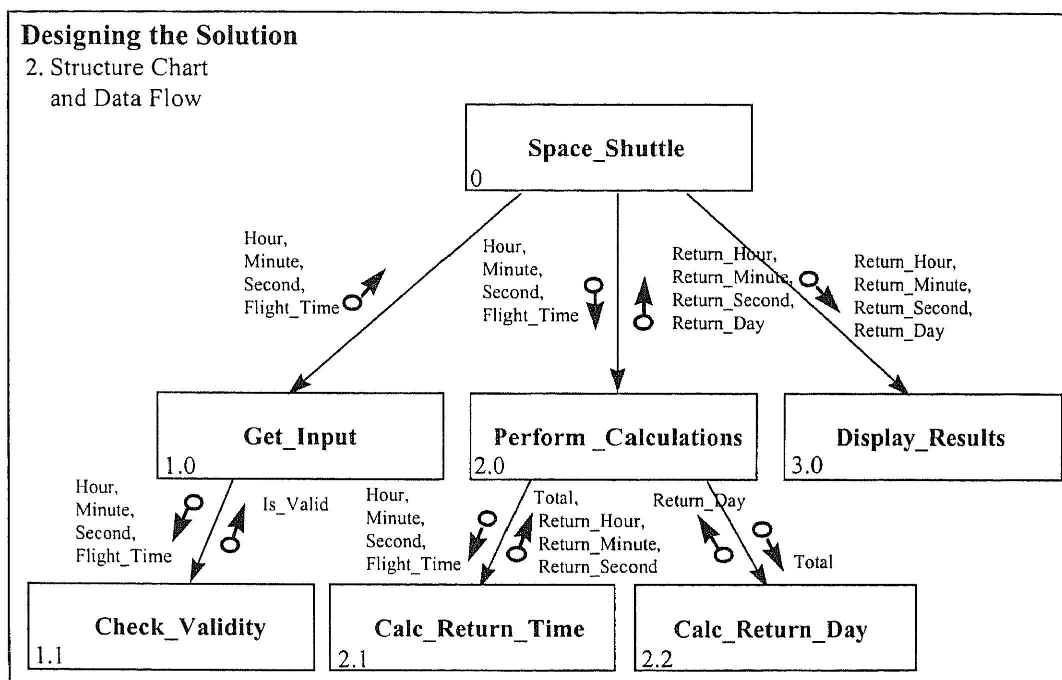


Figure 39 Structure chart and data flow

5.3.3.3 Modules Specification and Algorithm Construction

Constructing the algorithmic specification and logic for each module is the last step before translation to language code. Language-independent syntax is provided to represent module logic. The student may implement all modules in the sequence prompted by the system or may choose specific modules for implementation.

Information about each module, stored in the *reference database*, is displayed and used to complete solution specification. The student is requested to enter a short English description for each module including pre-conditions and post-conditions, which will be used as comments in the heading of the module during translation.

The computational requirements of the solution is accomplished by constructing syntactically correct statements. Remembering syntax details, while elementary to the expert, is often daunting to the beginner. This activity requires that the student has detailed syntactical knowledge of language constructs. SOLVEIT provides the student with access to an *algorithmic constructs toolbox* containing syntax templates used to define and construct the logic for these modules. The modules can be saved and reused in subsequent problem solving and program development sessions. Previously written modules appropriate for the present problem may be reused and combined with new modules created to meet the requirements of the current problem. General purpose modules developed in this stage are stored in a library of functions for reuse. *Figure* 40 to *Figure* 43 show module specifications and algorithmic logic based on structure chart hierarchy and data flow.

Designing the Solution
3. Module Specifications

1.0 Get_Input_Data
Data Received: Hours, Minutes, Seconds, Flight_Time from keyboard
Information Returned: Hours, Minutes, Seconds, Flight_Time by reference.
Logic: Gets hours, minutes, seconds and flight time from user.

1.1 Check_Validity
Data Received: Hours, Minutes, Seconds, Flight_Time.
Information Returned: 1 for valid input 0 otherwise.
Logic: Check variables Hours, Minutes, Seconds, Flight_Time to see if input falls within predefined ranges.

Figure 40 Module specification for input subproblem

Designing the Solution
3. Module Specifications

2.0 Perform_Calculations
Data Received:Hours, Minutes, Seconds, Flight_Time.
Information Returned: Return_Hour,Return_Minute,Return_Second, Return_Day by reference.
Logic: Calls Calc_Return_Time() and Calc_Return_Day(long int, int&).

2.1 Calc_Return_Time
Data Received: Hours, Minutes, Seconds, Flight_Time.
Information Returned: Return_Hour, Return_Minute, Return_ Second, Return_Day by reference
Logic: Calculates Return_Hour, Return_Minute, Return_ Second from Blast off Hours, Minutes, Seconds, Flight_Time, and Refueling constants

2.2 Calc_Return_Day
Data Received: Total flight time (in seconds)
Information Returned: Day of return (by reference).
Logic: Calculates how many days after blastoff rocket returns.

Figure 41 Module specification for computation subproblem

Designing the Solution
3. Module Specifications

3.0 Dispaly_Results
Data Received: Return_Hour, Return_Minute, Return_Second, Return_Day.
Information Returned: none
Logic: Displays the returning flight time and day on the computer screen.

Figure 42 Module specification for ouput subproblem

Designing the Solution
4. Algorithmic Logic

```
0.0  Do steps 1.0 to 3.0
1.0  Get Input:  Hours, Minutes, Seconds, Flight_Time from keyboard
        1.1 Check Validity
                1.1.1 Check Hours,Minutes,Seconds, and Flight_Time for range validity.
                1.1.2 if valid Continue from step 2.0.
                1.1.3 else Repeat from step 1.0
2.0 Perform Calculations
        2.1 Calc Return Time
                2.1.1 Convert to seconds and add to get preliminary flight time
                      Temp = (3600 * Hours) + (60 * Minutes) + Seconds + Flight_Time
                2.1.2 Calculate refuel time needed and add to total time
                      Total = Temp + (  ( Temp / TIME_TO_REFUEL ) * REFUEL_TIME )
                2.1.3 Convert back to Hours,minutes,seconds
                      Return_Hour = Total / 3600
                      Return_Minute = Temp / 60
                      Return_Second = Temp % 60
        2.2 Calc Return Day
                2.2.1 Return_Day = Total / 86400
3.0 Display Results
        3.1 Display:  Return_Day,Return_Hour, Return_Minute, and Return_Second
4.0 Check to see if user wants to repeat process
        4.1 if yes Repeat from step 1.0
        4.2 else end
```

Figure 43 Algorithmic logic

This concludes the problem solving stages within SOLVEIT and the student continues with the program development stages by transferring into a programming

language environment (i.e. Pascal, C, C++ compilers) to translate the algorithmic specification and logic into code.

5.3.4 Translating the Solution

The algorithmic logic created in the *module logic specification editor* is stored in the *reference database* as a text file, but cannot be compiled directly. In this stage, the algorithmic logic created in the design stage is copied into the syntax editor of the target programming environment and is used as the basis for code translation. By the end of this stage, syntactically correct code that can be compiled using a language compiler is produced and is ready to be tested, as shown in *Figure* 44.

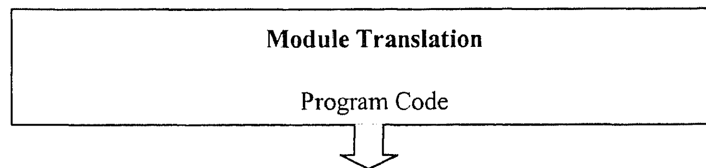


Figure 44 Outcome of solution translation

5.3.4.1 Code Translation

The algorithmic logic produced in the design stage is used as the basis for code translation. The work done in previous stages amounts to a solution for the problem, but additional work is required to be able to execute the solution on a computer. Using the facilities of the programming environment, the student can initially translate the logic into a program shell that displays the entire skeletal program, which can be then visually verified. Final translation can proceed either on a module-by-module basis or complete translation can take place.

5.3.5 Testing the Solution

The translation stage produces a program that can be compiled, executed, and tested using the tools of a programming language compiler. In this stage, the syntax and logic correctness of the program must be verified and any errors removed. By the end of this stage, a syntactically correct solution is ready to be executed and to produce the results, as shown in *Figure* 45.

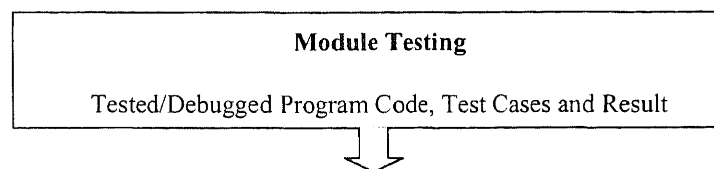


Figure 45 Outcome of solution testing

5.3.5.1 Code Testing

Test data is constructed and used for comprehensive program verification to uncover program errors, both syntactical and logical, and to fix them when found. This requires that the program does not violate any language rules and its output is consistent with problem requirements and design specifications. With SOLVEIT, the student follows an organized process of problem formulation, planning and design before the code is produced. This makes it possible to compare the problem requirements, solution specifications and program results.

5.3.6 Delivering the Solution

After execution and testing, the student returns to SOLVEIT and completes the process by carrying out the delivery stage. By the end of this stage, a project turn-in package to be delivered to the instructor is assembled, as shown in *Figure* 46.

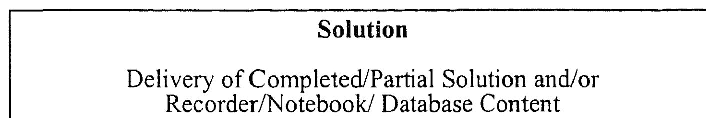


Figure 46 Delivery of complete solution

5.3.6.1 The File Manager

The student has access to the information produced in all stages of SOLVEIT, the content of the *reference database*, a complete transcript of the *project notebook/graphics editor* and the *recorder* showing the result of interactions between the student and the system. The student can choose to print the result of the entire project, saves it to a disk file, or select the parts needed.

The end result is a fully documented program based on problem requirements and solution specifications. The *file manager* is used to produce a complete package consisting of the outcome of all stages of the process and, if required, the content of *reference database*, *project notebook*, and *recorder*.

5.4 Implementation of SOLVEIT

The implementation, deployment and testing of an initial prototype version of SOLVEIT based on the specifications described in this chapter took place over a period of 2 years.

The system was used in a problem solving and programming course for the first time starting in the Fall of 1996. Initial system testing were performed during the Summer and Fall of 1996. The system was modified as a result this testing (see sections 6.1.1 and 6.1.2). Integration into the curriculum took place in the Fall of 1996 and Spring of 1997. Evaluation was performed during these two semesters. The evaluation design is presented in Chapter 6 and the results are reported in Chapter 7.

5.5 Summary

When first learning problem solving and programming in a setting such as the introductory course in computer science, students must develop the necessary skills that enable them to carry out all of the involved activities. As documented in Chapter 2, some problem solving and program development activities are supported by existing tools that are intended to assist novice programmers. Programming, in its implementation stage, is one of the activities that has been the beneficiary of such automation. SOLVEIT addresses the entire problem solving and program development process and advocates the need for further automation. It provides the students with a system that emulates the types of activities and interactions that take place when solving a problem using the Dual Common Model described in Chapter 4. This environment can be an integral part of the curriculum. The system serves mostly as a facilitator of the problem solving process. The students remain active participants and self-regulated learners who ask questions, formulate answers, explore alternatives, and, more importantly, learn from their own experiences. The teacher also remains involved in the learning process. The problem, its

solution and the student thought processes during problem solving can be obtained by the teacher for assessment and feedback, through the aggregation of an electronic portfolio.

CHAPTER 6

EXPERIMENTAL DESIGN: TESTING AND EVALUATION

A key motivation for the development of educational computing systems is improving students' ability to learn independently. Most efforts in this area have focused on the theoretical aspects of systems and not on measuring their effectiveness for students (see Chapter 3). In our research, we have developed and implemented a plan for testing and evaluating SOLVEIT and for studying its impact on students' problem solving and program development abilities, their cognitive skills, knowledge, perception, and attitudes and motivation.

This chapter discusses the two components of this study: (1) system pre-deployment testing and (2) system evaluation. For testing, a description of user testing and protocol analysis is presented. For evaluation, a description of the experiment including the hypotheses and research questions used in the evaluation, the subjects, the design, instrumentation and data collection are presented.

6.1 Introduction

SOLVEIT was designed as a problem solving and program development environment that takes into consideration the problem solving skills required by students learning how to program, as well as the specific knowledge related to the use of a programming language. The system provides tools that the students use in *formulating* the problem; *planning* and *designing* the solution; and *monitoring* and *evaluating* the solution's

progress. SOLVEIT encourages students to *understand* the problem and its requirements and to *think* about possible solutions before engaging in implementation details.

This study sought to verify the claims made regarding the cognitive model of SOLVEIT and to investigate the impact resulting from using the SOLVEIT environment as a support tool for problem solving and program development.

6.2 System Testing

Pre-deployment testing was strictly concerned with system functionality and the readiness of SOLVEIT to be integrated into the classroom. It began in the Summer of 1996 with an initial version of SOLVEIT. This covered user testing and protocol analysis as described in T*able* 21.

Table 21 The testing plan

User	Testing Method
1. Students in Summer Program (EOSEP). 2. Students in Summer Program (SPCS).	User Testing
1. Graduate Students in Bridge Program. 2. Freshmen Students in Winter Session Program.	Protocol Analysis

Two distinct student populations were selected to obtain a broad view of student feedback. One group was comprised of students in the Summer Program in Computer Science (SPCS), a program designed for highly motivated high school students in grades nine through twelve. The other group included students in the computer science course of the Educational Opportunity Summer Enrichment Program (EOSEP), which is designed for underprepared high school seniors accepted to attend NJIT as freshmen. These students, as well as other students, are the system's expected future users when taking their first course in computer science.

6.2.1 User Testing

SOLVEIT was installed in two laboratories used by students in both summer programs to solve assigned programming projects. Sixty two students, 16 in SPCS and 46 in EOSEP, participated in testing. Teaching assistants and tutors selected to work with students in the lab solicited and compiled user feedback. The results were used to correct bugs and to enhance the system and produce a new version before it was integrated into the curriculum for first semester evaluation in Fall 1996.

The system was tested again with 16 first year students, registered to take the first course in computer science in the Spring 1996 semester, enrolled in a preparatory winter session. Students solved a complete problem using a new version of SOLVEIT and the instructor and two teaching assistants solicited and compiled user feedback. The testing results, along with results of protocol analysis (discussed below), performed at the same time, were used to produce a new system to be used in the second semester evaluation in Spring 1997.

6.2.2 Protocol Analysis

The method to assess the usability and to understand how students formed their mental model of SOLVEIT was protocol analysis (Goodwin, 1987; Turoff & Hiltz, 1997). This method involves asking potential users of the system to perform a predetermined task using the system and at the same time "think out loud" about what they are doing (Newell & Simon, 1972; Carrol, et al. 1985).

Again, two distinct student populations were selected to obtain more conclusive feedback on the user interface and the ease of use of system functionality using the thinking out loud method. The first group of students consisted of two graduate students, whose undergraduate degrees were not in computer science, taking a bridge course in program design and data structures. The second group was made up of three students, selected randomly from the winter session group discussed above. Students participating in protocol analysis gave a running commentary on what they were attempting to do, what type of problems they encountered, and any other task related thoughts. Each session was tape recorded and analyzed. Both groups encountered significant problems with the user interface of the *information elicitation* and the *goal decomposition* tools. Some minor problems with inconsistencies in the placement of *help*, *next* and *back* buttons. Corrective actions were taken by redesigning the screen interface for both information elicitation and goal decomposition tools. Also, a reorganization of command buttons was made on all screens. Problems with the goal decomposition tool were uncovered again during an informal protocol analysis session with the same graduate

students that lead to more changes and simplification of the user interface. No other major problems were reported.

6.3 Evaluation Method

The principal method of evaluation was controlled experimentation designed to understand and thereby improve the learning and teaching of problem solving and program development to novice students. There are many different ways to design the evaluation plan, with no single correct approach (Frechtling, 1992). Some studies require qualitative analysis; others require quantitative analysis; most benefit from a combination of the two (Herman, Morris, & Fitz-Gibbon, 1987). The testing of hypotheses and research questions of this study required analysis of students' performance on homework, quizzes and exam; self-administered questionnaires and reports; and in-class students' observation and monitoring. Therefore a combination of qualitative, quantitative and anecdotal reporting was selected for this study (Fowler, 1993; Love, 1991). *Table* 22 describes the evaluation approach.

Table 22 The evaluation approach

User	Evaluation Method	Type of Data
Students Taking First Course in Computer Science.	Controlled Experiment	Qualitative, Quantitative, and Anecdotal

An experiment to test the hypotheses, answer the research questions and analyze the results was conducted over two semesters. Students were advised of the evaluation plan

and how it was to proceed; those who agreed to participate were asked to read and sign a Protection of Human Subjects consent form and to answer the pre-test questionnaire included in Appendix 1. A post-test questionnaire was also administered at the end of the semester (also can be found in Appendix 1).

6.3.1 Hypotheses and Research Questions

Hypotheses and research questions were designed to assess whether the tools within the SOLVEIT environment aid the students in their search for the solution, producing better results, enhancing their perception, attitude, and motivation, and in the development of skills and knowledge necessary for problem solving and program development. The hypotheses were designed to test the relationship between the various tools of the system and students' performance. The major assumption of the hypotheses was that students using SOLVEIT will perform better on problem solving and program development tasks than students not using the system. The research questions were designed to examine unforeseen effects not directly related to the system. *Table* 23 describes the hypotheses and *Tables* 24 and 25 describe the research questions. Detailed discussion of hypotheses and research questions, the rationale for their selection, their measures and impact, the procedure and instruments for their verification is provided in section 6.3.

Table 23 Process hypotheses

Hypothesis (process)	Measures	Impact	Assessment
1. Students in the experimental group will: 1.1 show clearer understanding of problem, and will 1.2 identify problem's facts better than students in the control group.	Problem Formulation and Facts Identification	Solution Planning and Solution Correctness	Problem Description, Information Elicitation and Organization
2. Students in the experimental group will: 2.1 engage in planning more often than students in the control group, and will 2.2 apply early decomposition of problem goals into subgoals.	Solution Planning and Goal Decomposition	Solution Design and Solution Correctness	Plan Outline, Goal and Data Refinement
3. Students in the experimental group will demonstrate a higher level of competence in problem decomposition, inter-module organization, refinement, and specification skills than students in the control group.	Solution Design and Module Decomposition/ Organization/ Specification	Program Composition and Solution Quality	Design Charts and Algorithmic Module Logic

Table 24 Product research questions

Research Question (product)	Measures	Impact	Assessment
1. Will students in the experimental group produce more efficient, or better suited, programs compared to students in the control group?	Code Efficiency	Program Composition	Program Code
2. Will students in the experimental group produce more complete and robust programs compared to students in the control group?	Code Reliability	Program Correctness	Test Data
3. Will students in the experimental group produce clearer and more understandable solutions compared to students in the control group?	Code Readability	Program Comprehension and Modification	Program Documentation
4. Will students in the experimental group produce more accurate solution specifications, program code and results consistent with problem requirements compared to students in the control group?	Overall Solution Correctness	Program Testing and Debugging	Solution Specifications, Program Code and Results

Table 25 Subjective research questions

Research Question (subjective)	Measures	Impact	Assessment
5. Will students in the experimental group have a more favorable perception of their learning experience compared to students in the control group?	Students' Satisfaction	Students' Morale	Students' Feedback and Questionnaires
6. Will students in the experimental group exhibit better attitude and increased motivation toward learning problem solving and programming compared to students in the control group?	Students' Commitment	Students' Performance	Students' Records, Feedback and Questionnaires

6.3.2 Subjects

The evaluation took place in the Fall semester of 1996 and Spring semester of 1997. Three sections of the first course on problem solving and programming taken by students majoring in computer science, information systems, mathematics and physics (see Appendix 2 for course syllabus and related material) were used each semester. In the first semester, two sections were the control and one received the experimental treatment. In the second semester, one was the control and two were experimental. *Table* 26 describes the assignments of sections to conditions in the two semesters.

