AN INVESTIGATION OF STUDENTS' KNOWLEDGE, SKILLS AND STRATEGIES DURING PROBLEM SOLVING IN OBJECT-ORIENTED PROGRAMMING

by

Hester Maria Havenga

AN INVESTIGATION OF STUDENTS' KNOWLEDGE, SKILLS AND STRATEGIES DURING PROBLEM SOLVING IN OBJECT-ORIENTED PROGRAMMING

by

HESTER MARIA HAVENGA

submitted in accordance with the requirements
for the degree of

DOCTOR OF PHILOSOPHY IN MATHEMATICS, SCIENCE AND TECHNOLOGY EDUCATION

in the subject

TECHNOLOGY EDUCATION

at the

UNIVERSITY OF SOUTH AFRICA

PROMOTER: PROF E MENTZ

JOINT PROMOTER: PROF MR DE VILLIERS

JUNE 2008

Abstract

The object-oriented paradigm is widely advocated and has been used in South African universities since the late 1990s. Object-oriented computer programming is based on the object-oriented paradigm where objects are the building blocks that combine data and methods in the same entity.

Students' performance in object-oriented programming (OOP) is a matter of concern. In many cases they lack the ability to apply various supportive techniques in the process of programming. Efficient knowledge, skills and strategies are required during problem solving to enhance the programming process. It is often assumed that students implicitly and independently master these high-level knowledge, skills and strategies, and that teaching should focus on programming content and coding structures only. However, to be successful in the complex domain of OOP, explicit learning of both programming and supportive cognitive techniques is required.

The objective of this study was to identify cognitive, metacognitive and problem-solving knowledge, skills and strategies used by successful and unsuccessful programmers in OOP. These activities were identified and evaluated in an empirical research study. A mixed research design was used, where both qualitative and quantitative methods were applied to analyse participants' data. As a qualitative research practice, grounded theory was applied to guide the systematic collection of data and to generate theory.

The findings suggest that successful programmers applied significantly more cognitive-, metacognitive- and problem-solving knowledge, skills and strategies, also using a greater variety, than the unsuccessful programmers. Since programming is complex, we propose a learning repertoire based on the approaches of successful programmers, to serve as an integrated framework to support novices in learning OOP. Various techniques should be used during problem solving and programming to meaningfully construct, explicitly reflect on, and critically select appropriate knowledge, skills and strategies so as to better understand, design, code and test programs. Some examples of teaching practices are also outlined as application of the findings of the study.

Key terms:

Cognition; Constructivism; Grounded theory; Knowledge; Metacognition; Object-oriented programming; Problem solving; Qualitative methods; Quantitative methods; Skill; Strategy; Successful programmer; Unsuccessful programmer.

Acknowledgements

My sincerest thanks and appreciation go to:

- My promoter, Prof Elsa Mentz, for her guidance in leading me in this study. Her valuable insight, detailed feedback and support in the research process enabled me to complete this study;
- My joint promoter, Prof Ruth de Villiers, for her leadership, insight, rigorous comments and for teaching me a research approach;
- The following individuals for their support in one or another way: Prof P Engelbrecht, Prof HJ Steyn, Prof FJ Potgieter, Prof JL de K Monteith, Prof BW Richter, Prof H Kruger, Prof G Drevin, Prof B Smit, Dr G Koekemoer, Dr R Goede, Dr S Ellis, Dr G Reitsma, Ms J Viljoen, Ms L le Roux, Ms C Van Rensburg, Ms C Moraal;
- My dad, Paul and my mum, Lina for their support and motivation;
- My husband Kobus and daughters Roelien and Ane, and my son Johan, for their understanding when I was not always available to them;
- Soli Deo gloria.

Table of contents

Title Page ... ii
Abstract ... iii
Acknowledgements ... iv
Table of contents ... v
List of figures ... x
List of tables ... xi
List of program segments ... xii
Appendices ... xii
CD ... xii
Glossary of terminology ... xiii

1. Theoretical background and real-world problem statement

1.1 Introduction ... 1
1.2 Background ... 2
1.3 Problem statement, research question and subquestions ... 3
1.4 Research objectives ... 4
1.5 Delineation and limitations ... 5
1.6 Research framework and methodology ... 5
1.7 Data to be collected and research instruments ... 6
1.8 Significance of the study ... 7
1.9 Brief chapter overviews ... 7
1.10 Conclusion ... 9

2. Research design and methodology

2.1 Introduction ... 10
2.2 Epistemological paradigm, research design and methodology ... 12
2.3 The interpretivist paradigm ... 13
2.4 Research practice – grounded theory ... 16
  2.4.1 Overview ... 16
  2.4.2 The process of generating a grounded theory ... 16
2.5 Research considerations with regard to this study ... 19
  2.5.1 Relevance of interpretivism ... 19
  2.5.2 Relevance of grounded theory ... 21
  2.5.3 Reliability, validity and reflexivity ... 22
2.6 The positivist paradigm ... 24
  2.6.1 Relevance of the positivist paradigm ... 24
  2.6.2 Reliability and validity ... 24
2.7 Research methods: data collection techniques ... 25
  2.7.1 Research plan and participants ... 25
  2.7.2 Object-oriented computer program ... 26
  2.7.3 Written document – participants' thinking processes ... 27
  2.7.4 Questionnaire ... 28
  2.7.5 Ethical aspects ... 29

2.8 Research methods: data analysis techniques ... 29
2.8.1 Computer program analysis ... 30
2.8.2 Textual document analysis – using the support of *Atlas.ti* ... 30
2.8.3 Questionnaire data analysis ... 30
2.9 Qualitative data analysis software – *Atlas.ti* ... 31
2.9.1 Application of *Atlas.ti* ... 31
2.9.2 The harmony between grounded theory and *Atlas.ti* ... 38
2.10 Chapter conclusion ... 40

3. Cognitive, metacognitive and problem-solving knowledge and skills in object-oriented programming

3.1 Introduction ... 42
3.2 Object-oriented programming ... 44
3.2.1 The need to change to the object-oriented paradigm ... 44
3.2.2 The origin of object-oriented programming languages ... 45
3.2.3 An overview of object-oriented programming ... 45
3.2.3.1 Object ... 46
3.2.3.2 Class ... 47
3.2.3.3 Attributes and methods ... 47
3.2.3.4 Constructors and destructors ... 48
3.2.3.5 Abstractions and associations ... 48
3.2.3.6 Polymorphism and dynamic binding ... 50
3.2.3.7 Advantages and disadvantages of object-oriented programming ... 50
3.2.4 Programming notations and models ... 51
3.2.4.1 Patterns in object-oriented programming ... 52
3.2.4.2 UML – an important graphical notation ... 52
3.2.4.3 CRC cards ... 53
3.2.5 Problem and design spaces in object-oriented programming ... 53
3.3 Cognitive knowledge and skills in object-oriented programming ... 54
3.3.1 Memory, comprehension, reasoning, decision making, creative and critical thinking in object-oriented programming ... 54
3.3.1.1 Memory and cognitive load ... 54
3.3.1.2 Comprehension, reasoning, decision making, creative and critical thinking ... 57
3.3.2 Bloom's taxonomy ... 60
3.3.3 Some practical means of cognitive support ... 63
3.4 Metacognitive knowledge and skills in object-oriented programming ... 66
3.4.1 Metacognitive knowledge in general ... 66
3.4.2 Metacognitive knowledge in object-oriented programming ... 68
3.4.3 Some practical examples of metacognitive support ... 69
3.5 Problem-solving knowledge and skills in object-oriented programming ... 70
3.5.1 Factors that relate to the level of difficulty of problems ... 70
3.5.1.1 The structuredness of problems ... 70
3.5.1.2 The complexity of problems ... 71
3.5.1.3 The dynamicity of problems ... 71
3.5.1.4 The domain specificity or context of problems ... 71

3.5.2 Steps in problem solving ... 74
3.5.2.1 Problem understanding ... 74
3.5.2.2 Program designing ... 74
3.5.2.3 Program coding ... 75
3.5.2.4 Program testing ... 75
3.5.3 Level of expertise and problem solving ... 76
3.5.4 Some practical means of support during problem solving ... 78
3.6 Chapter conclusion ... 80

4. Cognitive, metacognitive and problem-solving strategies in object-oriented programming

4.1 Introduction ... 81
4.2 Strategic aspects of performance ... 82
4.3 Cognitive strategies ... 84
4.3.1 Rehearsal strategy in object-oriented programming ... 85
4.3.2 Elaboration strategy in object-oriented programming ... 86
4.3.3 The organisation-and-integration strategy in object-oriented programming ... 88
4.4 Metacognitive strategies ... 90
4.4.1 Planning strategy in object-oriented programming ... 91
4.4.2 Monitoring strategy in object-oriented programming ... 92
4.4.3 Regulation strategy in object-oriented programming ... 93
4.4.4 Reflection in object-oriented programming ... 95
4.5 Problem-solving strategies in object-oriented programming ... 97
4.5.1 Problem-solving strategies during programming ... 97
4.5.1.1 Bottom-up strategy ... 98
4.5.1.2 Top-down strategy ... 98
4.5.1.3 Integrated strategy ... 99
4.5.1.4 As-needed strategy ... 99
4.5.1.5 Trial-and-error strategy ... 99
4.6 Chapter conclusion ... 100

5. Empirical research and data analysis

5.1 Introduction ... 102
5.2 Analysis of participants' computer programs and thinking processes ... 104
5.2.1 Measurement criteria ... 105
5.2.2 Example 1: A poor program ... 110
5.2.2.1 Cognitive knowledge and skills ... 111
5.2.2.2 Metacognitive strategies ... 113
5.2.2.3 Problem-solving strategy ... 114
5.2.2.4 Application of measurement criteria in the Delphi program ... 114
5.2.3 Example 2: An excellent program ... 118
5.2.3.1 Cognitive knowledge and skills ... 119
5.2.3.2 Metacognitive strategies ... 122
5.2.3.3 Problem-solving strategy ... 124
5.2.3.4 Application of the measurement criteria in the Java program ... 125
5.2.4 Evaluation of the computer programs: Participants 31 and 32 ... 129

5.2.5 Quantitative analysis - statistical methods 132
5.2.6 Evaluation of all participants' computer programs and thinking processes 134
5.2.6.1 Cognitive knowledge and skills 134
5.2.6.2 Metacognitive strategies 135
5.2.6.3 Problem-solving strategies 136
5.2.6.4 Values allocated for object-oriented programming 136
5.2.6.5 Correlations between various constructs 140
5.2.7 Knowledge, skills and strategies used by successful programmers 141
5.2.8 Application of interpretivism and positivism in Section 5.2 146
5.3 Qualitative analysis of participants' thinking processes using *Atlas.ti* software 147
5.3.1 The coding process of two detailed examples in *Atlas.ti* 148
5.3.1.1 The coding process of Participant 29's thinking processes 149
5.3.1.2 The coding process of Participant 32's thinking processes 150
5.3.2 The organisation of codes into families and identification of themes 152
5.3.3 Theme 1: Cognitive knowledge, skills and strategies 154
5.3.4 Theme 2: Metacognitive knowledge, skills and strategies 158
5.3.5 Theme 3: Problem-solving knowledge, skills and strategies 160
5.3.6 Theme 4: Errors and problems during programming 162
5.3.7 Theme 5: Additional support during programming 163
5.3.8 Application of interpretivism and grounded theory in Section 5.3 and the generation of themes 163
5.4 Statistical analysis – questionnaire 168
5.4.1 Biographical information 168
5.4.2 Closed-ended questions 169
5.4.2.1 Cognition knowledge and skills 170
5.4.2.2 Metacognitive strategies 175
5.4.2.3 Problem-solving strategies 178
5.4.2.4 Mean values, standard deviations and correlation of various constructs 181
5.4.3 Open-ended questions 182
5.4.4 Application of positivism and interpretivism in Section 5.4 186
5.5 Triangulation between different analysis methods 187
5.6 Measures to ensure rigour and quality of data 199
5.6.1 Qualitative measures 199
5.6.2 Quantitative measures 200
5.7 Overview of the research findings 200
5.7.1 Cognitive knowledge, skills and strategies 201
5.7.2 Metacognitive knowledge, skills and strategies 202
5.7.3 Problem-solving knowledge, skills and strategies 203
5.7.4 Application of knowledge and skills in object-oriented programming 204
5.8 Chapter conclusion 204

6. Discussion and conclusion

6.1 Introduction 206
6.2 Discussion of the findings of this study 207
6.2.1 Response to Subquestions 1.1 and 2.1: Cognitive knowledge, skills and strategies 208
6.2.2 Response to Subquestions 1.2 and 2.2: Metacognitive knowledge,

skills and strategies 208
6.2.3 Response to Subquestions 1.3 and 2.3: Problem-solving knowledge, skills and strategies 209
6.2.4 Response to Subquestion 3.1: Differences between unsuccessful and successful programmers 210
6.2.5 Performance patterns of unsuccessful and successful participants 212
6.2.5.1 Imbalances between the constructs 212
6.2.5.2 Performance profile of unsuccessful and successful participants 214
6.3 A learning repertoire of knowledge, skills and strategies for object-oriented programming 217
6.3.1 Research methodology applied 217
6.3.2 A proposed learning repertoire for the effective learning of OOP 217
6.4 Application of this study to teaching and learning 221
6.5 Recommendations and future research directions 226
6.6 Chapter conclusion 226

References 228

List of figures

Figure 1.1	Structure of the thesis	8
Figure 2.1	An overview of the research design	11
Figure 2.2	An example of selected text with the associated code	21
Figure 2.3	Various tools available in *Atlas.ti*	32
Figure 2.4	Different steps in the analysis of participants' thinking processes with *Atlas.ti*	33
Figure 2.5	One hermeneutic unit with many primary documents	34
Figure 2.6	*Atlas.ti* qualitative software	34
Figure 2.7	An extract of text from a primary document in *Atlas.ti*	35
Figure 2.8	Examples of codes in *Atlas.ti*	36
Figure 2.9	Examples of quotations associated with the code 'Java:constructor'	36
Figure 2.10	An example of a memo in *Atlas.ti*	37
Figure 2.11	An example of families in *Atlas.ti*	37
Figure 2.12	An example of a network structure in *Atlas.ti*	38
Figure 3.1	Various goals, knowledge, skills, strategies and their application in an object-oriented program	43
Figure 3.2	The Vehicle class, Bus subclass and Bus-object	47
Figure 3.3	A vehicle's registration number, subdivided into chunks	64
Figure 3.4	An example of a memory diagram	64
Figure 3.5	An example of a semantic network	65
Figure 4.1	Various goals, knowledge, skills, strategies and their application in an object-oriented program	82
Figure 4.2	An example of a reminder in Delphi	86
Figure 4.3	*Watch* and *Trace* functions displaying the value of the variable 'year' currently in memory during program execution	94
Figure 5.1	Design form of the first application program of P31	117
Figure 5.2	Compilation of P31's program showing numerous errors	117
Figure 5.3	Java output from the *Date class and Test class* programs of P32	127
Figure 5.4	P29's data is assigned to the DATE_CLASS hermeneutic unit for analysis within the primary document	149
Figure 5.5	An example of the coding process and highlighted text of P29	150
Figure 5.6	P32's data is assigned to the DATE_CLASS hermeneutic unit for analysis within the primary document	151
Figure 5.7	An example of the coding process and highlighted text of P32	152
Figure 5.8	Grouping of codes into the possible coded 'families'	153
Figure 5.9	Selected codes in the problem-solving coded family	153
Figure 5.10	An integrated representation of the themes that emerged from participants' thinking processes	166
Figure 5.11	Extraction from the Cognitive knowledge, skills and strategies category	166
Figure 6.1	Possible imbalances in unsuccessful participants' thinking	213
Figure 6.2	Possible imbalances in successful participants' thinking	214
Figure 6.3	Performance profile of unsuccessful participants	215
Figure 6.4	Performance profile of successful participants	215
Figure 6.5	A learning repertoire of cognitive, metacognitive and problem-solving knowledge, skills and strategies in an OOP task	219

List of tables

Table 1.1 Research questions and subquestions 4
Table 2.1 A summary of principles in the Information System field 15
Table 2.2 The application of Klein and Myers' (1999) seven principles in this study 20
Table 2.3 The research plan in this study 26
Table 2.4 Requirements for writing the *Date class* program 27
Table 2.5 Data collection and analysis methods 29
Table 2.6 A summary of grounded theory concepts and their associated methods or tools in *Atlas.ti* 40
Table 3.1 The taxonomy of Benjamin Bloom et al. (1973) 61
Table 3.2 Analysis of cognitive skills in object-oriented programming 62
Table 3.3 Causes of errors during programming 73
Table 3.4 Categories for ensuring program correctness 75
Table 3.5 Examples of expertise during problem solving 77
Table 5.1 Measurement criteria for the analysis of Delphi and Java programs and thinking processes 106
Table 5.2 Analysis of Delphi and Java programs and thinking processes of all the participants 108
Table 5.3 Examples of P31's cognitive knowledge and skills (or the lack thereof) 111
Table 5.4 Examples of P31's metacognitive strategies (or the lack thereof) 113
Table 5.5 Examples of P31's problem-solving strategies (or the lack thereof) 114
Table 5.6 Marks allocated to P31 for OOP 118
Table 5.7 Examples of cognitive knowledge and skills used by P32 119
Table 5.8 Examples of metacognitive strategies used by P32 123
Table 5.9 The problem-solving strategy used by P32 125
Table 5.10 Marks allocated to P32 for OOP 128
Table 5.11 Summary and comparison of the marks allocated to P31 and P32 131
Table 5.12 Mean values and standard deviations for cognitive knowledge and skills 135
Table 5.13 Mean values and standard deviations for metacognitive strategies 136
Table 5.14 Frequencies for selected problem-solving strategies 136
Table 5.15 Summary statistics for OOP knowledge and skills 137
Table 5.16 Mean values and standard deviations for OOP knowledge and skills 139
Table 5.17 Correlations between cognition, metacognition and OOP knowledge and skills 140
Table 5.18 Allocated values of successful programmers 142
Table 5.19 Means, standard deviations and practical significance of unsuccessful and successful programmers 143
Table 5.20 Theme 1: Cognitive knowledge, skills and strategies – codes in *Atlas.ti* with associated quotations from participants' thinking processes 155
Table 5.21 Theme 2: Metacognitive knowledge, skills and strategies – codes in *Atlas.ti* with associated quotations from participants' thinking processes 158
Table 5.22 Theme 3: Problem-solving knowledge, skill and strategies – codes in *Atlas.ti* with associated quotations from participants' thinking processes 160
Table 5.23 Theme 4: Errors and problems during programming – codes in *Atlas.ti* with associated quotations from participants' thinking processes 162
Table 5.24 Theme 5: Additional support during programming – codes in *Atlas.ti* with associated quotations from participants' thinking processes 163
Table 5.25 Programming language currently used 169

Table 5.26 Questions about cognitive knowledge and skills 171
Table 5.27 Questions about metacognitive strategies 176
Table 5.28 Questions about problem-solving strategies 179
Table 5.29 Mean values and standard deviations for cognition, metacognition and problem-solving sections 181
Table 5.30 Correlations between constructs 182
Table 5.31 Open-ended questions in the questionnaire 182
Table 5.32 Strategies, plans or useful 'tricks' that participants used when writing a program 184
Table 5.33 Problem-solving steps used by participants in a programming task 184
Table 5.34 Supportive memory representation techniques that participants used during a programming task 185
Table 5.35 Triangulation between different analysis methods: P31's data 188
Table 5.36 Triangulation between different analysis methods: P24's data 190
Table 5.37 Allocated values of average participants 192
Table 5.38 Triangulation between different analysis methods: average participants' data 193
Table 5.39 Triangulation between different analysis methods: P29's data 195
Table 5.40 Triangulation between different analysis methods: P32's data 197
Table 6.1 Research questions and subquestions 207
Table 6.2 Facilitator practices in teaching cognitive knowledge, skills and strategies 223
Table 6.3 Facilitator practices in teaching metacognitive knowledge, skills and strategies 224
Table 6.4 Facilitator practices in teaching problem-solving knowledge, skills and strategies 225

List of program segments

Program 5.1 Delphi program segment from the *Date class* program of P31 (First attempt) 115
Program 5.2 Delphi program segment from the *Date class* program of P31 (Second attempt) 116
Program 5.3 Java program segment from the *Date class* program of P32 125
Program 5.4 Java program segment from the *Test class* program of P32 127

Appendices

Appendix A Consent form 239
Appendix B Ethical approval 240
Appendix C Programming assignment 241
Appendix D Codes in *Atlas.ti* 244
Appendix E The questionnaire and mark sheet 249
Appendix F Data of Participant 31, an unsuccessful programmer 256
Appendix G Data of Participant 32, a successful programmer 260
Appendix H Article accepted for the *South African Computer Journal* 272

CD .... inside back cover

1. A Java solution
2. A Delphi solution
3. Exported HTML file with codes, memos and families in *Atlas.ti*

Glossary of terminology

Class: a template or design pattern for an object.

Cognition: the mental processes used in the acquisition, storage, transformation and application of knowledge.

Constructivism: a theory that emphasises how people construct their own learning and understanding of the world through experiences, and by interpretation of those experiences.

Expert: a knowledgeable person with superior skills in a particular field.

Grounded theory: using guidelines for the organisation of data, theory is developed that provides relevant interpretations, applications, predictions and explanations.

Interpretivism: refers to knowledge that is intentionally obtained by means of the interpretation and the meaning of constructs through a person's lived experience.

Knowledge: information acquired through experience or education.

Metacognition: explicit knowledge of one's own cognitive strengths and weaknesses that affect memory performance.

Novice: a person who is inexperienced and new in a particular field.

Object (in the context of OOP): contains both data and operations in the same entity.

Object-oriented programming (OOP): a computer programming language based on the object-oriented approach, whereby objects have the responsibility of carrying out specific operations to solve a problem.

Positivism: focuses on science as an approach that verifies and confirms empirical observations by means of measurable ways.

Problem solving: a complex cognitive process required to find an answer.

Skill: the ability to do a particular task.

Strategy: a designed plan to achieve a purpose and to solve a problem.

Successful programmer: a person who has achieved the outcomes and has dealt efficiently with problems.

Triangulation: a method used by qualitative researchers to check and establish validity in their studies.

Unsuccessful programmer: a person who did not achieve the stated outcomes.

1 Theoretical background and real-world problem statement

1.1 Introduction

There are different approaches, called paradigms, to the way a programmer analyses, designs and implements a computer program (Ragonis & Ben-Ari, 2005:203). The object-oriented paradigm is based on a fundamentally different view from that of its predecessors (Satzinger & Ørvik, 2001:2). Under the object-oriented approach, a computer system is viewed as a collection of interacting objects with specific attributes and methods, where objects have the responsibility of carrying out specific tasks (Garrido, 2003:26-27). Object-oriented programming (OOP) is based on the object-oriented paradigm where objects are the building blocks that combine data and methods in the same entity. This paradigm should lead to easier maintenance and improved quality, productivity and flexibility (Satzinger & Ørvik, 2001:2,9).

However, the learning and teaching of OOP is multidimensional and complex (Govender & Grayson, 2006:1687). OOP is more abstract and comprehensive than its predecessors and requires more depth to grasp (Northrop, 1992:247). It involves a rich environment in which specific programming words, statements and constructs come together to be integrated in a tightly defined way to solve a problem efficiently.

In the learning of object-oriented programming, the student must know what objects are required, what behaviours they should exhibit, and what interactions occur between them (§3.2). In order to understand the learning processes entailed in OOP, it is important to investigate students' programming processes and products to determine which high-level knowledge, skills and strategies are actually used and which are optimally required. Furthermore, educators need to play supportive roles that facilitate the acquisition of appropriate activities and techniques as students learn to apply their knowledge, skills and strategies in programming.

1.2 Background

Traditional programming languages such as Turbo Pascal, use a DOS (Disk Operating System)-based operating system and many errors occurred when such programs were executed within a Windows-based environment (§3.2.1). Programming in a DOS environment was difficult without a user-friendly graphical interface, and programming problems became more complex (Yousoof, Sapiyan & Kamaluddin, 2006:259). In addition, data could easily be modified outside of its scope as global data in a program where it was uncontrolled and unpredictable (Weisfeld, 2004:7-8).

The object-oriented paradigm addresses some of these problems. OOP involves solving problems by identifying the real-world objects of a problem and creating programming objects to simulate those objects and their processes (Sebesta, 2004:92). OOP includes data abstraction that encapsulates the behaviour of data and hides access to data (Satzinger & Ørvik, 2001:44). Furthermore, inheritance and dynamic binding were added to enhance the reuse of existing software (Sebesta, 2004:458; Weisfeld, 2004:7-8) (§3.2.3). As a result of various advantages, OOP became important in many applications and in software development.

The object-oriented paradigm is widely advocated (Or-Bach & Lavy, 2004:82) and is used in international higher education institutions. It was introduced in South African universities from the late 1990s. Popular OOP languages that are currently taught are C#, Java, C++ (BSc or BSc Engineering) and Delphi (BEd and in-service teachers with Information Technology as their major subject).

Students' performance in programming is a matter of concern and the programming courses have a high attrition rate (Govender & Grayson, 2006:1687). Ala-Mutka (2004:9) points out that the biggest problem of novices is not their understanding of basic concepts but rather learning how to apply these in a program. Since both BSc and BEd students have problems in the learning of OOP, they should learn how to apply supportive knowledge, skills and strategies to facilitate the process. These techniques are explicitly required in Information Technology or Computer Science to support the learning of object-oriented programming. In this regard, Haden and Mann (2003:70) mention the need for an accurate way of understanding early programming performance to support students in becoming so-called 'good-programmers'.

1.3 Problem statement, research question and subquestions

It is often assumed that students implicitly and independently master the required high-level knowledge, skills and strategies that foster effective programming, and that teaching should focus on programming content and coding structures only. However, to be successful in the complex domain of OOP, explicit learning of both facets is required.

Knowledge relates to information and skills acquired through experience or education. In addition, it refers to what someone already knows (Concise Oxford English Dictionary, 2004:789; Neath & Surprenant, 2003:223). A *skill* can be defined as the ability to do a particular task (Concise Oxford English Dictionary, 2004:1351), while a *strategy* is a designed plan to achieve a specific purpose in the long term (Concise Oxford English Dictionary, 2004:1425). A strategy includes a sequence of activities that are gradually automatised. It is a dynamic process with *problem solving* as its aim (Gu, 2005:1, 6, 10, 16). Since various high-level knowledge, skills and strategies are involved in the programming process, the main research question is:

Which knowledge, skills and strategies are used during problem solving in object-oriented programming?

This question is divided into subquestions, some of which relate primarily to theoretical matters, and were answered by means of the literature study. Other subquestions relate to actual experiences when students undertake object-oriented computer programming. These questions were answered by taking certain theoretical concepts that emerged from the literature study and using them to structure further exploration in original empirical investigations.

Previous studies emphasise the role of a particular domain in supporting programming, e.g. the cognitive (Ala-Mutka, 2004:2), metacognitive (Glaser, 1999:91-92) or problem solving (Sternberg, 2006:424) domain. However, this study follows an approach in which cognitive, metacognitive and problem-solving knowledge, skills and strategies are integrated to support the learning of OOP. Cognition includes a wide range of mental processes used in the acquisition, storage, transformation and application of knowledge (Sternberg, 2006:157). Metacognitive knowledge is explicit knowledge of an individual's own cognitive processes, strengths and weaknesses, beliefs and conditions that affect memory performance (Gravill, Compeau & Marcolin, 2002:1055; Koriat, 2002:267). Problem solving refers to all processes involved with the aim of 'finding an answer' (Concise Oxford English Dictionary, 2004:1374).

The subquestions are shown in Table 1.1, along with references in parentheses to the relevant chapter(s) in the study.

Table 1.1: Research questions and subquestions

Research questions and subquestions	Chapters involved
1. Which *knowledge and skills* are used during problem solving in object-oriented programming? 1.1 Which cognitive knowledge and skills are used in OOP? 1.2 Which metacognitive knowledge and skills are used in OOP? 1.3 Which problem-solving knowledge and skills are used in OOP?	Literature study (3), Empirical research (5), Discussion and conclusion (6)
2. Which *strategies* are used during problem solving in object-oriented programming? 2.1 Which cognitive strategies are used in OOP? 2.2 Which metacognitive strategies are used in OOP? 2.3 Which problem-solving strategies are used in OOP?	Literature study (4), Empirical research (5), Discussion and conclusion (6)
3. What are the differences between the ways in which unsuccessful and successful programmers apply supportive knowledge, skills and strategies in OOP? 3.1 What are the differences between the ways in which unsuccessful and successful programmers apply cognitive, metacognitive and problem-solving knowledge, skills and strategies in OOP? 3.2 What contribution can be made to the practices of teaching and learning OOP by applying the knowledge, skills and strategies used by successful programmers?	Empirical research (5), Discussion and conclusion (6)

1.4 Research objectives

There is a need for the refinement of research on object-oriented design and programming that explores the complexities of OOP (Bergin, Reilly & Traynor, 2005). There is also a need to offer guidelines regarding appropriate activities to support the learning of OOP (Or-Bach & Lavy, 2004:82). The main objective of this study is to identify knowledge, skills and strategies that support students during problem solving in object-oriented programming. The first specific objective is an attempt to identify cognitive, metacognitive and problem-solving knowledge, skills and strategies used by successful and unsuccessful programmers in OOP. Secondly, novices should be supported in learning OOP. The driving force behind this study is the aim of improving the teaching and learning of OOP and to support students during programming.

1.5 Delineation and limitations

The focus of this research is object-oriented programming, with the scope of the study being limited to OOP students in higher education taking Information Technology or Computer Science as a major subject. For programming languages, the BEd (Bachelor of Education) students used Delphi and the BSc (Bachelor of Science) students used Java. Although there are notable differences between these two languages, both are based on the object-oriented paradigm. The following aspects were *not* the focus in this study:

- programming paradigms other than OOP;
- differences in language constructs of various OOP languages; and
- the curriculum in OOP.

1.6 Research framework and methodology

A theory provides a framework for interpreting observations (Schunk, 2000:3) and orientation to the study (Henning, Van Rensburg & Smit, 2004:25, 26). Strauss and Corbin (1998:15, 22) describe theories as sets of well-developed related concepts and themes, which constitute an integrated framework that can be used to explain or predict phenomena. The research paradigm in this study is constructivist problem solving (§2.2, Fig. 2.1). Constructivism emphasises that individuals construct knowledge through their actions in what they learn and understand (Schunk, 2000:229). This research approach can help to understand human thought and action, and to produce deep insights into phenomena (Klein & Myers, 1999:67) such as object-oriented programming.

In this study, a mixed methodology is used that includes both qualitative and quantitative methods in the analysis of the participants' computer programs, thinking processes and questionnaires (Fig. 2.1). Both interpretivism and positivism are applied. The study is mainly interpretivist involving interpretation of participants' thinking processes and programs, while the positivist approach is used to combine interpretivism with statistically significant effects for further clarification.

Various research practices can be used to guide interpretivist studies and to provide consistency as the study progresses. One of these is grounded theory. The goal of grounded theory as a research design in this study is to guide data collection and to derive criteria for analysing the knowledge, skills and strategies of participants during computer

programming. In addition, grounded theory is an approach whereby theory and models can be generated inductively from the collection and analysis of contextual data (Strauss & Corbin, 1998:15; §2.4, §5.3, Chapter 6).

1.7 Data to be collected and research instruments

In order to answer the research questions, a literature study was used as well as empirical research. Literature studies were undertaken in the areas of cognition, metacognition, problem solving, knowledge and skills, strategies, object-oriented programming, interpretivism, and positivism. The subsequent empirical study focuses on identifying and evaluating knowledge, skills and strategies used by students during the process of computer programming.

Population

Participants were cohorts in the years 2005 and 2006 of BSc (Faculty of Science) and BEd (Faculty of Education) third-year students in Computer Science and Information Technology respectively. They participated willingly in this research. Data was collected while participants were designing and developing an object-oriented computer program to perform complex calculations with dates from the calendar. Their associated thinking processes were analysed as well. For the sake of convenience, all participants will be referred to in the male gender.

An object-oriented computer program

The participants were required to design and create an object-oriented program relating to a *Date class* (§2.7.2). This required mandatory calculations with dates, for example, precise determination of which years – over an extended period – were leap years. Apart from the specified functionality, the problem was open-ended and participants could decide to incorporate additional functionality and usability features. They also needed to design a *Test class* to determine whether the output of the *Date class* was correct. The programs could be done in either the Delphi or Java programming language. The students' computer programs were analysed and assessed by the researcher by means of specific criteria that emerged from the literature studies (Table 5.1, Table 5.2, §5.2).

Thinking processes

Participants were also requested to write down their thinking and problem-solving skills during the programming process. This exercise was supported by a framework (Table 2.4) to direct their thinking and problem-solving activities. In this way their thinking processes,

while undertaking computer programming, were made explicit for analysis. The researcher analysed the participants' thinking processes with the support of *Atlas.ti,* a software program and knowledge workbench for qualitative analysis (§2.9, §5.3).

Questionnaire

To extend the research, the participants of 2006 completed a questionnaire about their problem-solving processes during programming. It included closed-ended as well as open-ended questions on the programming experience (Appendix E). The closed-ended questions were statistically analysed by means of descriptive statistics (§5.2.5, §5.4). The open-ended responses were analysed by comparing the answers of successful participants (§5.2.7, Table 5.18) to those of unsuccessful participants and discussing the differences between them.

1.8 Significance of the study

This study holds both theoretical and practical value. The theoretical significance is the identification of specific knowledge, skills and strategies that can be applied with success in the teaching of OOP. The practical contribution is the generation of a synthesised framework, which integrates such knowledge, skills and strategies in a so-called 'learning repertoire'. The purpose of this framework is to support novices in the learning of OOP. In addition, some examples of teaching and learning practices in OOP are outlined. An article regarding this research has been submitted to and accepted for publication by the *South African Computer Journal* (Havenga, Mentz & De Villiers, 2008) (Appendix H).

1.9 Brief chapter overviews

With the background of Section 1.3, which presents the questions to be addressed and answered in this study, the following figure (Fig. 1.1) gives a brief overview of the structure of the thesis. Chapter 1 makes it clear exactly what was studied. In this regard, the following topics are addressed: the background, problem statement and research questions, objectives, delineation and limitations, framework and methodology, data to be collected and research instruments, significance of this study and a brief chapter overview. Chapter 2 gives a clear focus into the research design of this study with reference to the research paradigm and methodology, data collection and data analysis methods used to answer the research question. After an explanation of the approach and concepts of object-oriented programming in Chapter 3, the chapter focuses on the cognitive, metacognitive and problem-solving knowledge and skills required in OOP. Chapter 4 correspondingly focuses on

cognitive, metacognitive and problem-solving strategies in OOP. Chapter 5 describes how the empirical research of this study focuses on ascertaining and evaluating the knowledge, skills and strategies that students use during computer programming. In addition, differences between successful and unsuccessful programmers are addressed with reference to the respective forms of knowledge, skills and strategies that they applied during programming. Findings are presented in Chapter 5 and are further discussed in Chapter 6. Moreover, Chapter 6 answers the main research question and proposes a learning framework to support students in the learning of OOP, particularly with respect to the various activities involved in writing high-quality programs. Finally, the findings inform practice as the value of explicit teaching of the knowledge, skills and strategies used by successful programmers, is discussed.

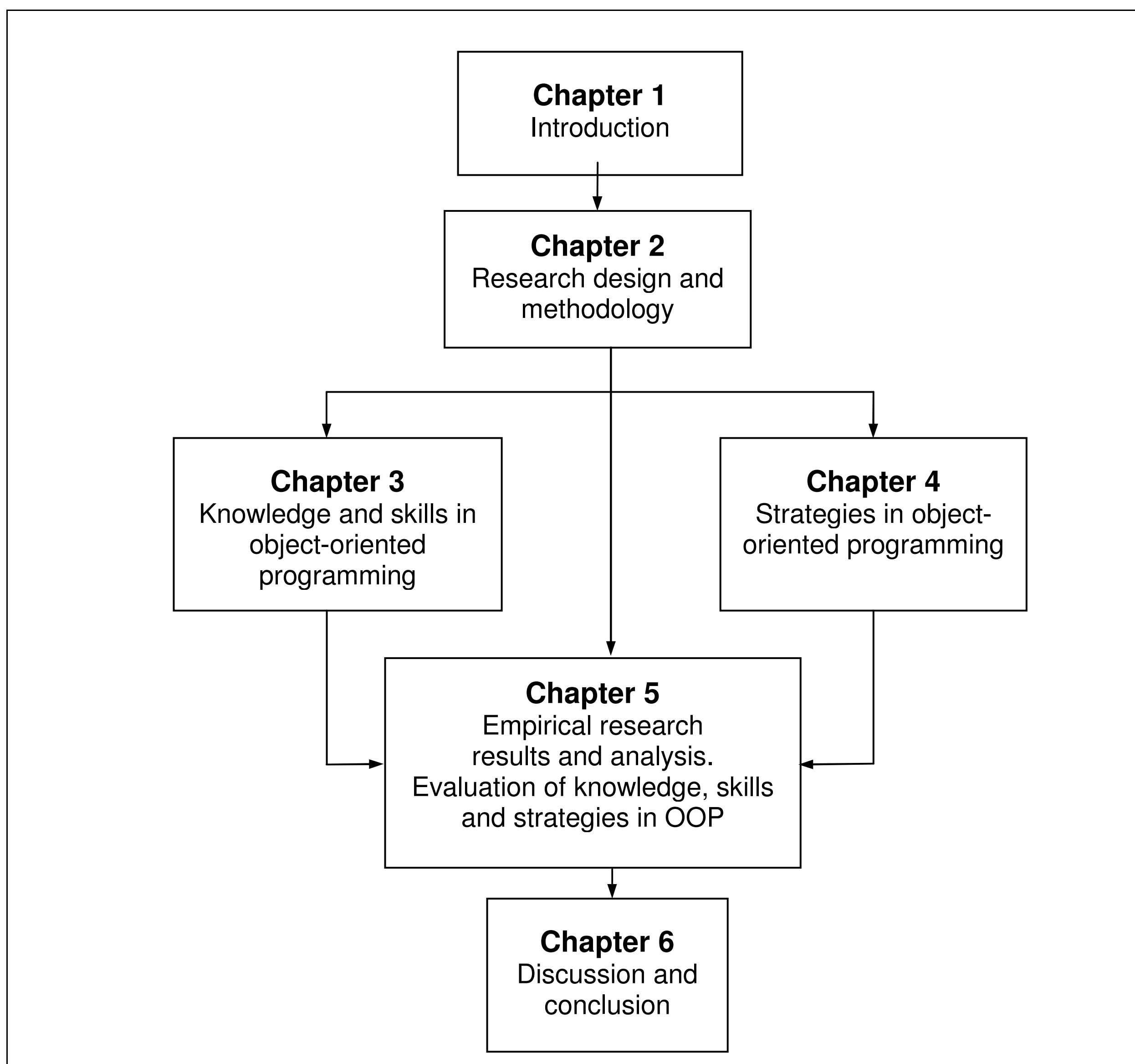


Figure 1.1: Structure of the thesis

1.10 Conclusion

To conclude, an increased understanding of the skills and strategic processes involved in learning object-oriented programming should lead to more effective instruction in computer programming and better learning on the part of students. The objective of this study is to contribute to the practice of teaching by explicitly including certain knowledge, skills and strategies that have been demonstrated to be valuable in object-oriented programming.

The next chapter gives an overview of the research design, paradigm, data collection and data analysis methods.

2 Research design and methodology

2.1 Introduction

The purpose of this chapter is to provide a clear focus into the research design of this study with reference to the research paradigm and methodology, data collection and data analysis methods used to answer the research question. Research is the origin of new ideas and ways of thinking about the world. Formally, research refers to the systematic investigation and study of materials and sources in order to establish facts and verify information (Concise Oxford English Dictionary, 2004:993; Neuman, 2002:23).

The ontological foundations of this research may be found in the design and construction of computer programs. Object-oriented programs (OOP) consist of data and operations that are both encapsulated in an object. Furthermore, a class is a blueprint for an object (Weisfeld, 2004:14; §3.2.3). In this regard, Reed (2003:256, 257) mentions that ontologically all abstractions of OOP refer to representations where classes correspond with concepts. In addition, instances (objects) and methods correspond with 'knowledge of units' and 'knowledge of actions' respectively.

Different epistemological paradigms can be used to explain or predict phenomena and guide the research study (Henning et al., 2004:25, 26, 117; Schunk, 2000:4). Fig. 2.1 presents an overview of the research design in this study. This research is constructivist-based within an interpretivist epistemology and emphasises the construction of knowledge in which students are actively involved (§2.2). Furthermore, this paradigm focuses on the construction of knowledge in situations where problem solving is required in object-oriented programming.

This research employs mixed methods (Johnson & Onwuegbuzie, 2004:15; Cupchik, 2001) whereby both qualitative and quantitative methodologies associated with interpretivism – and, to a lesser degree, positivism – are applied. The research paradigm will be discussed with reference to these qualitative and quantitative methodologies (§2.2, §2.3, §2.6) and grounded theory will be introduced as a research practice for collecting and organising data (§2.4). General research considerations with regard to this study are outlined in Section 2.5,

while the specific data collection and analysis techniques used will be discussed in Sections 2.7 and 2.8 respectively. Finally, the use of *Atlas.ti* software in this study is outlined in Section 2.9.

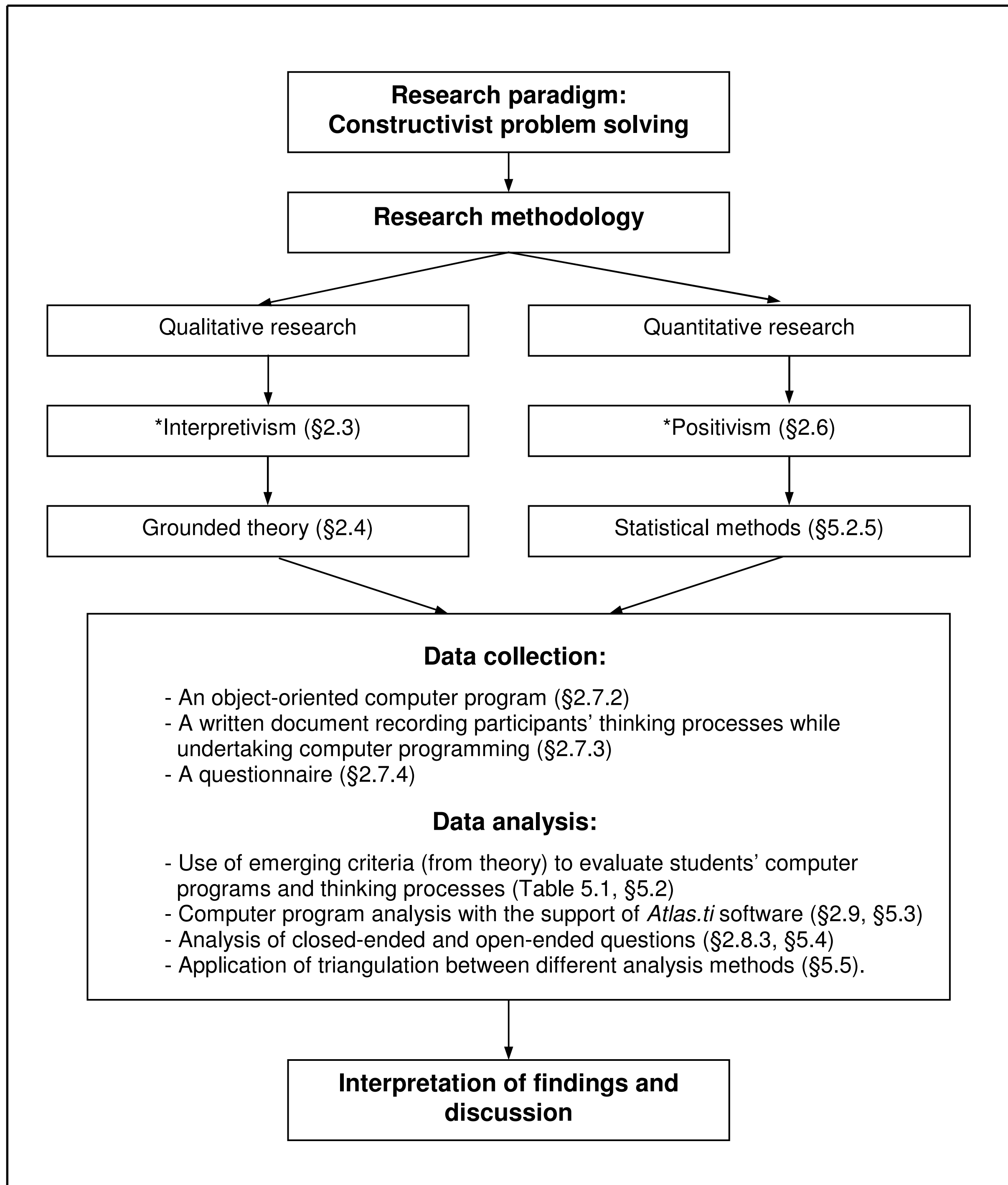


Figure 2.1: An overview of the research design

* Both interpretivism and positivism are included in the constructivist problem-solving paradigm; however this study is mainly interpretivist where the interpretations of participants' thinking processes and programs are required. In addition, the positivist approach is applied to combine the interpretivist approach with statistically significant effects for further clarification.

2.2 Epistemological paradigm, research design and methodology

This research is constructivist-based within an interpretivist epistemology, and subscribes to a learning philosophy that emphasises the active construction of knowledge. Constructivism is a theory that emphasises how people construct their own learning and understanding of the world through experiences and by interpretation and reflection on those experiences (Cupchik, 2001). In addition, the constructivist approach seeks to develop consensus about how to understand the focus of inquiry (Guba & Lincoln, 1989:44-45).

The epistemological paradigm, within which this study is located, as well as the concomitant research design and methodology may, for purposes of this study, be termed *constructivist problem solving.* As such, this paradigm focuses on the active construction of knowledge in situations where problem solving is required in object-oriented programming. According to Hadjerrouit (1999:172), the nature of object-oriented computer programming requires a constructivist approach to problem solving and to the design of a new program for which object-oriented concepts, language and problem-specific knowledge and skills are required. Furthermore, embedded in this paradigm is a rich problem-solving environment where the programming words, statements and constructs all come together and will be integrated in a very specific way to solve the problem efficiently.

This research paradigm was chosen to emphasise the role of the programmer behind the computer in constructing programming knowledge and solving the problem. In addition, it refers to the researcher's role in constructing knowledge about the students' programming. This constructivist problem-solving paradigm therefore has two perspectives. Firstly, it refers to the students in their construction of a computer program. In object-oriented programming, this implies that students should be actively involved in a particular form of problem solving required in constructing a computer program and applying programming constructs such as classes and objects. By using a programming language the programmer must be able to select, organise and integrate various programming statements and constructs in such a way that a new program is developed successfully.

Secondly, this paradigm refers to the researcher's task in the construction of knowledge regarding the students' programming 'constructs', as she interprets and reflects on those programming experiences. This implies a *metaconstructive* process where action, interpretation and reflection are involved. In this way, knowledge is constructed from the

students' intentions, meanings and various thinking processes during their programming of the problem.

In this study, a mixed methodology was applied where both qualitative and, to a lesser extent, quantitative, methods were used to address the research question (Fig. 2.1). In this regard, Weber (2004:iv, vi, vii) refers to 'deep similarities rather than deep differences' between positivism and interpretivism. Both approaches are concerned with the process of understanding the world. In addition, data can be analysed from both interpretivist and positivist perspectives. Qualitative and quantitative methods are not mutually exclusive (De Villiers, 2005b:13). In this study, the methods are complementary and parallel, as each covers various facets of the research process and together they address what researchers actually use in practice (Johnson & Onwuegbuzie, 2004:15; Cupchik, 2001). This implies that the richness and thick description of thinking processes are combined with statistically significant effects for further clarification.

The interpretivist paradigm, along with grounded theory as research practice, is discussed in Sections 2.3 and 2.4 and the positivist paradigm is outlined in Section 2.6.

2.3 The interpretivist paradigm

A research paradigm explains or predicts phenomena, provides orientation and guides the study (Henning et al., 2004:25, 26, 117; Schunk, 2000:4). Fig. 2.1 shows the mixed methodology and dual paradigms that underlie the structure of this study, guiding the researcher's approach and supporting explanations of the phenomena that emerge. Interpretivism is an appropriate paradigm for studies of complex human behaviour.

The interpretive approach originated in the social sciences, however it is increasingly applied in Information Systems for research on the design and development of applications (De Villiers, 2005b:12). The interpretivist paradigm refers to knowledge that is intentionally obtained by means of the *interpretation and meaning of constructs* through the lived experience of human beings. Thus interpretive research leads to subjective findings which may differ between researchers, where the researcher 'is an instrument' to make sense of what he or she perceives as important in the understanding of phenomena (De Villiers, 2005a:142; Weber, 2004:vi; Leedy & Ormrod, 2001:147). Interpretivism investigates research questions and aims to explain human experiences, as it focuses on the

understanding of phenomena that occur in a natural setting (De Villiers, 2005b:12). In this regard, Henning et al. (2004:20) mention that the researcher should look 'at different places and at different things' in order to understand the phenomenon.

Various research practices can be used within interpretive studies, for example case studies, ethnography, narratives and grounded theory. Grounded theory can be used to collect rich data from multiple sources, to refine concepts, to define properties of categories and to identify their relevant contexts (Charmaz, 2000:519). In the grounded theory approach, analysis aims to reflect the perspectives until the saturation of data has occurred (Glaser & Strauss, 1967:61; De Villiers, 2005a:148).

Interpretivism uses mainly qualitative methods to understand the phenomena. Qualitative research refers to methodological approaches that rely on non-statistical methods of data collection and analysis. Some examples of qualitative data are text, cultures and artefacts, documents, journals, interviews, fieldwork, memos, e-mails, and scenarios as resources (Berntsen, Sampson & Østerlie, 2004:3; Prasad & Prasad, 2002:7-8). The main purpose of qualitative methods in this study is to interpret, that is, to gain insights about computer programming so as to develop theoretical perspectives regarding the process of OOP programming.

However, it is important that interpretive research itself should be considered to determine whether this study indeed represents an interpretive approach. Klein and Myers (1999) propose a set of principles for conducting and evaluating interpretive field studies from the philosophical perspective of hermeneutics. These principles set standards to qualify and further ground interpretive research in Information Systems (IS). They are shown in Table 2.1 and are applied to the present study in Table 2.2 in §2.5.1.

Table 2.1: A summary of principles in the Information System field
(Klein & Myers, 1999:72, 73)

A set of principles for conducting and evaluating interpretive studies	
1	**The fundamental principle of the hermeneutic circle:** This principle suggests that human understanding is achieved by iteration considering the interdependent meaning of parts and the whole that they form, by iterating between the parts and the whole in a cyclic manner. This principle of human understanding is fundamental to all other principles.
2	**The principle of contextualisation:** Critical reflection is required on the social and historical background of the research settings, so that the intended audience can understand how the situation currently under investigation emerged.
3	**The principle of interaction between the researcher(s) and the subjects:** This requires critical reflection on how the research materials (or data) were socially constructed through the interaction between the researcher and the participants.
4	**The principle of abstraction and generalisation:** The idiographic details revealed by the interpretation of data resulting from Principles 1 and 2 should be related to theoretical, general concepts that describe human understanding and social actions.
5	**The principle of dialogical reasoning:** Dialogical reasoning requires sensitivity to possible contradictions that may emerge between the theoretical preconceptions guiding the research design and the actual findings (i.e. "the story that the data tells") with subsequent cycles of revision.
6	**The principle of multiple interpretations:** This requires sensitivity to possible differences in interpretations among the participants, as expressed in their multiple narratives of the same sequence of events under study. Similar to multiple-witness accounts.
7	**The principle of suspicion:** Sensitivity is required to possible 'biases' and systematic 'distortions' in the information collected from the participants.

Klein and Myers (1999) propose that these seven principles for the evaluation of interpretive research, should also serve as guidelines for conducting this type of research. The principles are practically applied to the context of the present study in Subsection 2.5.1.

2.4 Research practice – grounded theory

2.4.1 Overview

Various research practices can be used as methodologies for guiding interpretivist studies and to provide consistency as the study progresses (De Villiers, 2005b:17). One of these is grounded theory, which, as shown in Fig. 2.1, will be used in this study as a means of collecting and organising data.

Grounded theory was initially proposed in 1967 by Glaser and Strauss (1967) with the purpose of providing a research framework. In 1998, Strauss and Corbin (1998:21) developed a more detailed description of grounded theory and gave guidelines for the organisation of data using a set of well-developed categories. Appropriate use of grounded theory processes and methods compositely forms an explanatory scheme.

The main purpose of grounded theory is to commence with the data and use it to develop theory that provides relevant interpretations, applications, predictions and explanations (Leedy & Ormrod, 2001:154; Glaser & Strauss, 1967:1, 2, 249; Chapter 6). The grounded theory method specifies an analytic strategy to collect rich data from multiple sources. It involves a process of collecting data to fill conceptual gaps, applying constant comparative analysis, refining concepts, defining the properties of the categories and identifying their relevant contexts (Boychuk Duchscher & Morgan, 2004:607; Charmaz, 2000:519). Grounded theory is thus an approach where theory is generated inductively from the analysis of the data as concepts are formulated into a logical systematic and explanatory scheme (De Villiers, 2005b:22; Strauss & Corbin, 1998:21).

2.4.2 The process of generating a grounded theory

One of the major steps in grounded theory is the coding of data. In this regard certain terminology is explained in more detail:

- *coding* refers to the analytic processes through which data is fractured, conceptualised, and integrated to form theory (Strauss & Corbin, 1998:3);
- a *category* refers to a concept that represents a phenomenon (Strauss & Corbin, 1998:101);
- a *main category* refers to a core category to which other categories are related (Henning et al., 2004:132);

- a *theme* represents a 'chunk of reality' that can be used as a basis for an argument (Henning et al., 2004:107);
- a *theory* denotes a set of well-developed categories and/or themes that are systematically integrated to yield a rich, dense theory that explains some phenomenon (Boychuk Duchscher & Morgan, 2004:608; Henning et al., 2004:107).