Table 26 Assignments of sections to conditions

Semester	Course	Section	Type
Fall 96	CIS 113	01	Control
		03	Control
		05	Experiment
Spring 97	CIS 113	02	Experiment
		04	Experiment
		06	Control

6.3.3 Design

The experiment was a pre-post control group design covering two different methods of doing problem solving and programming. The first method used a traditional programming environment (C++ compiler) and the second method used both SOLVEIT for problem solving activities and a traditional programming environment (also C++ compiler) for program development. The traditional group was the control group and the SOLVEIT group was the experimental group. Both groups received *the same instruction, assignments, quizzes and exams*.

The independent variable is the integration of SOLVEIT environment into the course for students in the experimental group and the absence of SOLVEIT for students in the control group.

The dependent variables fall into three categories. Each category consists of a set of variables: (1) the problem solving process includes problem formulation, planning,

design, translation and testing (the last two tested in terms of product), (2) the product includes efficiency, reliability, readability and correctness, and (3) subjective includes perception, attitude and motivation. *Table* 27 describes the independent and dependent variables of the hypotheses and *Tables* 28 and 29 describe the independent and dependent variables of the research questions.

Table 27 Hypotheses' (process) variables

Hypothesis (process)	Independent Variable	Dependent Variable
1	SOLVEIT (presence or absence)	Problem Formulation
2		Solution Planning
3		Solution Design

Table 28 Research questions' (product) variables

Research Question (product)	Independent Variable	Dependent Variable
1	SOLVEIT (presence or absence)	Efficiency
2		Reliability
3		Readability
4		Correctness

Table 29 Research questions' (subjective) variables

Research Question (subjective)	Independent Variable	Dependent Variable
5	SOLVEIT (presence or absence)	Perception
6		Attitude and Motivation

All grading was done by course Teaching Assistants (TAs) who were not given any details of the evaluation plan, and who received training covering course content, teaching skills, grading criteria, method and instruments, given by the course instructor. The grading process was blind as to whether the student was in the experimental or control group. The instructor was in charge of the lecture and preparation of course material, programming assignments, quizzes and exams, but did not participate in any evaluation of students' performance. The TAs and the instructor met weekly to discuss the coverage of material and assigned work.

6.3.4 Instrumentation

Researchers analyzing students' problem solving abilities have suggested assessment methods and developed various evaluation instruments with comparable objectives and performance (Hartman, 1996; Meier, 1992; Szetela, 1987; Charles, Lester, & O'Daffer, 1987). *How to Evaluate Progress in Problem Solving* by Charles, Lester, and O'Daffer (1987) provides comprehensive guidelines for conducting evaluation plans for problem solving. Some of the techniques and instruments for measuring problem solving progress

provided in the book were adapted and revised to fit the needs of this evaluation. New instruments for evaluating the process and the product of problem solving and for students' self-assessment reports were devised. Each process and product instrument contains a description of possible outcome, the indicators to be examined, and a scoring scale. An additional instrument for quiz/exercise problems was also devised. These instruments were used initially to grade students work in both SPCS and EOSEP programs and in one section of the first course in problem solving and programming in Summer 1996. Uncovered problems were corrected and the instruments were revised before they were used during the experiment, in the following Fall and Spring semesters.

The pre-test questionnaire covered identifying, general and experience information. The post-test questionnaire covered identifying and general information, course content, problem solving and programming issues, and course outcomes. *Table* 30 describes the instrument for the hypotheses and *Tables* 31 and 32 describe the instruments for the research questions. *Table* 33 describes the consent form and pre-post test questionnaires.

Table 30 Hypotheses' (process) instruments

Instrumentation	
Hypotheses (process)	Four evaluation instruments (three for problems and one for exercise questions) covering possible stage outcome, the indicators to be examined, and a scoring scale of 0-4.

Table 31 Research questions' (product) instruments

Instrumentation	
Research Questions (product)	Four evaluation instruments covering possible outcome, the indicators to be examined, and a scoring scale of 0-2.

Table 32 Research questions' (subjective) instruments

Instrumentation	
Research Questions (subjective)	Students' self-reports covering their retrospective feedback on problem solving.

Table 33 Consent/pre/post test instruments

Instrumentation	
Overall course	Consent form explaining research project. Pre/post test questionnaires covering identifying and general information, course content, problem solving and programming issues, and course outcomes.

6.3.5 Data Collection

Recognizing that the evaluation of all students' problem solving and program development work can be a massive undertaking (Charles, Lester, & O'Daffer, 1987),

five programming assignments, five quizzes and two major exams of all students, in all sections of the course, were selected for evaluation. The data were obtained from multiple sources including: (1) students' pre and post-test questionnaires, (2) student performance on programming assignments, quizzes, exams, (3) students' self-assessment reports, and (4) in-class student observation and monitoring. *Table* 34 describes the data collection for the hypotheses and *Tables* 35 and 36 describe the data collection for the research questions.

Table 34 Data collection for hypotheses (process)

Data Collection		
Hypotheses (process)	Programming Assignments Quizzes	Throughout the semester
	Self-reports	Periodic
	Midterm Exam	Mid-semester
	Final Exam	End-of-semester
	Pre/post Questionnaires	Beginning/end-of-semester

Table 35 Data collection for research questions (product)

Data Collection		
Research Questions (product)	Programming Assignments Quizzes	Throughout the semester
	Midterm Exam	Mid-semester
	Final Exam	End-of-semester

Table 36 Data collection for research questions (subjective)

Data Collection		
Research Questions (subjective)	Self-reports Observation and Monitoring	Periodic
	Pre/post Questionnaires	Beginning/end-of-semester

6.4 Assessing the Effect of the SOLVEIT Environment on Students' Problem Solving and Program Development Skills

The study examined measures of the three sets of variables in the following categories: *process*, *product*, and *subjective*, as shown in *Table* 37. Detailed description of each set of variables is included in sections 6.3.1 to 6.3.3. Since SOLVEIT is based on a cognitive model for problem solving and program development (Dual Common Model described in Chapter 4), we hypothesized that the system will have positive benefits on facilitating the development of students' cognitive skills and abilities required for problem solving and programming. We also asked questions whether using the various tools of SOLVEIT would have cognitive benefits that translate into producing better solutions to a given problem, developing favorable perception, better attitude and increased motivation toward the learning of problem solving and programming.

Table 37 Definitions of dependent variables' categories

Category	**Definition**
Process	Problem solving and program development method and cognitive skills required to produce solution.
Product	Solution as a product of problem solving process.
Subjective	Perception, attitude, and motivation toward problem solving and programming.

6.4.1 Process Measures

Process measures focused on the approach to problem solving used by the students to understand, develop, and produce the solution. *Table* 38 describes the dependent variables for the process of problem solving and program development.

Table 38 Process variables

Variable	Description
Formulating the Problem	Understanding the question as well as the meaning of the problem's terminology, and the identification of its facts.
Planning the Solution	Determining if problem goals can be subdivided into subgoals, identifying the tasks to accomplish each goal and subgoal, and searching for a solution strategy.
Designing the Solution	Refining the solution strategy. Organizing and sequencing of goals and subgoals, and transforming tasks into algorithmic specifications.
Translating the Solution	Implementing solution by translating detailed design into programming language code. Executing code to produce results.
Testing the Solution	Developing and applying test data to verify program correctness and accuracy of produced results.

Three evaluation instruments were developed to assess students' problem formulation, planning and design skills, and to grade their work. Twelve points out of a total of twenty points were assigned for the first three stages of the process. Translation and testing stages were graded as the product of problem solving, and were worth the remaining eight of the twenty points. The following is a discussion of the process, the tasks performed and a presentation of the hypotheses developed for this set of variables.

6.4.1.1 Formulating the Problem

The process of problem solving requires a variety of cognitive skills and begins with formulating the question. Problem formulation requires the understanding of the question as well as the meaning of the problem's terminology, and the identification of its facts. Problem understanding requires the processing of information. The techniques of verbalization and visualization are contributing factors in creating an *initial* understanding of the problem. For example, making a drawing, talking, or answering questions about the problem aid the task of problem understanding. However, problem formulation evolves with the transformation of the given problem statement into a precisely formulated model. Developing a precise model of the problem requires elicitation and organization of all relevant information and elimination of irrelevant information.

SOLVEIT requires the student to describe the problem to be solved in written and, if possible, visual form. The student also interacts with the system and answers questions about the problem. This prompts the student to think and construct interpretations about problem facts, conditions and constraints, and presumably facilitates problem understanding. Using the information elicitation tool of SOLVEIT, students also gather information about the goal, givens, unknowns, conditions and constraints of the problem. This should enable the student to form a concise and meaningful model of both the initial and goal states of the problem. The following hypothesis is presented in this domain.

Hypothesis # 1: Problem formulation

Students in the experimental group will:
1.1 show clearer understanding of problem, and will
1.2 identify problem's facts better than students in the control group.

Measures: Problem formulation and facts identification. Outcome of problem formulation stage of students in experimental and control groups for programming assignments, quizzes and exams was judged by course TAs. *Table* 39 shows the instrument for assessing students' problem formulation skills.

Impact: Solution planning and solution correctness. Students in the experimental group will develop better skills at understanding the problem and identifying the information needed to solve it. This will also provide students in the experimental group with knowledge needed to represent a precise and complete model of a problem's initial and goal states. A higher level of understating of the problem and an accurate representation are likely to lead to a correct solution.

Assessment: Using the given problem, the students are expected to write, in their own words, the statement of the problem, to answer questions about the problem and to identify and organize the information needed to solve the problem. This was graded using a scale of 0-4. Also, the post-test questionnaire included a series of problem formulation questions.

Table 39 Instrument for assessing students' problem formulation skills

The Process - Formulating the Problem		
Outcome	**Indicator**	**Scoring Scale**
Excellent representation of problem and complete identification of relevant facts, indicating full understanding, required to solve the problem.	Problem is clearly and correctly stated. All goals, givens, and unknowns are identified.	4
Reasonable representation of problem and identification of almost all relevant facts, indicating adequate understanding, required to solve the problem.	Problem is correctly stated. Most goals, givens, and unknowns are identified.	3
Incomplete representation of problem and/or identification of facts, indicating some understanding, but not enough to solve the problem.	Problem is partially stated and/or some facts are identified.	2
Inappropriate representation of problem and inability to identify relevant facts, indicating complete misunderstanding, required to solve the problem.	Problem statement is incorrect and meaningless facts are identified.	1
Lack of problem representation and identification of relevant facts, indicating complete misunderstanding, required to solve the problem.	No problem representation/fact identification attempted or completely irrelevant work.	0

6.4.1.2 Planning the Solution

Planning is a cognitive activity where the development of an appropriate solution strategy begins. The student considers various alternatives to determine the course of action suited to achieve the goal of the problem, subdivides the goal into subgoals, and identifies the tasks needed to accomplish each subgoal. The relevant information identified in the previous stage is related to the various subgoals and its role and meaning are defined. This enables the student to begin the process of carrying out the strategy to progress

toward meeting each subgoal of the problem and eventually producing the complete solution.

Using SOLVEIT, the student first outlines the strategy to solve the problem. Next, the student explicitly refines the goal into subgoals and defines the tasks associated with each subgoal. To complete the planning stage, SOLVEIT also requires the student to transform the given information into formal data representation. The following hypothesis is presented in this domain.

Hypothesis # 2: Solution Planning

Students in the experimental group will:

2.1 engage in planning more often than students in the control group, and will

2.2 apply early decomposition of problem goals into subgoals.

Measures: Solution planning and goal decomposition. Outcome of planning stage of students in experimental and control groups for programming assignments, quizzes, and exams was judged by course TAs. *Table* 40 shows the instrument for assessing students' solution planning skills.

Impact: Solution design and solution correctness. Strategic planning, early decomposition of problem goal into subgoals, and data representation should produce sufficient details essential for a design that, if properly implemented, will lead to a correct solution. Students in the experimental group should develop better skills at planning the solution and outlining the strategy needed to implement it. This experience will also provide students with the opportunity to learn the needed skills of selecting and using problem solving strategies. An accurate representation of the problem and a complete

identification of relevant information followed by a carefully planned solution is likely to lead to a correct solution.

Assessment: Using the information produced in problem formulation, the students are expected to produce a clear and appropriate plan for a solution strategy, decompose the goal into subgoals and represent the data that will lead to the design stage. This was graded using a scale of 0-4. Also, the post-test questionnaire included a series of solution planning questions.

Table 40 Instrument for assessing students' planning skills

The Process - Planning the Solution		
Outcome	**Indicator**	**Scoring Scale**
Excellent planning strategy and refinement of goals that will lead to a correct solution for the problem.	Detailed and clear planning. Complete goal refinement , task identification, and data representation.	4
Reasonable planning strategy and refinement of goals that could lead to a correct solution for the problem.	Adequate planning. Sufficient goal refinement, task identification, and data representation.	3
Incomplete planning strategy and/or some evidence of goal refinement, but not enough to solve the problem.	Partially correct planning and/or some goal refinement, task identification, and data representation.	2
Inappropriate planning strategy and complete lack of adequate goal refinement necessary to solve the problem.	Incorrect planning and meaningless goal refinement.	1
Lack of planning and refinement necessary to solve the problem.	No planning/refinement attempted or completely irrelevant work.	0

6.4.1.3 Designing the Solution

Design is a cognitive activity where the student organizes and refines the components of the solution strategy, and defines specifications to be translated into code. There are two levels of design. The first is a high-level design where a framework structure for a solution to the problem is produced, typically in visual or outline form. This involves the organization and sequencing of subgoals, the determination of whether the subgoals require further refinement, the establishment of relationships among the various solution components, and the association between data and subgoals. Subsequently, detailed design transforms subgoals into corresponding algorithmic specifications and the solution logic is readied to be translated into programming language syntax.

SOLVEIT supports a modular design methodology that allows the student to decompose and represent the problem in terms of smaller subproblems. Using a structured chart representation, the subproblems are presented visually as modules along with a data description table showing the data flow between the various modules. The algorithmic logic and module specification details are constructed within SOLVEIT. The following hypothesis is presented in this domain.

Hypothesis # 3: Solution design

> *Students in the experimental group will demonstrate a higher level of competence in problem decomposition, inter-module organization, refinement, and specification skills than students in the control group.*

Measures: Solution design and module decomposition, organization and specification. Outcome of design stage of students in experimental and control groups for programming

assignments, quizzes, and exams was judged by course TAs. *Table* 41 shows the instrument for assessing students' solution design skills.

Impact: Program composition and solution quality. Modular organization, sequencing, and refinement provide a complete overview of the various solution components which direct the formation of the algorithmic solution and simplify program composition by providing a framework for control and data flow. Students in the experimental group will gain tactical skills in problem decomposition, modular organization and refinement required for problem solving and program development. An accurate understanding of the problem and a carefully planned and designed solution is likely to lead to a good quality solution.

Assessment: Using the result of the planning stage, students are expected to produce a well organized, refined and specified design, including charts and algorithmic specifications, that will lead to the implementation of a good quality solution. This was graded using a scale of 0-4. Also, the post-test questionnaire included a series of solution design questions.

Table 41 Instrument for assessing students' design skills

The Process - Designing the Solution		
Outcome	**Indicator**	**Scoring Scale**
Excellent design strategy and module specifications that will lead to a good quality solution for the problem.	Complete module decomposition, organization, and detailed specifications.	4
Reasonable design strategy and module specifications that could lead to a solution for the problem.	Sufficient module decomposition, organization, and sufficient specifications.	3
Incomplete design strategy and/or some evidence of module specifications, but not enough to solve the problem.	Partial design and/or some module specifications.	2
Inappropriate design strategy and specifications that will not lead lead to a solution for the problem.	Improper module decomposition, organization, and specifications.	1
Lack of design and specifications necessary to solve the problem.	No design/specifications attempted or completely irrelevant work.	0

6.4.1.4 Translating and Testing the Solution

To complete the problem solving and program development process, students implement the solution by translating the detailed design into programming language code, which is then tested and executed to produce the result. The translation and testing tasks are evaluated in terms of solution quality and correctness using four research questions. The evaluation plan for translation and testing is provided below under "product measures."

6.4.2 Product Measures

The effect of the instructional context (i.e. the use of SOLVEIT) on the problem solving and program development outcome was also investigated. Product measures examined the result of problem solving - the solution. *Table* 42 describes the dependent variables for the product of problem solving and program development.

Table 42 Product variables

Variable	Description
Efficiency	Finding and implementing a well suited solution for a problem.
Reliability	Whether or not a program provides a complete and robust solution to the problem.
Readability	Clarity of program solution covering both program documentation and programming style.
Correctness	Merit and validity of any solution to a problem hinges on whether the correct answer has been found.

Unlike process measures where we claimed, in forming the hypotheses, that using the various tools of SOLVEIT will have positive cognitive benefits, with product measures we simply asked questions and searched for answers on whether such an advantage will translate into producing a better solution to a given problem.

Research questions are more appropriate than hypotheses for these measures because SOLVEIT focuses primarily on the process of problem solving, and not on the product itself. Two factors are considered in the evaluation of the solution: quality and correctness. Four evaluation instruments were developed to assess student solution

quality and solution correctness, and to grade their work. Eight points out of a total of twenty points were assigned for the product of problem solving. The following is a discussion of the product and the expected outcome, and a description of research questions developed for this set of variables.

6.4.2.1 Solution Quality

Any solution that correctly meets the requirements of the problem is considered an effective solution. The quality of solution, however, is a measure that extends beyond effectiveness and includes efficiency, reliability and readability. The research questions investigated in solution quality covered all of these three variables.

Efficiency: Efficiency in problem solving is finding and implementing the best suited solution for a problem. Often, there are several possibilities to consider when solving a problem, with some more appropriate and more efficient than others. Choosing the most appropriate algorithms, data structures and control structures for a specific problem situation leads to efficiency. Solution time, the length of time spent solving the problem, is another factor of efficiency, as are execution time and memory requirements. Solution time and machine time/space factors, however, are secondary when dealing with relatively small programs such as those in the first course on problem solving and programming and were not considered here.

Research question # 1: Efficiency of solution

Will students in the experimental group produce more efficient, or better suited, programs compared to students in the control group?

Measures: Code efficiency. Data structures, algorithms, control structures, and language constructs of the solution code of students in experimental and control groups for sample programming assignments, quizzes and exams were judged by course TAs. *Table* 43 shows the instrument for assessing solution efficiency.

Impact: Program composition. The answer to this research question will show whether problem solving skills developed by students in the experimental group will lead to composing more efficient programs more often than students in the control group could. This can be shown if the coding of students in the experimental group demonstrate the use of more appropriate algorithms, choice of data structures, and control structures more often than students in the control group.

Assessment: Using the problem's requirements, the students are expected to produce the best suited and most efficient problem solution by selecting the most appropriate algorithms and constructs for the given problem. This was graded using a of 0-2.