The purpose of codes is to capture the meaning of data and to classify a large amount of textual data (Muhr, 2004:32). Codes serve to provide new perspectives on data and to guide the researcher during the analysis process. Related codes can be grouped into categories, which develop inductively, guided by the data. The process of category refinement is continuous until the main categories are identified and integrated into a coherent whole. The collection and analysis of data is therefore a recursive process until saturation has occurred. Grounded theory strives towards verification through the process of saturation, which is achieved when no further evidence emerges and when multiple behaviours indicate similar properties and patterns (De Villiers, 2005b:24; Goulding, 1998:52).

In a grounded theory study, data analysis occurs through various coding procedures: open coding, axial coding, selective coding and the development of a theory (Leedy & Ormrod, 2001:155; Charmaz, 2000:514-516). These different procedures are explained in more detail.

- *Open coding*

  Open coding is the first analytic process of coding. Its purpose is to mark text or other informative data and to associate codes with the marked segments of text. Data is decomposed into parts (concepts) and is marked line-by-line and coded as shown in the example in Fig. 2.2 (Subsection 2.5.2). During the process of open coding, the following types of questions should be asked: What is going on? What is the participant saying? Data is therefore decomposed and the parts are compared and grouped together in a new way into a category. This process of grouping concepts into higher levels of abstraction is called categorising and this forms the basis of grounded theory construction (Henning et al., 2004:131). The selection of groups for comparison highlights the various similarities and differences, which is vital for defining the different categories (Glaser & Strauss, 1967:55).

- *Axial coding*

  Axial coding is defined as 'the process of relating categories to their subcategories' (Strauss & Corbin, 1998:124). The purpose is to refine the information about each category and its subcategories, determining more about a category in terms of its

conditions, context, strategies and consequences (Leedy & Ormrod, 2001:155). Different codes are linked and the focus is on relationships between the various categories, implemented by this linking of associated codes (§2.5.2). However, Henning et al. (2004:132) mention that linking between various categories may be implicit or hidden and could take place on a conceptual level only.

- *Selective coding*

  Selective coding is the process whereby a main category is selected to which other categories are related (Henning et al., 2004:132; Neuman, 2002:423). It is a process where categories are organised around a central explanatory concept where major categories are related (Strauss & Corbin, 1998:161). Finally, each category may become a new theme (§5.3). It is a process of integrating and refining the emerging theory. Strauss and Corbin (1998:161) mention various techniques that can be used to facilitate the integration process, namely writing a storyline, using diagrams or using computer programs.

- *Interpretation and the development of a theory*

  Qualitative interpretation is constructed from the findings. According to Denzin and Lincoln (2000:23), the process of interpretation is critical as theory, methods, actions and policies all come together. An understanding of what constitutes a theory is important. Boychuk Duchscher and Morgan (2004:608) and Strauss and Corbin (1998:21) emphasise that theory denotes a set of well-developed categories that are systematically integrated in yielding a rich, dense theory that explains some phenomenon. The theory must be usable in practical applications, should provide a perspective on behaviour, and should guide and provide a style for research. Thus, the grounded theory approach can be applied in Information Systems in the process of generating theory. This, in turn, can lead to models and/or representations to explain specific phenomena in the domain under consideration (De Villiers, 2005b:25; Chapter 6).

Glaser and Strauss (1967:3-5) cite four criteria for a well-constructed grounded theory. These criteria will be applied in Chapter 5:

- fit: the categories and properties should fit the realities being studied;
- work: in order to work, the theory should explain variations in behaviour;
- relevance: this is achieved when a grounded theory both fits and works; and
- modifiability: the emerging theory is open to adaptation as new data is integrated.

2.5 Research considerations with regard to this study

The relevance of interpretivism and grounded theory to this study is discussed in the following subsections. The issues of reliability, validity and reflexivity are also addressed.

2.5.1 Relevance of interpretivism

The main research objective of interpretive research is the interpretation of actions or meanings of participants' lived experience (§2.3, §5.2, §5.3). Interpretivism can be applied to investigate meanings such as the thinking processes of programmers during the process of object-oriented programming, as was done in the present study. With reference to the award-winning paper of Klein and Myers (1999:67-94), their scope is limited to interpretive research in the context of field studies that also include in-depth case studies. The present study does not include case studies. However, the set of principles in Table 2.1 are applicable to this study, because the interpretation and meaning of 'lived experience' – in this case, thinking processes – is fundamental to the goal of understanding OOP thinking processes, so as to help students to program better and lecturers to teach better. The rigorous use of these principles can also support reliability, validity and grounding in interpretivist research (Pozzebon, 2004:281).

Table 2.2 is an extension of Table 2.1. In Table 2.2, each of the seven principles of Klein and Myers (1999:72-73) is used as a header, followed in each case by a description of how the principle is applied in this study. Cross-references in parentheses indicate the appropriate subsection/s of the empirical work, which is described and discussed in Chapter 5.

A question in §2.3 queries whether or not this study is indeed an interpretive approach. The information in Table 2.2 indicates that this is the case and interpretive research is relevant to this study.

Table 2.2: The application of Klein and Myers' (1999) seven principles in this study

	Application of a set of principles for conducting and evaluating interpretive studies
1	**The fundamental principle of the hermeneutic circle:** Achieving human understanding by iterating between the parts and the whole that they form. *Neither the parts (programming statements) nor the whole (the computer program) can be understood without reference to each other. Referencing specific programming statements in the program is necessary to understand the purpose of the entire program (§5.2, §5.3).*
2	**The principle of contextualisation:** Reflection on the social and historical background of the research setting to explain the current situation under investigation. *Participants who took Computer Studies or some other computer-programming course prior to their university studies, might demonstrate superior problem-solving skills and strategies during the programming process.*
3	**The principle of interaction between the researcher(s) and the subjects:** Reflection on how the data was constructed through the interaction between the researcher and the participants. *How are computer programs constructed and how are they related to the associated thinking processes? Data will emerge from interaction between the researcher and participants. The researcher will direct the students' thinking and programming by means of specific requirements for the calculations with dates and by requesting the students to reflect and record their thinking processes (Table 2.4; Appendix C).*
4	**The principle of abstraction and generalisation:** Relating the idiographic details revealed by the interpretation of data resulting from Principles 1 and 2 to theoretical, general concepts that describe human understanding and social actions. *The students' specific thinking processes and programming statements must be related to generic problem solving in object-oriented programming (§5.2, §5.3).*
5	**The principle of dialogical reasoning:** Sensitivity to possible contradictions between the theoretical preconceptions guiding the research design and actual findings that may emerge from subsequent cycles of revision. *The researcher is sensitive to possible contradictions between theoretical foundations and the data that emerges from the research. She will acknowledge such and will make the basis of the research process as transparent as possible to the reader (§5.2, §5.3, §5.4).*
6	**The principle of multiple interpretations:** Sensitivity to possible differences in interpretations among the participants, in the same sequence of events under study. Similar to multiple-witness accounts where all participants recall an event as they personally perceived it. *It requires sensitivity to analyse the different interpretations by participants of the same computer program and the different approaches to constructing it (§5.2, §5.3).*
7	**The principle of suspicion:** Sensitivity to possible 'biases' and systematic 'distortions' in the information collected from the participants. *Biases and distortions in the participants' written thinking processes should be identified. Furthermore, there should be explicit avoidance of bias on the part of the researcher, in her interpretation of the qualitative data (§5.3).*

2.5.2 Relevance of grounded theory

Grounded theory is an approach whereby theory and models can be generated inductively from the collection and analysis of contextual data. It can be applied in Information System (IS) research, including areas such as OOP, to synthesise a theoretical framework, which leads to models (De Villiers, 2005b:25; Chapter 6). The initial goal of grounded theory as a research design in this study is therefore to guide data collection, and to derive criteria for analysing the knowledge, skills and strategies of participants during computer programming. Such criteria emerged inductively from the literature study (Table 5.1).

The subsequent steps of a grounded theory process, introduced in §2.4.2, will be revisited in the section below, and applied to the present study.

- *Open coding*

  In open coding, data is decomposed into parts, marked line-by-line and coded. During the process of open coding, the following questions must be asked: What is the participant saying? Which actions did the participant take during programming? In the example of Fig. 2.2, the programming statement below (extracted from a 'thinking processes' document – see §2.7.3) could be coded as indicated after the arrow. The colons in the *New code* separate further descriptions for the purpose of elaborating the meaning of the code:

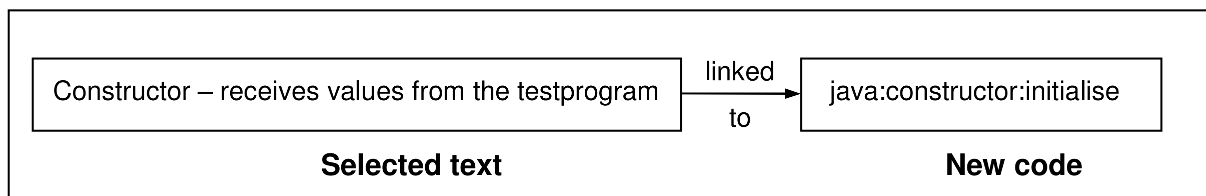


Figure 2.2: An example of selected text with the associated code

  The code *java:constructor:initialise* was later categorised in the cognitive knowledge, skills and strategies category, as an indication of the application of knowledge and skills in OOP.

- *Axial coding*

  In axial coding, different categories are combined in new ways to make connections. The researcher focuses on the categories that may be clustered together or may be subdivided into subcategories (Fig. 5.10). Several closely related concepts can be organised into a major topic of interest. Furthermore, it may suggest dropping or

adding a theme or examining other themes in more depth (Neuman, 2002:423). For example, the theme *Cognitive knowledge, skills and strategies* is examined in more depth in Chapter 5.

- *Selective coding*

  During selective coding, a main category is selected to which other categories are related. Selective coding involves scanning data and previous codes and looking selectively for cases that illustrate themes (Henning et al., 2004:132; Neuman, 2002:423). The selection of groups for comparison makes the similarities and differences distinct. Using axial and selective coding in *Atlas.ti,* a category can be coded as a coded family; and a coded family usually becomes one theme in the study (Henning et al., 2004:137, §2.9 §5.3). For example, a coded family named *Cognitive knowledge, skills and strategies* is a theme in §5.3.3 and is comprised of several different codes that relate to aspects within the cognitive domain.

- *Interpretation and the development of a theory*

  Qualitative interpretation with a final thematic pattern is constructed from findings (Henning et al., 2004:106). During the process of interpretation, different methods and actions all come together to motivate and defend the interpretations and to develop an emerging theory on the programs and thinking processes in OOP. Chapters 5 and 6 give details on such an emerging theory relating to participants' thinking processes.

The reliability and validity of qualitative analysis methods are addressed in the next subsection. Furthermore, reflexivity with reference to the researcher's beliefs will be briefly outlined.

2.5.3 Reliability, validity and reflexivity

- *Reliability and validity*

  Interpretivism uses mainly qualitative methods to understand phenomena and to describe, interpret and build theory (Leedy & Ormrod, 2001:101,102). However, multiple methods and/or sources of analysis should be used to ensure reliable and valid data. Reliability refers to consistency or repetition of the same results under identical or similar conditions. Validity refers to the truthfulness i.e., whether the findings of the study are true (accurately reflecting the real situation) and certain (whether the findings have been backed by evidence) (Guion, 2002:1; Neuman, 2002:164,171). However, in

practice, qualitative researchers apply reliability and validity principles in different ways than those used by quantitative researchers. One method used in qualitative research is triangulation.

- *Triangulation*

  Triangulation is a method used by qualitative researchers to check and establish validity in their studies (Golafshani, 2003:597; Guion, 2002:1). Du Plooy (2001:38) defines triangulation as the combination of two or more data-collection methods, and advises the use of multiple sources of information to obtain data. Different types of triangulation can explain the richness of human behaviour and are suitable where a complex phenomenon is studied (Cohen, Manion & Morrison, 2000:112,115). Guion (2002:1) distinguishes between five different types of triangulation, namely data triangulation, investigator triangulation, theory triangulation, methodological triangulation and environmental triangulation. In this study, methodological triangulation was used. Methodological triangulation establishes validity between different methods e.g., the use of qualitative and quantitative methods (Guion, 2002:2). It was applied in the present study by using both qualitative analysis with the aid of *Atlas.ti* (§2.9, §5.3), and various statistical methods (§5.2, §5.2.5) to analyse participants' computer programs and thinking processes (§5.5). Further data was also elicited by means of a questionnaire.

- *Reflexivity*

  Reflexivity is a subjective concept that refers to personal experience, which influences the thoughts and meanings of a researcher. The researcher's beliefs guide the research work and her background influences the interpretation of data (Williamson, 2006:86; Denzin & Lincoln, 2000:19). The present researcher has been involved in Computer Science/Information Technology for many years as an examiner of provincial matriculation examination papers. She is also a lecturer for BEd Computer Science students. This study gives the researcher the opportunity to build on personal experience, to support the students and to establish trust and effective communication, as she gains new insights through reflective research. In such a situation, any form of bias on the part of the researcher must be avoided, as also mentioned in Principle 7 in Table 2.1 and 2.2. In the mixed methodology of this study, a positivist approach was used to obtain further insights by means of statistical analysis. This is discussed in the next section.

2.6 The positivist paradigm

2.6.1 Relevance of the positivist paradigm

The positivist paradigm (Fig. 2.1) refers to knowledge that is absolute and objective. Such knowledge is discovered by means of systematic investigation of phenomena in a controlled environment. Positivism focuses on science as an approach that verifies and confirms empirical observations by means of measurable ways where findings are 'true' (Cupchik, 2001). In this paradigm, the observer is separated from the observed findings. This paradigm uses mainly quantitative methods to analyse data. Data collection comprises measurements such as those obtained from experiments, which are frequently processed by statistical analysis. Furthermore, a quantitative study may confirm or reject the hypothesis that was tested (Cupchik, 2001).

The positivist paradigm was applied in this study to add another facet of analysis by 'measuring' data to explain the success of participants (e.g., Table 5.2, Table 5.15, Table 5.26 - 5.28) and to ensure reliability and validity of data. The statistics used in this study include the following: factor analysis, reliability testing, descriptive statistics (mean value and standard deviation), and practical significance (effect size and correlation), all of which are outlined in Subsection 5.2.5 and applied in Sections 5.2 and 5.4.

2.6.2 Reliability and validity

Quantitative research measures variables with the purpose of explaining, predicting and controlling phenomena (Leedy & Ormrod, 2001:101). Various statistical techniques can be used to check reliability and validity of data, and inferences are based on actual and objective experiences (Johnson & Onwuegbuzie, 2004:14).

In this study, the mixed method research design, combining interpretivism and positivism, provided a "complementary and parallel" approach to "promote shared responsibility" for research quality and to cover various facets of the research process in practice (Johnson & Onwuegbuzie, 2004:15, 24; Cupchik, 2001). Leedy and Ormrod (2001:101) emphasise that by using both quantitative and qualitative research methods, a researcher is not limited and can "learn more about the world". The data collection and analysis techniques used in this study are described in §2.7 and §2.8 respectively to explain the process in more detail.

2.7 Research methods: data collection techniques

In this section, the research plan and the participants are discussed briefly to set the context, after which, the various data collection techniques used in this study, are outlined. Finally, there is a brief mention of the ethical aspects involved.

2.7.1 Research plan and participants

The empirical research shown in Fig. 2.1 was done over two years of study, namely 2005 and 2006, investigating situations where participants gained experience in object-oriented computer programming. In the second year, the research was extended by means of a questionnaire, which required participants to answer specific questions about their programming experiences.

- *Data collection in 2005*
  - Students were required to write a *Date class* program, an object-oriented computer program to execute complex calculations with dates (§2.7.2, §5.2). The text of the *Date class* task is given in Appendix C. The programs were collected as data and retained by the researcher for subsequent analysis.

  - Associated with each *Date class* programming task, students recorded their thinking processes during the programming experience (§2.7.3). These also became data in the form of documents (Fig. 5.4, Fig. 5.6).

- *Data collection in 2006*
  - The next cohort of students was also required to write the *Date class* program to execute complex calculations with dates (§2.7.2, §5.2). The programs were collected as data and retained by the researcher for subsequent analysis.

  - Associated with each *Date class* programming task, students recorded their thinking processes during the programming experience (§2.7.3). These also became data in the form of documents (Fig. 5.4, Fig. 5.5).
  - In addition, in 2006, a questionnaire with closed-ended and open-ended questions was given to participants to extend the research (§2.7.4).

The scenario and the data collection techniques are summarised in Table 2.3:

Table 2.3: The research plan in this study

Year	Participants	n	Programming language	Data collection techniques
2005	BEd 3rd year	11	Delphi	1. *Date class* program: a computer program regarding dates and leap years 2. Written document of participants' thinking processes during programming
	BSc 3rd year	17	Java	
2006	BEd 3rd year	3	Delphi	1. *Date class* program: a computer program regarding dates and leap years 2. Written document of participants' thinking processes during programming 3. Questionnaire
	BSc 3rd year	17	Java	
Total		**48**		

The participants in this study were third-year Computer Science students (Table 2.3). The first group (BEd students) were from the Faculty of Education, and used Delphi as an object-oriented programming language (rows 1 and 3 in Table 2.3). The second group (BSc students) were from the Faculty of Science and used Java as an object-oriented programming language (rows 2 and 4 in Table 2.3). The data collection techniques will now be discussed in more detail.

2.7.2 Object-oriented computer program

Each participant was required to design and create an object-oriented program relating to a *Date class* (Table 2.4, §5.2, Appendix C). This required certain calculations with dates. It was an open-ended question and participants had to decide personally which calculations were necessary in the program. At the very least, it was compulsory for the program to determine which years were leap years and the difference between any two dates in the range 1 January 1800 to the current date. Programmers had to bear in mind that specific months have 31 days and others 30 days, and that, when a year is a leap year, February has 29 days instead of 28 days. The participant also had to design a *Test class* or 'driver' program to determine whether the output of the *Date class* program was correct. Both programs could be written in either the Delphi or Java programming language. Table 2.4 (an extract from the programming task in Appendix C) shows important requirements, which were given to participants to direct their programming:

Table 2.4: Requirements for writing the *Date class* program

Date class
Include the following: Variables Constructor Input: today's date
Use the following methods (Java); procedures and functions (Delphi): setTodaysDate (format: yyyymmdd) getDay( ); getMonth( ); getYear( ) isLeapYear( ) – test for leap years dateDifference( ) – calculate the difference between two dates
Application or *Test class:* Instantiate an object Decide which method of input will be used (files/streams/ components, etc.) Decide what exception handling is necessary if any dates are incorrect

Requirements for the determination of leap years

Fundamental to the program was that students had to calculate when a year was a valid leap year. Leap years are required so that the calendar is in alignment with the earth's motion around the sun. A leap year is a year with one extra day inserted into February; therefore, a leap year has 366 days, with 29 days in February instead of the usual 28 days. In the Gregorian calendar, which is the calendar used by most modern countries, the following rules determine which years are leap years:

> Every year divisible by 4 is a leap year;
> However, a year divisible by 100 is not a leap year;
> There is one exception: if the year is also divisible by 400, then it is a leap year.

The years 1800, 1900, 2200 are not leap years but the years 2000, 2004, 2400 are leap years (Thorsen, 2007).

2.7.3 Written document – participants' thinking processes

During the process of programming the *Date class* task, participants were required to reflect on and to write down their thinking and problem-solving processes. This exercise was supported by the use of a framework to direct these related processes. The complete text of

the programming assignment is given as Appendix C. This reflection and the written records made the participants' thinking processes during OOP explicit for subsequent analysis.

2.7.4 Questionnaire

The purpose of the questionnaire in 2006 was to extend the research and to determine students' cognitive and practical behaviour during the problem-solving process involved in writing the computer program. No ideal questionnaire including questions about knowledge, skills and strategies in object-oriented programming was readily available; therefore, a new questionnaire was designed for this study based on theoretical concepts that emerged from the literature studies in Chapters 3 and 4.

The questionnaire was divided into categories for (i) cognitive knowledge and skills, (ii) metacognitive strategies and (iii) problem-solving strategies. For example, the category for cognitive knowledge and skills comprised various skills from Bloom's taxonomy. For each of these six cognitive skills (knowledge, comprehension, application, analysis, synthesis and evaluation, §3.3.2), three questions were compiled that related it to the OOP domain. These three questions per skill were included to enhance reliability, and all three therefore needed to be consistent and had to measure the same construct or issue (e.g., knowledge). In compiling the questionnaire, the questions referring to one specific issue were deliberately not grouped according to the categories or subcategories, but were distributed throughout the questionnaire to enhance reliability. The questionnaire and mark sheet are given in Appendix E.

A pilot version of this questionnaire was given to a small group of participants, comprising one honours student, one fourth-year student, and three Computer Science lecturers. The purpose was to check the clarity of each question, to eliminate complexities, and to gain feedback on technical issues such as layout and numbering within the questionnaire and the time necessary to complete it. It was also important to determine which questions were ambiguous or confusing. The pilot questionnaire served well and certain improvements were made.

2.7.5 Ethical aspects

This study carries the approval of the Dean of the Faculty of Education and the Head of the School of Computer Science to conduct this research with students as participants. Each student participated willingly in the study and completed a consent form (Appendix A). The consent form also guaranteed confidentiality of participants. Moreover, this study carries the approval of the Ethical Committee and Research Director of the tertiary education institution where this research was conducted (Appendix B).

2.8 Research methods: data analysis techniques

The main purpose of the qualitative methods used in this study was to gain insight into the nature of problem solving in OOP and to develop theoretical perspectives regarding this phenomenon (Chapter 6). The data analysis techniques are discussed briefly.

Table 2.5 indicates the various types of data collected, namely computer programs, textual documents and questionnaire data, relating them to their associated data analysis methods:

Table 2.5: Data collection and analysis methods

Data collection methods	Data analysis methods
Object-oriented computer programs (*Date class* and *Test class)*	- Computer programs analysed manually, using a framework of measurement criteria that emerged from the theoretical literature studies (Table 5.1) - Factor analysis and sample adequacy (§5.2.5) - Descriptive statistics (mean, standard deviation) (Table 5.12, Table 5.16, Table 5.19) - Reliability (Cronbach-alpha values >= 0.5) (§5.2.6) - Practical significance with reference to effect size and correlation (Table 5.17, Table 5.19, §5.2.5)
Written documents describing participants' thinking processes	Textual document analysis using the *Atlas.ti* software program and knowledge workbench for qualitative analysis (§2.9, §5.3)
Questionnaire data	*Closed-ended questions:* - Factor analysis and sample adequacy (§5.2.5) - Descriptive statistics (mean, standard deviation) (§5.2.5, §5.4) - Reliability (Cronbach-alpha values >= 0.5) (Table 5.26 – 5.28) - Correlation (Table 5.30) *Open-ended questions:* a discussion of each question (§5.4.3)

2.8.1 Computer program analysis

All participants' computer programs (§2.7.2) were analysed qualitatively using a framework of specific criteria (Table 5.1) that emerged from the literature study chapters (Chapters 3 and 4). Analysis of the *Date class* programs in the programming languages Delphi and Java was done according to these measurement criteria, which relate to the various knowledge, skills and strategies used in OOP. For each category in Table 5.1, the participant received a mark (score) out of 4, except for the problem-solving section where participants could have used more than one strategy for which 8 marks were therefore allocated. In addition, quantitative analysis was done, including factor analysis and sample adequacy; the determination of descriptive statistics such as mean values with standard deviation of all categories (§5.2.5); reliability of constructs; and the practical significance of various constructs to report measures such as effect size and correlation.

2.8.2 Textual document analysis – using the support of *Atlas.ti*

Each participant's thinking processes (§2.7.3) were also analysed. This qualitative process was done by means of *Atlas.ti* software, which provided support to the researcher during the analysis process as well as during the interpretation and organisation of participants' thinking processes. It also facilitated the application of grounded theory in practice as described in §2.5.2 and §5.3. The use of *Atlas.ti* for the purpose of textual analysis is explained more extensively and specifically in Section 2.9.

2.8.3 Questionnaire data analysis

To extend the research and data collection of 2005, the participants of 2006 completed a questionnaire about their problem-solving processes during programming (§2.7.4). This was done after they had completed their *Date class* programming tasks. It also provided a means of collecting relevant personal information from each participant.

Questionnaires frequently make use of rating scales to determine behaviour and opinions. The Likert scale originates from Rensis Likert, who developed this technique for the assessment of attitudes (Neuman, 2002:182). The present questionnaire employs a Likert scale on the continuum: 'never', 'seldom', 'often' and 'always', using a 4-point scale so as to

avoid any middle options. There were also several open-ended questions. The questionnaire (Appendix E) comprises the following subsections:

- Cognitive knowledge and skills;
- Metacognitive strategies; and
- Problem-solving strategies.

Mean values of each category were calculated and findings will be reported in Chapter 5. The findings of the open-ended questions are discussed in Subsection 5.4.3.

2.9 Qualitative data analysis software – *Atlas.ti*

2.9.1 Application of *Atlas.ti*

Atlas.ti is powerful software that supports the researcher in handling large amounts of data during the process of qualitative analysis. Various types of data can be analysed, including textual, graphical, audio and video data (Muhr, 2004:2, 5). The present researcher decided to use *Atlas.ti* due to the large amount of data (48 participants) and to optimise the analysis process with various tools incorporated in *Atlas.ti* (Fig. 2.3). A complete description of the analysis process is shown in §5.3.1 and §5.3.2, along with two detailed examples.

This section is included to explain the researcher's activities during the *Atlas.ti* analysis process. The description is illustrated with screen displays. It provides snapshots of the process to set the context for the full empirical study of Chapter 5.

- *The researcher's task during analysis*
  Each participant's thinking processes were saved as a primary document (PD) and analysed by the researcher with the support of *Atlas.ti*. The researcher opened and scrutinised each participant's document (Fig. 2.6, Fig. 2.7). Various notable connotations or significances emerged repeatedly (or, in some cases, less commonly) from the different student's texts. The researcher defined codes to represent each of these connotations, then she highlighted and selected segments of text from each PD and assigned the appropriate codes to them. The purpose of the codes, therefore, is to capture the meaning of data. For example, 'Design a constructor for the Date.java class' in a participant's document is assigned the following code: 'java:constructor' (Fig. 2.7). This process was conducted for all participants' data and thereafter the codes were organised by the researcher into possible 'families' that could subsequently

become themes. The researcher was responsible for the decisions made in the analysis process, however, the use of *Atlas.ti* software supported the researcher in various ways.

- *The support of Atlas.ti software*

  Various techniques are available to support the analysis process of large amounts of data. Tools incorporated in *Atlas.ti* are: a text search tool, auto coding tool, query tool, redundant code analyser, and HTML and XML generators. Furthermore, tools such as an object manager, network editor and code tree help the researcher to navigate through the data structures and concepts (Muhr, 2004:35). Fig. 2.3 shows various tools incorporated in *Atlas.ti:*

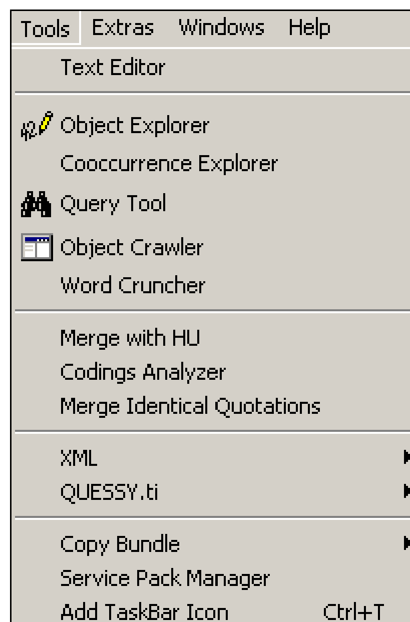


Figure 2.3: Various tools available in *Atlas.ti*

- *The Atlas.ti methodology*

  Muhr (2004:3) mentions additional principles of the *Atlas.ti* methodology: visualisation, integration, serendipity and exploration. Visualisation is possible in *Atlas.ti* to support human perception by displaying relationships between objects. Integration refers to the hermeneutic unit that integrates all the primary documents (Fig. 2.5). Serendipity refers to 'finding something without having searched for it' (Muhr, 2004:3), and exploration is the directed process of getting data as part of constructive activities.

- *Two principal modes of Atlas.ti*

  The two principal modes or levels of working with *Atlas.ti* were both applied in this study. The *textual level* refers to the researcher's activities, such as the coding of text

(Fig. 2.7) and the adding of comments (§2.8, Fig. 5.5, Fig. 5.7). The *conceptual level* refers to model building by the researcher, whereby codes are linked to diagrams to form networks (Fig. 2.12; Muhr, 2004:25, 26) and possible themes (Fig. 5.10). These activities all support the process of developing a grounded theory. Fig. 2.4 consolidates the researcher's role by giving an overview of the steps during this qualitative analysis process (§5.3):

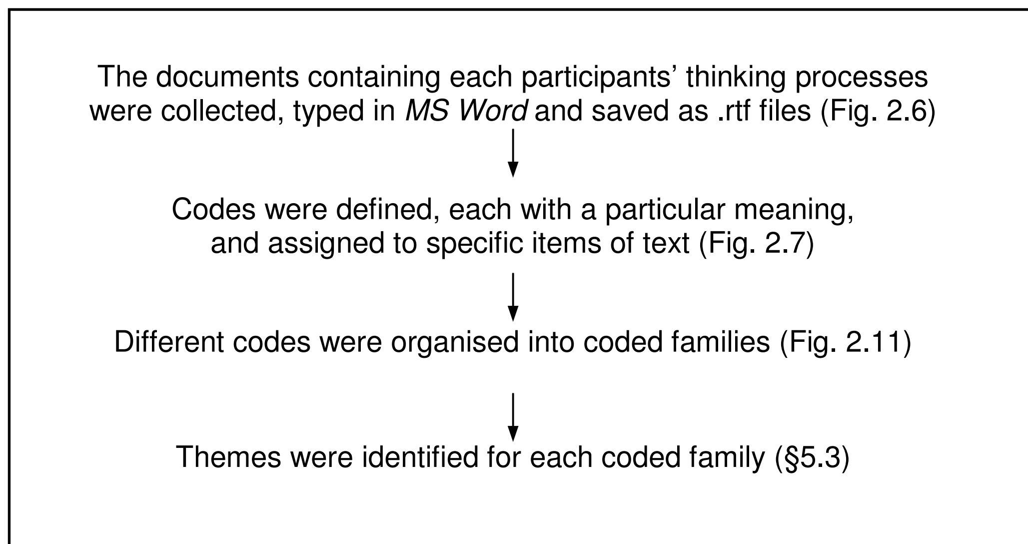


Figure 2.4: Different steps in the analysis of participants' thinking processes with *Atlas.ti*

In cases where the students' documents were written in the Afrikaans language, the selected text was translated into English for the purposes of this study. After the codes had been allocated to different segments in the thinking processes of participants (Fig. 2.7), related codes were grouped or categorised into families that became possible themes (§5.3). Each theme could be represented with a network structure generated by *Atlas.ti* (Fig. 2.12). Output could be displayed in an *Atlas.ti* editor, sent to the printer, or saved as a file. The entire hermeneutic unit (comprising all the participants' documents) could be exported in an XML-format. A CD with this entire file is included in the back cover of this thesis as part of the study.

A description of the main workspace in *Atlas.ti* follows, along with a sequence of illustrations.

- *Primary documents*

  Each participant's thinking processes were saved as a primary document (PD) and analysed by the researcher with the support of *Atlas.ti*. Within *Atlas.ti,* all the different primary documents are linked to *one* hermeneutic unit (HU) (Fig. 2.5) and can be selected from a drop-down list (Fig. 2.6).

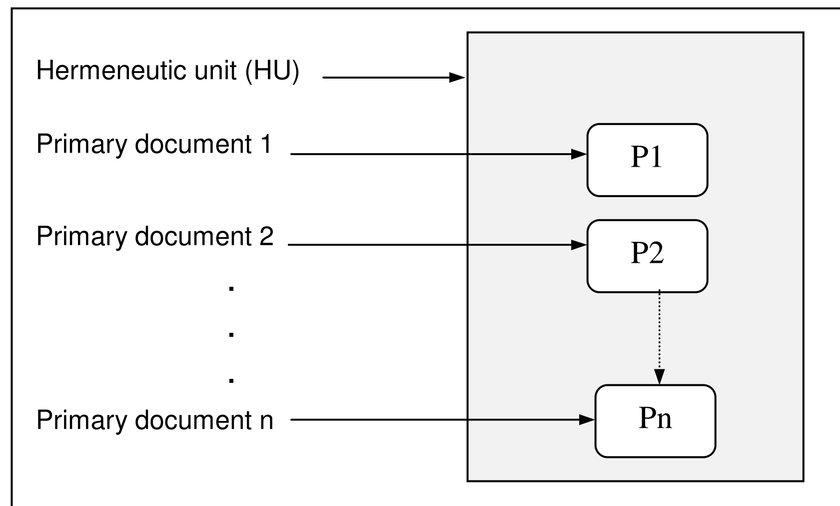


Figure 2.5: One hermeneutic unit with many primary documents
(adapted from Muhr, (2004:66))

When clicking on the top left drop-down arrow of the screen captured in Fig. 2.6, the available primary documents are shown and a specific document can be selected from the drop-down list to display the contents of that primary document. On the left-hand side of Fig. 2.6, the name of the first primary document (P1:1BEDBJ05.rtf) is highlighted.

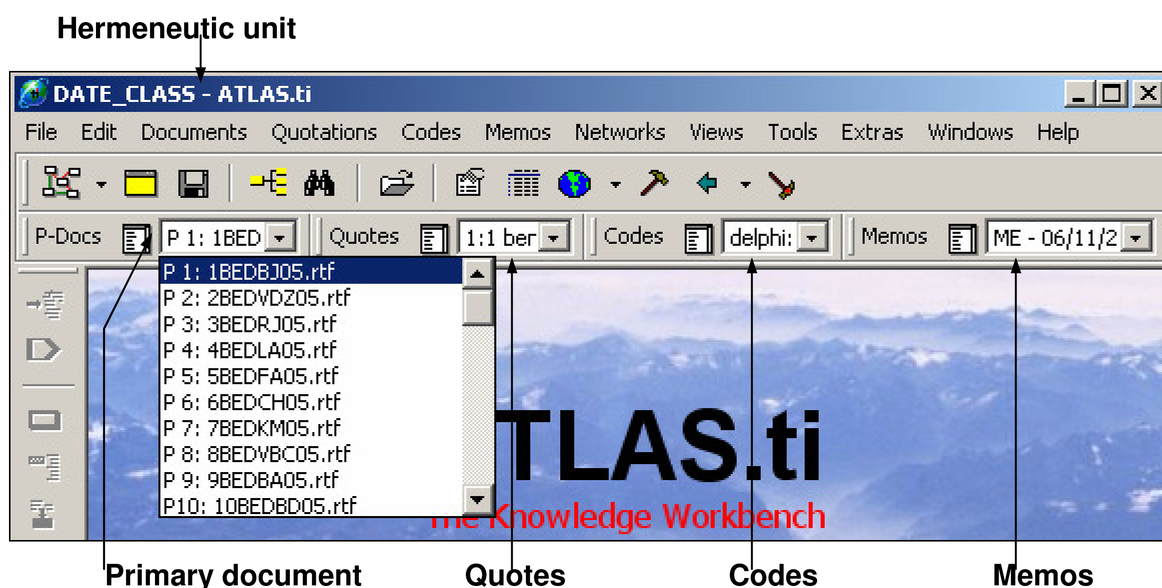


Figure 2.6: *Atlas.ti* qualitative software

In order to edit primary documents within *Atlas.ti,* the word-processing documents should be saved in rich text format (.**rtf**), which is a standard feature in most word processing programs. As shown in Fig. 2.7, which displays a text extract from a PD,

the documents (students' thinking processes) were typed and saved in rich text format before they could be assigned as primary documents to *Atlas.ti* in a HU.

- *Coding process*

  Various types of coding can be used in *Atlas.ti* (Muhr, 2004:116,117):
  - *Open Coding* creates a new code;
  - *In-Vivo Coding* creates a code from a selected text;
  - *Code-by-List* selects existing codes from the code list; and
  - *Quick Coding* codes with the currently selected code.

Fig. 2.7 portrays an extract from the coding process of Participant 32's (P32) thinking processes. In the window on the left-hand side, specific lines of text were selected and highlighted by the researcher for the purpose of assigning codes to them. These lines were originally written in Afrikaans by the participant, and have been translated into English immediately below the highlighting.

In the window on the right-hand side, an area is shown with the codes and memos. The numbers {8-1} in parentheses associated with the highlighted code 'java:constructor {8-1}' refer to: (groundedness:density) where groundedness is the number of quotations associated with this specific code (namely 8) and density is the number of codes connected to this code (namely 1) (Muhr, 2004:118). The numbers (11:17) in parentheses after the unhighlighted code 'java:constructor (11:17)' refer to the row number (11) where the highlighted text associated with a specific code starts, and the row number (17) where that text ends. Furthermore, in a code, 'java:...' refers to BSc students who used Java as programming language and 'delphi:...' refers to BEd students who used Delphi as the programming language.

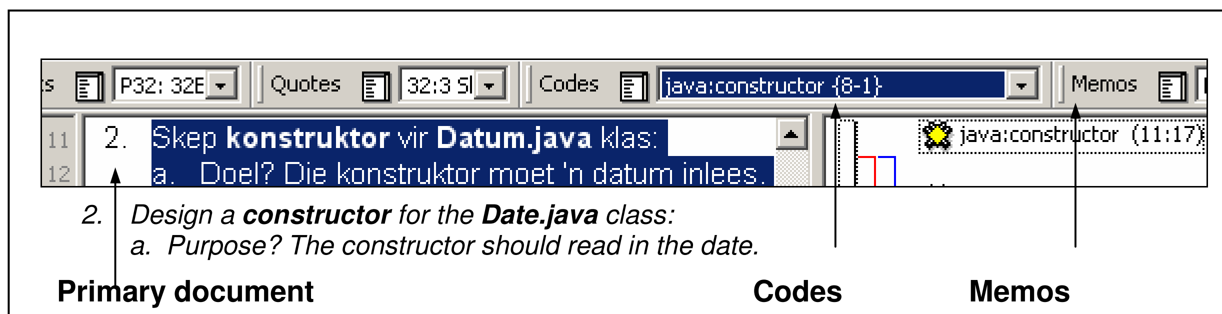


Figure 2.7: An extract of text from a primary document in *Atlas.ti*
(with the participant's text translated from Afrikaans to English below)

Fig. 2.8 shows examples of various codes in *Atlas.ti*

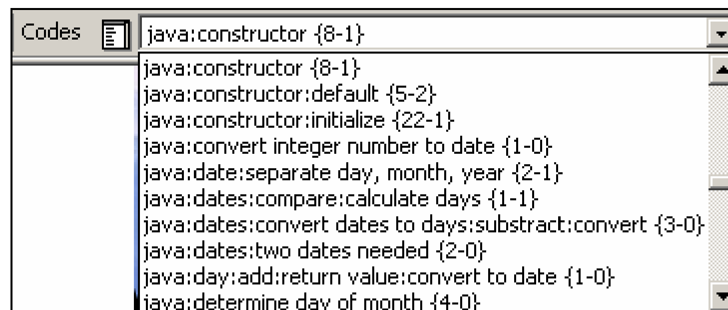


Figure 2.8: Examples of codes in *Atlas.ti*

- *Quotations*

  Quotations are selected pieces of text that explain thinking during programming. During the process of coding, different quotations can be selected from the primary document (Fig. 2.7, Fig. 2.9) and marked as such. A quotation list can be printed to display all occurrences of a particular code in the form of its associated quotations from all the primary documents (n=48).

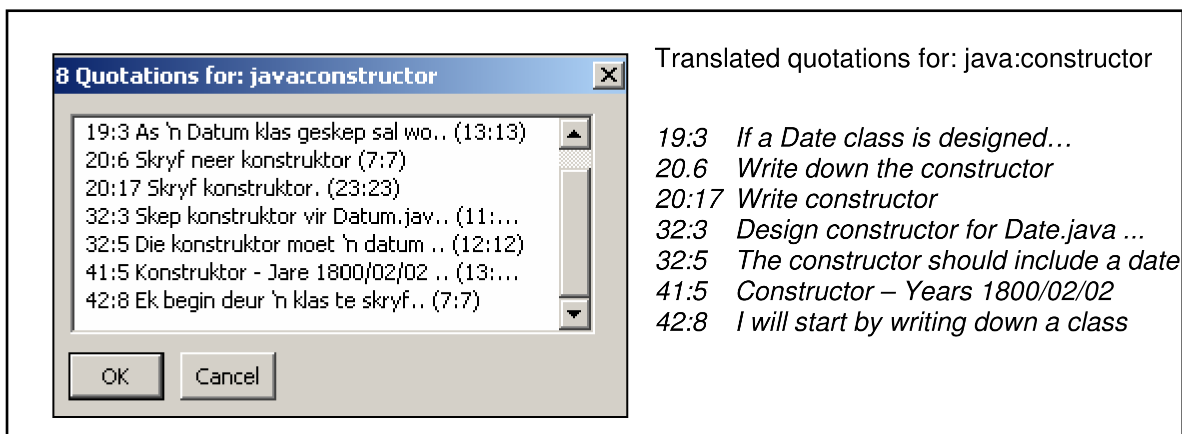


Figure 2.9: Examples of quotations associated with the code 'Java:constructor'

- *Memos*

  Memos are similar to codes, but their main purpose is to capture the *researcher's* comments and thoughts about data (Fig. 2.10) rather than content of the textual data in hand. They can also be attached to quotations. Memo writing entails the making of reflective notes regarding the data (Henning et al., 2004:132), and can also be used to provide more information about the codes. Charmaz (2000:517) describes memo writing as an intermediate step between coding and the completed analysis. It serves to keep the researcher focused, to connect categories and to define the memos' purpose in a bigger picture. Furthermore it aids in interpretation of data.

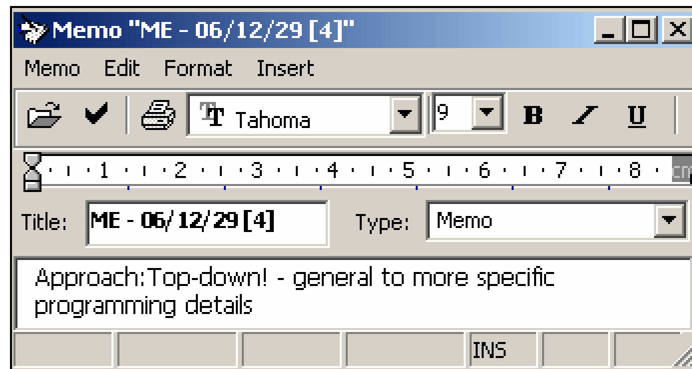


Figure 2.10: An example of a memo in *Atlas.ti*

- *Families and networks*

  After the codes have been assigned to different segments in all the primary documents, related codes are grouped or categorised as a coded family to form a possible theme. Fig. 2.11 shows examples of five families that emerged from the *Date class* HU (§5.3).

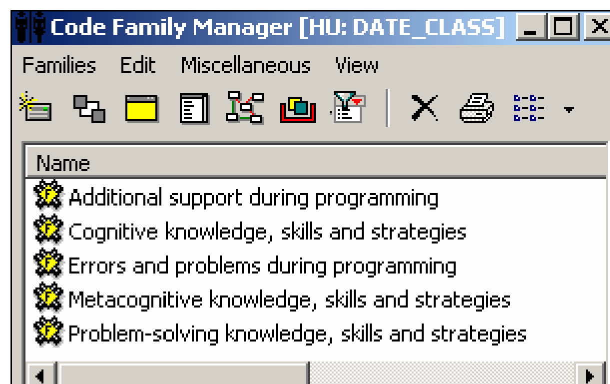


Figure 2.11: An example of families in *Atlas.ti*

Each theme can be represented with a network structure in *Atlas.ti* (Fig. 2.12). Network structures allow conceptualisation by connecting similar elements in a visual diagram (Muhr, 2004:33). In *Atlas.ti,* the network editor provides a method to create and manipulate these network structures. 'CF' at the bottom of Fig. 2.12 refers to the 'Coded Family':

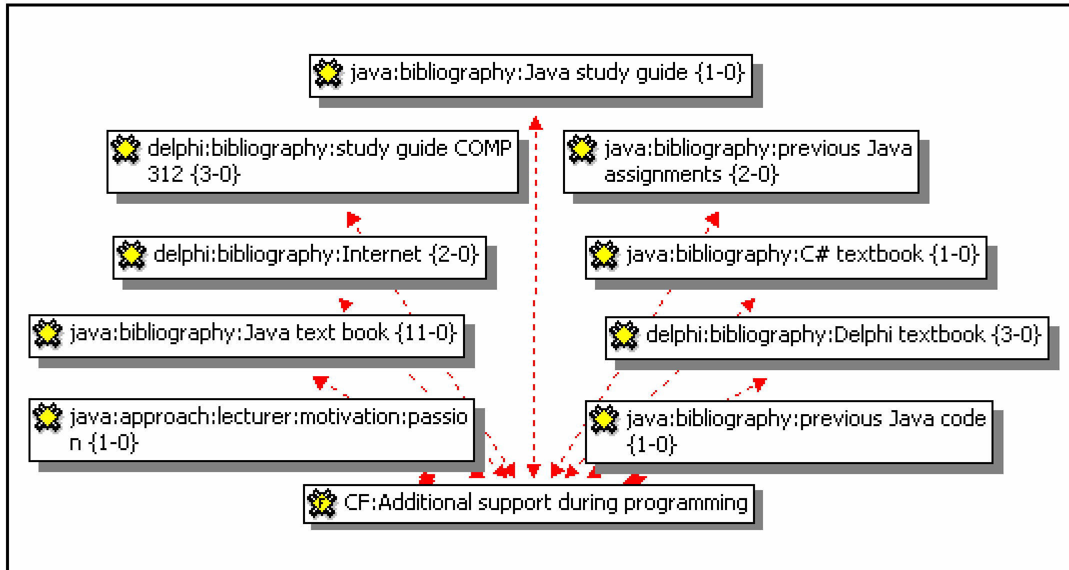


Figure 2.12: An example of a network structure in *Atlas.ti*

A final thematic pattern is constructed from which the researcher should perceive the overall picture. This view should emphasise the meaning between various categories, as well as the purpose of each specific category (Henning et al., 2004:106, 107, Fig. 5.10).

To summarise this section, *Atlas.ti* is an integrated collection of various tools that support the researcher in the process of qualitative analysis. Within *Atlas.ti,* all participants' data, in the form of primary documents, is linked to one hermeneutic unit. Various quotations, codes and memos can be respectively selected, assigned, or created, in the process of refining data from multiple sources into themes to explain and clarify students' thinking processes during OOP.

2.9.2 The harmony between grounded theory and *Atlas.ti*

It is important to note that grounded theory had an explicit influence on the design of the qualitative analysis software of *Atlas.ti* (Henning et al., 2004:114, 117, 122; §2.5.2). The role of *Atlas.ti* in the present grounded study is outlined in more detail below.

The qualitative analysis in this study involved the interpretation of the documents setting out the students' thinking processes in object-oriented programming. Coding processes are central both to grounded theory and to most of the software packages used as tools for this purpose (Henning et al., 2004:130). Different forms of grounded theory coding – open coding, axial and selective coding – are available in *Atlas.ti,* and were used to analyse the

documents as explained in §2.5.2. The selection and grouping of related codes highlighted various similarities and differences, which were vital for defining the different categories (Glaser & Strauss, 1967:55). Grounded theory strives towards verification through the process of saturation, which is achieved when no further evidence emerges (Goulding, 1998:52). In the analysis process with *Atlas.ti*, the researcher decided to continue with the analysis process until all the participants' thinking processes had been processed. In fact, saturation of data did not occur until near the very end.

Grounded theory analysis entails the inductive refinement of categories to more abstract levels and the integration of categories into a coherent whole that can 'explicate, clarify, illuminate and also explain' (Henning et al., 2004:116, 117) various processes. The use of axial and selective coding in *Atlas.ti* supports the coding of categories into coded families, and a coded family usually becomes one theme in the study (Henning et al., 2004:137, §2.5.2, §5.3).

Table 2.6 summarises the main grounded theory concepts and their associated methods or tools in *Atlas.ti*.

Table 2.6: A summary of grounded theory concepts and their associated methods or tools in *Atlas.ti*

Grounded theory concept	Methods or tools in *Atlas.ti* that implements the grounded theory concept
Open coding The analytic process through which concepts are identified and their properties and dimensions are discovered in the data (Strauss & Corbin, 1998:101).	**Coding** The procedure of associating a quotation with a code. Various types of coding can be used in *Atlas.ti* (Muhr, 2004:116,117, §2.9.1): • *Open coding* (creates a new code); • *In-Vivo coding* (creates a code from selected text); • *Code-by-list* (selects existing codes from the code list); and • *Quick coding* (codes with the currently selected code).
Axial coding The process of relating categories to their subcategories, linking categories at the level of properties and dimensions (Strauss & Corbin, 1998:123)	**Linking of codes** The linking of various categories takes place on a conceptual level to determine relationships among categories (Muhr, 2004:214).
Selective coding The process of integrating and refining the theory (Strauss & Corbin, 1998:143).	**Families** Families are used in *Atlas.ti* to classify a group of objects. Coded families are a group of codes that belong together (Muhr, 2004:191, 192).
Building theory A theory is a set of well-developed concepts related through statements of relationship, which together constitute an integrated framework that can be used to explain or predict phenomena (Strauss & Corbin, 1998:15).	**Model-building activities** In *Atlas.ti,* the conceptual level focuses on model-building activities such as designing a network that links various nodes (Muhr, 2004:26, 107, 211).

2.10 Chapter conclusion

The purpose of this chapter was to provide a clear focus on the research design of this study with particular reference to the paradigm and methods used to answer the research question. The overall epistemology, research design and methodology were introduced in §2.2, while the role of the interpretive paradigm was outlined in Section 2.3. Grounded theory was used in this study as the main research practice for the collection and organisation of data (§2.4).

Section 2.5 outlined the relevance of interpretivism and grounded theory in this study, while the role of the positivist paradigm was summarised in Section 2.6. Sections 2.7 and 2.8 addressed methods of data collection and analysis respectively. Reliability and validity were discussed with reference to qualitative and quantitative research methods. Statistical analysis of the questionnaire was discussed in Section 2.8. Section 2.9 was devoted to explaining how participants' thinking processes were analysed with *Atlas.ti* software. This fairly lengthy section is intended to set the scene for the empirical analysis presented in Chapter 5. Finally, the relationship between grounded theory and *Atlas.ti* was mentioned. The purpose of using *Atlas.ti* in this study was outlined, and its application was described and illustrated.

Chapters 3 and 4 comprise the literature study component of this research. In Chapter 3, the cognitive, metacognitive and problem-solving knowledge and skills necessary in OOP are considered, laying a foundation for Chapter 4, which discusses the respective cognitive, metacognitive and problem-solving strategies.

3 Cognitive, metacognitive and problem-solving knowledge and skills in object-oriented programming

3.1 Introduction

It is not well understood how people learn to program and solve a problem in computer science (Traynor & Gibson, 2004:2). According to Deek (1999:1), learning to program is a complex cognitive task that includes learning the programming language, comprehending existing programs, modifying written programs, composing new programs and using debugging techniques. In the process of learning object-oriented programming (OOP), the student must know which objects, behaviours and interactions are important in the problem domain. There is a need for the refinement of research that explores the difficulties of OOP. There is also a need for guidelines about specific types of knowledge and skills to support the learning of OOP (Or-Bach & Lavy, 2004:82; Staats & Blum, 1999:13).

Efficient knowledge and skills on the part of the programmer are necessary during the processes of problem solving, decision making, planning and critical thinking in OOP. *Knowledge* relates to information and skills acquired through experience or education and also refers to what someone knows (Concise Oxford English Dictionary, 2004:789; §1.3). Declarative knowledge refers to the knowledge of facts while procedural knowledge refers to knowledge of procedures that can be implemented in a task (Sternberg, 2006:229). Both types of knowledge are important in OOP. A *skill* can be defined as the ability to do a particular task (Concise Oxford English Dictionary, 2004:1351; §1.3).

Fig. 3.1 shows various types of knowledge and skills, whose application in OOP is explored in this chapter. This includes the cognitive, metacognitive and problem-solving *knowledge and skills* necessary in OOP. The shaded blocks in Fig. 3.1 present the goal, various types of knowledge and skills, and their application in OOP that will be addressed in this chapter.