Table 43 Instrument for assessing solution efficiency

The Product - Solution Quality: Efficiency		
Outcome	**Indicator**	**Scoring Scale**
Well suited solution is produced.	Most appropriate algorithms, data structures, control structures, and language constructs for this problem situation are chosen.	2
Minimally acceptable solution is produced.	Program accomplishes its task, but lacks coherence in choice of either data and/or control structures.	1
Unacceptable solution quality is produced.	Program solution lacks coherence in choice of both data and control structures.	0

Reliability: Reliability refers to whether or not a program provides a complete and robust solution to the problem. The programmer must ensure that the program will function properly under all possible test cases, work for all valid input, and anticipate and respond to all invalid input.

Research question # 2: Reliability of solution

Will students in the experimental group produce more complete and robust programs compared to students in the control group?

Measures: Code reliability. Solution and program execution results of students in experimental and control groups for programming assignments, quizzes and exams were judged by course TAs. *Table* 44 shows the instrument for assessing solution reliability.

Impact: Program correctness. The answer to this research question will show whether or not problem solving skills developed by students in the experimental group will lead to developing solutions that will produce more reliable programs. This can be shown if the programs of students in the experimental group, when tested at various stages of their development, survive exhaustive code testing necessary to verify program correctness more often than students in the control group.

Assessment: Using the problem's specification and the produced solution, the students are expected to develop test data suited for verification of program reliability. Each module must be verified to ensure that unexpected mistakes, such as syntax and run-time errors, are unlikely to appear. This was graded using a scale of 0-2.

Table 44 Instrument for assessing solution reliability

The Product - Solution Quality: Reliability		
Outcome	**Indicator**	**Scoring Scale**
Robust solution is produced.	Program functions properly under all test cases. Works for all valid input, and responds to all invalid input.	2
Minimum requirement solution is produced.	Program functions under limited test cases or works only for valid input and fails to respond to invalid input.	1
Unacceptable solution quality is produced.	Program fails under most test cases.	0

Readability: Readability is a function of both program documentation and programming style, and is necessary for ensuring the clarity of program solution.

Comprehending and modifying programs is facilitated when comments and explanations are embedded within the code to explain approaches and techniques used to solve the problem. Maintenance would be difficult without adequate documentation. Program style is enhanced by establishing and adhering to coding conventions and guidelines, and contributes to producing readable solutions.

Research question # 3: Readability of solution

> *Will students in the experimental group produce clearer and more understandable solutions compared to students in the control group?*

Measures: Code readability. Documentation and style of solution code of students in experimental and control groups for programming assignments, quizzes and exams were judged by course TAs. *Table* 45 shows the instrument for assessing solution readability

Impact: Program comprehension and modification. The answer to this research question will show whether students in the experimental group will develop the documentation skills and habits that are essential for understanding programs and, if necessary, change program logic or functionality, or possibly adapt previously written code to solve a new problem. This can be shown if code written by students in the experimental group demonstrates effective use of comments and explanations within the code describing approaches and techniques used to solve the problem and of user documentation more often than students in the control group.

Assessment: The students are expected to produce programs that are easily read and understood. Each module must be documented using comments to explain code and data

definition statements, and adopting a consistent programming style that enhances readability. This was graded using a scale of 0-2.

Table 45 Instrument for assessing solution readability

The Product - Solution Quality: Readability		
Outcome	**Indicator**	**Scoring Scale**
Clear and understandable solution is produced.	Program includes commented code, meaningful identifiers, indentation to clarify logical structure, and user instructions.	2
Minimally documented solution is produced.	Program lacks clear documentation and/or user instructions.	1
Unacceptable solution quality is produced.	Program is totally incoherent.	0

6.4.2.2 Solution Correctness

The merit and validity of any solution to a problem hinges on whether the correct answer has been found - in other words, the solution must work. It is possible that mistakes can be made at different stages of the problem solving and program development process. The resulting product is either (1) a correct solution, (2) a partial solution, or (3) an error-ridden solution. The research question investigated covers the correctness of solution specification, code and results.

Research question # 4: Correctness of solution

> *Will students in the experimental group produce more accurate solution specifications, program code and results consistent with problem requirements compared to students in the control group?*

Measures: Overall solution correctness. Solution specifications, program code and execution results of students in experimental and control groups for programming assignments, quizzes, and exams were judged by course TAs. *Table* 46 shows the instrument for assessing solution correctness.

Impact: Program testing and debugging. The answer to this research question will show whether problem solving skills developed by students in the experimental group will lead to producing more accurate solution specifications and code consistent with the problem requirements and, therefore, correct results more often than students in the control group.

Assessment: Using problem requirements the students are expected to produce solution specifications, program code and correct results that satisfy the actual problem being solved. In addition to the test scenario developed to test program reliability, each module's specification and code must be carefully examined to ensure program and result correctness consistent with problem requirements. This was graded using a scale of 0-2.

Table 46 Instrument for assessing solution correctness

The Product - Solution Correctness		
Outcome	**Indicator**	**Scoring Scale**
Appropriate solution is produced.	Correct solution specifications, program code and results consistent with problem requirements.	2
Incomplete solution is produced.	Partial solution specifications/program code and/or some results.	1
No solution or totally inappropriate solution is produced.	No solution specifications/ program code, or results inconsistent with problem requirement.	0

Finally, the instruments presented in the sections on process and product are suited for evaluation of complete problem solving and program development sessions, such as programming assignments and exam problems. *Table* 47 presents an instrument designed for assessing other types of problems, such as those given in short quizzes and exercise questions. This instrument uses a scale of 0-4.

Table 47 Instrument for assessing short quizzes and exercise questions

Quiz/Exercise Problems		
Outcome	**Indicator**	**Scoring Scale**
Excellent solution indicating full understanding of the problem, its requirements, and its constraints.	Answer is clearly and correctly stated.	4
Reasonable solution indicating adequate understanding of the problem, its requirements, and its constraints.	Answer is correct, with minor omissions or inaccuracies.	3
Incomplete solution indicating some understanding of the problem, its requirements, and its constraints, but not enough to completely solve the problem.	Answer is partially stated and/or some evidence of correct result is shown or invalid assumptions are made.	2
Inappropriate solution indicating complete misunderstanding of the problem, its requirements, and its constraints.	Answer does not make any sense.	1
Lack of solution indicating complete misunderstanding of the problem, its requirements, and its constraints.	No attempt at solving problem.	0

6.4.3 Subjective Measures

In addition to the performance-based measures of the process and the product of problem solving, the effect of the instructional context on students' perception, attitude and motivation was also investigated. These measures are examined from the viewpoints of both the students and the teachers. *Table* 48 describes the subjective dependent variables.

Table 48 Subjective variables

Variable	Description
Perception	Students' own feelings and assessment of the environment and methodology for learning problem solving and programming.
Attitude and Motivation	Performance may be reflected in students commitment to the course and is evidenced in their attitude toward learning, and their motivation for achievement.

As with product measures, we questioned whether using the various tools of SOLVEIT would have cognitive benefits and whether such advantages would translate into developing favorable perception, better attitude and increased motivation toward the learning of problem solving and programming specifically and the field of computer science generally. Two methods were used to evaluate these measures: (1) students' self-reporting and (2) observations and monitoring of students' behavior.

Self-reporting allows the students to participate in an on-going evaluation of their progress by providing information about their performance on homework problems, quizzes, or exams and the difficulties they may be encountering in the course. Self-reporting was restricted to selected experiences in problem solving, was not used for course grading, and only anecdotal reporting of results will be provided. The course pre/post-test questionnaires, also a self-reporting mechanism, included questions related to these measures.

Observations and monitoring of students' activities covered class attendance, completion and quality of assigned work, and interest in course topics. The following is a discussion of subjective measures and a description of research questions developed for this set of variables.

6.4.3.1 Perception

Perceptions are formed based on observations and interpretations of knowledge about particular situations or experiences (Solso, 1988) which lead to acceptances or rejections. Students' perception about the learning environment is an important satisfaction measure of their learning goals (Gagne, 1988). Perception, as well as attitude (Rokeach, 1972), is difficult to determine and its evaluation depends on the students' own recognition and communication of their beliefs and feelings. The research question investigated whether the use of SOLVEIT improved student confidence regarding learning, increased student satisfaction in their learning experience, and enhanced the relevance of learning goal.

Research question # 5: Perception of students

> *Will students in the experimental group have a more favorable perception of their learning experience compared to students in the control group?*

Measures: Students' satisfaction. Course questionnaires and students' periodic self-assessment reports on their experience with problem solving and program development were examined to uncover successes, difficulties or other relevant information.

Impact: Students' morale. The answer to this research question will show whether or not the instructional context for students in the experimental group will lead to developing

favorable perceptions of learning problem solving and programming, the methodology, or anything else about the course.

Assessment: To evaluate students' perception toward the learning experience, students were asked to reflect on specific problem solving experiences by assessing their own performance through open-ended comments immediately after completing a programming assignment or an exam problem. Also, the course questionnaires included a series of perceptual questions. *Table* 49 shows the instrument used by students' for self-assessment.

Table 49 Instrument for students' self-assessment reports

Self-assessment Report
Completing this report provides the instructor and the TA with a retrospective feedback on your success, difficulties, feelings or anything else you wish to comment regarding the problem you have just solved. This enables you to communicate your thoughts to us throughout the semester. Please be as candid and informal as you wish. Answers can be as short (or as long) as you feel is necessary (use the back side of this form or attach additional sheets). The following is intended to give you some direction to the report:
1. When I first saw the problem ... 2. I formed the solution by ... 3. This problem solving experience ... 4. Anything else?

6.4.3.2 Attitude and Motivation

The research literature contains many definitions for attitude and researchers recognize that it is not a valid scientific construct and it is difficult to determine what data to include or exclude as part of an attitude (Rokeach, 1972). Nevertheless, Rokeach offers the following definition: "An attitude is a relatively enduring organization of beliefs around an object or situation predisposing one to respond in some preferential manner."

Successful learning experiences can shape students' general attitudes toward learning and their motivation for achievement will have impact on their performance in school (Gagne, 1988). For example, the use of a specific teaching method or a tool can have cognitive benefits as well as positive effects on students' attitude (Mager, 1968; Papert, 1980). In addition to students' general attitude, the motivation to accomplish the tasks required in a course is important. The research question investigated whether differences regarding school in general and problem solving and programming in particular occur regarding students' attitude and motivation as a result of the use of SOLVEIT.

Research question # 6: Attitude and motivation of students

Will students in the experimental group exhibit better attitude and increased motivation toward learning problem solving and programming compared to students in the control group?

Measures: Students' commitment. Students' answers on course questionnaires and their course record for attendance in lecture/recitation-laboratory sessions, quality of course

work and submission of homework on time were analyzed as indicators of students' commitment to the course.

Impact: Students' performance. The answer to this research question will show whether or not the instructional context for students in the experimental group will lead to developing better attitude toward what they are learning and increased motivation regarding their responsibilities to the course requirements.

Assessment: To evaluate students' attitudes and motivation toward problem solving and programming, an observational and monitoring record of students' commitment and performance was maintained in addition to the overall course grading database. Also, the course questionnaires included a series of questions on attitude and motivation. The differences between the two groups may be observed in students' commitment to the course and their interest in the topic, even though it may be hard or unexciting to them, as well as in students' feedback on their learning experience.

CHAPTER 7

EXPERIMENTAL RESULTS AND ANALYSIS

An experiment to evaluate the effectiveness of SOLVEIT for students taking the first course on problem solving and programming was conducted over two semesters. The impact of the new methodology and tools was measured by testing a collection of hypotheses and research questions.

This chapter presents the results of the experiment conducted to test the hypotheses and to answer the research questions of the study. Data were collected from two main sources: (1) course questionnaires and (2) students' performance on course requirements. Questionnaire data were related to identifying background, experience and general information as well as information about the course, problem solving and programming. Only end-of-semester course questionnaires were administered in the Fall semester, but pre/post-course questionnaires were administered in the Spring semester. Students' performance data consisted primarily of results on five problem solving and programming projects, five quizzes, a midterm and a final examination.

A collection of statistical procedures were used to analyze the results including analysis of variance, cross-tabulations and chi square tests. The results and analysis are presented in sections 7.2 to 7.5 using a combination of tables and charts.

The results presented in this chapter are based only on first semester evaluation. Data from second semester evaluation is still being analyzed, but initial analysis show similar trends as the first semester. Some results of second semester evaluation are

included in section 7.6 of this chapter and final analysis will be reported in scholarly journals and conference proceedings.

7.1 Descriptive Statistics

The evaluation took place in the Fall 1996 and Spring 1997 semesters. The students were split into two conditions as shown in *Table* 50. The reported results are based only on the performance and response of those students who completed the experiment. There was 81 students (48 students in the control sections and 33 students in the experimental section) in the Fall semester and 105 students (30 in the control section and 75 in the experimental section). Students who did not do any of the assigned course work, whether they officially withdrew or just did not show up for class, were removed from the statistics file. Students who withdrew after the midterm exam were kept in.

Table 50 Distribution of subjects across conditions

Semester	Condition	Students
Fall 96	Control	48
	Experiment	33
Spring 97	Control	30
	Experiment	75

7.1.1 Demographic Information from Fall 1996 Semester

Individual characteristics were collected via an end-of-semester course questionnaire that was completed by 64 students (41 in the control group and 23 in the experimental group) who took the final exam. The following is a discussion of some important information.

The students answered questions regarding their background. Nearly all students (96.9%) were NJIT undergraduates. The majority of them were full-time students (92%), taking 12 to 19 credits (90%), in either their first (67%) or second-year (22%), and most were majoring in computer and information science (70%).

Most students age fell between 18 and 22 (93%), 83% of them were males and 17% females, less than half (48%) reported that English was their first language, and the overall ethnic composition was as follows: 8% African Americans, 33% Asian or Asian-Americans, 23% Hispanics, and 33% Whites.

The students also answered questions regarding their experiences with computers and programming. Most were frequent users of computers (69%), most also had some programming before this course (73%) with BASIC (23%) and Pascal (18%), or a both (16%) being the most popular languages. Students' self-assessment of programming skills was as follows: 29% for poor, 32% for average, 22% for good, 13% for excellent and 5% were not sure. As for problem solving skills, the results were as follows: 13% for poor, 29% for average, 37% for good, 19% for excellent and 3% were not sure.

Finally, most students expected to graduate with a degree in computer science (67%) and find employment within their chosen field (67%).

7.1.2 A Comparison of Students in the Control and Experimental Groups

A host of cross-tabulations and chi square tests were performed on the results of the course questionnaire items. No significant difference was found in any of the categories regarding background or experience of the students in the two groups.

7.2 Reliability and Validity of Performance Assessment Instruments

It is important that the evaluation instruments are appropriate for the specific application in order for the results to be meaningful and useful. While all measurements are subject to fluctuations that influence their reliability and validity (Rosenthal & Rosnow, 1991), these two important characteristics are essential for any type of instruments used for evaluation. Reliability refers to the consistency of results obtained using a certain method and validity refers to the appropriateness of the interpretation made of such results (Gronlund, 1985).

The results of any performance-based evaluation should be viewed as a combination of the student's ability level and the limitations of the overall method used. Ideally, the results must truly reflect the student's ability and, although sometimes difficult, attempts can be made to minimize errors (Moore, 1983). There are four main types of reliability tests that can be used to establish consistency of results. Specific types, or a combination, of reliability tests can be useful for certain situations. One in particular, internal consistency (Rosenthal & Rosnow, 1991) or inter-rater reliability, is essential when students' performance is being judged by an instrument, as was the case in this study. The reliability of scoring using the seven *performance assessment instruments* (see Tables 39 to 41 and Tables 43 to 46) developed for this evaluation was established in

two separate courses before these instruments were used in this study. Two Teaching Assistants (TAs) working in the Summer Program in Computer Science (discussed in Chapter 6) and the instructor for the summer program (also the instructor for the course used in this study) used the instruments to grade seven programming assignments and four exams. The two TAs, who had previous teaching experience, and the instructor graded all of the programming assignments and exams separately. The instruments were revised three times based on feedback and consensus among the TAs and the instructor. Inter-rater reliability tests were performed on the grading of the last programming assignment and exam. The correlation coefficient for the three graders were calculated to find the degree of grading agreement. *Table* 51 shows the reliability coefficients for the three graders ranging from 0.82 to 0.95.

Table 51 Inter-rater reliability for the three graders

	TA 1	**TA 2**	**Instructor**
TA 1	1.00	0.82	0.92
TA 2	0.82	1.00	0.95
Instructor	0.92	0.95	1.00

Validity is an indicator of an instrument's quality. There are three types of validity tests that can be used to establish whether an instrument measures what it says it measures (Moore, 1983). As with reliability, specific types or a combination of validity

tests can be useful for certain situations. Content validity is appropriate in this case. Unlike reliability, no numerical coefficient is obtained with content validity (Moore, 1983). To assure content validity, Rosenthal and Rosnow (1991) suggest that when creating an assessment instrument, a list of skills or material that the students should master must be made. This list should be used to create any required instruments. Moore (1983) recommends that individuals with expertise in the field should examine the instrument and analyze it to determine what it measures. Both techniques were used to develop the *seven performance assessment instruments*. The instructor who teaches all on-campus sections of the first course and the instructor for the distance learning sections collaborated on the development and the revision of these instruments. Other methods were also used. *How to Evaluate Progress in Problem Solving* by Charles, Lester, and O'Daffer (1987) contains instruments already tested and used for measuring problem solving progress. Some of these instruments and techniques were adapted for this evaluation. The course questionnaires were used primarily to gather information, but were not used for grading.

7.3 Test of Process Hypotheses

Three hypotheses were developed for "process measures" which focused on students' approach to problem solving in formulating the problem and in planning and designing the solution. Students' performance was judged based on five quizzes, five programming assignments, a midterm exam and a final exam.

Programming assignments were graded out of a total of twenty points. Twelve points were assigned for the first three stages of the process divided equally among

problem formulation, solution planning and solution design. The remaining eight were split equally among reliability, efficiency, readability and correctness of the solution. Quizzes focused mostly on the choice of control and data structures and were graded out of a total of four points. The common midterm and final exams consisted of three sections: (1) the first section was similar in style and grading system to the quizzes, contained four problems and was worth 40 % of the total grade; (2) the second section was similar in style and grading system to the programming assignments, contained one large problem and was worth 40 % of the total grade; (3) the third section covered general computer science topics and was worth the remaining 20 % of the total grade. Sample copies of assignments, quizzes, midterm and final exams are included in Appendix 3.

To test the *process* hypotheses, a spectrum of analysis of variance (ANOVA) and means comparison for students' performance on programming assignments, quizzes and exams were performed. Results are reported below.

7.3.1 Results from Fall 1996 Semester

The results from the Fall semester were interesting. The students performed equally in a unilateral analysis of programming assignments and quizzes. But the experimental group performed much better on some problems of the midterm exam; in particular, those problems in section one which focused on the program development skills related to the choice of solutions' control and data structures.

Significantly better results were attained in the final exam performance. The final exam problems in section one, focusing on the choice of solutions' control and data structures, and the problem in section two that dealt with formulation, planning and

design skills resulted in dramatic significant differences between the experimental and control groups. The experimental group performed extremely well.

We tabulate below a selection of the ANOVA and means results showing the significance levels for problem 2.1 of the final exam, designed specifically to test students' comprehensive skills in the process of problem solving and program development.

7.3.1.1 Analysis of Problem Formulation Hypothesis

Hypothesis # 1: *Students in the experimental group will:*

1.1 show clearer understanding of problem, and will

1.2 identify problem's facts better than students in the control group.