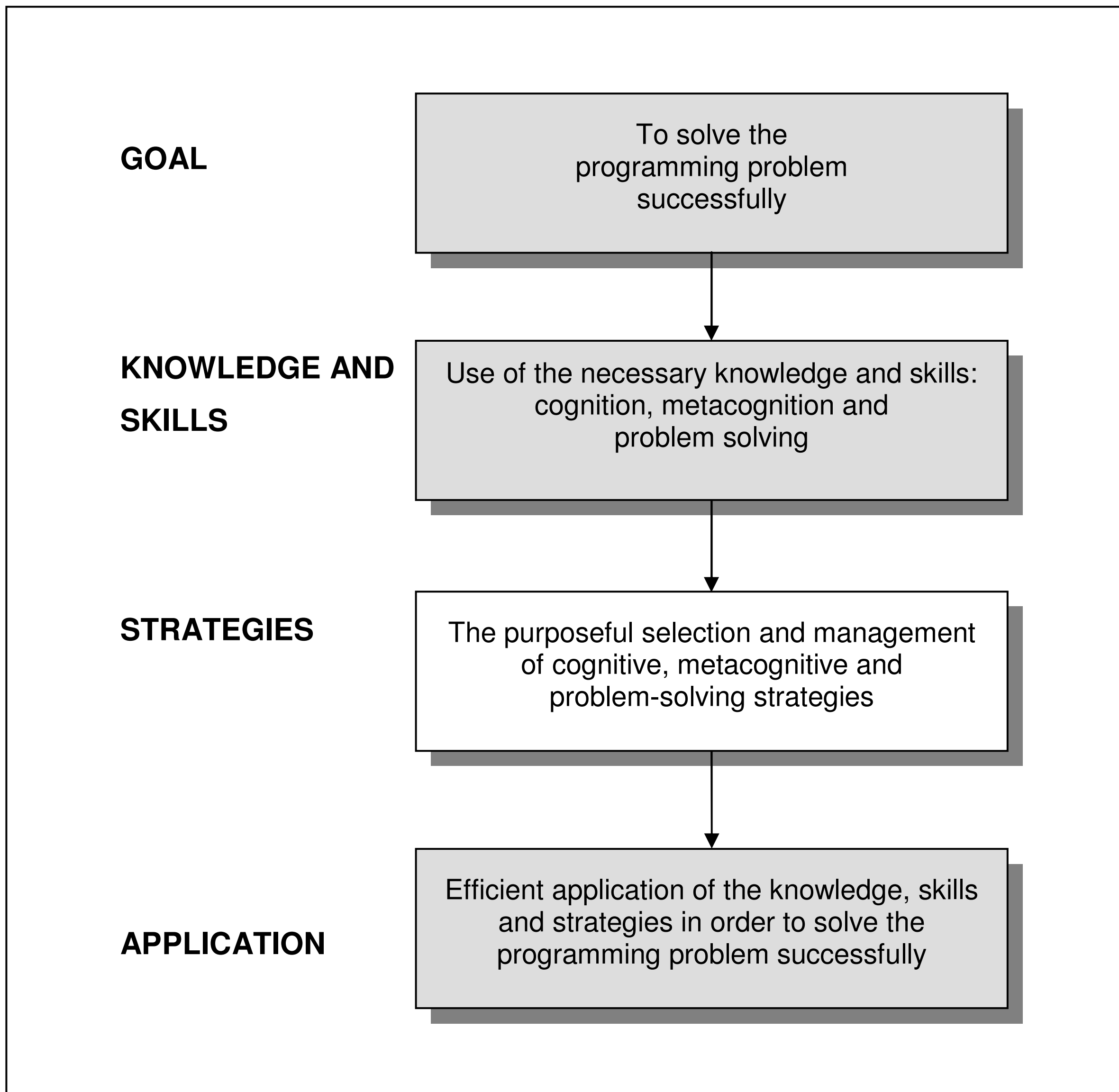


Figure 3.1: Various goals, knowledge, skills, strategies and their application in an object-oriented program

After an explanation of the approach and concepts of object-oriented programming (§3.2), this chapter focuses on the cognitive, metacognitive and problem-solving *knowledge and skills* necessary in OOP. These three topics are addressed in Sections 3.3, 3.4 and 3.5 respectively. Chapter 4 builds upon Chapter 3 as it correspondingly focuses on cognitive, metacognitive and problem-solving *strategies* in OOP.

Furthermore, various guidelines and practical means of support during the learning of OOP are discussed in some detail in different sections of the chapter.

3.2 Object-oriented programming

There are different approaches, called paradigms*,* to the way a programmer analyses, designs and implements a program (Ragonis & Ben-Ari, 2005:203). The object-oriented paradigm is widely advocated internationally in many higher education institutions (Or-Bach & Lavy, 2004:82), and was introduced in South African universities a few years ago. This section elaborates on the object-oriented paradigm, the origin of OOP languages, programming notations and models as well as problem and design spaces in OOP.

3.2.1 The need to change to the object-oriented paradigm

Changing to the object-oriented paradigm was advisable due to many problems in software development:

- Programming languages need a specific platform or operating system. Most programs must run on the Windows platform, but traditional programming languages, such as Turbo Pascal need a DOS-based operating system and many errors occurred when such programs were executed within a Windows-based environment (§1.2);
- Programming in a DOS environment was difficult without a user-friendly graphical interface;
- In the former procedural paradigm, the main program lay in control. Since global declaration of data was permitted, it resulted in difficulty in modifying and testing of programs because the data may change in any procedure (Rosson & Alpert, 1990:356);
- Programming problems became more complex and the programs consequently became cumbersome (Yousoof et al*.,* 2006:259).

The object-oriented approach addresses some of these problems. Object-oriented programming is based on this approach, where objects are models of real-world entities that have the responsibility of carrying out specific tasks to solve the problem (Garrido, 2003:26-27). Most software applications can run on a Windows platform, have a user-friendly graphical interface, and each object is responsible for its own behaviour (Shalloway & Trott, 2002:6,12-15).

3.2.2 The origin of object-oriented programming languages

The origin of OOP can be traced to SIMULA 67, but it was later fully developed in the programming language Smalltalk in the late 1960s. The program units in Smalltalk are objects, which encapsulate local data and methods. All computing is done by the sending of messages to an object to invoke a method (Sebesta, 2004:92, 93).

C++ was built on the programming language C to support OOP. It was designed by Bjarne Stroustrup at Bell Laboratories in 1980. In 1990, Sun Microsystems designed Java. The fundamental goal was to provide greater simplicity and reliability than C++. Anders Hejlsberg designed Delphi, which first appeared in 1995. It is a Windows programming development tool, which extended Borland Pascal language by providing object-oriented support and a graphical user interface for programming. The latest OOP language is C#, which was developed along with the .NET platform in 2000 by Anders Hejlsberg (Sebesta, 2004:92, 93, 106).

3.2.3 An overview of object-oriented programming

This subsection gives an overview of the building blocks of an object-oriented program. Moreover, key features of an OOP language will be discussed. It concludes with a summary of the advantages and disadvantages of an OOP language.

The main building block of the object-oriented approach is an object, which is a self-contained entity with both data and methods. This means that an object stores attributes and determines its behaviour in a program with methods (Satzinger, Jackson & Burd, 2004:175). Methods replace the procedures of traditional programming and are invoked by messages send to the object. An object may change its own values by receiving messages. For example, the method *setName(String myName)* can be invoked and the object responds by assigning a *String* type.

Objects are based on classes, which indicate the specific type of an object. A class is therefore the blueprint of an object. When an object is instantiated, the class specifies how an object is built and defines the attributes and methods of all objects created with that specific class (Weisfeld, 2004:14). A class is divided into an interface and implementation section. The interface provides communication between a unit and other units. Within the interface, the instantiation and operation of objects are defined. For example, the method

```
function getName:String;  // return a name
```

is defined in the interface section. Only the *public* attributes and methods are part of the interface (Weisfeld, 2004:18). Details should rather be contained in the implementation section. The user should not be able to make any changes to the implementation part, which is *private* to the unit and contains the inner details of the class declared, i.e.:

```
function TForm1.getName:String;
begin
  Result := fName;
end;
```

The focus of object-oriented programming is to enhance reusability and extensibility (Sebesta, 2004:384). Class libraries are examples of highly reusable classes. These can be used for easy implementation of additional functionality, for example, *uses Sysutils* in Delphi for string/integer conversion or *import JOptionPane* in Java to create a standard dialog box that prompts the user for a value. Extensibility can be implemented by using inheritance to extend and modify existing classes (Weisfeld, 2004:76). For example the Bus and Motorcar subclasses can inherit from the Vehicle parent class some general attributes and properties.

The greatest benefit of object-oriented programming is that the user can interact with the computer by manipulating objects on the computer screen. In this regard, Heines and Schedlbauer (2007:1) mention that graphical user interface (GUI) programming provides a vehicle to teach OOP effectively. The GUI is the interactive part of a program, where the user comes in contact with the complete system via the interface where input and output occured (Satzinger et al., 2004:531).

The basic elements and features of the object-oriented approach are defined and discussed in more detail.

3.2.3.1 Object

As already stated, an object is the main building block within the object-oriented paradigm, where objects have the responsibility of carrying out specific tasks (Garrido, 2003:26-27). Some real-world examples of an object are a bus, a book or a student. Objects are therefore items that people think about, identify, act upon or apply concepts to (Satzinger & Ørvik, 2001:17).

3.2.3.2 Class

Students should gain a clear understanding of the rich concepts ‘object’ and ‘class’ (Eckerdal & Thuné, 2005:89, 92, 93). A class is a general category, whereas an object is a specific instance (Satzinger & Ørvik, 2001:39). Similar objects are grouped together into classes that also specify the type of the object. A class can also be described as a template or design pattern for a possible object (Eckel, 2003:35; Weisfeld, 2004:14). Particular objects are called instances or instantiations of a class, for example, the bus with registration number, PGK456GP is an instance of the Bus subclass, as shown in Fig. 3.2

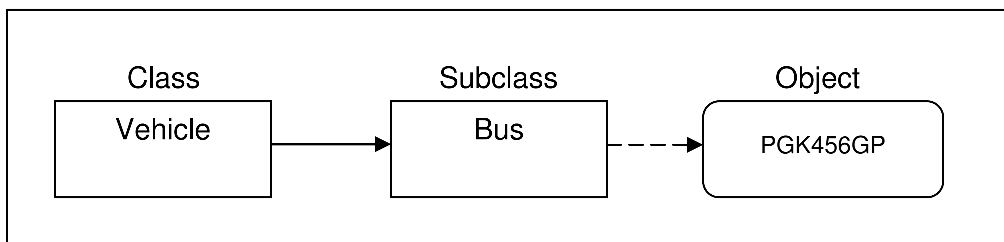


Figure 3.2: The Vehicle class, Bus subclass and Bus-object

3.2.3.3 Attributes and methods

In the object-oriented approach, both the attributes and operations are equally important. Attributes are descriptive properties of an object, for example, a bus object has the properties *registrationNumber*, *routeNumber* and *maintenanceDate*. These attributes are stored within an object and represent the state of the object.

An operation determines the behaviour of an object or what the object can do. Examples of possible methods for the bus object are *getRegistrationNumber( )*, *changeRoute(..)* and *getMaintenanceDate( )*. A method may be invoked by sending a message to it that might change the state of the object. The method name, parameters and return type of a method are required (Schach, 2005:19; Sebesta, 2004:459; Weisfeld, 2004:10). Methods are specified as *public, protected* or *private* depending on their availability to one or more classes or subclasses.

Both the attributes and methods are encapsulated or bundled in an object to control the access to objects (Weisfeld, 2004:8). Each object is therefore a self-contained entity to carry out its role (Lewis, Pérez-Quiñones & Rosson, 2004:18). Information hiding differs from encapsulation because information hiding is a technique whereby details of each class’s performance are hidden from other classes to isolate the effects of changing design decisions (Lewis et al., 2004:18).

3.2.3.4 Constructors and destructors

Two special methods play important roles in OOP, namely constructors and destructors. Constructors are called when creating new objects to allocate memory and initialise variables. Delphi uses the constructor *Create* to create an object. In Java, all objects are explicit heap dynamic i.e., created explicitly on the heap during runtime, and are allocated with the *new*-operator (Sebesta, 2004:487).

Destructors are called to reclaim heap storage and destroy objects. Java does not make use of a destructor but uses implicit garbage collection i.e., there is no need for the programmer to write code for the destructor (Sebesta, 2004: 432, 440, 478). Delphi uses the *Free* method to destroy an object. *Free* automatically checks for a nil reference before calling the *Destroy* method (destructor).

A programming language is object-oriented if it supports abstraction, inheritance and polymorphism (Sebesta, 2004:458; Stroustrup, 1995:2). These principles will be discussed in more detail in the following subsections.

3.2.3.5 Abstraction and associations

Abstraction is the ability to define and use variables and operations that allow the details to be ignored. The purpose of abstraction is to simplify the presentation of entities during programming and to reduce complexity (Sebesta, 2004:16, 428, 429). Two different kinds of abstraction are distinguished: process abstraction and data abstraction.

Process abstraction refers to the calling of a subprogram (method/procedure/function) without providing the details of the subprogram (Sebesta, 2004: 428,429). For example, the details of a *sortArray(int[ ] myArray)* method are hidden from the Java programming environment.

Data abstraction refers to the declaration of the type and the operations on objects that are contained in a single unit to restrict data access by means of messages sent to the methods (Schach, 2005:19, 185; Sebesta, 2004:420). Global declaration of data leads to difficulty in modifying programs, because the data may change within different methods (§3.2.1). The solution is data abstraction. In Java, an abstract data type is declared and defined in one single unit named a class. In Delphi, a user-defined unit is an example of an abstract data type. For example,

*TCalculate = **class**(TObject);*

Different abstraction principles are used in the object-oriented approach, namely classification and instantiation; generalisation and specialisation; and aggregation and composition (Satzinger & Ørvik, 2001:92-95).

Classification and instantiation

Classification refers to the principle by which different things can be classified in a category according to similar properties. Similar objects are grouped together into classes that also specify the type of the object. Objects of a specific class, such as Vehicle, represent a set of properties that are typical to the Vehicle class. Although the bus, PGK456GP has properties that are typical of 'bus vehicles', it has some particular properties that are specific to the bus PGK456GP (Fig. 3.2), such as its colour and registration number.

Generalisation and specialisation

A class should not be designed for a particular task but rather for a particular kind of task (Lewis et al., 2004: 18). In programming, a special case of the generalisation relationship is inheritance, which is considered as an *is-a* relationship, where the derived classes inherit attributes and methods from the parent class (Schach, 2005:499-502; Weisfeld, 2004:27). In biological terms, a child inherits certain characteristics from its parents. In the same way, inheritance provides a framework in OOP for hierarchies where the derived class or subclass inherits attributes and methods from its parent class or superclass. However, in OOP the class with more functionality (subclass) inherits from the class with less functionality (parent class) because less functionality is included in a more general class (Hadar & Leron, 2008:44).

Inheritance makes reuse possible and reduces the amount of duplicate program code (Sebesta, 2004:459). The purpose of inheritance relationships is that new object types need not be designed from scratch, but are variations of an existing class.

For example: A Bus *is-a* Vehicle.

In this simple statement, 'Bus' can be considered a subclass and 'Vehicle' a class. Furthermore, the *is-a* relation indicates an inheritance relationship between Bus and Vehicle, indicating that the attributes and methods applicable to the Vehicle class are also 'available' to the Bus subclass. In the case of single inheritance, a class can inherit from one class only. However, some problems can be solved with multiple inheritance. When a subclass has more than one parent class, there is a situation of multiple inheritance (Schach, 2005:195; Sebesta, 2004:459-460).

Aggregation and composition

Aggregation refers to an association that specifies the whole-part or *has-a* relationship where an object is composed from other objects (Satzinger & Ørvik, 2001:43, 94). For example, a Bus *has-a* steering wheel, one engine and many seats.

Composition is a stronger form of aggregation among the different parts. In other words, if the whole is deleted, so will be all the parts. Aggregation is like a collection of things, whereas composition implies that an object is part of the containing object (Shalloway & Trott, 2002:34-35).

3.2.3.6 Polymorphism and dynamic binding

The word polymorphism means multiple forms and implies different forms of methods. In OOP, this means that different objects may respond individually to the same message, therefore the same method may indicate different implementations. With polymorphism, we can implement new types that share common logic (Weiss, 2000:43). Polymorphism also supports greater abstraction, where a single message can evoke different behaviours (Rosson & Alpert, 1990:357).

It is sometimes necessary that a method of the parent class may be overridden by its derived class. This happens when a method call is bound dynamically (during program execution) to the method in the proper class. These methods have the same name and similar operations. This is called dynamic binding (Sebesta, 2004:460-461).

3.2.3.7 Advantages and disadvantages of object-oriented programming

To summarise, the transition to the object-oriented paradigm occurred due to certain strengths of OOP:

- In the object-oriented paradigm, each object is responsible for its own behaviour and such responsibilities must be clearly defined in the program (Shalloway & Trott, 2002:6, 12-15).
- The object-oriented paradigm considers the attributes and operations to be equally important and emphasises information hiding. This makes development and maintenance easier and promotes reuse (Schach, 2005:18-23).
- The organisation into a hierarchy of a superclass with one or more subclasses is a characteristic. This reduces the amount of programming code (Satzinger & Ørvik, 2001:5).

- The availability of user-friendly interfaces (Graphical User Interfaces or GUIs) has revolutionised the support for end-users working on desktop applications (Satzinger & Ørvik, 2001:7). There is a natural harmony between GUIs and the visual programming languages that support users and that ease the process of programming. Examples of visual programming languages are the object-oriented languages, Delphi and Visual Basic, where different buttons or objects can be organised on a visible form.
- The object-oriented paradigm addresses quality, productivity and flexibility (Satzinger & Ørvik, 2001:9). Quality refers to the measure of the standard of a product. Inheritance enhances productivity, because many lines of code can be eliminated and can rather be inherited. Moreover, in terms of flexibility, extension of a particular class is easier in OOP than in the procedural paradigm. Maintenance is therefore easier in the object-oriented paradigm.
- Many authors mention that OOP is more natural than the procedural programming paradigm, because an object refers to a 'thing' in real life (Satzinger & Ørvik, 2001).

The object-oriented paradigm also has some disadvantages:

- OOP requires the ability on the part of the programmer to grasp complex patterns of interactions between objects. Neubauer and Strong (2002:280, 285) refer to the need for an object-oriented programmer to capture requirements, recognise patterns, model visually and think critically.
- Learning the object-oriented paradigm takes longer than the 'normal' learning curve. Furthermore, problems are harder to solve (Schach, 2005).
- The programming environment can be complex or even confusing (Kölling, 1999:8, 9).

Different notations and models were designed for the object-oriented approach to support the learning of computer programming and these will be discussed in more detail in the next subsection.

3.2.4 Programming notations and models

Various notations and models can be used to help students direct their thinking during OOP. The following programming notations and models will be discussed in more detail: patterns applied in OOP, UML as a graphical notation, and CRC cards.

3.2.4.1 Patterns in object-oriented programming

The concept of patterns was originated by the architect Christopher Alexander, who defined a pattern as a discovered solution that has been tried and tested over a considerable period of time in order to solve problems (Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King & Angel, 1977). In the context of architecture, Alexander et al. (1977) describe the use of 253 recognition patterns as a 'pattern language' to support understanding of the problem domain and to create a design after the problem domain has been understood (Shalloway & Trott, 2002:xviii, xi).

During a presentation in 1996 at OOPSLA (*ACM Conference on Object-Oriented Programs, Systems, Languages and Applications*), Christopher Alexander (Alexander, 1996:3) transferred this architectural concept to the context of OOP and emphasised the relevant features of a pattern language. He pointed out that it:

- has a moral component, that is, it plays a significant role in human life;
- creates morphological coherence in the things which are made; and
- it is generative and allows people to create a complete coherent structure by following certain steps.

These features direct our thinking in OOP towards the creation of software that is significant and dynamic in a way that allows programmers to generate a complete software system by following certain steps (Alexander, 1996:3).

Beck and Cunningham (1989:1) use some of Alexander's ideas (1977) regarding patterns and apply them in the OOP language Smalltalk. In the 1990s, Gamma, Helm, Johnson and Vlissides wrote about 'design patterns', which are solutions to specific problems in object-oriented design. They describe various design patterns in OOP (Gamma et al., 1995:2).

3.2.4.2 UML – an important graphical notation

Grady Booch presents a language-independent notation for documenting the design of a system. Booch (1991:156) designed a standard for expressing object-oriented thinking by means of UML (Unified Modelling Language). UML is mainly a graphical notation used to communicate concepts. It supports object-oriented design by means of various diagrams that are language-independent and no programming code is involved. UML communicates important object-oriented concepts, interactions and behaviours of a system to ensure that the software system is built correctly (Fowler, 2000:5-9). However, UML is not executable and, even when using it, it is sometimes difficult for students to implement design applications in a programming language (Schulte & Niere, 2002:1).

3.2.4.3 CRC cards

CRC cards (Class-Responsibility Collaboration cards) refer primarily to a technique for teaching people how to work with objects. CRC cards help explore the interaction between classes and key responsibilities of a class. Beck and Cunningham used CRC cards to teach Smalltalk (an object-oriented language) (Beck & Cunningham, 1989). The absence of a complex notation makes CRC cards valuable as a technique for learning object-orientation (Fowler, 2000:9, 77).

To summarise, different notations, patterns and models, some of which have been briefly described, are used to support the learning of OOP. Despite these tools, there are still programmers who find it difficult to design and program in the object-oriented domain (Schulte & Niere, 2002:1). An overview of the problem and design spaces in OOP follows.

3.2.5 Problem and design spaces in object-oriented programming

The problem space is a space containing the programmer's mental representations in terms of specific *requirements* of a program (Purao, Bush & Rossi, 2001:2). The design space contains the mental representations of the programmer's interpretation in terms of specific *solutions* to the program (Purao et al., 2001:2). In terms of the problem space in OOP, the programmer needs to understand that the program requires the use of objects that interact with each other by means of message sending. In terms of the design space in OOP, the programmer needs to interpret the program and represent a correct solution for the problem.

Purao et al. (2001:3-5) make the interesting claim that the problem space and design space are not on the same plane and they represent two different dimensions during programming. Cleenewerck (2003:1) believes that students get overwhelmed in the amount of objects and messages and he calls this phenomenon 'lost in object space', as a description of the situation where students complain about the complexity of OOP. The development of a solution needs interpretation, application and expansion of knowledge and skills during programming. Programmers have to learn systematic techniques to develop a program from a conceptual model of the problem domain (Bennedsen & Caspersen, 2004). Support by means of cognitive knowledge and skills can play an important role in programming. The importance of cognition is discussed next, along with the associated knowledge and skills.

3.3 Cognitive knowledge and skills in object-oriented programming

The term cognition includes a wide range of mental processes used in the acquisition, storage, transformation and application of knowledge. Formally, cognition can be defined as the mental action of acquiring knowledge through thought, experience and the senses (Concise Oxford English Dictionary, 2004:278). Cognition focuses on "what learners know and the way they come to know it" (Schunk, 2000:24). Ragonis and Ben-Ari (2005:209) emphasise that the processes and understanding that take place during the execution of a task are more important than the achievements.

Efficient cognitive knowledge and skills are necessary during the processes of problem solving, decision making, critical thinking and reasoning in OOP. Students must understand the knowledge and apply different skills in OOP in order to do so successfully. A plethora of high-level skills is involved in the programming process. However, the focus in this section is on memory, comprehension, reasoning skills, and decision making, as well as on creative and critical thinking. Bloom's taxonomy and different means of support are also discussed.

3.3.1 Memory, comprehension, reasoning, decision making, creative and critical thinking in object-oriented programming

The effective use of memory skills and comprehension is vital during the processes of the learning and writing of computer programs. A brief overview is given of the memory and reasoning skills required in comprehension.

3.3.1.1 Memory and cognitive load

Memory refers to different processes and dynamic mechanisms of storing within, retaining in, and retrieving information from the brain (Sternberg, 2006:157). There are different models of memory; however working memory and long-term memory will be discussed in more detail.

- *Working memory and cognitive load*

  Working memory refers to an integrated part of memory that stores and manipulates information temporarily. It has a multi-functional character in order to combine processing and storage, and to facilitate different cognitive activities such as reasoning, comprehension and learning (Baddeley, 2003:829). Working memory is that memory

which stores incoming information immediately, but for a short time span only (Sternberg, 2006:170). It is thus limited in duration and, with regard to capacity, it can hold only a small amount of information. It is also described as the immediate memory where basic cognitive operations are carried out (Neath & Surprenant, 2003:69). Different areas in the left and right brain hemispheres respectively are involved in the working memory (White & Sivitanides, 2002:62).

Working memory is a basic component of intellectual achievements and higher cognitive functions such as reasoning, language processing and problem solving and it is a critical component of intelligence (Sternberg, 2006:498). Jonides and Nee (2006:181,192) mention that in many cognitive tasks, the major difference between individuals concerns the variation in their working memory capacity and the associated issue of how many items can be retrieved. Yousoof et al. (2006:259) suggest that the processing of information to solve problems also occurs within working memory. Working memory thus plays an important role in the programmer's interpretation of the program (§3.2.5).

The cognitive load of a task is related to the interactivity between various elements within the working memory that can influence the storage of information. Yousoof et al. (2006:262) claim that cognitive load is the core complex area that needs to be addressed during learning to program. Additionally, ill-structured problems (§3.5.1.1) – many OOP problems are ill structured – present "immense cognitive loads" for programmers (Tan, Turgeon & Jonassen, 2001:97). In OOP, different aspects increase cognitive load such as learning the syntax, understanding the semantics, making decisions and designing an algorithm, identifying classes and collaborations, and finally, the creation of a complete program consisting of many objects with specific subtasks (Cleenewerck, 2003). If cognitive load can be managed in a systematic way, then many programming difficulties can be overcome (Yousoof et al., 2006:259).

- *Long-term memory, encoding, retrieval and reconstruction*

  Long-term memory stores information over a long period, but in such a way that it can be recalled. Long-term memory is organised into a network of interconnected nodes of information that structure material into mental images. During the learning process, these structures are modified, as new information is integrated with prior knowledge. It is important that the information is systematically organised in order to improve memory skills (Schunk, 2000:144). The hippocampus is an important part of the brain that is involved in the integration and consolidation of information and the transfer of newly

synthesised information into long-term memory. It plays an important role in complex learning (Sternberg, 2006:187).

The structure of memory changes due to the continual construction and reconstruction of knowledge. Knowledge can be represented with a schema. The schema is dynamic and organises knowledge at different levels and reflects an individual's knowledge, experience and expectations about an aspect (Neath & Surprenant, 2003:264,245). Yousoof et al. (2006:259) and Sternberg (2006:218) mention that the organisation of knowledge in the mental network of an expert is more complex and more extensive than that of a novice. Experts will therefore solve problems more easily than novices (Table 3.5, §3.5.3).

Different processes are involved in the transfer of information to long-term memory. Deliberate attention to information and the creation of associations between new and prior information may increase transfer (Sternberg, 2006:197). Important processing occurs during the encoding and retrieval of information (Neath & Surprenant, 2003:103).

Encoding involves the processing of new information and preparation for storage in long-term memory (Schunk, 2000:143). The manner in which knowledge is encoded determines which retrieval cues will activate such knowledge. Different types of information are also encoded in long-term memory, for example: semantic encoding, visual encoding and acoustic encoding (Sternberg, 2006:168,196). For example, in computer programming, semantic encoding is required for the 'programming word' *length,* which has different meanings for strings and for arrays in Java. In the case of strings, length is a *method,* and in the case of arrays, length is a *field*. Visual encoding is required for the programming types such as *Byte* (starting with a capital letter – a wrapper class) and *byte (*written in lower case – presenting number values*),* which are different in Java, since it is a case-sensitive programming language. Delphi, on the other hand, is not a case-sensitive programming language.

Retrieval refers to the active triggering of associations in memory to recall relevant information (Schunk, 2000:157). Categorisation can effect retrieval of information (Sternberg, 2006:207). For example, the categorisation of different types of control structures (*for, while, switch*) in Java may support the retrieval of information from memory.

3.3.1.2 Comprehension, reasoning, decision making, creative and critical thinking

- *Comprehension*

  Comprehension in programming includes all the activities involved in the writing, modifying and debugging of a program (Zhang, 2005:4, 10). Comprehension is based on the role of concepts. Concepts are units of human knowledge that can be processed in memory. In an object-oriented program, concepts may become possible objects that can be implemented in programming code (Rajlich & Wilde, 2002:271, 272).

  Processes of language comprehension are important in programming and include the syntax and semantics of a language. Syntax refers to the grammar and structure of sentences (Sternberg, 2006:323). In a programming language, syntax refers to expressions, statements and program units. The output of the following Java statements will differ, because of the difference in syntax:

```
System.out.println(1+2+3);          // the answer is the numerical value 6
System.out.println("1"+"2"+"3");    // the answer is the string 123
```

  Semantics refers to the meaning of words and it is necessary to encode meanings through concepts within the human memory. In a programming language, semantics refers to the meaning of expressions, statements and program units (Sebesta, 2004:114). The meaning, or semantics, of a statement is related to the syntax of that statement (Sebesta, 2004:114). For example, the meaning of the following Delphi statement is that program statements should be repeated:

```
For k := 10 downto 1 do
begin
  ...
end;
```

  Since OOP involves more than merely applying the syntax and understanding the semantics, a programmer must think in terms of objects. Which objects exist in the problem domain? What is their behaviour and how will these objects communicate? Different types of reasoning skills are involved in a programming task.

- *Reasoning*

  Reasoning refers to the mental processes involved in problem solving to explain, classify, determine sources, infer and evaluate (Schunk, 2000:288-292). Different types of reasoning can be distinguished: inductive reasoning, deductive reasoning and analogical reasoning. Logical thinking is important during reasoning and Govender and Grayson (2006:1692) emphasise that the ability to engage in sound logical thinking should be a prerequisite for programming.

  Inductive reasoning

  This type of reasoning occurs where general rules are developed from knowledge and previous examples (Schunk, 2000:290). Inductive reasoning involves searching for a rule or extrapolating on an existing rule. For example:

  A *Bus* can transport people
  A *Bus* is a *Vehicle*
  Therefore, a *Vehicle* can transport people

  Inductive reasoning is not always appropriate or accurate, for example, an invalid induction is: a dolphin is a mammal and a dolphin can swim, therefore a mammal can swim. Similarly, in the Bus example above, if the term Vehicle was replaced by Container in lines 2 and 3, it would be an example of flawed reasoning. In OOP inductive reasoning is used to determine general 'rules' to identify objects that can respond to the same message in different ways. This implies polymorphism, which allows more flexibility in the design and provides generic programming (Garrido, 2003:239). For example, a method call 'drive' is interpreted differently by a bus, a bicycle and a train (§3.2.3.3).

  Deductive reasoning

  This is the converse of inductive reasoning. During deductive reasoning, the student proceeds from general concepts to specific conclusions (Schunk, 2000:291). This type of reasoning involves the combination of existing information by specific mental operations. A well-known example of deductive reasoning is a syllogism. A syllogism consists of premises and a conclusion (Schunk, 2000:291) for example:

  All *Buses* are *Vehicles*
  PGK456GP is a *Bus*
  Therefore, *PGK456GP* is a *Vehicle*

In OOP, deductive reasoning is used when possible subclasses inherit attributes and methods defined by the superclass (parent class). The subclass Bus *is-a* Vehicle, implies that the Bus inherits properties and methods from the Vehicle class.

Analogical reasoning

During analogical reasoning, similarities are determined between concepts. For example, a programmer determines similarities and differences between classes and applies them to new experiences. A programmer can abstract a solution from a previous problem and relate that information to a new problem. In the successful use of analogical reasoning, students must be able to extract objects from the problem domain (§3.2.5), compare them to their prior knowledge and recognise similarities between the new program and previous programs (Staats & Blum, 1999:14).

- *Decision making*

  We are constantly making judgements and decisions every day. The word decision refers to the conclusion reached after consideration (Concise Oxford English Dictionary, 2004:768). During decision making, different options are considered one by one and a selection is made according to the acceptability, or the elimination, of different options (Sternberg, 2006:443,444). Boy (2005:2) refers to different levels of decision making. Sometimes immediate decision making is necessary and sometimes decision making may require considerable time when it uses complex cognitive activity and resources to decide on an action. It will take extensive time to program an entire software system to fulfil all requirements. Decision making is important during the programming process, for example, a decision should be made about which type of iteration loop such as *for…* or *while…* would be appropriate for a specific problem.

- *Creative thinking*

  Creativity refers to the process of producing something that is both original and worthwhile (Sternberg, 2006:429,437). The creative individual may develop an idea into a meaningful contribution and with recognised value. Some characteristics of creative people include high motivation, dedication to standards of excellence, self-discipline, many hours of hard work and the application of insight, divergent thinking and risk-taking (Sternberg, 2006:437). In OOP the identification of classes can be classified as a creative process, and the steps during programming, as a logical activity (White & Sivitanides, 2002:62).

- *Critical thinking*

  Critical thinking refers to logical thinking, reasoning, classification and analogies. It includes intricate problem-solving skills and strategies to addresses complex issues and to improve their outcomes. There are no general rules or principles to apply to the solving of ill-structured problems. For such problems, critical thinking skills are necessary (Tan et al., 2001:97). Many OOP problems are ill structured (§3.5.1.1) and there are multiple ways to solve these problems i.e., there are different acceptable solutions. Programmers must work through problems, generate their thoughts, and then analyse these thoughts critically to design accurate, high-quality programs. Tan et al. (2001:97, 98) emphasise the value of argumentation as a process that engages learners in critical thinking. It is an important facet of a problem-solving strategy to provide opportunities for programmers to form arguments for their preferred solutions.

Comprehension, reasoning, decision making, creative and critical thinking involve working through problems, determining appropriate programming variables, selecting statements and types, determining their logical order in a class and/or method, and putting it all together in the design of accurate, high-quality programs. These are definitely not trivial tasks for a novice programmer!

To structure different cognitive abilities involved in OOP, Bloom's taxonomy (Bloom, Krathwohl & Masia, 1973:186-193) will be used, which presents various categories of learning in the cognitive domain. The higher levels relate to forms of thinking that are much harder to master.

3.3.2 Bloom's taxonomy

Bloom et al. (1973) define different types of learning in the cognitive domain. Bloom's taxonomy presents six categories of learning that may also be applied to problem solving. These categories are hierarchically ordered; each level progresses to higher levels of learning. The cognitive domain includes *knowledge*, *comprehension*, *application*, *analysis*, *synthesis* and *evaluation* as shown in Table 3.1 (Grant, 2003:96; Bloom et al., 1973:186-193).

Table 3.1: The taxonomy of Benjamin Bloom et al. (1973)

Category	Description
Knowledge	Specific facts and ways of remembering
Comprehension	Grasping the meaning of material
Application	The use of previously-learned material in new situations
Analysis	The breaking down of material into parts or sub-problems
Synthesis	The combining of parts to form a new or original whole
Evaluation	Judging the value of material

Bloom's taxonomy was revised by Anderson and Krathwohl (2001) to include new developments in cognitive psychology. However, for the purpose of this study, the original taxonomy of Bloom will be applied in the programming domain.

Categories of cognitive learning are important in order to determine the depth and level of cognitive skills required during a task such as computer programming. Bloom's taxonomy (1973:186-193) is used in this study, because:

- its framework, or hierarchical structure, is suitable for determining and evaluating the *range of cognitive abilities* used during OOP (Oliver, Dobele, Greber & Roberts, 2004:227);
- it is a mature way of *analysing the cognitive depth* and *difficulty of learning* or performing a given task (Oliver et al., 2004:227; Xu & Rajlich, 2004:176);
- *problem solving* is one of the *key aspects of programming* and is on the higher levels of Bloom's taxonomy (Govender & Grayson, 2006:1687); and
- it can *provide insight* into differences in problem solving between novices and experts (Zant, 2005).

High cognitive demands characterise the learning of OOP. Table 3.2 shows the application of Bloom's taxonomy to the understanding, designing, coding and testing of an object-oriented program.

Table 3.2: Analysis of cognitive skills in object-oriented programming (synthesised by the author)

OOP Implementation	Cognitive skills	Bloom et al., (1973)
UNDERSTAND - Define and remember facts (Zant, 2005) e.g. define an object. - Understand and interpret a problem. Construct an internal representation of the problem (Matlin, 2002:362).	- Recall information - Understand and interpret a problem.	Knowledge Comprehension
DESIGN - Recall previous knowledge of OOP classes and objects. - Identify possible classes and methods. - Construct a design in the context of familiar designs (e.g. use case diagrams). - Analyse the programming problem and compare possible designs. - Design classes with general properties and methods and design additional methods. - Evaluate a design.	- Recall information - Interpret a programming problem - Use previous designs (diagrams, use cases) - Analyse and compare solutions - Categorise, combine, and modify a design - Interpret and evaluate the proposed design.	Knowledge Comprehension Application Analysis Synthesis Evaluation
CODE - Recall previous designs, understand concepts (Zant, 2005). - Understand and interpret the design. - Construct programs in the context of familiar solutions (Zant, 2005). - Analyse the programming problem in subproblems. - Develop solutions to new and complex problems (Zant, 2005). - Implement exception handling and debugging techniques (Xu & Rajlich, 2004:176). Evaluate the programming solution.	- Recall information - Understand and interpret the design - Use previous solutions in a new program - Analyse and compare solutions - Categorise, combine, design and modify the program - Interpret and evaluate the solution.	Knowledge Comprehension Application Analysis Synthesis Evaluation
TEST - Recall previous programs and their output. - Interpret and judge the program to determine whether the problem was solved correctly. - Apply previous debugging techniques to identify possible errors. - Analyse the solution and check the code rigorously. - Test the output using test data. - Modify the solution if necessary. Implement debugging techniques and evaluate the solution (Xu & Rajlich, 2004:176).	- Recall information - Understand and interpret a solution - Use previous solutions in a new program - Analyse and compare solutions - Categorise, combine, design, modify - Interpret and evaluate the thinking and problem solving.	Knowledge Comprehension Application Analysis Synthesis Evaluation

This taxonomy gives an overview of the depth of cognitive skills involved during programming. As explained, Table 3.2 demonstrates how different levels of Bloom's taxonomy are applied during the understanding, designing, coding and testing stages of an object-oriented program. These are complex processes involving the application of many cognitive skills in order to solve the problem successfully. Note that knowledge is the first skill mentioned in Bloom's taxonomy, however various types of knowledge are required during all the processes of programming.

During the process of understanding a programming problem, knowledge and comprehension skills are needed to understand and interpret the problem. During the design, code and test steps of programming, *all* levels of Bloom's taxonomy are necessary. The importance of Bloom's taxonomy is emphasised by Zant (2005) who claims that it is difficult for a novice programmer to become an expert without progressing through each of the six levels. Xu and Rajlich (2004:176) emphasise that during the debugging of a program, all levels of Bloom's taxonomy are necessary. However, it is important to note that the application of these levels may vary in different programming contexts and at different stages.

Students must understand the program as a whole, but should also apply knowledge and skills to program the logical details of each method and determine the associations between classes. Cognitive skills such as attention, memory, reasoning, problem solving and practical programming skills must be applied in a specific way to solve the problem successfully (§3.3.1, §5.2.3, Program 5.3, Program 5.4). Some practical means of support are given in the next subsection.

3.3.3 Some practical means of cognitive support

Practical means of cognitive support include chunking and the construction of memory diagrams, semantic networks and schemas. These factors are addressed in the bullets following, with the appropriate subcategories of Bloom's taxonomy indicated in parentheses at the end of each point.

- *Chunking*

  Chunking is the meaningful combination of information to improve retrieval of information (Gu, 2005:16; Schunk, 2000:140). In OOP, this could mean that various necessary attributes are combined and encapsulated in a specific object. For example,

a vehicle's registration number may be chunked in a meaningful way as shown in Fig. 3.3: *[Knowledge, Comprehension]*

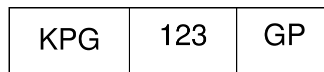


Figure 3.3: A vehicle's registration number, subdivided into chunks

- *Memory diagrams*

  A memory diagram focuses on key points relating to the meaning of a program fragment. It helps students in the understanding of object-oriented programming concepts (Holliday & Luginbuhl, 2003). For example, Fig. 3.4 represents the reference of myName to the String "Peter" (in Java). Using such diagrams, will help students understand the meaning of a 'reference' to objects (Java does not use pointers but rather references to objects). *[Comprehension]*

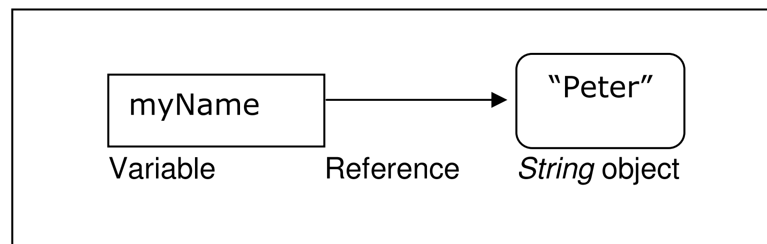


Figure 3.4: An example of a memory diagram

- *Semantic networks*

  A semantic network or semantic net is a graphical notation for representing knowledge of the problem domain by means of interconnected nodes. This can help the programmer to determine specific requirements and possible solutions in the problem space and design space (§3.2.5). A node contains the concept, and links connect nodes with other nodes. A semantic network has three kinds of elements: concepts, relationships that link two associate concepts, and instances of a relation – that is two concepts linked by a specific relation (Huan Keat, 2004).

  A semantic network can support learners in organising their knowledge, reduce cognitive load during problem solving and can support object-oriented learning (Jonassen, 2004:61). It can be implemented in object-oriented design where a network exhibits different relationships. For instance, an *is-a* relationship between concepts can be represented (Huan Keat, 2004:2). Fig. 3.5 shows an example of a semantic network where a Bus *is-a* Vehicle and the Bus also *has-a* steering wheel, has wheels and can

move (§3.2.3.6). These are indications of different relationships between the Bus, a Vehicle, the steering wheel, wheels and the attribute of movement, all portrayed in a single diagram. *[Analysis]*

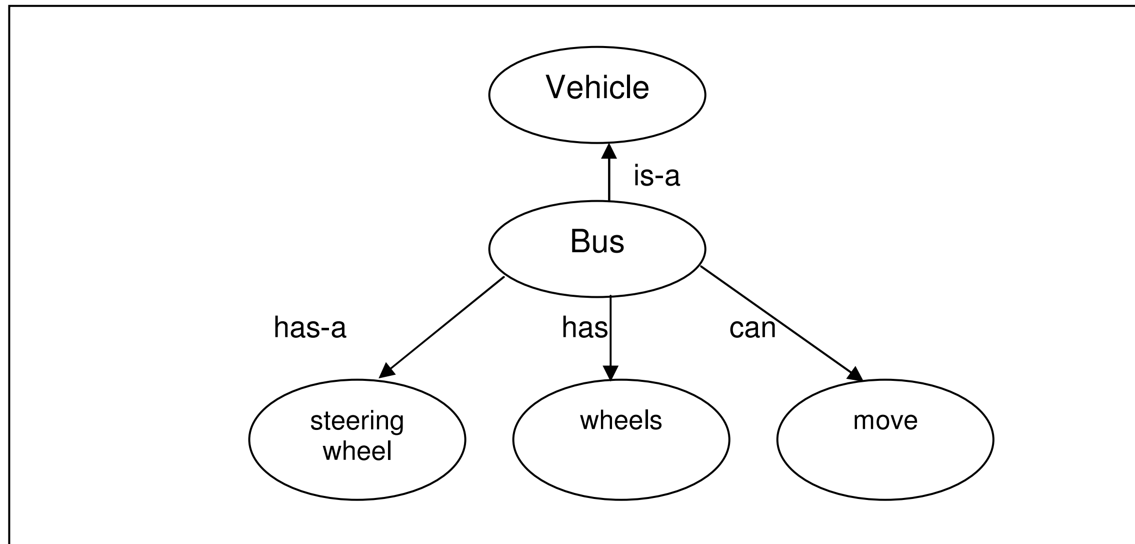


Figure 3.5: An example of a semantic network
(adapted from Mentz, 1998:73)

- *Schemas*

  A schema organises and represents information meaningfully (Schunk, 2000:145). Different schemas can be used to represent object-oriented information. An application domain schema represents the objects and functions for a specific problem. Object schemas include structural details of an object (attributes and methods) (Détienne, 2002:60, 61). *[Analysis, Synthesis]*

This section overviewed cognitive knowledge and skills during programming. The importance of working memory and cognitive load, long-term memory, comprehension, reasoning, decision making, creative and critical thinking were emphasised. Bloom's taxonomy (Bloom et al*.,* 1973) was used to structure categories of learning in the cognitive domain with reference to the programming domain. Moreover, some practical means of cognitive support during programming were given. In the next section, the importance of metacognitive knowledge and skills is outlined.

3.4 Metacognitive knowledge and skills in object-oriented programming

The term metacognition refers to the conscious planning, control and evaluation of one's own cognitive processes, such as thoughts, memories and actions that engage in learning processes (Shimamura, 2000:313; Sternberg, 2006:197). It relates to the knowledge, skills and strategies that are required for a specific task, as well as how and when to apply these activities to complete the task successfully (Schunk, 2000:180-181). Metacognitive activity includes awareness by learners of the strengths and weaknesses of their abilities and the management of learning and task development, particularly in a complex environment such as OOP (Gravill et al*.,* 2002:1055).

Metacognition consists of metacognitive knowledge and metacognitive control of experiences (Flavell, 1979:907-909). Metacognitive knowledge refers to the acquired knowledge about cognitive processes, while metacognitive control refers to the use of different metacognitive strategies.

Since metacognition plays a critical role in successful programming (Gravill et al., 2002:1055; Staats & Blum, 1999:13), it is important to study metacognitive activity to ensure that students can better apply and control their cognitive resources. The focus in this section is on general metacognitive knowledge and skills, metacognitive knowledge and skills in OOP and, finally, different means of metacognitive support. Metacognitive control by means of different strategies will be addressed in Chapter 4.

There is a relationship between metacognition and working memory (Shimamura, 2000:315). Working memory is involved in the processes of temporary activation and storage of information and these processes must be controlled by means of selection and manipulation of information. The frontal cortex is an important part of the brain that contributes to metacognition (Shimamura, 2000:315).

3.4.1 Metacognitive knowledge in general

Metacognitive knowledge, skills and strategies improve performance, and students should learn to apply these activities in specific situations (Schunk, 2000:184).

Metacognitive knowledge refers to explicit knowledge about an individual's own cognitive strengths and weaknesses, beliefs, conditions and variables that affect memory performance, encoding and retrieval strategies. It also addresses the effects of all these on learning and remembering (Koriat, 2002:267). Metacognitive knowledge involves knowledge of a person, knowledge of a task and knowledge of different strategies (Flavell, 1979), each of which is considered in the points following.

- *Knowledge of a person*

  The programmer must have knowledge of himself. This may include:
  - motivation;
  - intrinsic goal orientation (Bergin et al., 2005:82; Panaoura & Philippou, 2005);
  - interests, strengths, weaknesses (Gravill et al., 2002:1055);
  - level of expertise in a specific domain (Ertmer & Newby, 1996:4);
  - prior knowledge and previous experiences (Ertmer & Newby, 1996:8);
  - judgement and beliefs about personal learning (Ertmer & Newby, 1996:6); and
  - the learner's understanding of his own memories and the way he learns (Panaoura & Philippou, 2005).

- *Knowledge of a task*

  Learners must have knowledge regarding the task they are undertaking. This includes knowledge about the characteristics of a memory task that make it easier or harder (Schneider & Lockl, 2002:227). It is also important to estimate knowledge about a task. Gravill et al. (2002:1055) refer to the importance of correct estimation: without effective metacognitive knowledge, over-estimation or under-estimation of specific tasks may occur. Ertmer and Newby (1996:5) found in their research that experts tend to apply metacognitive knowledge and strategies during a task and are more aware of the conditions under which such knowledge and actions should be applied.

- *Knowledge of strategies*

  Knowledge about metacognitive strategies is needed in order to plan different strategies. These strategies will be discussed in Chapter 4.

3.4.2 Metacognitive knowledge in object-oriented programming

Metacognition is an important component in problem solving (Hammouri, 2003:576) and this could be one of the reasons for performance differences in OOP. Staats and Blum (1999:13) emphasise that understanding the object-oriented paradigm requires an increase in different types of cognitive and metacognitive skills. This subsection addresses the specific application to OOP of Flavell's (1979:906-911) categories of metacognitive knowledge, which were introduced in the previous subsection.

- *Knowledge of the programmer*

  Factors such as the type of personality, strengths and weaknesses in problem solving, interest in computer programming, study possibilities, employment opportunities in the field of computing, and different learning styles can affect the way a student understands new material. It may also affect the student's performance in computer programming (Grant, 2003:97). For example, the right-brain hemisphere deals with the creative and artistic processing, whereas the left hemisphere deals with the logical processing (White & Sivitanides, 2002:62). In OOP, the identification of classes is more creative (right hemisphere) and the steps during programming are more logical (left hemisphere).

- *Knowledge of the programming task*

  Prior knowledge of the programming task, level of expertise and previous OOP experience may influence a student's behaviour and the ability to complete a new program successfully. Knowledge about the cognitive processes involved in understanding, designing, coding and testing computer programs fosters successful programming (Table 3.2). Previous experience and knowledge about former programming tasks will make planning for a new task easier. Another factor is that novices pay considerable attention to debugging computer programs. They lack in-depth knowledge about programming and try to solve errors by means of repetitive debugging efforts. Carbone, Mitchell, Gunstone and Hurst (2002:2) suggest that novices should explicitly aim to use metacognitive skills in the initial design of a computer program.

- *Knowledge of programmer's own strategies*

  Programmers must have knowledge about strategy-selection mechanisms, and strategy execution. This is addressed further in §4.5.1 in Chapter 4.

Different forms of metacognitive knowledge and skills are important during the understanding, designing, coding and testing of a program. Various practical approaches to metacognitive support are discussed in §3.4.3 and §4.4.

3.4.3 Some practical examples of metacognitive support

In this subsection, various practical examples of metacognitive support within the complex domain of learning to program are discussed. As in an earlier subsection, the different types of metacognitive knowledge are indicated in parentheses at the end of each point.

- *Journal or reflective diary*

  The purpose of a journal is to give learners the opportunity to make brief notes about their feelings, frustrations, expectations, goals and problems in specific aspects of programming. This is an effective way to reflect on the programming task. Students may use a weekly diary to reflect upon their plans during the week (Fekete, Kay, Kingston & Wimalaratne*,* 2000:145). *[Knowledge of the programmer, knowledge of a task]*

- *Goal setting, outcomes and self-motivation*

  Goal setting and self-motivation can contribute to effective learning during programming (Gravill et al.*,* 2002:1063). A goal reflects the purpose of doing a task and refers to the quantity and quality of performance, whereas goal setting refers to establishing an objective to serve as the aim of one's actions (Schunk, 2000:100, 104). Outcomes expectations are personal beliefs about the outcomes to be achieved, and motivation is the process of goal-directed behaviour with the aim of succeeding in a task (Schunk, 2000:106, 334). The novice programmer should be motivated and set various goals related to the completion of a programming project. *[Knowledge of the programmer, knowledge of a task]*

- *Metacognitive scaffolding*

  Metacognitive scaffolding refers to ways of supporting students and providing guidance during the learning process (Xun & Land, 2004:5, 12-13). It involves providing just enough support to novice programmers so that they will use the processes, yet still be able to perform when the support is removed, as they gravitate towards independent programming. In other words, programmers should be able to develop and use their knowledge and skills to generate programs on their own when the scaffolding support is

removed. For example, the methods required in a new class can initially be given to novices to direct their thinking. However, they should subsequently be able to identify their own methods in a new class without any support. *[Knowledge of a task]*

To summarise this section, metacognitive activity includes awareness by learners of the strengths and weaknesses of their own abilities in developing an object-oriented program. Finally, they should demonstrate the correctness of their own programs. Many hours of practicing programming will enhance the development of such metacognitive processes.

Necessary knowledge and skills in problem solving will be discussed in the next section.

3.5 Problem-solving knowledge and skills in object-oriented programming

Problem solving means "finding a way out of difficulty" (Polya, 1981:ix) or "finding an answer" (Concise Oxford English Dictionary, 2004:1374). Solving problems, sometimes difficult ones, is part of daily decision making. However, problem solving is a complex cognitive process where various possible solutions must be identified in order to select the best one to achieve a goal (Schunk, 2000:191). Since the process is complex, students require support during this process. Different kinds of problem situations require different strategies and interactions. This section considers factors that relate to the level of difficulty of programs, steps in problem solving, expertise and problem solving and some practical means of support during problem solving.

3.5.1 Factors that relate to the level of difficulty of problems

Some problems are more difficult to solve than others. Jonassen (2004:3-9) mentions four factors that relate to the level of difficulty of problems. Problems vary according to their structuredness, complexity, dynamicity and domain specificity. Each will be outlined in more detail.

3.5.1.1 The structuredness of problems

A well-structured problem is a problem in which the goal is clearly stated and all the information required to solve it is present. However, in an ill-structured problem there is uncertainty about which concepts are necessary to solve the problem. This kind of problem

is complex, open-ended, not well defined, and some of the information necessary to solve it is missing or there are several possible solutions to the problem (Xun & Land, 2004:5,7; Jonassen, 2003b; Ormrod, 2003:280; Tan et al., 2001:97). Moreover, the problem space (§3.2.5) of ill-structured problems may be larger than that of well-structured problems and there may be multiple problem spaces available (Xun & Land, 2004:8).

Students learn to represent well-structured problems, but fail to solve ill-structured problems. Solving ill-structured problems poses its own set of cognitive and metacognitive requirements to the problem solver (Xun & Land, 2004:8). Sternberg (2006:406) suggests that ill-structured problems can be described as insight problems. This means that learners should restructure their representation of the problem to be solved. An example of an ill-structured problem in OOP is the requirement mentioned in §2.7.2 of this study, that students write a *Date class* program to include some of the many possible calculations with dates. At the very least, this program had to determine leap years and the difference between two dates, and there are many other possible calculations that an enterprising problem solver could think of. This is an open-ended and real-world problem that is difficult to solve.

3.5.1.2 The complexity of problems

Jonassen (2004:67) classifies complexity of problems by the number of issues, functions or variables involved in the problem. In the context of solving OOP problems, the student should know which objects are relevant and which behaviours and interactions exist between these objects. These various dimensions make decisions difficult.

3.5.1.3 The dynamicity of problems

The dynamicity of a problem indicates whether it is impacted by a change in conditions. For example, changes in one factor may impact on others, causing variable changes, in turn, in other factors (Jonassen, 2004). When the problem itself is changing, a continuous understanding and searching for new solutions is necessary (Jonassen, 2003b:5). In OOP, the state of an object may change due to the receiving of messages. Memory allocation, exception handling and threads are also examples of programming activities that may result in a change in programming conditions.