The ANOVA performed on students' *problem formulation* grades reveals a significant difference (> 99.9% level of confidence) between the two groups, as shown in *Table* 52. Based on these results, the hypothesis is **supported.**

Table 52 ANOVA and means report for problem formulation

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F2F	Between Groups	43.032	1	43.032	15.652	.000
	Within Groups	175.952	64	2.749		
	Total	218.985	65			

Means Report

F2F

Control	Mean	1.74
	N	42
	Std. Deviation	1.87
Experiment	Mean	3.42
	N	24
	Std. Deviation	1.18
Total	Mean	2.35
	N	66
	Std. Deviation	1.84

7.3.1.2 Analysis of Solution Planning Hypothesis

Hypothesis # 2: *Students in the experimental group will:*

2.1 engage in planning more often than students in the control group, and will

2.2 apply early decomposition of problem goals into subgoals.

The ANOVA performed on students' *solution planning* grades reveals a significant difference (99.6% level of confidence) between the two groups, as shown in *Table* 53. Based on these results, the hypothesis is **supported.**

Table 53 ANOVA and means report for solution planning

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F2P	Between Groups	18.721	1	18.721	9.056	.004
	Within Groups	132.310	64	2.067		
	Total	151.030	65			

Means Report

F2P

Control	Mean	1.48
	N	42
	Std. Deviation	1.61
Experiment	Mean	2.58
	N	24
	Std. Deviation	1.06
Total	Mean	1.88
	N	66
	Std. Deviation	1.52

7.3.1.3 Analysis of Solution Design Hypothesis

Hypothesis # 3: *Students in the experimental group will demonstrate a higher level of competence in problem decomposition, inter-module organization, refinement, and specification skills than students in the control group.*

The ANOVA performed on students' *solution design* grades reveals a significant difference (99.0% level of confidence) between the two groups, as shown in *Table* 54. Based on these results, the hypothesis is **supported**.

Table 54 ANOVA and means report for solution design

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F2D	Between Groups	15.823	1	15.823	7.036	.010
	Within Groups	143.935	64	2.249		
	Total	159.758	65			

Means Report

F2D

Control	Mean	1.69
	N	42
	Std. Deviation	1.63
Experimrnt	Mean	2.71
	N	24
	Std. Deviation	1.23
Total	Mean	2.06
	N	66
	Std. Deviation	1.57

In summary, students' performance on the final exam section that dealt with comprehensive problem solving and program development skills clearly shows a significant difference (99.8% level of confidence) between the two groups, as shown in *Table* 55, with students in the experimental group earning much higher grades.

Table 55 ANOVA and means report for overall problem solving and program development skills

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F2 Tot	Between Groups	316.680	1	316.680	10.753	.002
	Within Groups	1884.911	64	29.452		
	Total	2201.591	65			

Means Report

F2 Total

Control	Mean	8.57
	N	42
	Std. Deviation	6.21
Experiment	Mean	13.13
	N	24
	Std. Deviation	3.63
Total	Mean	10.23
	N	66
	Std. Deviation	5.82

7.4 Test of Product Research Questions

Four research questions were developed for "product measures" which focused on the efficiency, reliability, readability and correctness of students' solutions. With process measures, we hypothesized that the tools of SOLVEIT will have positive effect on students' performance. With product measures, because SOLVEIT focuses explicitly on

the *process* of problem solving and not the *product*, we asked whether there will be any effect.

To test the *product* research questions, a spectrum of ANOVA and means comparison for students' performance using a combined grade for efficiency, reliability, readability and correctness on programming assignments, quizzes and exams were performed. Results are reported below.

7.4.1 Results from Fall 1996 Semester

The results from the Fall semester for research questions are similar to what was reported for the hypotheses. The students again performed equally on programming assignments and quizzes. But the experimental group performed uniformly better on all problems in section one of the midterm exam and three out of four problems in the same section of the final exam. This section was designed to evaluate the *product* of problem solving.

We tabulate below a selection of the ANOVA and means results showing the significance levels for the problems in section one of both the midterm and final exams, designed specifically to test students' problem solving and program development skills related to the choice of solution's control and data structures. The four research questions are discussed together because one combined grade was assigned for each of the midterm and final exam questions of this section.

7.4.1.1 Analysis of Product Research Questions for Midterm and Final Problems

Research question # 1: *Will students in the experimental group produce more efficient, or better suited, programs compared to students in the control group?*

Research question # 2: *Will students in the experimental group produce more complete and robust programs compared to students in the control group?*

Research question # 3: *Will students in the experimental group produce clearer and more understandable solutions compared to students in the control group?*

Research question # 4: *Will students in the experimental group produce more accurate solution specifications, program code and results consistent with problem requirements compared to students in the control group?*

The ANOVA performed on students' grades for problem 1.1 of the midterm reveals significant difference (99.5% level of confidence) between the two groups, as shown in *Table* 56.

Table 56 ANOVA and means report for midterm question 1.1

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
M1.1	Between Groups	87.552	1	87.552	8.258	.005
	Within Groups	826.948	78	10.602		
	Total	914.500	79			

Means Report

M1.1

Control	Mean	5.15
	N	48
	Std. Deviation	3.33
Experiment	Mean	7.28
	N	32
	Std. Deviation	3.14
Total	Mean	6.00
	N	80
	Std. Deviation	3.40

The ANOVA performed on students' grades for problem 1.2 of the midterm reveals significant difference (99.9% level of confidence) between the two groups, as shown in *Table* 57.

Table 57 ANOVA and means report for midterm question 1.2

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
M1.2	Between Groups	96.302	1	96.302	10.872	.001
	Within Groups	690.885	78	8.858		
	Total	787.187	79			

Means Report

M1.2

Control	Mean	3.79
	N	48
	Std. Deviation	2.37
Experiment	Mean	6.03
	N	32
	Std. Deviation	3.71
Total	Mean	4.69
	N	80
	Std. Deviation	3.16

The ANOVA performed on students' grades for problem 1.3 of the midterm reveals significant difference (99.3% level of confidence) between the two groups, as shown in *Table* 58.

Table 58 ANOVA and means report for midterm question 1.3

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
M1.3	Between Groups	68.252	1	68.252	7.778	.007
	Within Groups	684.448	78	8.775		
	Total	752.700	79			

Means Report

M1.3

Control	Mean	2.65
	N	48
	Std. Deviation	2.63
Experiment	Mean	4.53
	N	32
	Std. Deviation	3.41
Total	Mean	3.40
	N	80
	Std. Deviation	3.09

The ANOVA performed on students' grades for problem 1.4 of the midterm reveals significant difference (> 99.9% level of confidence) between the two groups, as shown in *Table* 59.

Table 59 ANOVA and means report for midterm question 1.4

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
M1.4	Between Groups	164.502	1	164.502	14.025	.000
	Within Groups	914.885	78	11.729		
	Total	1079.387	79			

Means Report

M1.4

Control	Mean	3.29
	N	48
	Std. Deviation	3.19
Experiment	Mean	6.22
	N	32
	Std. Deviation	3.75
Total	Mean	4.46
	N	80
	Std. Deviation	3.70

The ANOVA performed on students' grades for problem 1.1 of the final reveals significant difference (99.3% level of confidence) between the two groups, as shown in *Table* 60.

Table 60 ANOVA and means report for final question 1.1

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F1.1	Between Groups	104.762	1	104.762	7.779	.007
	Within Groups	861.905	64	13.467		
	Total	966.667	65			

Means Report

F1.1

Control	Mean	2.38
	N	42
	Std. Deviation	3.26
Experiment	Mean	5.00
	N	24
	Std. Deviation	4.30
Total	Mean	3.33
	N	66
	Std. Deviation	3.86

The ANOVA performed on students' grades for problem 1.2 of the final reveals significant difference (99.2% level of confidence) between the two groups, as shown in *Table* 61.

Table 61 ANOVA and means report for final question 1.2

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F1.2	Between Groups	93.870	1	93.870	7.453	.008
	Within Groups	806.073	64	12.595		
	Total	899.943	65			

Means Report

F1.2

Control	Mean	3.67
	N	42
	Std. Deviation	3.51
Experiment	Mean	6.15
	N	24
	Std. Deviation	3.61
Total	Mean	4.57
	N	66
	Std. Deviation	3.72

The ANOVA performed on students' grades for problem 1.3 of the final reveals significant difference (99.9% level of confidence) between the two groups, as shown in *Table* 62.

Table 62 ANOVA and means report for final question 1.3

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F1.3	Between Groups	98.894	1	98.894	11.420	.001
	Within Groups	554.216	64	8.660		
	Total	653.110	65			

Means Report

F1.3

Control	Mean	3.81
	N	42
	Std. Deviation	2.73
Experiment	Mean	6.35
	N	24
	Std. Deviation	3.30
Total	Mean	4.73
	N	66
	Std. Deviation	3.17

In summary, students' performance on the midterm and final exams sections that dealt with efficiency, reliability, readability and correctness of students' solutions clearly shows a significant difference (99.9% level of confidence for both midterm and final exams) between the two groups, as shown in *Tables* 63 and 64, with students in the experimental group earning much higher grades. Based on these results, the answer to these research questions is **significantly positive.**

Table 63 ANOVA and means report for section one of the midterm

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
M1Tot	Between Groups	1548.008	1	1548.008	12.980	.001
	Within Groups	9302.542	78	119.263		
	Total	10850.6	79			

Means Report

MID1Tot

Control	Mean	15.08
	N	48
	Std. Deviation	9.67
Experiment	Mean	24.06
	N	32
	Std. Deviation	12.58
Total	Mean	18.68
	N	80
	Std. Deviation	11.72

Table 64 ANOVA and means report for section one of the final

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
F 1Tot	Between Groups	1179.682	1	1179.682	11.552	.001
	Within Groups	6535.591	64	102.119		
	Total	7715.273	65			

Means Report

F1 Tot

Control	Mean	11.94
	N	42
	Std. Deviation	8.58
experiment	Mean	20.73
	N	24
	Std. Deviation	12.37
Total	Mean	15.14
	N	66
	Std. Deviation	10.89

In addition, students overall grade for the three sections of the final exam also reveals significant difference (96.8% level of confidence) between the two groups as shown in *Table* 65.

Table 65 ANOVA and means report for overall final grade

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
FTOT	Between Groups	2131.082	1	2131.082	4.815	.032
	Within Groups	28323.9	64	442.562		
	Total	30455.0	65			

Means Report

FTOT

Control	Mean	44.08
	N	42
	Std. Deviation	21.05
Experiment	Mean	55.90
	N	24
	Std. Deviation	21.02
Total	Mean	48.38
	N	66
	Std. Deviation	21.65

Finally, overall course performance of students in the Fall 1996 semester reveals considerable differences in letter grades earned by students. *Table* 66 shows the percentage of each letter grade earned by students in the two groups.

Table 66 Letter grade distribution for Fall 96 semester

LETTER GRADE Crosstabulation

			GROUP		Total
			Control	Experiment	
LETTER G	A	Count	12	13	25
		% within GROUP	25.0%	39.4%	30.9%
	B+	Count	1	3	4
		% within GROUP	2.1%	9.1%	4.9%
	B	Count	6	2	8
		% within GROUP	12.5%	6.1%	9.9%
	C+	Count	9		9
		% within GROUP	18.8%		11.1%
	C	Count	6	3	9
		% within GROUP	12.5%	9.1%	11.1%
	D	Count	4	1	5
		% within GROUP	8.3%	3.0%	6.2%
	F	Count	6	7	13
		% within GROUP	12.5%	21.2%	16.0%
	Inc	Count	1		1
		% within GROUP	2.1%		1.2%
	W	Count	3	4	7
		% within GROUP	6.3%	12.1%	8.6%
Total		Count	48	33	81
		% within GROUP	100.0%	100.0%	100.0%

7.5 Test of Subjective Research Questions

Two research questions were developed for "subjective measures" which focused on students' perception, attitude and motivation. *Process* hypotheses and *product* research

questions examined the performance-based differences between the two groups. *Subjective* differences were also examined but judged based on students' self-reporting as well as students observation and monitoring.

For students' self-reporting, we intended to examine two sources: (1) the periodic students' self-assessment reports and (2) the post-test questionnaire.

The first request for students self-reporting was to be returned with the third programming assignment. The result of this request was not encouraging. Seven students submitted the self-assessment reports, with more than half of these reports turned-in at least a week after the assignment submission. A second attempt to collect self-assessment information was made during the midterm exam. The report was to be submitted with the problem in section two that dealt with students' comprehensive skills and abilities in the problem solving and program development process. This second request was met with even less student enthusiasm than the first request. Only three students returned their reports.

As a result of the first semester experience, the instrument for self-reporting was redesigned to be brief, and a note was added encouraging the students to be informal and honest in their feedback. As with the first semester, the third assignment was the first attempt at collecting self-assessment reports in the second semester evaluation. The results were nearly identical to the first semester. Since these reports were not part of the announced course grading criteria, it was decided by the instructor and TAs that no further requests for the reports will be made.

The feedback from the post-test questionnaire was easier to obtain. All students who took the final exam also completed the questionnaire. Students' perception of their

satisfaction with the quality of course content and their perception of their own problem solving and programming skills was evaluated. Students' attitude and motivation toward learning problem solving and programming and their commitment to the course was also evaluated.

In addition to the attitude and motivation feedback obtained from the post-test questionnaire, a students' observation and monitoring record was maintained. Class attendance, quality of produced work and timely submission of homework were used as indicators of students' commitment to the course.

To test the *subjective* research questions, a spectrum of ANOVA, cross-tabulations and chi square tests for students' answers on the post-test questionnaire were performed. Anecdotal reporting is provided for the results of students observation and monitoring. Results are reported below.

7.5.1 Results from Fall 1996 Semester

The results from the Fall semester for *subjective* research questions are mixed. No significant difference is found for most of students' answers on the post-test questionnaire. The results of only two items show significant difference. The first item is question # 4 in the section on *problem solving and programming* dealing with verbalization and the second item is question # 13 in the section on *course outcomes* dealing with discussing issues related to problem solving and programming with other students outside of the class.

We tabulate below the ANOVA and means results for the two questions of the post-test questionnaire, regarding verbalization and discussion of issues related to

problem solving and programming, showing the significant difference and we also provide results from the observation and monitoring record.

7.5.1.1 Analysis of Subjective Research Questions

Research question # 5: *Will students in the experimental group have a more favorable perception of their learning experience compared to students in the control group?*

Only one question dealing with students' perception of their problem solving and program development skills shows a significant difference. The ANOVA performed on students' response for question # 4 of the post-test questionnaire in the section on *problem solving and programming* dealing with verbalization reveals significant difference (96.7% level of confidence) between the two groups, as shown in *Table* 67.

Table 67 ANOVA for question on post-test questionnaire dealing with verbalization

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
Verbalize	Between Groups	5.965	1	5.965	4.786	.033
	Within Groups	76.035	61	1.246		
	Total	82.000	62			

Research question # 6: *Will students in the experimental group exhibit better attitude and increased motivation toward learning problem solving and programming compared to students in the control group?*

Only one question dealing with students' attitude and motivation toward problem solving and program development shows a significant difference. The ANOVA performed on students' response for question # 13 of the post-test questionnaire in the section on *course*

outcomes dealing with discussing issues related to problem solving and programming with other students outside of class reveals significant difference (> 99.9% level of confidence) between the two groups, as shown in *Table* 68.

Table 68 ANOVA for question dealing with discussing problem solving and programming with students outside of class

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
PS Discuss	Between Groups	27.317	1	27.317	19.595	.000
	Within Groups	86.433	62	1.394		
	Total	113.750	63			

Other evidence of students' attitude and motivation toward learning problem solving and programming was investigated. For example to examine students' commitment to the course, we decided to use the dates on which the five unannounced quizzes were given as attendance indicators and the rate of submission of the five programming assignments as a motivation indicator. Records on late assignments were kept beginning with the third assignment. No significant difference was found in either attendance, rate or timely submission of assignments. The quality of work, however, of students in the experimental group was better. This is demonstrated by the results of the process hypotheses and product research questions.

In summary, perception, attitude and motivation of students in the experimental group appeared to be the same compared to students in the control group. Given these results, the answer to these research questions is **not significantly positive.**

7.6 Preview of Results from Spring 1997 Semester

This section provides a preview of the results from the Spring semester. Initial analysis also indicates that overall student performance in the experimental group is better than students in the control group. In this section, we present some evidence of this result. Further analysis will be completed and reported in the future.

Students' total grade for the semester in the three sections reveals significant difference (98.7% level of confidence) between the two groups as shown in *Table* 69.

Table 69 ANOVA and means report for overall total grade for Spring 97 semester

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
TOTAL GRADE	Between Groups	5055.310	1	5055.310	6.372	.013
	Within Groups	72988.0	92	793.348		
	Total	78043.3	93			

Means Report

TOTAL GRADE

1	Mean	53.75
	N	26
	Std. Deviation	31.58
2	Mean	70.15
	N	68
	Std. Deviation	26.78
Total	Mean	65.61
	N	94
	Std. Deviation	28.97

Overall course performance of students in the Spring 1997 semester reveals

considerable differences in letter grades earned by students. *Table* 70 shows the percentage of each letter grade earned by students in the two groups.

Table 70 Letter grade distribution for Spring 97 semester

LETTER GRADE Crosstabulation

			GROUP		
			Control	Experiment	Total
LETTERG	A	Count	5	20	25
		% within GROUP	16.7%	26.7%	23.8%
	B+	Count	3	8	11
		% within GROUP	10.0%	10.7%	10.5%
	B	Count	1	13	14
		% within GROUP	3.3%	17.3%	13.3%
	C+	Count	5	13	18
		% within GROUP	16.7%	17.3%	17.1%
	D	Count	6	12	18
		% within GROUP	20.0%	16.0%	17.1%
	F	Count	5	2	7
		% within GROUP	16.7%	2.7%	6.7%
	Inc	Count	1		1
		% within GROUP	3.3%		1.0%
	W	Count	4	7	11
		% within GROUP	13.3%	9.3%	10.5%
Total		Count	30	75	105
		% within GROUP	100.0%	100.0%	100.0%

7.7 Summary of Hypotheses and Research Questions Test

Based on the results and analysis from the Fall semester and the initial results from the Spring semester the process hypotheses are found to be supported, the product research

questions are found to be significantly positive and the subjective research questions appear to be not significantly positive. *Table* 71 summarizes the results of the hypotheses and research questions test.

Table 71 Summary of hypotheses and research questions test

Hypothesis (process)	**Result**
1. Students in the experimental group will: 1.1 show clearer understanding of problem, and will 1.2 identify problem's facts better than students in the control group.	Supported
2. Students in the experimental group will: 2.1 engage in planning more often than students in the control group, and will 2.2 apply early decomposition of problem goals into subgoals.	
3. Students in the experimental group will demonstrate a higher level of competence in problem decomposition, inter-module organization, refinement, and specification skills than students in the control group.	

Research Questions (product)	**Result**
1. Will students in the experimental group produce more efficient, or better suited, programs compared to students in the control group?	Significantly positive
2. Will students in the experimental group produce more complete and robust programs compared to students in the control group?	
3. Will students in the experimental group produce clearer and more understandable solutions compared to students in the control group?	
4. Will students in the experimental group produce more accurate solution specifications, program code and results consistent with problem requirements compared to students in the control group?	

Research Questions (Subjective)	Result
5. Will students in the experimental group have a more favorable perception of their learning experience compared to students in the control group?	Not significantly positive
6. Will students in the experimental group exhibit better attitude and increased motivation toward learning problem solving and programming compared to students in the control group?	