3.5.1.4 The domain specificity or context of problems

Problems also differ according to their domain specificity (Jonassen, 2003b:6). OOP has a complex domain and particular difficulties occur due to the requirement to define and use complicated structures or operations, such as classes and methods. Objects have the

responsibility of carrying out specific tasks to solve the problem (Garrido, 2003:26-27). Objects should therefore be programmed with the necessary functionality to implement the appropriate methods.

There are many factors that make programming a complex task for novice programmers who then tend to make errors. Possible causes of errors in programming are listed in Table 3.3, structured into categories according to the three main themes of this chapter, namely: cognitive-, metacognitive- and problem-solving issues.

Table 3.3: Causes of errors during programming

Cognitive issues that lead to programming errors

- There is a major cognitive load involved in programming (Tan, Biswas & Schwartz, 2006:828, 830; Yousoof et al., 2006:262).
- Novices lack the problem domain knowledge to understand the problem and hence to propose a solution (Traynor & Gibson, 2004:2).
- Novice programmers have limited knowledge (Ala-Mutka, 2004:2).
- Novices have difficulty with logical thinking (Chmura, 1998:56).
- Students have difficulty developing a strategy for decomposing a problem into subproblems or components (Keefe, Sheard & Dick, 2006:1).
- Students do not fully understand the intricate mechanisms of programming, such as parameter passing (Keefe et al., 2006:1).
- Novices have problems in implementing abstract programming techniques in different situations (Keefe et al., 2006:1).
- Students do not understand OOP concepts (Keefe et. al., 2006:1).
- Students struggle to decompose the full program into smaller steps (Chmura, 1998:56).
- Certain topics are isolated and consequently little is known about them (Schulte & Niere, 2002:1).

Metacognitive issues that lead to programming errors

- Novice programmers fail to reflect and articulate on the abstract object-oriented principles (Schulte & Niere, 2002:2).
- Students should reflect on the correctness of their programming and should submit a broad range of test cases (Edwards, 2004:26).

Problem-solving issues that lead to programming errors

- Programming students confuse problem solving with coding (Traynor & Gibson, 2004:2).
- Students do not think in object structures (Schulte & Niere, 2002:1).
- There is a gap between the abstract approach and the hands-on knowledge necessary to implement the program (Schulte & Niere, 2002:2).
- There are many hierarchical levels of complexity (Pressing, 1999:3).
- The functionality of object-oriented programs is distributed over many objects, each with specific subtasks to perform (Cleenewerck, 2003:2).
- Certain problems vary according to the level of dynamicity involved (Jonassen, 2003b).
- There is a tendency to develop a fixation on a particular type of problem, with the result that students lack the ability to solve new problems in different ways (Sternberg, 2006:412).

3.5.2 Steps in problem solving

Different kinds of problems may be solved differently. Students can, however, be guided by means of specific steps to support the particular problem-solving process in hand. Xun and Land (2004:8) identify the major processes for problem solving in ill-structured domains as problem presentation, generation and selection of solutions, making justifications, and monitoring or evaluating solutions. However, Deek, Turoff and McHugh (1999:332-335) focus explicitly on the problem-solving models used in programming and propose an integrated methodology that combines problem solving and programming. They identify the following steps in their problem-solving model: formulating the problem, planning the solution, designing the solution, translating the solution, testing the solution and delivering the solution. The following problem-solving steps are selected to be discussed in more detail: problem understanding, program designing, program coding and program testing (Table 3.2).

3.5.2.1 Problem understanding

During the process of reading a problem description, the programmer uses memory and skills to form concepts. Concepts are units of knowledge that can be processed within working memory. Understanding involves asking questions to fully comprehend the problem, that is, to determine the unknown or to define exactly what is being asked in the question (Schunk, 2000:195). Problem understanding implies the construction of an internal representation of the problem (Matlin, 2002:362). During problem understanding a decision must be made about the relevant information that will be required to solve this particular problem.

3.5.2.2 Program designing

Appropriate representation of the problem is important and therefore students must construct a meaningful conceptual framework of the problems as a foundation for design. It is necessary for learners to organise and display problems to enhance their mental representations and problem-solving processes (Jonassen, 2003a:366). The more students are able to represent and model problems, the better they will be able to transfer those skills to well- and ill-structured problems (Jonassen, 2003a:364). OOP requires a clear decision about what objects, behaviours and interactions are needed (§3.2.3). Program design in this context may involve design of the problem domain (e.g. classes and objects), operating environment (e.g. database management, persistent objects and SQL), and the user interface (e.g. the graphical user interface, menus, buttons) (Satzinger & Ørvik, 2001:143-146).

3.5.2.3 Program coding

Coding a solution involves the use of logical reasoning and deduction, as one applies the language syntax and programming constructs and synthesises a whole new program to solve the problem (Stamouli & Huggard, 2006:109-118; §3.3.1.2). In the process of coding the solution, knowledge and skills relating to one kind of problem can be transferred to another. However, transfer can be either negative or positive. Negative transfer occurs when the experience of solving a previous problem makes it harder to solve a subsequent one. Positive transfer occurs when solving a previous problem makes it easier to solve the new one (Sternberg, 2006:413). The various diagrams generated during the design process must be interpreted and 'translated' into programming code, as the different forms of knowledge and practical skills are applied in the context of a specific programming language.

3.5.2.4 Program testing

Program testing refers to checking the correctness of the solved problem. Stamouli and Huggard (2006:113) mention different categories for ensuring program correctness: syntactical correctness, functional correctness, design correctness and input/output validation and performance correctness. These are shown in Table 3.4.

Table 3.4: Categories for ensuring program correctness
(Stamouli & Huggard, 2006:114)

Category	Description
Syntactical correctness	A program is syntactically correct when it compiles without errors.
Functional correctness	A program should fulfil the requirements of the problem specification.
Design correctness	A program should be correctly structured to enable extensibility.
Input/output validation and performance correctness	A program should cater for invalid input and should be optimised in terms of length and execution time.

Finally, practical skills are required during program testing to debug a computer program and understand error messages (Chmura, 1998:56). The best predictor skill to use during program testing is experience. Prior exposure to solve similar problems will support skills during program testing (Jonassen, 2004:13).

To support the understanding of successful problem solving, various differences between experts and novices in OOP are discussed in §3.5.3.

3.5.3 Level of expertise and problem solving

A novice programmer is one who is undergoing the process of learning the required knowledge and skills for programming, while an expert programmer is one who successfully applies knowledge and skills to solve a programming problem effectively (Govender & Grayson, 2006:1689). Experts and novices differ in the way they solve problems. Some of the abilities and approaches of experts are indicated in Table 3.5, structured (as in Table 3.3) under the main themes of this chapter, namely cognitive-, metacognitive- and problem-solving knowledge and skills (specifically in the context of programming) (§5.2.3). Note that the cognitive categories are ordered according to levels of Bloom's taxonomy as indicated in parentheses.

Table 3.5: Examples of expertise during problem solving

Cognitive knowledge and skills

- Expert programmers possess a well-organised, carefully-learned knowledge structure (Ala-Mutka, 2004:2; Glaser, 1999:91-92) [*knowledge].*
- Experts perceive large, meaningful patterns that guide their thinking with rapid pattern recognition (Glaser, 1999:91-92) *[knowledge, comprehension].*
- Experts rely on the recall of re-organised material, forming meaningful chunks (Yousoof et al., 2006:259; Matlin, 2002:136; §3.3.3) [*knowledge, comprehension].*
- Experts apply their knowledge to the goal structure of the problem (Glaser, 1999:91-92) *[application].*
- Experts work through different levels of abstraction and some identify further sub-problems (Rosson & Alpert, 1990:349) *[application, analysis].*
- Experts can decompose a large procedure into smaller units (Or-Bach & Lavy, 2004:84) [*analysis].*
- Experts are better at reconstructing missing portions of information from material that is partially remembered (Matlin, 2002:136) *[synthesis].*
- Experts show high accuracy in solving the problem and reaching the outcomes (Sternberg, 2006:426) *[evaluation]*.

Metacognitive knowledge and skills

- Experts use selective memory search strategies and effective metacognitive processes (Rosson & Alpert, 1990:349).
- The experts' knowledge enables them to use self-regulatory processes with great skill – they monitor their own problem-solving activities (Glaser, 1999:91-92).

Problem-solving and programming knowledge and skills

- Experts can solve a problem quickly, although they often appear to spend more time in problem representation (Sternberg, 2006:424).
- Experts define methods with associated classes, whereas novices tend to define classes first and thereafter methods (Pennington, Lee & Rehder, 1995:210).
- Experts' problem-solving skills entail selective search of memory – they have fast access pattern recognition and representational capability (Sternberg, 2006:424).
- Experts can access multiple possible interpretations (Sternberg, 2006:424-425).
- Experts apply various problem-solving strategies and plan sufficiently (Deek, 1999:2).

3.5.4 Some practical means of support during problem solving

There are various practical means of support during problem solving, some of which have already been mentioned in §3.3.3 and §3.4.3. This subsection emphasises applying program documentation, understanding example programs, using help support, using the Internet, understanding a program and completing partial solutions, using trace tables, creating test data and using pair programming. Note that the steps in problem solving are indicated in parentheses.

- *Program documentation*

  This includes comments in the program that explain and highlight the purpose of a program or program segments. Good documentation also serves to ease the process of program maintenance. Making a program more understandable and more readable, supports maintenance, testing and debugging tasks. Nurvitadhi, Leung and Cook (2003:13) found in their research that method comments in Java increase low-level program understanding. This facilitates the task of reusing and extending by means of inheritance (Nurvitadhi et al., 2003:16). An example of program documentation is indicated by '//'; for example:

  import javax.swing // the JOptionPane class *[Problem understanding]*

- *Example programs*

  The use of example programs supports understanding and problem solving within the programming process. Such examples can be used as templates to support thinking during the programming of a new problem. *[Problem understanding]*

- *Help* support

  Most development environments include a Help facility. Delphi's *Help system* contains extended descriptions and example code, and educators may regard this as a source of many potential learning opportunities. Searching for help by using the *OnClick*-event handler will, for example, provide the student with a full example of such an event handler, explain its association with actions such as the *OnExecute*-method and describe how and when to use it. *[Program coding]*

- *The Internet*

  Learners can download various examples of similar problems and study those examples explaining the program code. Many websites are available to support the learning of programming, examples being:

  http://delphi.about.com/ and http://java.sun.com/ *[Program coding]*

- *Understanding a program and completing partial solutions*

  Learning support can be provided by supplying partially completed solutions to a program and requiring the students to complete the missing parts (Yousoof et al., 2006:260). For example, students could receive a partially completed Delphi or Java solution and be required to complete a method in order to sort different bus routes according to the *driverNumber*. *[Program coding, Program testing]*

- *Trace tables*

  A trace table can be used to track the order of program execution line-by-line. *[Program coding, Program testing]*

- *Test data*

  Students should be able to explicitly demonstrate the correctness of their programs (Edwards, 2004; Or-Bach & Lavy, 2004:85). This can be done by using test data and submitting test cases. Students must explicitly learn a testing approach. Edwards (2004:27) refers to the practice of test-driven development, a technique whereby students write test cases before proceeding with new code. This can increase a student's confidence and comprehension of the programming process and also provide positive feedback. *[Program coding, Program testing]*

- *Pair programming*

  To support comprehension, students may program together in collaborative pairs, which is known as pair programming. In a study by McDowell, Werner, Bullock and Fernald, (2002:38), it was found that students who programmed in pairs, produced better programs and completed the course at higher rates than students who programmed independently. Pair programming is a programming style in which two programmers work together continuously on the same program to solve the problem (Williams, Wiebe, Yang, Ferzli & Miller, 2002:197). One participant is the *driver* who operates the keyboard and the other participant is the *navigator* who constantly reviews code and watches for potential upcoming problems during programming of the solution. From time to time, participants should swap roles within a pair. Keefe et al. (2006:8) emphasise that pair programming is successful when both students participate actively in the process. During pair programming students can improve their problem-solving skills, the quality of programming code and the testing of their programs (Keefe et al., 2006:2, 3). Students working together should understand the program as a whole and should apply various programming skills in a specific way to help each other to solve the problem successfully. *[Program coding, Program testing]*

3.6 Chapter conclusion

Learning to program is a complex and multi-dimensional task. Particularly, in an object-oriented domain it involves the application of differing knowledge and skills during the problem-solving process. The better we understand the problems that can potentially occur during programming, the better we can teach programmers which skills to use and how to apply them in generating an object-oriented program.

This chapter has given an overview of various aspects of object-oriented programming (§3.2). Different aspects of programming were addressed with reference to cognition, metacognition and problem-solving knowledge and skills in the object-oriented domain. The cognitive requirements of the programming process were discussed with reference to the role of memory, thinking, reasoning in OOP (§3.3); and the importance of Bloom's taxonomy in OOP (§3.3.2). In the section on metacognitive knowledge and skills, the importance of self-knowledge, task-knowledge and strategy knowledge was emphasised (§3.4). In addition, various types of problems and different steps in problem solving were discussed (§3.5).

Furthermore, some guidelines and practical means of cognitive support (§3.3.3), metacognitive support (§3.4.3), and support during problem solving in OOP (§3.5.4) were pointed out.

Students can use strategies to help them to achieve specific goals. Chapter 4 builds on Chapter 3 and will correspondingly focus on the role of cognitive, metacognitive and problem-solving strategies during OOP.

4 Cognitive, metacognitive and problem-solving strategies in object-oriented programming

4.1 Introduction

Chapter 3 outlined the roles of cognitive, metacognitive and problem-solving *knowledge and skills* in object-oriented programming (OOP). This chapter builds further on this foundation by discussing specific cognitive, metacognitive and problem-solving *strategies* that provide support to students during the learning of OOP.

A *strategy* is a designed plan to achieve a specific purpose in the long term (Concise Oxford English Dictionary, 2004:1425, §1.3). It is a set of procedures to accomplish a cognitive task and involves putting different skills together to achieve a specific outcome (Kirkwood, 2000:512; Lemaire & Fabre, 2005:12; Schunk, 2000:382). Gu (2005:1, 6, 10, 16) suggests that a strategy is a dynamic process with problem solving as its aim. It involves different activities such as reflecting on the progress and outcome of a task. Strategies are an important part of the learning experience (Filcher & Miller, 2000:61).

The focus in this chapter is on cognitive, metacognitive and problem-solving *strategies* and their application in OOP. These three strategies are addressed in Sections 4.3, 4.4 and 4.5 respectively. Fig. 4.1 is a follow-up of Fig 3.1. The shaded blocks in Fig. 4.1 indicate various goals and strategies that are explored in this chapter and their application in OOP.

Furthermore, various guidelines and practical ways of using strategies during the learning of OOP, are discussed in some detail in various parts of the chapter.

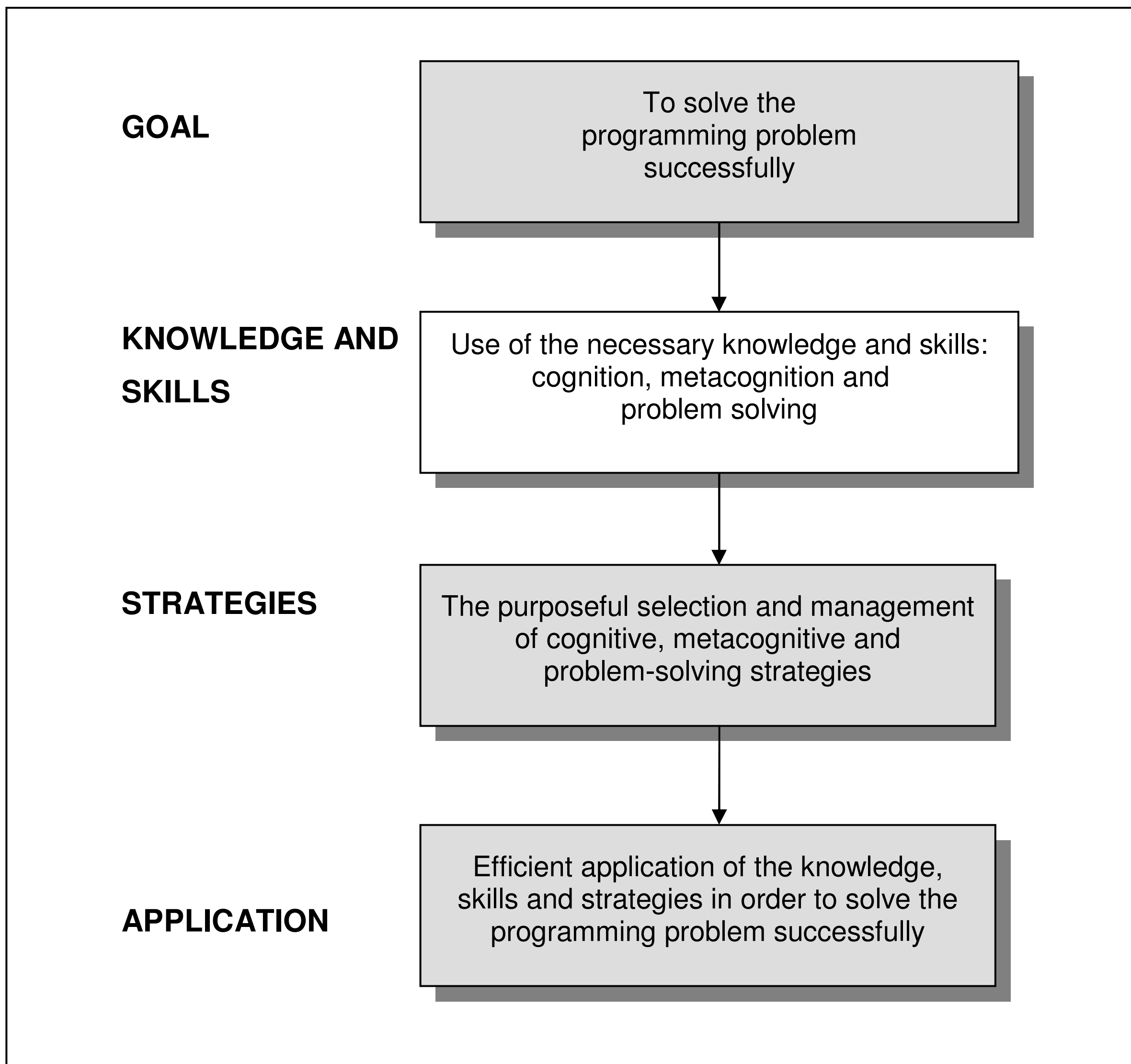


Figure 4.1: Various goals, knowledge, skills, strategies and their application in an object-oriented program

4.2 Strategic aspects of performance

Various factors influence the use of strategies. Lemaire and Fabre (2005:15) propose a conceptual framework for investigating strategic aspects of performance and refer to strategy repertoire, strategy execution and strategy selection. In addition, Gu (2005:9-10) refers to strategy performance that can influence the task. A brief overview of each of these aspects will be given with reference to this study.

- *Strategy repertoire*

  The strategy repertoire refers to the sum of all strategies that may be involved in a specific task. Programming is complex and the application of multiple strategies is necessary. Gu (2005:7) mentions that, because of the dynamic nature of strategies, the ideal strategy is made up of metacognitive and cognitive strategies. However, problem solving is a major component of the programming process, so problem-solving strategies are also included in this study. Therefore, the repertoire of strategies in this study includes cognitive, metacognitive and problem-solving strategies appropriate for OOP (Fig. 6.5).

  It is difficult in specific tasks to 'see' specific evidence of strategic use and people are sometimes unaware of the strategies that they use (Lemaire & Fabre, 2005:17). However, by analysing the computer programs and thinking processes of different students, inferences can be made with regard to the strategic approaches students are using that could explain their variations in performance (§5.2).

- *Strategy execution*

  Strategy execution focuses on the actual use of strategies and also considers how rapidly and accurately programmers perform (Lemaire & Fabre, 2005:15). Fast and accurate programming performance is required during a practical programming test or in an examination. During strategy execution, the retrieval of data is necessary. In the process of organising information during learning, the use of strategies may support retrieval of knowledge (Sternberg, 2006:207, 221).

- *Strategy selection*

  Strategy selection addresses how people choose between strategies for a given task. Selection is influenced by the complexity of the problem, as well as by the awareness and use of different strategies and expertise (Lemaire & Fabre, 2005:15). Furthermore, the selection of strategies can explain individual and situational differences in a task (Morris & Schunn, 2005:33). Various strategies may be used in combination for a programming task and it is therefore difficult to determine precisely the effectiveness of a strategy. However, learners tend to choose strategies that they believe will result in the most effective performance (Roberts & Newton, 2005:132).

- *Strategy performance*
  Strategy performance depends on factors such as the learners (their values, skills, attitudes, prior knowledge and motivation), the programming problem (structuredness, complexity and dynamicity, domain specificity (§3.5.1)) and the learning environment, such as the explicit teaching of strategies (Gu, 2005:12). When sequencing specific activities repetitively in various situations and tasks, experts preserve the effect and take up less working capacity (Gu, 2005:9). Through practice, experts may automate various processes during the application of different strategies. Experts form rich organised schemas of information and consolidate sequences of steps into routines that require little control, thereby freeing their working-capacity to better monitor their performance (Sternberg, 2006:425).

The next section will discuss cognitive strategies, while §4.4 overviews metacognitive strategies, and §4.5 addresses specific problem-solving strategies, all in the context of OOP. These sections will give an overview of the strategies, as well as discussing their role and application in OOP. In addition, some practical means of implementing these strategies will be discussed in each section.

4.3 Cognitive strategies

Section 3.3 introduced the concept of cognitive knowledge and skills in OOP. This section builds on that concept by addressing specific strategies. A cognitive strategy is a plan for orchestrating cognitive resources efficiently in a way that is goal-directed, actively selected and situation-specific (Schunk, 2000:382; Weinstein & Meyer, 1991:17). Cognitive strategies help us to remember, select and organise information within memory (Schunk, 2000:382).

The hippocampus in the brain is the part of memory that acts as a rapid learning system. It temporarily maintains new experiences until they are assimilated more permanently (Sternberg, 2006:100). However, working memory is limited in duration (§3.3.1.1) and information is lost if not learned well. The recall of information can be improved by cognitive strategies, which maintain information in working memory (Schunk, 2000:139-140). Strategies represent sequences of cognitive knowledge and skills that should produce a response that takes up less working memory capacity and requires less time to perform (Gu, 2005:9; Hertzog & Robinson, 2005:112; §3.3.1). The cognitive strategies that will be discussed are rehearsal, elaboration and organisation (Bergin et al., 2005:82).

4.3.1 Rehearsal strategy in object-oriented programming

Rehearsal strategies involve using different techniques to support memory activities. Kayashima, Inaba and Mizoguchi (2005) claim that rehearsal is vital for maintaining content in working memory. It involves "keeping information active" in memory and supporting learners in selecting important information (Sternberg, 2006:197; Bergin et al., 2005:82; §3.3.1). Rehearsal strategies are useful for learning in complex domains, such as programming, but they entail more than the mere repetition of information. During the process of rehearsal, the programmer attends to and selects important information from text (Bergin et al., 2005:82). When students use specific programming constructs repeatedly during practice, for example:

the ***for...*** or ***do...while*** iteration

the syntax of these constructs is remembered more easily. Another example of the application of the rehearsal strategy occurs when the programmer reads and rereads a problem description to select possible objects. If this task is done repetitively, the details are grasped and the selection of objects becomes more trivial.

Some practical means of implementing the rehearsal strategy are paying focused attention, as well as distributed practice and using reminders:

- *Focused attention*

  Attention refers to the process of focusing on a limited amount of information from the enormous amount of information available through various cognitive processes (Sternberg, 2006:62). The programmer should concentrate and maintain awareness of the programming problem in hand. The problem description should be read and reread continuously to support selection of the possible objects (§3.2.3). In this way, the programmer can actively focus on specific information to answer the question and actively plan future programming actions.

- *Distributed practice*

  Distributed practice refers to learning sessions, which are spaced over time. Memory is less effective when learning sessions are crammed together in a very short space of time (Sternberg, 2006:198, §6.4). This implies that students should have regular programming sessions at reasonable intervals to optimise their learning and to enhance their application of programming skills in various contexts.

- *Using reminders*

  Reminders are often used as external memory aids to increase the likelihood that people will remember specific information (Sternberg, 2006:202). When typing the word *button1.* in an event handler, Delphi responds by displaying available properties and methods (procedures and functions) from which the user can select, as shown in Fig. 4.2.

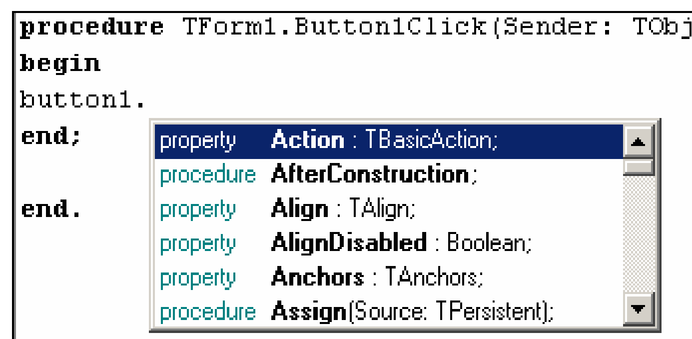


Figure 4.2: An example of a reminder in Delphi

4.3.2 Elaboration strategy in object-oriented programming

Elaboration strategies help learners to integrate new information with prior knowledge, move information into long-term memory, and make it more meaningful (Sternberg, 2006:199) (§3.3.1). By keeping information active in working memory, elaboration increases and supports the permanent storage of information in long-term memory (Schunk, 2000:156; Vögele & Wild, 2003:3). Elaboration strategies also support the encoding and retrieval processes, because they link to prior knowledge (Schunk, 2000:144). General elaboration strategies include: questioning, note taking (Schunk, 2000:384), creating analogies and summarising (Bergin et al., 2005:82), all of which expand information by adding something new to make it more meaningful. During elaboration, a programmer must connect new information to prior knowledge. For example:

> Application of the *FloatToStr* function (real number conversion to the String type) in a Delphi program, when the programmer has existing knowledge about the *IntToStr* function (integer number conversion to the String type).

Elaboration strategies also include the use of analogies between different programming solutions. Successful analogical use implies the creation of mental models whereby students extract objects from the problem, compare them to their personal knowledge base, and recognise relevant similarities between the new problem and previous problems solved (Staats & Blum, 1999:14).

Some practical approaches include generative note taking, summarising, asking questions, and creating analogies. These approaches are discussed in more detail.

- *Generative note taking*

  Generative note taking involves more than copying information. Rather, generative note taking should elaborate, integrate information and focus on the learning outcomes (Bergin et al., 2005:82; Schunk, 2000:387). This means that a programmer should take notes and construct meaningful detail about good programming practice and style in a way that refers to syntax, grammar style, semantics, documentation and prompt lines (Malik & Nair, 2003:71).

- *Summarising*

  In summarising, a learner may use his own words to emphasise the main ideas in text (Schunk, 2000:385) and to present information concisely (Bergin et al., 2005:82). Summaries should be of restricted length, and should include important points only. For example, students can summarise the benefits of the object-oriented approach and may note points such as how to enhance the quality, productivity and flexibility of information systems. During summarising, the student must understand the content of a problem and explain it briefly in his own words.