CHAPTER 8

CONCLUDING REMARKS AND FUTURE WORK

This chapter concludes with a summary of the evaluation results and their implications; proposed enhancements to SOLVEIT which would allow it to be used in subsequent courses, under different programming paradigms, and in other learning environments; a plan for long-term follow-up evaluation for later courses and additional instructors; and a summary of the research contributions.

8.1 Summary of Evaluation Results

The results of the evaluation from the Fall 1996 semester suggest that students in the experimental group acquired a higher-level of competence in both problem solving and program development skills than the control group. While the experimental group's scores on quizzes and programming assignments were statistically similar to the control group, the experimental group's midterm and final exam scores showed statistically significant improvements; indeed, some of the differences were dramatic. The initial results from the Spring 1997 semester are comparable to the Fall's, but with more significantly positive outcomes for the subjective research questions dealing with perception, attitude and motivation.

8.1.1 A Closer-Look at the Results

The significant differences between the exam results as opposed to the results on quizzes and programming assignments reflect the different characteristics of these activities and

the contexts in which they were given. The quizzes were intended to encourage students to keep up with assigned readings and provide the instructor and TAs with quick feedback on difficulties with comprehension of the material. The quizzes also tended to focus in an ad hoc manner on specific concepts, were short and administered in the class immediately following the introduction of the concept, typically before students could practice writing programs using the new ideas. Since both the experimental and control groups had the same level of exposure to the material and minimal practical experience with the concepts when the quizzes were given, one would expect the performance of the two groups on the quizzes to be comparable.

Programming assignments were more complex than the quizzes and required the understanding of several problem solving and program development concepts. They were given to provide students with problem solving situations that required problem formulation, solution planning and design skills, as well as substantial practice with the language control and data structures being studied. These assignments were solved in stages, and then submitted within a one or two week time frame, depending on their complexity. Students were allowed to discuss the problem with others, use problems solved in class as guiding examples, and seek help from the instructor, TAs, and tutors in the school's learning center. Thus, since every student tended to have access to a comparable level of assistance, it is unsurprising that their results were comparable.

The exams were different than the quizzes and programming assignments. Students solved exam problems under *uniform circumstances*. Exams were announced in advance and given at the same time to all sections. Each student had to rely solely on their own knowledge of the subject, their preparedness and experience, that is, they had to

show independence. These two factors, uniformity and independence, suggest that the statistically significant differences in the performance of the experimental and control groups on the exams was due to the independent variable; namely, their access to SOLVEIT or not.

8.1.2 Experimental Problems

The implementation as well as the results of this evaluation were successful. The only significant problem with the experiment was self-reporting, where students were asked to participate in an on-going evaluation of their progress by providing feedback on their performance on homework problems, exams, and any difficulties they might be encountering in the course. However, students' reports on what took place in class were not easily obtained. The self-assessment technique required a written response which was viewed as burdensome by the students. Since these responses were not even used for grading, the response was accordingly minimal. We refer to Charles, Lester, and O'Daffer, (1987) who describe important factors that hinder students' self-reporting that appear to apply directly in this case. For example, their work indicates that students may resent spending time on activities that are not graded and not directly related to the course work; students may also simply not remember all the important information about their experiences; and others may not possess the writing skills necessary for such a task. The last factor may be also be a relevant one in this situation. The course is normally taken in the first year at the same time as English composition, with some students taking remedial composition. Indeed, according to the results of the pre-test questionnaire, 52% of the students reported that English was their second language.

8.1.3 Effects Related to the Experimental Design

Students in all sections involved in the evaluation and the two TAs for the course were fully aware of the experiment, but not aware of any specific details of the design. Despite this, the so-called selection and originator effects are relevant.

The *selection effect* refers to a situation where students selected to participate in a special project may perform better because of the attention they receive, either because those selected feel better for receiving the attention, or because the extra attention is itself correlated with success. One could thus ask: how will the method work when it becomes the method everyone is using?

The *originator effect* refers to the situation where the developer of an idea is enthusiastic about the idea, wants to see it succeed, and may understand the experimental approach better than the standard one. One could then ask: how well will the new method work when it is used by someone else, who understands it only as well as, or perhaps less well than, the standard approach?

Both questions, regarding selection effect and originator effect, will be addressed as part of the future work planned to begin in the Fall 1997 semester.

8.2 Enhancements to SOLVEIT

Four different types of enhancements are envisioned for SOLVEIT:

1) Enhancements to improve its usability in the first course on problem solving and programming;

2) Enhancements to extend the system to address issues arising in subsequent courses on data structures and algorithms (dynamic data structures);

3) Enhancements to the extend the system's functionality to accommodate different program development paradigms;

4) Enhancement of the system to the Distance Learning environment.

8.2.1 Enhancing Functionality

The current version of SOLVEIT can benefit from several additional functions that will enhance its usefulness for students in the first course on problem solving and programming: a hypertext help system, a seamless transition into the programming environment, and a database of sample problems and solutions.

8.2.1.1 Hypertext Help System

SOLVEIT's current help facility consists of text descriptions of the various tools and their functionality. An improved help system, including a glossary of terms, is needed. Current hypertext-based Help facilities typically include links in each help document to allow students to navigate through in-depth information on a specific subject. An additional useful function would be to allow the students to add their own descriptions or clarifications to the Help database, using hypertext links.

8.2.1.2 Transition from SOLVEIT into the Programming Environment

Currently, once the student completes the problem solving stages within SOLVEIT and produces an algorithmic solution ready for translation and compilation, the language compiler is manually invoked. A seamless integration of SOLVEIT with the target programming environment would invoke the compiler from within SOLVEIT, passing

the pseudocode algorithm specification directly to the compiler. The student at that point would then work within the programming environment to generate executable code and results, and under appropriate conditions or at the student's choice, control would be returned to SOLVEIT.

8.2.1.3 Database of Sample Problems and Solutions

Another function that would be useful for both the student and the instructor is a facility to allow cataloging and presentation of sample problems and their solutions. This could be used as a guide for solving similar problems and could include a variety of problems that could be solved using SOLVEIT, ranging from the simple to the more complex.

8.2.2 Extending the Approach to Subsequent Courses

Extending the technique to other courses requires additional research. In particular, the functions and tools that need to be added to the environment must be carefully identified. For example, the second course deals with complex data structures and more elaborate algorithms, and so will require a greater emphasis on data modeling and choice of control structures. Correct specification of the new functions will require reexamination of the underlying problem solving and program development model.

For example, for programs written in the first course, the calling structure and module decomposition closely correspond with one another, while for more complicated programs, user-defined functions can be called at multiple, semantically different call locations, with the same effect but for different purposes. This raises problems both for the logic flow description and inter-module communication.

Complex data structures also introduce new difficulties. Even for simple lists, verifying correctness has an added dimension; namely, verifying that the data structure primitives do the correct thing, particularly for dynamic structures. Program testing where there are pointers is also significantly more difficult. Similarly, generating test sets for complicated control flows (sequences, nested conditionals, nested loops, recursion) is also more difficult.

8.2.3 Restructuring Functionality for Alternative Programming Paradigms

SOLVEIT was designed to be used in a top-down structured design paradigm. Using this methodology, a problem to be solved is decomposed into smaller subproblems, these subproblems are then further decomposed, and so on. A structure chart is used to represent the various functional components of the solution (modules) and their relationships. An alternative methodology is the object-oriented design paradigm. Instead of decomposing the problem into functional components, autonomous objects are identified according to abstractions in the problem domain to perform well-defined operations upon certain data (Booch, 1991). The current facilities of SOLVEIT that need to be adapted and the new ones that need to be designed to support object-oriented design will be investigated and developed.

8.2.4 Restructuring Functionality for Distance Learning

In the traditional classroom setting, the instructor has opportunities to facilitate the learning process directly and indirectly through feedback, reinforcement material, and the integration of necessary tools. Such support mechanisms do not, for the most part, exist

for the typical distant learning student, who is often working in isolation, and acquiring information from videotapes, electronic interactions, and print material. Distance Learning could be greatly enhanced by improving the range of interaction available to both teachers and students. Additional SOLVEIT facilities will be identified and developed to support students learning problem solving and programming through the distance leaning environment.

8.3 A Plan for Long-Term Evaluation

The effect of the instructional context on further learning will be investigated. The impact of SOLVEIT on students' performance when it is no longer used in subsequent courses will be examined in terms of learning outcome variables, described in *Figure* 47, that includes the acquisition, retention, and transfer of knowledge, skills and cognitive strategies required for advanced problem solving and program development beyond the first course.

8.3.1 Learning Outcome Measures

As with product and subjective variables, we ask questions whether the use of the various tools of SOLVEIT will have cognitive benefits and whether such advantages will translate into acquiring, retaining, and transferring superior skills and knowledge. We also ask whether the students will demonstrate better metacognitive strategies and more creative thinking.

Variable	Definition
Learning outcomes	knowledge acquisition and understanding knowledge retention and transfer Cognitive strategies: monitoring the problem solving process and creative thinking

Figure 47 Learning outcome variables

8.3.1.1 Knowledge Acquisition and Understanding

In addition to problem solving and program development skills, students must also acquire a thorough understanding of the theory of programming languages, including alternative programming paradigms, such as object-orientation and functional programming. Knowledge of programming and problem solving methodology, as demonstrated by the ability to understand problems, plan and design solutions, compose, comprehend, test and debug, document, and modify programs, constitute the basic skills required of programmers, and are already addressed by SOLVEIT. Understanding more complex programming languages issues would require addressing topics such as data structures and algorithms, axiomatic semantics, formal methods, and alternative programming paradigms. The appropriate research questions would measure student acquisition and understanding of such concepts, demonstrated as usual when the knowledge is applied appropriately (Moore & Newell, 1973). The time it takes to acquire and understand knowledge could also be an additional measuring factor.

The following research questions regarding knowledge acquisition and understanding suggest themselves:

1.1 *Will there be differences between the experimental group and control group in acquiring and understanding concepts related to data structures, algorithm design, axiomatic semantics, formal methods, etc.?*

1.2 *Will the method apply equally well to other programming languages and paradigms, such as object-orientation and functional programming?*

1.3 *Will there be differences between the experimental group and control group the time its takes to acquire an understanding of concepts related to data structures and algorithm design.*

8.3.1.2 Knowledge Retention and Transfer

A key goal of education is to allow students to make the transition from guided to independent learning, and be able to transfer knowledge and strategies from old to new problems (Greeno, Collins, & Resnick, 1995). The independent learner must demonstrate not only self-instruction and self-regulation of learning (as considered in the next section), but also retention and transfer of knowledge (Schoenfeld, 1992), since the very ability to make use of acquired knowledge depends on retention. Both classroom teaching and instructional tools enable students to acquire knowledge and skills, in each case assisting students in the learning process. Students learning programming, however, also need the opportunity to practice problem solving strategies and program development skills independently of the teacher and the tool in order to master such strategies and enhance their transferability (Gagne & Driscoll, 1988). Short term

evidence of knowledge and skill retention and transfer is demonstrated through quizzes, exams, and problem solving and programming assignments. However, long term retention and transfer requires evaluation of student performance in various situations, multiple courses, and over a long period of time.

Two related research questions suggest themselves regarding knowledge retention and transfer:

1.1 *Will there be any differences between students in the experimental group and students in the control group in retaining and transferring knowledge about problem solving, program development and programming language concepts?*

1.2 *How long (as students continue to take future courses) will the difference between the experimental group and the control group be seen?*

1.3 *Will the difference carry over into areas other than Computer Science courses?*

8.3.1.3 Cognitive Strategies: Monitoring the Problem Solving Process

Metacognition, or cognitive strategies, refers to techniques used to guide and monitor thinking. Metacognition guides the knowledge of one's own thought processes and the regulation of theses processes during problem solving. Students can learn more effectively when they are aware of their own thinking processes and develop the ability to monitor their understanding of the tasks they perform. Realizing there is a need to assess, and perhaps modify, the problem solving strategy requires that students develop effective skills for monitoring and evaluating their thinking and feedback on their progress. *Internal feedback* is an important indicator of the progress of the problem solving process and is triggered as a result of the problem solver's own comprehension of what has been,

is being, and remains to be done. *External feedback*, such as that from an instructional environment for problem solving, can heighten student's awareness and develop metacognitive skills for monitoring and evaluating their own thinking strategies through the problem solving process.

The following research questions suggest themselves regarding monitoring of the problem solving process:

1.1 *Will students in the experimental group demonstrate more monitoring of their thinking processes compared to students in the control group?*

1.2 *Will students in the experimental group demonstrate better abilities to evaluate their understanding of the tasks they perform, compared to students in the control group.*

8.3.1.4 Cognitive Strategies: Creative Thinking

The strategy a student uses to arrive at a solution and the solution itself are the product of creative thinking, a cognitive activity the research literature generally defines as the production of novel and useful solutions to a problem (Couger, 1995). Creative thinking is related to factors such as fluency and originality (Guilford, 1967; Mayer, 1988), where *fluency* is defined as the ability to form multiple solutions satisfying the requirements of a problem, and *originality* is defined as the ability to generate unusual solutions.

The following research questions suggest themselves regarding creative thinking:

1.1 *Will students in the experimental group exhibit more fluency in problem solving as demonstrated by their ability to form multiple solutions satisfying the requirements of a problem, compared to students in the control group?*

1.2 *Will students in the experimental group exhibit more originality in problem solving as demonstrated by an ability to generate unusual solutions, compared to the control group?*

8.4 Summary of Contributions

This dissertation has proposed a framework for an integrated problem solving and program development environment that addresses the needs of students learning programming. Several objectives have been accomplished by this research:

- The tasks required for program development were defined. These are: developing the skills for composing and comprehending programs, testing and debugging solutions, and documenting and modifying programs. Essential problem solving skills such as understanding the problem and its requirements and devising a solution, as well as a practical command of programming language constructs are also required before a program can be written and its solution tested.
- A literature review to determine the actual difficulties involved in learning the tasks of program development was performed. These difficulties are: deficiencies in problem solving strategies and tactical knowledge; ineffective pedagogy of programming instruction, and misconceptions about syntax, semantics, and pragmatics of language constructs.
- A comprehensive study of environments and tools developed to support the learning of problem solving and programming was performed. Twenty nine different systems were studied, described, and classified in four categories: programming environments,

debugging aids, intelligent tutoring systems, and intelligent programming environments.

- A careful analysis and critique of these tools was performed, which uncovered limitations that have prevented them from accomplishing their goals. These are: absence of problem solving/software engineering frameworks, overemphasis on language syntax, inadequate user interface, incomplete rules-and-errors knowledge base, simple problem domain, limited classroom evaluation, failure to integrate the tools into the curriculum, impeded creativity and development of higher order thinking skills.
- An extensive study of problem solving methodologies developed in this century was carried out and a *common model* for problem solving was produced. The tasks of program development were integrated with the common model for problem solving. Then, the cognitive activities required for problem solving and program development were identified and integrated to form a *Dual Common Model for problem Solving and Program Development.*
- The Dual Common Model was used to define the functional specifications for a problem solving and program development environment which was designed, implemented, tested, and integrated into the curriculum.
- The development of the new environment for learning problem solving and program development was followed by the planning of a cognitively oriented assessment method and the development of related instruments to evaluate the process and the products of problem solving. A detailed statistical experiment to study the effect of this environment on students' problem solving and program development skills,

including system testing by protocol analysis, and performance evaluation of students based on research hypotheses and questions was designed and implemented over two semesters.

- The results of the evaluation suggest that students in the experimental group acquired a significantly higher-level of competence in both problem solving and program development skills than the control group. Based on the analysis from the two semesters, the *process* hypotheses are found to be supported, the *product* research questions are found to be significantly positive and the *subjective* research questions appear to be not significantly positive.

APPENDIX 1

SURVEY INSTRUMENTS

CONSENT STATEMENT

Title of Research Project: An Integrated Environment for Problem Solving and Program Development

Investigator: Fadi P. Deek

I acknowledge that on ____/____/____, I was informed by Fadi P. Deek of New Jersey Institute of Technology of the research project on an "Integrated Problem Solving and Program Development Environment". This phase of the project consists of collecting information on student performance in the introductory course in computer science.

I was told with respect to my participation in this project that:

1. a. Confidentiality of replies will be fully protected.

   b. Data on individuals will be used for statistical analysis only.

   c. Quotations will not be identifiable unless the participant explicitly gives permission to quote.

   d. No risks to the students are involved.

2. The following procedures are involved:

   a. Distribution of course questionnaires and possible participation in interviews.

   b. Collection of data such as placement records, SAT scores and other standardized tests, student grades, semester GPA, and transcripts for analysis.

3. The following benefits are expected by my participation:

   a. An opportunity to experience alternative learning methods for problem solving and programming.

   b. An opportunity to contribute to the evaluation and possible enhancement of a Problem Solving and Program Development System whose goal is to improve the teaching and learning of problem solving and programming for first-year students.

I am fully aware of the nature and extent of my participation in this project, and I hereby agree, with full knowledge and awareness of all of the foregoing, to participate in the project. I further acknowledge that I have received a complete copy of this consent statement.

I also understand that I may withdraw my participation in the project at any time.

Signature of Subject	Printed Name of Subject
Address of Subject	City, State, ZipCode
Student ID #	Telephone Number

PRE-COURSE QUESTIONNAIRE

The purpose of this questionnaire is to gather information about students in the course. If you feel that any of these questions invade your privacy, you are of course free to decline to answer them.

IDENTIFYING INFORMATION

Name: ____________ ____________ Instructor: __________________
last first
Local Telephone: __________________ TA: __________________

Email: ______________ Course #: ______________ Section #: ____________
Date: ___/___/___

BACKGROUND INFORMATION:

1. I am:
   [ ] an NJIT undergraduate student
   [ ] a Rutgers (Newark) undergraduate student
   [ ] a non-matriculated student
   [ ] a graduate student (in bridge program)
   [ ] Other (please specify) _________________
2. Major: _________________

   Academic standing:
   [ ] Freshman
   [ ] Sophomore
   [ ] Junior
   [ ] Senior

   Status: [ ] Full-time taking ___ credits
   [ ] Part-time taking ___ credits

3. Age (at last birthday): ___

4. Is English your native or first language? [ ] Yes [ ] No

5. **(Optional)** Ethnic background. Please check one:

   [ ] African American
   [ ] Asian or Asian-American
   [ ] Hispanic
   [ ] White
   [ ] Other (please specify) _________________

6. Gender: [ ] Male [ ] Female

EXPERIENCE INFORMATION:

7. How frequently have you used computers in the past, for any kind of applications?

   [ ] Never
   [ ] Occasionally
   [ ] Frequently

8. How many *programming* courses have you taken previously?

   [ ] None
   [ ] One
   [ ] Two or more

   [ ] taken in High school [ ] taken in college

9. What *programming languages* do you know (either self-taught or learned in school)?

   [ ] BASIC
   [ ] Pascal
   [ ] C/C++
   [ ] Assembly
   [ ] LOGO
   [ ] Other (please specify) ________________

10. How would you describe your *programming* skills?

   [ ] Poor
   [ ] Average
   [ ] Good
   [ ] Excellent
   [ ] Not sure

11. How would you describe your *problem solving* skills?

   [ ] Poor
   [ ] Average
   [ ] Good
   [ ] Excellent
   [ ] Not sure

12. For each of the following pairs of words, please circle the response that is closest to your **current feelings about learning problem solving and programming**. For example, for the first pair of words, if you feel learning *programming* is "stimulating" and not "dull", circle "1"; "4" means that you are undecided/neutral or think they are equally likely to be stimulating or dull; "3" means that they are slightly more stimulating than dull, etc. (please circle the number from 1 - 7)

Stimulating	1	2	3	4	5	6	7	Dull
Demanding	1	2	3	4	5	6	7	Obliging
Fun	1	2	3	4	5	6	7	Dreary
Easy	1	2	3	4	5	6	7	Difficult
Threatening	1	2	3	4	5	6	7	Not threatening
Efficient	1	2	3	4	5	6	7	Inefficient
Relevant	1	2	3	4	5	6	7	Irrelevant
Frustrating	1	2	3	4	5	6	7	Not frustrating

GENERAL INFORMATION:

13. How easy/difficult do you expect this course to be? Please circle a number from 1-5:

EASY 1 2 3 4 5 DIFFICULT

14. Average number of hours per week that you expect to study for **this course**?

[ ] Less than 30 minutes
[ ] 30 minutes to an hour
[ ] 1 - 3 hours
[ ] 4 - 6 hours
[ ] 7 - 9 hours
[ ] 10 or more hours

15. Average number of hours per week that you expect to study for **all of your other courses** this semester (use a similar scale as above)?

___

16. What grade do you expect to receive in this course?

    [ ] A   [ ] B   [ ] C   [ ] D   [ ] F

17. When I graduate from college, I probably will have majored in ______________________

18. I expect to pursue a career as a ______________________

THANK YOU !!!

POST-COURSE QUESTIONNAIRE

IDENTIFYING INFORMATION

Name: ____________ ____________ Instructor: ____________________
last first
Local Telephone: ___________________ TA: ___________________

Email: ______________ Course #: ______________ Section #: _____________
Date: ___/___/___

COURSE CONTENT

Please respond to each of the following statements by circling the number (from 1-5) that best indicates your agreement or disagreement with each statement.

Strongly Disagree	Disagree	Undecided	Agree	Strongly Agree
1	2	3	4	5

1. Course content was interesting to me.
1-----------------------2---------------------3----------------------4-----------------------5

2. Course content is valuable.
1-----------------------2---------------------3----------------------4-----------------------5

3. Course goals were clear to me.
1-----------------------2---------------------3----------------------4-----------------------5

4. I thought the problems assigned were difficult.
1-----------------------2---------------------3----------------------4-----------------------5

5. The problems were interesting to me.
1-----------------------2---------------------3----------------------4-----------------------5

6. I learned a great deal from the assignments.
1-----------------------2---------------------3----------------------4-----------------------5

7. This course was a waste of time.
1-----------------------2---------------------3----------------------4-----------------------5

8. How would you rate the content of this course? (check one).
[ ] Very easy [ ] Easy [ ] Just right [ ] Difficult [ ] Very difficult

9. How would you rate this course over-all? (check one).
[ ] Excellent [ ] Very good [ ] Good [ ] Fair [ ] Poor

PROBLEM SOLVING AND PROGRAMMING

Please respond to each of the following statements by circling the number (from 1-5) that best indicates your agreement or disagreement with each statement.

Never	**Hardly ever**	**Sometimes**	**Most of the time**	**Always**
1	2	3	4	5

1---------------------------2-------------------------3---------------------------4---------------------------5

1. Before I start solving a problem, I think about what I should know or understand.
1---------------------------2-------------------------3---------------------------4---------------------------5

2. Before I start solving a problem, I think about what I already know about the problem's topic.
1---------------------------2-------------------------3---------------------------4---------------------------5

3. Before I start solving a problem, I think about whether I have seen this problem before, or one like it.
1---------------------------2-------------------------3---------------------------4---------------------------5

4. Before I start solving a problem, I talk about it to myself, or other people in my class.
1---------------------------2-------------------------3---------------------------4---------------------------5

5. Before I start solving a problem, I verify my understanding of the question before I go on.
1---------------------------2-------------------------3---------------------------4---------------------------5

6. Before I start solving a problem, I try to find all the information I need to solve the problem.
1---------------------------2-------------------------3---------------------------4---------------------------5

7. When preparing to solve a problem, I draw sketches or graphs.
1---------------------------2-------------------------3---------------------------4---------------------------5

8. When preparing to solve a problem, I organize the information, gathered from the problem statement, by their relevant categories.
1---------------------------2-------------------------3---------------------------4---------------------------5

9. When preparing to solve a problem, I think about different ways to find the solution.
1---------------------------2-------------------------3---------------------------4---------------------------5

10. When preparing to solve a problem, I think about what tasks I should do first, second, third, ...?
1------------------------2----------------------3-----------------------4------------------------5

11. When preparing to solve a problem, I break the problem into parts.
1------------------------2----------------------3-----------------------4------------------------5

12. When solving a problem, I consider different approaches.
1------------------------2----------------------3-----------------------4------------------------5

13. When solving a problem, I organize and order the tasks that need to be done.
1------------------------2----------------------3-----------------------4------------------------5

14. When solving a problem, I check to make sure that I understand the problem.
1------------------------2----------------------3-----------------------4------------------------5

15. When solving a problem, I periodically check to make sure I am on the right path to a solution.
1------------------------2----------------------3-----------------------4------------------------5

16. After solving a problem, I try to verify my solution.
1------------------------2----------------------3-----------------------4------------------------5

17. After solving a problem, I check to make sure that I responded to everything that was asked.
1------------------------2----------------------3-----------------------4------------------------5

18. After solving a problem, I think about what I have learned that will improve my problem solving skills.
1------------------------2----------------------3-----------------------4------------------------5

19. I have the skills needed to solve problems in this course.
1------------------------2----------------------3-----------------------4------------------------5

20. By taking my time to think about the problem before I write the program, I am able to develop better solutions.
1------------------------2----------------------3-----------------------4------------------------5

21. I know where to begin working on a problem.
1------------------------2----------------------3-----------------------4------------------------5

22. My analysis and design of solutions is systematic.
1------------------------2----------------------3-----------------------4------------------------5

23. I am satisfied with the quality of my solutions.
1------------------------2----------------------3-----------------------4------------------------5

OUTCOMES OF THE COURSE

C. Please respond to each of the following statements by circling the number (from 1-5) that best indicates your agreement or disagreement with each statement.

Strongly Disagree	Disagree	Undecided	Agree	Strongly Agree
1	2	3	4	5

1. I became more interested in problem solving and programming.
1------------------------2----------------------3-----------------------4------------------------5

2. I gained a good understanding of the concepts covered.
1------------------------2----------------------3-----------------------4------------------------5

3. My skills in problem solving and programming increased.
1------------------------2----------------------3-----------------------4------------------------5

4. My ability to critically analyze problems improved.
1------------------------2----------------------3-----------------------4------------------------5

5. I was motivated to do my best work in this course.
1------------------------2----------------------3-----------------------4------------------------5

6. I always completed the assigned readings.
1------------------------2----------------------3-----------------------4------------------------5

7. I was motivated to do additional readings.
1------------------------2----------------------3-----------------------4------------------------5

8. I participated actively in class discussions.
1------------------------2----------------------3-----------------------4------------------------5

9. I was stimulated to do my best effort on the assignments.
1------------------------2----------------------3-----------------------4------------------------5

10. I did not gain additional understanding of problem solving and programming by doing the assignments.
1------------------------2----------------------3-----------------------4------------------------5

11. I solved the problems just to get a grade.
1------------------------2----------------------3-----------------------4------------------------5

12. The programming assignments aided my learning.
1------------------------2----------------------3-----------------------4------------------------5

13. I frequently discuss issues related to problem solving and programming with other students outside of class.

1------------------------2----------------------3-----------------------4------------------------5

Strongly Disagree	**Disagree**	**Undecided**	**Agree**	**Strongly Agree**

1------------------------2----------------------3-----------------------4------------------------5

14. Solving the problems was a good learning experience.

1------------------------2----------------------3-----------------------4------------------------5

15. I always completed the assignments on time.

1------------------------2----------------------3-----------------------4------------------------5

16. I thought the quizzes and exams were difficult.

1------------------------2----------------------3-----------------------4------------------------5

17. I increased my confidence with computers.

1------------------------2----------------------3-----------------------4------------------------5

18. This class was a good learning experience.

1------------------------2----------------------3-----------------------4------------------------5

19. It is hard to do well in a problem solving and programming course.

1------------------------2----------------------3-----------------------4------------------------5

20. Once I master problem solving and programming, I will do better in my computer science, math, and science courses.

1------------------------2----------------------3-----------------------4------------------------5

21. I need to learn problem solving and programming well so that I can become a good computer scientist.

1------------------------2----------------------3-----------------------4------------------------5

22. My hard work in this course paid off. I expect to get a good grade.

1------------------------2----------------------3-----------------------4------------------------5

23. It is important for me to learn programming.

1------------------------2----------------------3-----------------------4------------------------5

24. I feel frustrated when solving problems and writing programs.

1------------------------2----------------------3-----------------------4------------------------5

25. If I cannot find the answer immediately I give up.

1------------------------2----------------------3-----------------------4------------------------5

26. My problem solving skills are not as good as other students.
1------------------------2----------------------3-----------------------4------------------------5

27. I made the right decision by choosing computer science as my major.
1------------------------2----------------------3-----------------------4------------------------5

28. Problem solving and programming is hard or not exciting.
1------------------------2----------------------3-----------------------4------------------------5

29. For each of the following pairs of words, please circle the response that is closest to your **current feelings about learning problem solving and programming**. For example, for the first pair of words, if you feel learning *programming* is "stimulating" and not "dull", circle "1"; "4" means that you are undecided/neutral or think they are equally likely to be stimulating or dull; "3" means that they are slightly more stimulating than dull, etc. (please circle the number from 1 - 7)

Stimulating	1	2	3	4	5	6	7	Dull
Demanding	1	2	3	4	5	6	7	Obliging
Fun	1	2	3	4	5	6	7	Dreary
Easy	1	2	3	4	5	6	7	Difficult
Threatening	1	2	3	4	5	6	7	Not threatening
Efficient	1	2	3	4	5	6	7	Inefficient
Relevant	1	2	3	4	5	6	7	Irrelevant
Frustrating	1	2	3	4	5	6	7	Not frustrating

GENERAL INFORMATION:

1. How easy/difficult did you find this course? Please circle a number from 1-5:

EASY 1 2 3 4 5 DIFFICULT

2. Average number of hours per week that you studied for **this course**?

[ ] Less than 30 minutes
[ ] 30 minutes to an hour
[ ] 1 - 3 hours
[ ] 4 - 6 hours
[ ] 7 - 9 hours

[ ] 10 or more hours

3. Average number of hours per week that studied for **all of your other courses** this semester (use a similar scale as above)?

___

4. How many credits including this course did you **complete** this semester? ___ (insert number)

5. What grade do you expect to receive in this course?

[ ] A [ ] B [ ] C [ ] D [ ] F

6. What is your major? ________________________

THANK YOU !!!

APPENDIX 2

COURSE SYLLABUS AND RELATED MATERIAL

CIS 113 INTRODUCTION TO COMPUTER SCIENCE I COURSE SYLLABUS

Fadi P. Deek
Computer and Information Science Department
Room 4406 GITC, phone number: 596-2997
e-mail: deek@cis.njit.edu
http://www.cis.njit.edu/~fadi

1. Course Description

Fundamentals of Computer Science are introduced, with emphasis on programming methodology and problem solving. Topics include basic concepts of computer systems, software engineering, algorithm design, programming languages, and data abstraction, with applications. A high level language (C++) is fully discussed and serves as the vehicle to illustrate the concepts in the course.

This class meets twice a week: for a lecture (three 40-minutes periods) given by myself, and a recitation/laboratory session (two 40-minutes periods) given by a Teaching Assistant (TA). Please consult my Web Page (URL given above) for my office hours and other important information.

Your TA is Edward Maybert. His office is located in room 2505 GITC (phone number: 642-4883, e-mail: exm7575@megahertz.njit.edu).

Your TA's office hours:

Thursday	9:15 - 9:55 a.m.	(Room 2505 GITC)
	4:00 - 5:55 p.m.	(Room 2505 GITC)

If these hours conflict with your schedule, you may talk to me and your TA after the class session or request an appointment through the department's secretary or by sending an e-mail request to me and/or to the TA.

2. Class Notes

Class notes are on reserve in the NJIT library and my be borrowed to review or copy. An electronic version can be found in my Web Page. You may download class notes to your local disk.

3. Course Requirements

5 programming assignments	25%
Midterm examination (date to be announced in class)	25%
Final Examination (date to be announced in class and in the VECTOR)	30%
5 Quizzes	10%
Class participation	10%

4. Tutoring

There is plenty of help available in this class. Your instructor and TA will answer questions related to topics covered in class, programming assignments, or any other related questions. Additionally, structured tutoring programs for students who request it or need it is also available.

The CIS department runs a tutorial center located in room 4308 GITC. Tutors will help students with troublesome program problems.

The University Learning Center located in University Hall also runs a comprehensive CIS tutorial program coordinated by me. Tutors at the center will assist the students in course and homework related problems and provide them with tutorial sessions.

5. Academic Integrity

The work you do and submit is expected to be the result of your effort ONLY. You may discuss the high level (general) solution of a problem. However, cooperation should not result in one or more students having possession of a copy of all or part of a program written by another student. The penalty for violating the university's code may include failure in the course and probation.

6. Lecture Outline and Textbook Readings

Topic 1 **The Machine: Hardware and Software**
- Text 1: Ch. 1
- Text 2: Ch. 0, Sec. 1.1-1.4, 2.1-2.4, 3.1-3.3

Topic 2 **Introduction to Problem Solving and Programming**
- Text 1: Ch. 2 and Ch. 3
- Text 2: Sec. 4.1-4.3, 5.1

Topic 3 **Modular Design and Abstraction**
- Text 1: Sec. 4.1 and 4.2
- Text 2: Sec. 5.3, 5.4

Topic 4 **Control Structures: Sequential, Selective, and Repetitive**
- Text 1: Sec. 5.1-5.4 and Sec. 6.3
- Text 2: Sec. 4.4, 5.2

Topic 5 **More on Modular Design: Module Communications**
- Text 1: Sec. 4.3-4.5

Topic 6	**More on Control Structures**		
		Text 1:	Sec. 5.5, 6.1, 6.2, 6.4, 6.6-6.7

***** MIDTERM EXAM *****

Topic 7	**Introduction to Software Engineering**		
		Text 2:	Ch. 6
Topic 8	**Data Abstraction: User Defined Types and the Class**		
		Text 1:	pp 427 and 428, Sec. 8.1-8.3
Topic 9	**More on Input and Output: Streams, Files, and Formatting**		
		Text 1:	Ch. 9
		Text 2:	Sec. 8.1 and 8.2
Topic 10	**Data Structures: Lists**		
		Text 1:	Sec. 10.1-10.3, and 10.8
		Text 2:	Sec. 7.1
Topic 11	**List Applications: Searching and Sorting**		
		Text 1:	Sec. 10.4
Topic 12	**Structures and pointers**		
		Text 1:	Sec. 7.1-7.2 and Sec. 11.1
Topic 13	**Dynamic Structures: Linked Lists**		
		Text 1:	Sec. 11.2 and 11.3
		Text 2:	Sec. 7.2

***** FINAL EXAM*****

Text 1: K.A. Lambert and D.W. Nance, "Understanding programming and problem solving with C++"
Text 2: J.G. Bookshear, "Computer Science: An Overview"

CIS 113 INTRODUCTION TO COMPUTER SCIENCE I
PROBLEM SOLVING AND PROGRAM DEVELOPMENT
DOCUMENTATION RULES

All assignments should be submitted in a format consistent with the problems presented and solved in the class lecture and recitation sessions. The following sections must be included in all of your programs:

0. The program should start with a paragraph of comments. This paragraph should always include:
   a) The name and number of the assignment.
   b) The student name, course/section and ID #.
   c) The instructor name and TA name.

1. Problem Formulation
   a) Describe the problem being solved in a few sentences.
   b) Identify the goal, givens, and unknowns.
   c) Other information as necessary.

2. Planning
   a) Describe the strategy you will use to solve the problem.
   b) Define data requirements:
      Input (describe the data that will be entered and processed by the program).
      Output (describe the expected output).
      Intermediate data.
   c) Refine goals into subgoals and identify tasks to be performed.
   d) Provide explanations of all formulae used.

3. Design
   a) Structure and data flow charts.
   b) Module specifications
   b) Algorithms.

4. Translation
   a) The problem solution is translated into C++ code.

5. Testing
   a) Provide comprehensive test run showing both input and output.

Naming of objects and functions should be meaningful. Declarations of objects that are logically related should be grouped together. All definition and declaration statements should be fully commented. Modules are to be documented in a similar way like the main program. Describe exactly what each function does, the input to the function, the logic, and the output produced. Define the type and role of each parameter

in a separate comment block at the top of each module. All input data must be checked for validity, when appropriate.

Sixty percent of the program grade is given for the problem solving stages and forty percent is for the program development stages. Submit all work. Points will be deducted for non-conforming programs.

Assignments should be submitted during recitations only. Late programs will be penalized and must be submitted to the TAs or instructor in-person. The penalty is as follows: the total grade that the assignment is marked out of 100% will be reduced by 100% for each day that it is late. For example, after one day the assignment will be marked out of 90%.

APPENDIX 3

SAMPLE ASSIGNMENTS, QUIZZES AND EXAMS

2/10/97

CIS 113 INTRODUCTION TO COMPUTER SCIENCE I

Programming Assignment # 2

NAME:____________________________ SID:______________________

Develop a solution for the following problem. Clearly document each stage of your problem solving and program development process as per class lecture and the handout on Assignment Rules.

Problem:

Hand-held calculator

Write a program to simulate a hand-held electronic calculator. (In other words, write a program that will cause your computer to behave as though it were a hand-held calculator). Your program should execute as follows.

Step 1: Display a prompt and wait for the user to enter an instruction code (a single character):

'+' for addition
'-' for subtraction
'*' for multiplication
'/' for division
'p' for power (exponentiation)
's' for square root
'q' for quit

Step 2 (if needed): Display a prompt and wait for the user to enter a type float number (which we will call the *left-operand*).
Step 3 (if needed): Display a prompt and wait for the user to enter a type float number (which we will call the *right-operand*).
Step 4: Display the accumulated result at any point during processing and repeat steps 1 through 3 (unless, of course, the instruction code 'q' was entered).

Use a separate module to prompt the user for the instruction code and to ensure that a valid code was entered. Also use a separate module for the entry of the left-operand and

the right-operand. Finally, use another module to perform the indicated operation (unless 'q' was entered). All output from the program should be displayed to the screen. As a result of testing phase, you are to submit a printout of the screen output.

Note: The implementation of the solution to this problem requires the use of sequential, selective and repetitive control structures.

Due date: Recitation period of the week of Feb. 17. Submit a disk copy and related documentation.

2/17/97

CIS 113 INTRODUCTION TO COMPUTER SCIENCE I

Quiz # 3

NAME:_____________________________ SID:_____________________

In this problem, you are asked to implement a module for the following task. Pay particular attention to your choice of ***parameters****, such as how many? what type? No input or output should be done inside the module unless you are specifically asked to do so.*

Problem

Some programming languages do not provide the exponentiation operator (sometimes indicated by **). However, for positive integer exponents, we can do exponentiation by repeated multiplication (using the * operator). Thus, 2^3 = 2 X 2 X 2 or 8; 5^5 = 5 X 5 X 5 X 5 X 5 or 3125; and so on. Implement a module that receives *x* and *n* and returns the value of x^n where the value of *x* is real and that of *n* is an integer greater than or equal to zero.

The main module should **only** include (1) the prototype, (2) the function call statement, and (3) the output statement.

CIS 113 INTRODUCTION TO COMPUTER SCIENCE I
FINAL EXAMINATION May 12, 1997

Name:______________________________ Student Id#________________

Section:_______ TA_____________

---

1. Be sure your test has:
   a. 10 pagefaces (including this one and 3 blank pages)
   b. 4 questions in Section I,
      1 question in Section II, and
      2 questions in Section III.
2. Remove the last (blank) page and use it for scratch work.
3. Do not begin until instructed to do so.
4. Do not sit next to anyone with whom you have studied.
5. Be sure your student ID number is on this page.
6. Use your time efficiently.

Section I: Short Answer, Programming Concepts

[40 Points Total]

1.1:	[10 Points]	________
1.2:	[10 Points]	________
1.3:	[10 Points]	________
1.4:	[10 Points]	________

Section II: Problem Solving and Program Development Skills

[40 Points Total]

2:	[40 Points]	________

Section III: Overview - Computer Science Concepts

[20 Points Total]

3.1:	[10 Points]	________
3.2:	[10 Points]	________

Total ________

Section I: Short Answer, Programming Concepts
[40 Points Total]

An important note. Please READ IT!!!

In this section, you are asked to implement a module for each of the following tasks. Pay particular attention to your choice of control structures, data structures, and parameters, such as how many? what type? and whether they are passed by copy (value), reference (variable), or returned by function result. No input or output should be done inside the module unless you are ***specifically*** *asked to do so.*

1.1 [10 Points] Implement a module that ***receives*** an N-element array of characters and the actual size of the array (i.e. how many valid elements are passed in the array), ***returns*** the number of capital letters within the array. The result should consist only of the number of the capital letters.

Do not use any library function calls pertaining to capital letter determination. You must code the entire solution yourself.

PS. In order to simplify your code you need to only check for the letters "A to F" inclusive.

1.2 [10 Points] One of the popular sorting methods is the *bubble sort*. It differs from the selection sort in that, instead of finding the element with the smallest value (for ascending order) and then performing an interchange, two elements are interchanged immediately upon discovering that they are out of order.

With this approach, at most **N** - 1 passes are required. During the first pass, LIST [0] and LIST [1] are compared, and if they are out of order they are interchanged; this process is repeated for elements LIST [1] and LIST [2], LIST [2] and LIST [3], and so on. This method will cause elements with the small values to "bubble up." After the first pass, the element with the largest value will be in the **N**th position. On each successive pass, the elements with the next largest value will be placed in the position **N** - 1, **N** - 2, ..., 3, 2, respectively.

After each pass through the array, a check can be made to determine whether or not any interchanges were made during the pass. If no interchanges have occurred during the last pass, then the array must be sorted and, consequently, no further passes are required.

Your task is to use the description above to develop a module for this sorting algorithm that ***receives*** an unordered N-element array of whole numbers and the actual size of the list (i.e. how many elements in the list), ***returns*** the list in ascending order.

1.3 [10 Points] Develop a module that ***accepts*** 2 numbers and an arithmetic operator (+, -, /, *) from the user. This input module ***returns*** the entered values to the calling module. The following module specifications must be used:

Get_Data: Get two numbers and an operator from user and return those values.
Input: num1, num2 - float values representing the numbers the user wants to perform operation on.
Operation - char value representing operation(+,-,*,/)
Output: Return num1, num2, operation
Logic: Read Input from keyboard, Error check those values and return values.

This module (Get_Data) uses a pre-defined function to check for validity of input. The prototype for this pre-defined module is as follows:

int ErrorCheck(float, float, char);

This function returns a -1 on an input error, a 0 otherwise. You are to use this function in your module to error check the user entered data and have the user re-enter the data if it contains an error.

1.4 [10 Points] Develop a module that ***receives*** and traverses a linked list of real numbers and ***returns*** the sum and average of all the numbers on the list. A single element of the list is defined by:

```
typedef CellType *CellPointer;
struct CellType
        {
        float number;
        CellPointer next;
        };
```

Also assume the following pointer variable declarations:

```
CellPointer head, // external pointer to the list
current;          // moving pointer
```

Remember that the last element's **next** field of the list contains **NULL.**

Section II: Problem Solving and Program Development Skills
[40 Points Total]

2. PROBLEM:

EARLY WARNING SYSTEM

Performance of freshmen students in CIS 113 is monitored in the first five weeks of the semester, and those students who are in danger of failing the class or receiving a low grade are sent a warning notice. The average of three quiz grades and two assignments are used to determine this feedback. They all are, quizzes and assignments, of equal weights (100 points each).

You are asked to design and implement a complete program that will calculate each student average, and then print out the student ID number, the average and a possible warning. Student's average grade is considered a passing one if it is a 60 point or higher. If the student's average grade is passing but with less than a 70 average, then the student's final grade may be in danger. The program should indicate that the student performance is marginal.

To store and process the data, create a list structure, where each component in the list can hold a student's ID number , assignment grades as well as quiz grades. Next, get in all students data (from the keyboard). Lastly, process each student as detailed above.

Your program must be modular, and communication should take place through parameters. Input, output, and error checking routines are be done in separate modules. Calculations should also take place within separate modules.

Show all work performed throughout the stages of the problem solving and program development process.

Section III: Overview - Computer Science Concepts
[20 Points Total]

3.1 [10 Points] When faced with a situation requiring the use of a repetition control structure, you may choose one of three constructs: the while, the do-while, and the for.

a) Summarize the distinctions among them.

b) Under which circumstances would you use any of them? Be specific.

3.2 [10 Points]
a) What is the difference between coupling and cohesion?

b) Which should be minimized and which should be maximized? and why?

REFERENCES

1. ACM Curriculum Committee on Computer Science, Curriculum 68: Recommendations for the Undergraduate Program in Computer Science, Communications of the ACM, 11 (3), pp. 151-197, March 1968.

2. ACM Curriculum Committee on Computer Science, Curriculum 78: Recommendations for the Undergraduate Program in Computer Science, Communications of the ACM, 22 (3), pp. 147-166, March 1979.

3. ACM/IEEE-CS Computing Curricula 1991, Report of the Joint Curriculum task force, ACM Press and IEEE Computer Society Press, New York, Feb. 1991.

4. Adam, A., and J.P. Laurent. "LAURA: A system to debug student programs", Artificial Intelligence, 15, pp. 75-122, 1980.

5. Anderson, J.R., The Architecture of Cognition, Cambridge, Massachusetts: Harvard University Press, 1983.

6. Anderson, J.R., and B. Reiser, "The LISP tutor", Byte, vol. 10 (4), pp. 159-175, 1985.

7. Anderson, J.R. (Ed.), Rules of the Mind, Hillsdale, New Jersey: Lawrence Erlbaum, 1993.

8. Anderson, J.R., A.T. Corbett, K.R. Koedinger, and R. Pelletier, "Cognitive tutors: lessons learned", The Journal of the Learning Sciences, 4 (2), pp. 167-207, 1995.

9. Anjaneyulu, K.S.R., "Bug analysis of Pascal programs", ACM SIGPLAN Notices, 29 (4), pp. 15-22, April 1994.

10. Barr, A., M. Beard and R.C. Atkinson, "A rationale and description of a CAI program to teach the BASIC programming language", Instructional Science, 4, pp. 1-31, 1975.

11. Barr, A., M. Beard and R.C. Atkinson, "The computer as a tutorial laboratory: the Stanford BIP project", International Journal of Man-Machine Studies, vol. 8, pp. 567-596, 1976.

12. Benbasat, I., and R.N. Taylor, "Behavioral aspects of information processing for the design of management information systems", IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-12 (4), July/August 1982.

13. Bereiter, C., and M. Scardamalia, "Cognitive coping strategies and the problem of inert knowledge", in S.S. Chipman, J.W. Segal, and R. Glaser (Eds.), Thinking and Learning Skills, Current Research and Open Questions, 2, pp. 65-80, Hillsdale, New Jersey: Lawrence Erlbaum, 1985.

14. Bertels, K., "A dynamic view on cognitive student modeling in computer programming", Journal of Artificial Intelligence in Education", 5 (1), pp. 85-105, 1994.

15. Bloom, B.S., (Ed.), Taxonomy of Educational Objectives, Handbook I: Cognitive Domain, New York, New York: McKay, 1956.

16. Bloom, B.S., "The 2 sigma problem: the search for methods of group instruction as effective as one-to-one tutoring", Educational Researcher, 13, page 3, June 1984.

17. Blum, B.I., "The life cycle-a debate over alternative models", ACM Software Engineering Notes, 7, pp. 18-20, October 1982.

18. Boehm, B.W., "Software engineering", IEEE Transactions on Computers, C-25, pp. 1226-1241, 1976.

19. Boehm, B.W., "A spiral model of software development and enhancement", IEEE Computer, 21 (5), pp. 61-72, 1988.

20. Bonar, J., and R. Cunningham, "Bridge: An intelligent tutor for thinking about programming", in J. Self (Ed.), Artificial Intelligence and Human Learning, Intelligent Computer Aided Instruction, pp. 391-409, London: Chapman and Hall, 1988.

21. Booch, G., Object Oriented Design with Applications, Redwood City, California: Benjamin/Cummings, 1991.

22. Brown, J.S., R.R Burgon and A. Bell, An Intelligent CAI System that Reasons and Understands, Cambridge, Massachusetts: Bolt Beranek and Newman, 1974.

23. Brusilovsky, P.L., "Turingal - The language for teaching the principles of programming", Proceedings of Third European Logo Conference, pp. 423-432, Parma, Italy, August 1991.

24. Brusilovsky, P.L., "Intelligent tutor, environment and manual for introductory programming", Educational and Training Technology International, 29 (1), pp. 26-34, 1992.

25. Brusilovsky, P.L., "Towards an intelligent environment for learning introductory programming", in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent environments for Learning programming, pp. 114-124, Berlin: Springer-Verlag, 1993.

26. Butler, D., and P. Winne, "Feedback and self-regulated learning: A theoretical synthesis", Review of Educational Research, 65 (3), pp. 245-281, 1995.

27. Calloni, B., and D. Bagert, "ICONIC programming in BACCII Vs textual programming: which is a better environment", in Proceedings of 25th SIGCSE Technical Symposium, ACM CSE Bulletin, 26 (1), pp. 188-192, 1994.

28. Carbonell, J.R., "AI in CAI: An artificial approach to computer-aided instruction", IEEE Transactions on Man-Machine Systems, MMS-11, pp. 190-202, 1970.

29. Carroll, J.M., and J.C Thomas, "Metaphor and the cognitive representation of computing systems", IEEE Transactions on Systems, man, and Cybernetics, SMC-12, (2), March/April 1982.

30. Charles, R., F. Lester, and P. O'Daffer, How to Evaluate Progress in Problem Solving, Reston, Virginia: National Council of Teachers of Mathematics, 1987.

31. Chestnut, H., Systems Engineering Methods, New York: Wiley, 1967.

32. Chi, M.T.H., R. Glaser, and E. Rees, "Expertise in problem solving", in R.J. Sternberg (Ed.), Advances in the Psychology of Human Intelligence, pp. 7-75, Hillsdale, New Jersey: Lawrence Erlbaum, 1982.

33. Clements, D.H., and S. Merriman, "Componential developments in LOGO programming environments", in R.E. Mayer (Ed.), Teaching and Learning Computer Programming, pp. 13-54, Hillsdale, New Jersey: Lawrence Erlbaum, 1988.

34. Corbett, A.T., and J.R. Anderson, "Student modeling in an intelligent programming tutor", in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent Environments for Learning programming, pp. 135-144, Berlin: Springer-Verlag, 1993.

35. Couger, J.D., Creative Problem Solving and Opportunity Finding, Danvers, Massachusetts: Boyd and Fraser, 1995.

36. Deek, F.P., and H. Kimmel, "Changing the students' role from passive listeners to active participants", in IEEE Proceedings of 23rd Frontiers in Education Conference, Washington, DC, pp. 321-325, 1993.

37. Deek, F.P., and H. Kimmel, "Enhancing the delivery of computer science instruction for first year engineering curriculum", in Proceedings of Fourth Conference on Engineering Education, Saint Paul, pp. 121-124, October 1995.

38. Denning, P., D. Comer, D. Gries, M. Mulder, A. Tucker, A. Turner, and P. Young, Report of the ACM Task Force on the Core of Computer Science, New York: ACM Press, 1989.

39. Descartes, R., Discourse on the Method of Rightly Conducting the Reason to Seek the Truth in the Sciences, 1637, tr. L.J. Lafleur, New York: Bobbs-Merrill, 1956.

40. Dewey, J., How We Think, Boston, Massachusetts: Heath, 1910.

41. Derry, S.J., and D.A. Murphy, "Designing systems that train learning ability: From theory to practice", Review of Educational Research, 56 (1), pp. 1-39, 1986.

42. Dijkstra, E., A Discipline of Programming. Englewood Cliffs, New Jersey: Prentice Hall, 1976.

43. Duncker, K., On Problem Solving, Psychological Monographs, 58 (5), Whole no. 270, 1945.

44. du Boulay, B., T. O'Shea, and J. Monk, "The black box inside the glass box: Presenting computing concepts to novices, International Journal of Man-Machine Studies, 14 (3), pp. 237-249, 1981.

45. Earnst, G.W., and A. Newell, GPS: A Case Study in Generality and Problem Solving. New York: Academic Press, 1969.

46. Ebrahimi, A., "Novice programmer error: language constructs and plan composition" International Journal of Human-Computer Studies, 41, pp. 457-480, 1994.

47. Eden, C., "Cognitive mapping", European Journal of Operational Research, 36, pp. 1-13, 1988.

48. Eisenstadt, M., B.A. Price, and J. Domingue, "Redressing ITS fallacies via software visualization", in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent Environments for Learning Programming, pp. 220-234, Berlin: Springer-Verlag, 1993.

49. Ennis, D., "Combining problem solving and programming instruction to increase the problem solving abilities in high school students", Journal of Research on Computing in Education, 26 (4), pp. 488-496, 1994.

50. Espinasse, B., "A cognitivist model for decision support: COGITA project, a problem formulation assistant", Decision Support Systems, 12, pp. 277-286, 1994.

51. Etter, D.M., "Engineering Problem Solving with ANSI C: Fundamental Concepts", Englewood Cliffs, New Jersey: Prentice Hall, 1995.

52. Fowler, F.J., Survey Research Methods, Newbury Park: California: Sage, 1993.

53. Forcheri, P., and M.T. Molfino, "Software tools for the learning of programming: A proposal", Computers Education, 23 (4), pp. 269-276, 1994.

54. Frechtling, J., (Ed.), User-Friendly Handbook for Project Evaluation: Science, Mathematics, Engineering and Technology Education, Arlington, Virginia: National Science Foundation, 1992.

55. Gagne, R.M., The Conditions of Learning, Fourth edition, New York: Holt, Rinehart and Winston, 1985.

56. Gagne, R.M., and M.P. Driscoll, Essentials of Learning for Instruction, Englewood Cliffs, New Jersey: Prentice Hall, 1988.

57. Gallopoulos, E., "Workshop on problem-solving environments: Findings and recommendations", ACM Computing Surveys, 27 (2), pp. 277-279, June 1995.

58. Gentner, D., and R. Landers, "Analogical reminding: a good match is hard to find", in Proceedings of International Conference on Systems, Man and Cybernetics, Tucson, Arizona, pp. 607-613, 1985.

59. Glinert, E., and S. Tanimoto, "Pict: an interactive graphical programming environment", IEEE Computer, 17 (11), pp. 7-25, 1984.

60. Grabel, D., (Ed.), Problem Solving: What Research Says to the Science Teacher, vol. 5, Washington, DC: National Science Teachers Association, 1989.

61. Grabiner, J., "Descartes and problem-solving", Mathematics Magazine, 68 (2), pp. 83-97, April 1995.

62. Graham, N., Introduction to Computer Science, St. Paul, Minnesota: West Publishing, 1985.

63. Greeno, J.G., "Natures of problem-solving abilities", in W.K. Estes (Ed.), Handbook of Learning and Cognitive Processes, 5, pp. 239-270, Hillsdale, New Jersey: Lawrence Erlbaum, 1978.

64. Greeno, J.G., A.M. Collins, and L.B. Resnick, "Cognition and learning", in D.C. Berliner and R.C. Calfee (Eds.), Handbook of Educational Psychology, pp. 15-45, Simon & Schuster Macmillan, 1996.

65. Gronlund, N.E, Measurement and Evaluation in Teaching, fifth edition, New York, New York: Macmillan Publishing Company, 1985.

66. Guimaraes, M., C. de Lucena, and M. Cavalcanti, "Experience using the ASA algorithm teaching system", ACM SIGCSE Bulletin, 26 (4), pp. 45-50, December 1994.

67. Hadamard, J., The Psychology of Invention in the Mathematical Field, Princeton, New Jersey: Princeton University Press, 1945.

68. Haga, H., and H. Kojima, "On the multimedia computer aided instruction system with an exercise facility for novice programmers", in B.Z. Barta, J. Eccleston, and R. Hambusch (Eds.), Computer Mediated Education of Information Technology Professionals and Advanced End-Users (A-35), pp. 155-163, Amsterdam: Elsevier Science North-Holland, 1993.

69. Halloun, I.A., and D. Hestenes, "Common sense concepts about motion", American Journal of Physics, 53, page 1056, 1987.

70. Hartman, H., Intelligent Tutoring, preliminary edition, Clearwater, Florida: H&H Publishing Company, 1996.

71. Hayes, J.R., "Teaching problem solving mechanism", in D.T. Tuma and F. Reif (Eds.), Problem Solving and Education: Issues in Taching and Research, pp. 141-147, Hillsdale, New Jersey: Lawrence Erlbaum, 1980.

72. Hayes, J.R., and H.A. Simon, "Understanding complex task instruction" in D. Klahr (Ed.), Cognition and Instruction, Hillsdale, New Jersey: Lawrence Erlbaum, 1976.

73. Henry, R.R., K.M. Whaley, and B. Forstall, "The University of Washington illustrating compiler", in Proceeding of ACM SIGPLAN on Programming Language Design and Implementation, White Plains, New York, pp. 223-233, June 1990.

74. Herman, J.L., L.L. .Morris, and C.T. Fitz-Gibbon, Evaluators Handbook, Newbury Park, CA: Sage, 1987.

75. Hesselberth, J., "Problem-solving in research and development", International Journal of Technology Management, 9 (2), pp. 253-260, 1994.

76. Hohmann, L., M. Guzdial, and E. Soloway, "SODA: A computer-aided design environment for the doing and learning of software Design", in Proceedings of Computer Assisted Learning 4th International Conference, Nova Scotia, Canada, pp. 307-319, June 1992.

77. Huff, A.S., (Ed.), Mapping Strategic Thought, Chichester, United Kingdom: Wiley, 1990.

78. Hoc, J.-M., and A. Nguyen-Xuan, "Language semantics, mental models, and analogy", in J.-M. Hoc, T.R.G Green, R. Samurcay, and D.J. Gilmore (Eds.), Psychology of Programming, London: Academic Press, 1990.

79. Hoc, J.-M., T.R.G. Green, R. Samurcay, and D.J. Gilmore (Eds.), Psychology of Programming, London: Academic Press, 1990.

80. Isoda, S., T. Shimomura, and Y. Ono, VIPS: a visual debugger, IEEE Software, 4 (2), pp. 8-19, 1987.

81. Johnson, D.M., The Psychology of Thought and Judgment, New York, New York: Harper, 1955.

82. Johnson, W.L., "Understanding and debugging novice programs", Artificial Intelligence, 42, pp. 51-97, 1990.

83. Johnson, W.L., and E. Soloway, "PROUST", Byte, 10 (4), pp. 179-190, 1985.

84. Johnson, W.L., and E. Soloway, "PROUST: Knowledge-based program understanding", Transactions on Software Engineering, SE-11 (3), pp. 267-275, March 1985.

85. Kingsley, H.L., and R. Garry, The Nature and Conditions of Learning, Englewood Cliffs, New Jersey: Prentice Hall, 1957.

86. Kohne, A., and G. Weber, "STRUEDI: A LISP-structure editor for novice programmers", in Proceedings of Second IFIP Conference on Human-Computer Inteaction, Sttutgart, Germany, pp. 125-129, 1987.

87. Lachman, R., J.L. Lachman, and E.C. Butterfield, Cognitive Psychology and Information Processing: An Introduction, Hillsdale, New Jersey: Lawrence Erlbaum, 1979.

88. Laubsch, J., and M. Eisenstadt, "Domain specific debugging aids for novice programmers", in Proceedings of Seventh International Joint Conference on Artificial Intelligence, Vancouver, Canada, pp. 964-969, August 1981.

89. Lauer, T.W., E. Peacock and A.C. Graesser (Eds.) Questions and Information Systems, Hillsdale, New Jersey: Lawrence Erlbaum, 1992.

90. Lemut, E., B. du Boulay, G. Dettori, (Eds.), Cognitive Models and Intelligent environments for Learning programming, Berlin: Springer-Verlag, 1993.

91. Levy, S.P., "Computer language usage in CS1: survey result", ACM SIGCSE Bulletin, 27 (3), pp. 21-26, September 1995.

92. Linn, M.C., and J. Dalbey, "Cognitive consequences of programming instruction: instruction, access, and ability", Educational Psychologist, vol. 20, pp. 191-206, 1985.

93. Lippert, R.C., "Expert systems: Tutors, tools, and tutees", Journal of Computer-Based Instruction, 16, pp. 11-19, 1989.

94. Love, A.J., (Ed.), Evaluation Methods Sourcebook, Ottawa, Canada: Canadian Evaluation Society, 1991.

95. Lyles, M.A., and I.I. Mitroff, "Organizational problem formulation: An empirical study", Administrative Science Quarterly, 25, pp. 102-119, 1980.

96. Mager, R.F., Developing Attitude Toward Learning, Belmont, CA: Fearon, 1968.

97. Marco, R.E., and M.M. Colina, "Programming languages and dynamic instructional tools: Addressing students' knowledge base", in S. Dijkstra, H.P.M. Krammer, J.J.G. van Merrienboer (Eds.), Instructional Models in Computer-Based Learning Environments. pp. 445-457, Berlin: Springer-Verlag, 1992.

98. Mayer, R.E, "The psychology of how novices learn computer programming", ACM Computing Surveys, 3 (1), pp. 121-141, March 1981.

99. Mayer, R.E., Thinking, Problem Solving, Cognition, New York: W.H. Freeman and Company, 1983.

100. Mayer, R.E., "Introduction to research on teaching and learning computer programming", in R.E. Mayer (Ed.), Teaching and Learning Computer Programming, pp. 1-12, Hillsdale, New Jersey: Lawrence Erlbaum, 1988.

101. Mayer, R.E., (Ed.), Teaching and Learning Computer Programming, Hillsdale, New Jersey: Lawrence Erlbaum, 1988.

102. McAllister, H.C., "Common sense problem solving and cognitive research", University of Hawaii at Manoa, World Wide Web Page, 1995.

103. McCalla, G., and K. Murtagh, "GENIUS: An Experiment in ignorance-based automated program advising", AISB Quarterly, pp. 13-20, Winter, 1990/91.

104. Meier, S.L., "Evaluating Problem Solving Processes", Mathematics Teacher, 85 (8), pp. 664-666, 1992.

105. Meier, S.L., R.L. Hovde, and R.L. Meier, "Problem solving: Teachers' perception, content area models, and interdisciplinary connections", Journal of School Science and Mathematics, 96 (5), pp. 230-237, 1996.

106. Miller, G., "The magical number seven, plus or minus two", Psychological Review, 63 (2), pp. 81-97, 1956.

107. Mimno, P.R., "Survey of CASE tools", in P.A. Ng and R.T. Yeh (Eds.), Modern Software Engineering, Foundations and Current Perspectives, pp. 323-350, 1990.

108. Mitroff, I.I., and M. Turoff, "Technological forecasting and assessment: science and/or mythology?", Technological Forecasting and Social Change, 5, pp. 113-134, 1973.

109. Moore, G.W., Developing and Evaluating Educational Research, Boston, Massachusetts: Little, Brown and Company, 1983.

110. Moore, J., and A. Newell, "How can MERLIN understand?", in Knowledge and Cognition L. Gregg (Ed.), Hillsdale, New Jersey: Lawrence Erlbaum, 1973.

111. Mukherjea, S., and J. Stasko, "Integrating algorithm animation capabilities within a source-level debugger", ACM Transactions on Computer-Human Interaction, 1 (3) pp. 215-244, 1994.

112. Myers, B.A., R. Chandhok, and A. Sareen, "Automatic data visualization for novice Pascal programmers", in Proceeding of IEEE Workshop on Visual Languages, Pittsburg, Pennsylvania, pp. 192-198, October 1988.

113. Navarat, P., and V. Rozinajova, "Making programming knowledge explicit", Computers Education, 21 (4), pp. 281-299, 1993.

114. Neal, L.R.,"A system for example-based learning", in Proceedings of CHI Conference on Human Factors in Computing Systems, Boston, Massachusetts, pp. 63-68, May 1989.

115. Newell, A., and H.A. Simon, Human Problem Solving, Englewood Cliffs, New Jersey: Prentice Hall, 1972.

116. Newell, A., "One final word", in D.T. Tuma and F. Reif (Eds.), Problem Solving and Education: Issues in Taching and Research, pp. 175-189, Hillsdale, New Jersey: Lawrence Erlbaum, 1980.

117. Ng, P.A., and R.T. Yeh (Eds.), Modern Software Engineering, Foundations and Current Perspectives, New York: Van Nostrand Reinhold, 1990.

118. Noon, J.P., (Ed.), Teaching CS1: What is the Best Language, Computer Science Product Companion, 3 (3), 1994.

119. Norman, D.A., "Cognitive engineering and education", in D.T. Tuma and F. Reif (Eds.), Problem Solving and Education: Issues in Teaching and Research, pp.97-107. Hillsdale, New Jersey: Lawrence Erlbaum, 1980.

120. Olsen, K.A., "The DSP system: A visual system to support teaching of programming", in Proceeding of IEEE Workshop on Visual Languages, Pittsburg, Pennsylvania, pp. 199-206, October 1988.

121. Ormerod, T., "Human cognition and programming, in J.-M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore (Eds.), Psychology of Programming, pp. 63-82. San Diego, CA: Academic Press Inc., 1990.

122. Osborn, A., Applied Imagination, New York: Scribner's Sons, 1953.

123. Page-Jones, M., The Practical Guide to Structured Systems Design, second edition, New Jersey: Yourdon Press, Prentice *Hall, 1988.*

124. Papert, S., Mindstorms: Children, Computers and Powerful Ideas, New York: Basic Books, 1980.

125. Parnes, S.J., Creative Behavior Guidebook, New York: Scribner's Sons, 1967.

126. Pea, R.D., and K. Sheingold, (Eds.), Mirrors of Minds: Patterns of Experience in Educational Computing, Norwood, New Jersey: Ablex, 1987.

127. Pennington, N., and B. Grabowski, "The tasks of programming", in J.-M. Hoc, T.R.G Green, R. Samurcay, and D.J. Gilmore (Eds.), Psychology of Programming, London: Academic Press, 1990.

128. Perkins, D.N., C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of learning in novice programmers", Journal of Educational Computing Research, 2 (1), pp. 37-56, 1986.

129. Perkins D.N., and F. Martin, "Fragile knowledge and neglected strategies in novice programmers", in E. Soloway and S. Iyengar (Eds.), Empirical Studies of Programmers, pp. 213-229, Norwood, New Jersey: Ablex, 1986.

130. Perkins, D.N., S. Schwartz, and R. Simmons, "Instructional strategies for the problems of novice programmers", in R.E. Mayer (Ed.), Teaching and Learning Computer Programming, pp. 153-178, Hillsdale, New Jersey: Lawrence Erlbaum, 1988.

131. Piaget, J., Genetic Epistemology, New York: Columbia University Press, 1970.

132. Poincare, H., The Foundations of Science, New York: Science Press, 1913.

133. Polya, G., How to Solve It, Princeton, New Jersey: Princeton University Press, 1945.

134. Polya, G., Mathematical Discovery: On Understanding, Learning and Teaching problem Solving, New York: Wiley, 1962.

135. Pressman, R., Software Engineering: A Practitioner's Approach, second edition, New York: McGraw-Hill, 1987.

136. Pylyshyn, Z.W., Computation and Cognition, Cambridge, Massachusetts: MIT Press, 1984.

137. Ramadhan, H., "An intelligent discovery programming system", in Proceedings of ACM Symposium on Applied Computing: Special Track on Visuality in Computing, Kansas City, USA, 1992.

138. Ramadhan, H., "Intelligent vs. unintelligent programming systems for novices", in IEEE Proceedings of the Sixteen Annual International Computer Software and Applications, pp. 375-380, Chicago, Illinois, 1992.

139. Ramadhan, H., and B. du Boulay, "Programming environments for novices", in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent Environments for Learning Programming. pp. 125-134. Berlin: Springer-Verlag, 1993.

140. Reiser, B.J., M. Ranney, M.C. Lovett, and D.Y. Kimberg, "Facilitating students' reasoning with casual explanations and visual representations, in Proceedings of the Fourth International Conference on Artificial Intelligence and Education, pp. 228-235, 1989.

141. Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views", IEEE Transactions on Software Engineering, SE-11 (3), pp. 276-285, March 1985.

142. Resnick, L.B., Education and Learning to Think, Washington, DC: National Academy Press, 1987.

143. Robillard, P.N., "Schematic pseudocode for program constructs and its computer automation by SCHEMACODE", Communication of the ACM, 29 (11), pp. 1072-1089, 1986.

144. Rogalski, J., and R. Samurcay, "Acquisition of programming knowledge and skills", Psychology of Programming, in J.-M. Hoc, T.R.G. Green, R. Samurcay, D. Gilmore (Eds.), pp. 157-174, London: Academic Press, 1990.

145. Rogalski, J., and R. Samurcay, "Task analysis and cognitive model as a framework to analyze environments for learning programming", in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent environments for Learning programming. pp. 6-19, Berlin: Springer-Verlag, 1993.

146. Rokeach, M., Beliefs, Attitudes, and Values, London: Jossey-Bass Publishers, 1972.

147. Rosen, K.H., Discrete Mathematics and its Applications, third edition, New York: McGraw-Hill, 1995.

148. Rosenthal, R. and R. Rosnow, Essentials of Behavioral Research: Methods and Data Analysis, second edition, New York: McGraw-Hill, 1991

149. Rubinstein, M., Patterns of Problem Solving, Englewood Cliffs: New Jersey, Prentice Hall, 1975.

150. Sage, M.P., and J.D. Palmer, Software Systems Engineering, New York: Wiley, 1990.

151. Schach, S.R., Software Engineering, second edition, Illinois: Asken Associates and Irwin, 1993.

152. Schoenfeld, A.H., "Explicit heuristic training as a variable in problem solving performance", Journal for Research in Mathematics Education, 10, pp. 173-187, May 1979.

153. Schoenfeld, A.H., Mathematical Problem Solving, Orlando, Florida: academic Press, 1985.

154. Schoenfeld, A.H., "Learning to think mathematically: Problem solving, metacognition, and sence making in mathematics, in D. Grouws (Ed.), Handbook for Research on Mathematics Teaching and Learning, New York: Macmillan, 1992.

155. Scholtz, J., and S. Wiedenbeck, "The role of planning in learning a new programming language", International Journal of Man-Machine Studies, 37, 191-214, 1992.

156. Scholtz, J., and S. Wiedenbeck, "An analysis of novice programmers learning a second language", in Proceedings of the Fifth Workshop on Empirical Studies of Programmers, pp. 187-205, Palo Alto, CA, 1993.

157. Shackelford, R., and A. Badre, "Why can't smart students solve simple programming problems?", International Journal of Man-Machine Studies, vol. 38, pp. 985-997, 1993.

158. Shih, W., and S.M. Alessi, "Mental models and transfer of learning in computer programming", Journal of Research on Computing in Education, 26 (2), pp. 154-175, Winter 1993-1994.

159. Shimomura, T., and S. Isoda, "Linked-list visualization for debugging, IEEE Software, 8 (3), 44-51, 1991.

160. Shneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Boston, Massachusetts: Little, Brown and Company, 1980.

161. Simon, H.A., The New Science of Management, New York, New York: Harper and Row, 1960.

162. Simon, H.A., "Information-processing theory of human problem solving", in W.K. Estes (Ed.), Handbook of Learning and Cognitive Processes, Hillsdale, New Jersey: Lawrence Erlbaum, 1978.

163. Simon, H.A., The Sciences of the Artificial, Cambridge, Massachusetts: MIT Press, 1969.

164. Simon, H.A., "Problem solving and education", in D.T. Tuma and F. Reif (Eds.), Problem Solving and Education: Issues in Teaching and Research, pp. 81-96. Hillsdale, New Jersey: Lawrence Erlbaum, 1980.

165. Skinner, B.F., "An operant analysis of problem solving", in B.F. Skinner (Ed.), Problem Solving: Research, Method, and Theory, Benjamin Kleinmuntz, New York: John Wiley and Sons, 1966.

166. Sleeman, D.H., and J.S. Brown, (Eds.), Intelligent Tutoring Systems, London: Academic Press, 1982.

167. Smith, G.F., "Defining real world problems: A conceptual language, IEEE Transactions on Systems, Man, and Cybernetics, 23 (5), pp. 1220-1234, 1993.

168. Snow, R.E., and J. Swanson, "Instructional psychology: Aptitude, adaptation, and assessment", Annual Review of Psychology, 43, pp. 583-626, 1992.

169. Soloway, E., K. Ehrlich, J. Bonar, and J. Greenspan, "What de novices know about programming", in A.N. Badre and B. Shneiderman (Eds.), Directions in Human-Computer Interaction, New York: Ablex, 1982.

170. Soloway, E., J. Spohrer, and D. Littman, *E unum pluribus*: generating alternative design, in R.E. Mayer (Ed.), Teaching and Learning Computer Programming, pp. 137-152, Hillsdale, New Jersey: Lawrence Erlbaum, 1988.

171. Spohrer, J., and E. Soloway, "Novice mistakes: are the folk wisdom correct?", Communications of the ACM, 29, pp. 624-632, 1986.

172. Stepien, W.J., S.A. Gallagher, and D. Workman, "Problem-based learning for traditional and interdisciplinary classrooms", Journal for the Education of the Gifted, 16 (4), pp. 338-357, 1993.

173. Sternberg, R.J., Beyond IQ: A Triarchic Theory of Human Intelligence, Cambridge, Massachusetts: Cambridge University Press, 1985.

174. Sumiga, J., "Programming and design, in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent environments for Learning programming, pp. 59-70, Berlin: Springer-Verlag, 1993.

175. Szetela, W., "The problem of evaluation in problem solving: Can we find solutions", Arithmetic Teacher, 35, pp. 36-41, November 1987.

176. Tremblay, J.P., and R.B Bunt, Introduction to Computer Science: An Algorithmic Approach, second edition, New York: McGraw-Hill, 1989.

177. Tucker, A.B., "New directions in the introductory computer science curriculum", in Proceedings of 25th SIGCSE Technical Symposium, ACM CSE Bulletin, 26 (1), pp. 11-15, 1994.

178. Turoff, M., and R. Hiltz, On the Design and Evaluation of Interactive Information Systems, in progress, 1997.

179. Tversky, A., and D. Kahneman, The framing of decisions and the psychology of choice, Science, 211, pp. 453-458, 1981.

180. Ueno, H., "Integrated intelligent programming environment for learning programming, IEICE Transactions on Information and Systems, E77-D (1), pp. 68-79, 1994.

181. Verdejo, M.F., and I. Fernandez, "Methodology and design issues in Capra: an environment for learning program construction, in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent environments for Learning programming, pp. 156-171, Berlin: Springer-Verlag, 1993.

182. Volkema, R.J., "Problem formulation in planning and design, Management Science, vol. 29, pp. 639-652, 1983.

183. Volkema, R.J., "Problem complexity and the formulation process in planning and design, Behavioral Science, 33, pp. 292-300, 1988.

184. Wales, C.E., A.H. Nardi, and R.A. Stager, Thinking Skills: Making a Choice, Center for Guided Design, West Virginia University, Morgantown, 1987.

185. Walker, H.M., The Limits of Computing, Boston, Massachusetts: Jones and Bartlett Publishers, 1994.

186. Wallas, G., The Art of Thought, New York: Harcourt Brace Jovanovich, 1926.

187. Weber, G., "Cognitive diagnosis and episodic modeling in an intelligent LISP-tutor, Proceedings of Intelligent Tutoring Systems-88, pp. 207-214, Montreal, Canada, 1988.

188. Weber, G., "Analogies in an intelligent programming environment for learning LISP", in E. Lemut, B. du Boulay, G. Dettori (Eds.), Cognitive Models and Intelligent environments for Learning Programming, pp. 210-219, Berlin: Springer-Verlag, 1993.

189. Weidenbeck, S., V. Fix, and J. Scholtz, "Characteristics of the mental representations of novice and expert programmers: an empirical study", International Journal of Man-Machine Studies, 39, pp. 793-812, 1993.

190. Weinberg, G.M., The Psychology of Computer Programming, New York: Van Nostrand Reinhold, 1971.

191. Wenger, E., Artificial Intelligence and Tutoring Systems, Los Altos, CA: Morgan Kaufmann Publishing, 1987.

192. Wetzel, G.F., and W.G. Bulgren, The Algorithmic Process: An Introduction to problem Solving, Chicago: The SRA Computer Science Series, 1985.

193. Whimbey, A., and J. Lochhead, Problem Solving and Comprehension: A Short Course in Analytical Reasoning, Philadelphia, Pennsylvania: The Franklin Institute Press, 1980.

194. Whimbey, A., "Think Aloud Pair Solving-TAPS: the key to higher order thinking in precise processing, Educational Leadership, 42 (1), pp. 67-70, 1987.

195. Wickelgren, W.A., How to Solve Problems, San Francisco: W.H. Freeman and Company, 1974.

196. Wirth, N., "Program development by stepwise refinement, Communications of the ACM, 14 (4), pp. 221-227, 1971.

197. Wirth, N., Algorithms + Data Structures = Programs, Englewood Cliffs, New Jersey: Prentice Hall 1975.

198. Yeh, R.T., "An alternative paradigm for software evolution" in P.A. Ng and R.T. Yeh (Eds.), Modern Software Engineering, Foundations and Current Perspectives, pp. 7-22, New York: Van Nostrand Reinhold, 1990.

199. Zelkowitz, M.V., B. Kowalchack, D. Itkin, and L. Herman, "A SUPPORT tool for teaching computer programming", in R. Fairley and P. Freeman (Eds.), Issues in Software Engineering Education, pp. 139-167, Berlin: Springer-Verlag, 1989.