- *Asking questions*

  Questioning is an important strategy to achieve higher-order learning outcomes (Schunk, 2000:386). Learners should ask the lecturer or trainer questions about programming and must then use the answers to elaborate on their previous knowledge. It is interesting that experts spend relatively more time than novices in figuring out a problem (Sternberg, 2006:424). The asking of questions supports programmers in this process, as they elaborate on the amount, organisation and use of knowledge required to solve a new programming problem.

- *Creating analogies*

  During analogies, similarities between concepts are identified. In this way, a programmer can abstract a solution from a previous problem and relate that information to a new situation. This means that the programmer should make meaningful links between formerly solved problems and a new programming problem. This requires knowledge of the specific domain to solve the problem (Schunk, 2000:198, 200).

4.3.3 The organisation-and-integration strategy in object-oriented programming

Bergin et al. (2005:85) believe that rehearsal, elaboration and organisational strategies are not as useful in the introductory learning of OOP as they are in other academic domains. More research is necessary in the particular case of OOP. The organisation strategy in the context of OOP also includes the integration of various knowledge and skills, whereby a programmer combines code to design a whole new program or software application. Therefore, the organisation-and-integration strategy is suggested in this programming context.

This strategy is appropriate, because it organises and combines different knowledge and skills with the goal of achieving a holistic problem solution. Neath and Surprenant (2003:102) emphasise that the organisation of information helps the organiser to comprehend and remember new information. Diagrams may be used to organise large amounts of information (§3.3.3). Caspersen and Kölling (2006:1) point out that some students do not naturally have the skills to combine different OOP constructs in an organised way. However, these skills can be learned. The organisation of information is essential for good problem solving. With the organisation-and-integration strategy, a programmer integrates all the necessary information within one program to solve the problem. Furthermore, experts perform better because they present new material in a coherent way (Sternberg, 2006:396,421). The organisation-and-integration strategy can support programmers in the combination of objects, methods and attributes within a class to form a new and correct programming solution.

Various examples of the organisation-and-integration strategy follow:

- *Organisation*
  During organisation, learners organise and build connections with the information they receive (Filcher & Miller, 2000:63). New facts are incorporated and require the reorganisation of prior knowledge (Rajlich & Wilde, 2002:272). This is particularly useful in the case of "complex material", where such organisation improves retrieval by linking associated items of relevant information (Schunk, 2000:156,387). As an example, if a programmer has prior knowledge about the primitive types such as *int* and *boolean*, the organisation strategy can help him to gain knowledge about *wrapper classes* (*Integer, Boolean*) and build connections with the new information (Wigglesworth & Lumby, 2000:119).

- *Outlining*

  Outlining is a popular strategy used to emphasise important points that are highlighted by bold text. Outlining is also important in a programming language. It can be applied either in the description of a programming problem to emphasise main points or in the code of a programming language where, for example, reserved words appear in bold within a computer program and cannot be redefined in any program. Examples of reserved words are *class, type, private, static* (Malik & Nair, 2003:24).

- *Categorical clustering*

  This strategy organises items into a set of categories (Sternberg, 2006:200). For example, the programmer might organise various programming types into categories, such as primitive data types: *char, byte, int, boolean*; and reference types, such as *array* and *class.*

- *Selecting the main idea*

  Basic understanding of the requirements of any new program includes identifying and selecting the necessary objects for that problem domain (Sicilia, 2006:6; Bergin et al., 2005:82; Satzinger et al., 2004:243). Students may highlight and select verbs in a problem description as an indication of possible methods for the new program. Underlining should be selective, to emphasise important concepts in the problem statement. This approach helps to ensure correct understanding of a programming problem. Students can underline nouns in a problem statement to identify possible objects (Sicilia, 2006:6), for example:

  The <u>XYZ-Company</u> is planning to reorganise <u>bus drivers</u>

- *Determining the flow of information*

  Programmers should determine the input, output, flow of information and parameter passing in a program (Satzinger et al., 2004:258). This can be done by means of diagrams (use case diagrams, system sequence diagrams) or by the use of arrows between method headings (without method bodies) to indicate possible parameter passing between different methods.

- *Separation*

  This entails the separation of the main concepts from the details. When using abstract classes in OOP, the abstract class cannot be instantiated and any subclass of the abstract class must provide implementations of all the inherited abstract methods (Sebesta, 2004:461).

Cognitive strategies, as described in this section, can help programmers to remember, select and organise information in memory. However metacognitive strategies, by means of which individuals monitor their own cognition, can also support the programmer in different ways and will be discussed in detail in §4.4.

4.4 Metacognitive strategies

To achieve the goal of successful programming, students build further on cognition and manage their own learning by means of metacognitive strategies. Metacognition, introduced in Section 3.4, is cognition about cognition and refers to mental operations that direct, monitor and support cognitive processes and that improve learners' abilities to achieve a goal (Kapa, 2001:318). Metacognition may influence behaviour, cognition and improve human learning (Schwartz & Perfect, 2002:5). It involves selective attention, conflict resolution, error detection and control (Shimamura, 2000:313).

A number of models for metacognition have been proposed (Gama, 2004; Ertmer & Newby, 1996; Flavell, 1979). For example, Ertmer and Newby (1996) refer to the following metacognitive strategies: planning, monitoring and *evaluation*. Bergin et al. (2005) discuss self-regulated learning with regard to introductory performance of students in their third level of introductory OOP and refer to planning, monitoring and *regulation* strategies. Moreover, Bloom's taxonomy is used in this study, which includes *evaluation* as the highest level of skills in the cognitive domain (§3.3.2, Table 3.1, Table 3.2). Therefore, to prevent possible confusion between evaluation as a cognitive skill or as a metacognitive strategy, the term *regulation* strategy will be used in this study with relation to the metacognitive domain.

Bergin et al. (2005:81, 82) found that students who perform well in programming use more metacognitive-management strategies than lower performing students. As stated above, this includes planning, monitoring and regulation of cognitive processes, as well as reflection on what occurs throughout the execution of a task. Each of these strategies will be discussed in more detail with reference to the task of programming.

4.4.1 Planning strategy in object-oriented programming

Planning strategies refer to the setting of goals, skimming text before reading, and analysis of different tasks (Bergin et al*.,* 2005:82). Planning should be conducted up-front to determine the sequence of specific information processing activities (Gilhooly, 2005:61; Kapa, 2001:319). The planning approach helps to activate prior knowledge, support comprehension, increase successful task completion and contribute in producing a quality solution (Ertmer & Newby, 1996:11). Planning is critical to the learning of programming. In fact, a lack of planning can result in poor understanding and trial-and-error strategies in attempts to solve the problem (Staats & Blum, 1999:15) (§4.5.1.5). In OOP, it is important to decide on an appropriate strategy to solve the particular problem. Planning can be enhanced with active questioning during or prior to the task (Staats & Blum, 1999:15). The analysis of a programming problem, identification of objects and methods, and the application of a strategy, are necessary to solve the problem and achieve the goal.

Successful problem solvers use more time and resources in planning their complete program than unsuccessful problem solvers do, and they spend more time initially on their design (Sternberg, 2006:396, Appendix G). This enables them to prevent certain potential errors proactively. Some practical examples of the planning strategy include the planning of the program and planning the debugging process.

- *Planning the program*

  Programmers should plan their programs in advance, set goals and consider the problem requirements. For this purpose, they can write planning details in a journal, making short notes on their intentions, including problems anticipated with specific aspects of the programming process.

- *Planning the debugging strategy*

  Students should plan their debugging process by determining in advance which debugging strategy/ies to use. They should think ahead to anticipate potential problems and take actions to avoid them, rather than to react to problem situations as they occur (Kirkwood, 2000:526,527).

4.4.2 Monitoring strategy in object-oriented programming

Learners use monitoring strategies to control their own programming activities. Monitoring applies to all processes that allow the individual to observe, reflect on and experience his own cognitive processes (Schwartz & Perfect, 2002:4, 5). It plays a vital role in the cognitive performance of complex problem solving. Monitoring guides the process of finding a solution as programmers aim to achieve their goals (Hertzog & Robinson, 2005:110,111).

During monitoring, a learner can transfer the use of a strategy from one task to another. For example a successful strategy in one specific problem could be used in an isomorphic problem. Problems are considered to be isomorphic when their formal structure is the same and only the content differs (Sternberg, 2006:223,400).

In the true spirit of metacognition, programmers should also be able to monitor their own strategic use. The application of an incorrect strategy may result in different and sometimes incorrect performance (Bergin et al., 2005:82; Lemaire & Fabre, 2005:19). For example, if the output of the program is wrong, another strategy should be chosen. It is therefore important to learn the conditions under which it is appropriate to apply each particular strategy. Some practical means of monitoring are given below:

- *Self-questioning*

  'Ask-yourself' questions serve a useful role in reflecting on one's knowledge and skills about the programming task. Students should query themselves to monitor whether they know the material they have studied. Hammouri (2003:576) emphasises that successful problem solvers use self-questioning consciously or unconsciously and monitor their own performance. Self-questioning enables students to direct their own problem-solving strategies. For example, learners could ask themselves questions about the accessibility of methods.

- *Help-seeking*

  Help-seeking refers to acquiring support when needed (Aleven, McLaren, Roll & Koedinger, 2004:227). Aleven et al. (2004) created a taxonomy for help-seeking activities to determine metacognitive behaviour. In programming, help-seeking can be done by finding help in a textbook or by on-line help, as was done by many students in this study during the programming of the *Date class* – see §5.3.7.

- *Monitoring their own strategy use*
  Programmers should monitor their own use of strategies in the programming process. For example, the plan should be monitored to note whether events evolved according to the initial plan, or not.

- *Investigating alternatives*
  Students tend to use the first solution that comes to their mind. They should monitor their performance and investigate alternative solutions to improve the quality of their programming, as well as the quality of their code. Such monitoring should include consideration of their approaches to correctness, testing and extendibility (Caspersen & Kölling, 2006:2).

4.4.3 Regulation strategy in object-oriented programming

Regulation strategies are the continuous modification of one's cognitive activity and self-evaluation to determine whether the problem is being solved successfully (Bergin et al., 2005:83). The chosen strategy must result in effective performance and fulfilment of the goal. Furthermore, all of the subgoals must be satisfied to ensure that the problem is solved.

Regulation also involves the identification of errors in a task (Bergin et al., 2005:82). Students should improve the accuracy of their self-judgement and refine their insight into the task (Bergin et al*.,* 2005:82; Roberts & Newton, 2005:132,154; Kapa, 2001:320). However, Sternberg (2006:224) posits that monitoring is a bottom-up process, keeping track of current understanding, whereas regulation is a top-down process of central control over these strategies. Using a strategy in a totally new way, on occasions, may support the sudden understanding of a problem as a learner gains new insight into the situation (Sternberg, 2006:406). Some practical means of applying the regulation strategy are the following:

- *Rereading and going back*
  During regulation, the programmer can go back and reread the problem description to check whether the problem was solved successfully, whether appropriate test data was used, and whether the correct output was delivered (Bergin et al*.,* 2005:82).

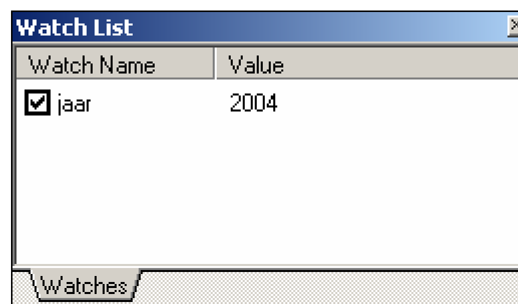
- *Identification of errors and demonstration of accuracy for quality control*
  Programming errors identified during the monitoring process should be changed and corrected. Edwards (2004:26) emphasises that programmers should personally demonstrate the correctness of their own code and they should do so by using more

than just debugging skills in the process. Programmers must be aware of various measures to determine the correctness of a program (rubrics, types of assessment, etc.).

- *Debugging and program tracing*

  Debugging techniques can support problem solving. Programmers should apply their knowledge of syntactic and semantic errors to debug a program and ensure correctness. For example, the *Trace*-function serves to show the next statement to be executed, while the *Watch*-function displays the value in memory at a specific time, thus identifying errors when the actual value of an item of data does not correspond with the anticipated value (Barrow, Miller, Malan & Gelderblom, 2005:367). The programmer can see what happens during program execution with the aid of tracing functions in the programming environment and can reflect on the way the problem was programmed. A programmer could, for example, trace the flow of a computer program to fully comprehend the working of a *while* loop and monitor program execution. Program tracing of each line in the program can be done in OOP languages such as *Delphi,* as shown in the code fragment of Participant 29 (P29) in Fig. 4.3:

Watch function:



Trace function:

```
procedure TDatum.IsSkrikkel( var Antw: String; Jaar:integer);
begin
  if ((Jaar mod 100 = 0) and (Jaar mod 400 = 0)) or (not(Jaar mod 100 =0)
      and (Jaar mod 4 = 0)) then

  Antw := ('Ja, '+ IntToStr(Jaar)+ ' is n skrikkeljaar !')
  else
  Antw := ('Nee, '+ IntToStr(Jaar)+' is nie n skrikkeljaar nie !');
end;
```

Figure 4.3: *Watch and Trace* functions displaying the value of the variable 'year' currently in memory during program execution

- *Reaction to feedback*

  Concrete feedback is important for the improvement of performance (Edwards, 2004:27) and occurs particularly in the interpretation of error messages during program execution (Shannon, 1999:xxiii). Some reactions to feedback include the improvement of program quality and accuracy. The purpose is to adjust a program to deliver the correct output. However, with no corrective feedback, the same mistakes may be perpetuated (Tan et al., 2006:828; Kapa, 2001:320).

- *Making predictions*

  Students should fully comprehend the source code, consider how programming statements would behave, and should predict how a change to their code would result in a change in program behaviour (Edwards, 2004:27). Moreover, students should hypothesise about what behaviour they expect from specific methods or from the complete program.

4.4.4 Reflection in object-oriented programming

This subsection addresses reflection, which includes the planning, monitoring and regulation processes addressed in the previous subsections. Reflection differs from the other three strategies, in that it is not a separate, subsequent activity, but rather a continuous activity that should take place through every step and phase of the programming process. Reflection is defined as "serious thought or consideration" and the word "reflect" means to "think deeply or carefully" (Concise Oxford English Dictionary, 2004:1208). It includes actions during the planning, monitoring and regulation of a task to control one's thought processes. Furthermore, reflection is a dynamic process of receiving feedback to successfully complete a task such as programming (Edwards, 2004:26). For example, after all syntax errors have been corrected, the program may still not give the correct output. In such cases, students must explicitly investigate the correctness of their code in terms of its semantics and reflect on their prior thinking to identify the cause of these errors (Edwards, 2004:26).

Reflection makes it possible for learners to apply their metacognitive knowledge, skills and strategies gainded from previous experiences and transfer them successfully into new programming situations. It allows learners to consider plans before, during and after engaging in a task (Breed, 2006, Ertmer & Newby, 1996:14).

Various models explain the importance of reflection during a task (Ertmer & Newby 1996:15; Schön, 1983:26). However, Gama's model, the *Reflection Assistant Model* (RA), focuses on metacognitive aspects in problem-solving environments (Gama, 2004:668-677). The RA is organised around different problem-solving stages: preparation to solve the problem, production stage and evaluation stage. Two additional conceptual stages were created: pre-task reflection and post-task reflection (Gama, 2004:671). Pre-task reflection refers to metacognitive aspects relevant when considering a new problem, as the learner becomes aware of useful strategies, available resources and the focused attention needed to succeed in the problem-solving activity. Post-task reflection refers to metacognitive aspects where learners create a space to ponder their actions during the completed problem-solving activities and compare them with the pre-task reflection (Gama, 2004:672).

Kapa (2001:317) found that learning was significantly more effective when provision was made for metacognitive support during each problem-solving step. With reference to Table 3.2 (§3.3.2), relating to cognitive skills used in the OOP process, programmers should reflect during:

- *Understanding*: to plan their own programming process and reflect on their planning decisions;

- *Designing*: to represent their problem mentally, relate new information to prior knowledge, monitor their own program design and make the necessary changes and reflect on their design. For example, they should check that all the necessary classes are included in their UML design (§3.2.4.2);

- *Coding*: to plan their own coding process, implement their design with programming code, continuously modify their program and regulate their own programming. For example, they should reflect on their problem-solving processes, which may result in a better approach for solving a specific problem;

- *Testing*: to interpret and judge the program output. This includes checking the accuracy of their own judgement, refining their personal insight, and the correction of syntax and semantic programming errors.

In this section, the metacognitive strategies of planning, monitoring, regulation and reflection were discussed. Programmers should apply metacognitive strategies to direct their solution

process. They should continuously reflect on their activities during the entire process of programming to achieve the goal that leads to the required outcome (Kapa, 2001:330).

Finally, important problem-solving strategies can be applied in OOP and will be addressed in §4.5.

4.5 Problem-solving strategies in object-oriented programming

The general concept of problem solving in OOP was considered in Section 3.5. This section narrows down the concept by focusing on specific problem-solving strategies that can be used to support OOP. Schunk (2000:196) distinguishes between general and specific problem-solving strategies. General strategies include trial-and-error, means-ends analysis, hill climbing or moving forward, divide-and-conquer, brainstorming and heuristics (Garton, 2004:46; Robinson-Riegler & Robinson-Riegler, 2004:498-499; Schunk, 2000:196). Cañas, Antolí, Fajardo and Salmerón (2005:96) claim that the application of appropriate problem-solving strategies reduces cognitive demands and speeds up performance. Such approaches include a pattern of repeatable actions that define a specific style of solving problems.

4.5.1 Problem-solving strategies during programming

In order to gain a deeper understanding of approaches to programming, there is a need to understand which strategies students follow during object-oriented program development. A programming strategy involves the understanding of how to apply programming knowledge and how to solve the underlining problem efficiently (De Raadt, Watson & Toleman, 2006:1).

In many cases, it is expected of novice programmers to use an implicit approach and to develop their own strategies as they learn to program. A study by De Raadt et al. (2006:2) investigated the explicit teaching of problem-solving strategies in programming. It was found that the results lay on a continuum ranging from “no teaching of any programming strategies at all” through to “explicit teaching of strategies to novice programmers”. Different problem-solving strategies are important during programming and are discussed in more detail.

4.5.1.1 Bottom-up strategy

A bottom-up strategy emphasises the detail and planning of individual parts that are combined to form larger components and to build higher levels of abstraction (Dunsmore, 1998:7). With this strategy, understanding progresses from the more *specific to the more general* (Shaft, 1995:26).

Bottom-up implies writing the code and then chunking program basics into higher-level abstractions (Zhang, 2005:6; Storey, Wong & Müller, 1997). In terms of OOP, this strategy focuses on the refinement of each method before proceeding with the programming of more classes/subclasses. Thus, a method in a class model is defined and refined before proceeding to other methods (Détienne, 1995:145).

For example, in the *Date class* program, discussed in detail in Chapter 5, the detail of a fundamental method *(DaysOfMonth( ))* is defined before proceeding with the next method (*isLeapYear*). In a study by Détienne (1995:137), it was anticipated that novice object-oriented programmers would use a bottom-up strategy. However, Corritore and Wiedenbeck (2000:139) emphasise that the bottom-up strategy is used more often by procedural programmers. Furthermore, expert programmers working in an unknown domain or task use a bottom-up strategy (Corritore & Wiedenbeck, 2000:139-148).

4.5.1.2 Top-down strategy

In the top-down strategy, high-level planning and understanding of the complete system are addressed up-front, without initially going into low-level details. The top-down strategy starts with goals and thereafter finds plans to achieve the goals. With a top-down approach, understanding progresses from the *general to the more specific* (Zhang, 2005:7; Brooks, 1983:546).

Pennington et al. (1995:198) found that expert object-oriented designers used the requirements of the problem to focus on the creation of classes and methods. They spent more time on the design of a program than novice object-oriented designers. Novices hope that the classes they have created will be useful later, while experts create classes only as needed. Within this strategy, a programmer makes certain hypotheses and then confirms or rejects them according to the evidence in program code (Rajlich & Wilde, 2002:271). All the methods are defined in the class before proceeding with the refinements of each method. For example, in a top-down design, the following methods should be defined in the *Date*

class task of this study before proceeding with the detail: *DaysOfMonth( ) and isLeapYear( )* etc.

Programmers with knowledge of the domain and task use a top-down, goal-oriented or hypothesis-driven strategy during understanding. Corritore and Wiedenbeck (2000:139) emphasise that the top-down strategy is used more by object-oriented-programmers in the early phases but that they tend to make increasing use of the bottom-up strategy during the maintenance of tasks.

4.5.1.3 Integrated strategy

The integrated strategy is applied where the programming code is developed in a more haphazard fashion. The programmer may switch between different strategies. An integrated strategy combines top-down and bottom-up strategies in different levels of abstraction. Conklin (2006:135) refers to the importance of the bottom-up approach in providing the basic knowledge needed for programming, as well as the role of the top-down approach in understanding the correct context of programming. This hybrid approach can be useful in large programs where systematic understanding is not possible (Pacione, 2004:23; Von Mayrhauser & Vans, 1997). One class is defined along with its methods before proceeding to the next class and its methods. It is important to integrate the pieces of information in OOP into a whole program (Kapa, 2001:317).

4.5.1.4 As-needed strategy

When the programmer attempts to minimise the amount of code that has to be understood, he uses the as-needed strategy. The focus in this strategy is not on the overall design, but on a specific part that has to be modified (Pacione, 2004:23; Corritore & Wiedenbeck, 2000:140). The as-needed approach makes it possible for a programmer to change only the required areas in a program (Young, 1996). However, these observations were made during program maintenance and not during the programming of a new task.

4.5.1.5 Trial-and-error strategy

This strategy is applied when a learner attempts to reach a solution without having had any explicit planning strategy. Trial-and-error construction occurs as successful tasks are established and unsuccessful tasks are abandoned (Schunk, 2000:32), leading to a gradual form of development. This strategy is used when a part of the solution fails to work, where the programmer has difficulties or is confused (Edwards, 2004:26). Novice Computer Science students rely on the trial-and-error strategy to fix errors and debug a program.

Edwards (2004:27) suggests that most novices focus on the development of programs and use synthesis skills to write a program, but advises that they should first master basic comprehension and analysis skills to avoid a trial-and-error strategy.

In this section, various problem-solving strategies were discussed: bottom-up, top-down, integrated, as-needed and trial-and-error. These strategies support the learning of problem solving in OOP. Some practical means of problem solving are mentioned.

- *Create a class with method stubs*
  Method stubs are method bodies, such as procedures and functions in Delphi that consist of minimal programming code. Novices are able to understand separate constructs, but lack the skills to organise these constructs in a coherent way. The aim is to break down all methods into smaller chunks to decrease the complexity of the program. For example, for methods that do not return values, the method body is empty and methods with return values consist only of a single return statement (Caspersen & Kölling, 2006:1, 2). This is probably an example of the integrated strategy, where both the top-down and bottom-up approaches are used.

- *Prevent strategy fixation*
  Problem solvers can fixate on a strategy that usually works in many different situations, but that is not necessarily appropriate for certain specific problems (Sternberg, 2006:411). For example, it is sometimes necessary to use the top-down strategy in a new program and the as-needed strategy in program maintenance tasks (§4.5.1.4)

4.6 Chapter conclusion

The role of this chapter, as a follow up to Chapter 3, is to examine the relationship between the theoretical concepts of cognitive, metacognitive and problem-solving knowledge and skills and their practical implementation in various strategies. Programmers must have knowledge about the different strategies and how to select and apply them in a programming task (§4.2). The role of cognitive strategies, rehearsal, elaboration and organisation-and-integration was mentioned in OOP (§4.3). In the section on metacognitive strategies, the importance of planning, monitoring, regulation and reflection was emphasised (§4.4). Furthermore, various practical problem-solving strategies that support program comprehension and development were discussed (§4.5). The selection and application of

various strategies has been described and practical examples have been given. Programmers should explicitly learn these strategies and apply them during programming.

Students can use knowledge, skills and strategies to help them to reach specific goals. In an empirical study, Chapter 5 will identify and evaluate the knowledge, skills and strategies that students use during computer programming.

5 Empirical research and data analysis

5.1 Introduction

Chapter 3 outlined the roles that cognition, metacognition and problem-solving knowledge and skills play in object-oriented programming. In Chapter 4, specific cognitive, metacognitive and problem-solving strategies were discussed that provide support to students during the learning of OOP. This chapter describes how the empirical research of this study focuses on ascertaining and evaluating the knowledge, skills and strategies that students use during computer programming. In addition, this chapter sets out to answer the main research question:

> ***Which knowledge, skills and strategies are used during problem solving in object-oriented programming?***

Chapter 2 addressed the applicability of interpretivism (§2.3, §2.5.1), grounded theory (§2.4, §2.5.2) and positivism (§2.6) to this study. Data collection methods and analysis methods were also discussed in Chapter 2 (§2.7, §2.8). The current chapter implements the research design and methodology of Chapter 2 as it considers the interpretation of participants' computer programs, their thinking processes and the questionnaire. The chapter also discusses the analysis of empirical materials as the raw data is converted to final patterns of meaning (§2.7, §2.8, Fig. 5.10).

The empirical research in this study was done over two years, namely 2005 and 2006, investigating situations where participants gained experience in object-oriented computer programming. The participants were third-year Computer Science students (Table 2.3) namely, BEd students from the Faculty of Education using Delphi as an object-oriented programming language and BSc students from the Faculty of Science, who used Java as an object-oriented programming language (§2.7.1). All participants (n=48) were required to complete the *Date class* programming task (*Date class* and *Test class,* Appendix C) and to record their thinking processes while they did the programming (§2.7). In the second year

(i.e., 2006) the research was extended by a questionnaire requiring the participants to answer specific questions about their programming experiences. For the sake of convenience, all participants will be referred to in the male gender. As stated in Chapter 2 (§2.9.1), the author translated the text of the students' thinking processes into English in cases where these had originally been written in the Afrikaans language. Where necessary, the original version was edited. The translated quotations are shown in italics in this chapter.

This study employed specific strategies to ensure that the main question was answered by the data collection and analysis processes; which are depicted diagrammatically in Fig. 2.1 in Section 2.1 and are summarised below:

Data collection techniques:

- Computer program and thinking processes: all participants were required to
  - complete the *Date class* and *Test* class programs; and
  - record their thinking processes during programming (§2.7, §5.2, §5.3, Table 2.5);
- Questionnaire: after the computer program had been written, the participants of 2006 completed a questionnaire focused on the experience of doing the programming task (§2.7.4, §5.4, Table 2.5).

Data analysis techniques:

- Each participant's program and thinking processes were analysed by specific measurement criteria, as indicated in Table 5.1 (§5.2);
- The participants' thinking processes were analysed with the aid of the *Atlas.ti software* introduced in Sections 2.9 (Table 2.5), (§5.3);
- Questionnaire analysis (§5.4): the closed-ended questions were analysed statistically, and are considered in Subsection 5.4.2. The open-ended questions are discussed in §5.4.3.

Description of the emerging theory:

Finally, coherence between the different data sources was investigated to identify final patterns of meaning and to describe the emerging theory that leads to a model to explain the specific phenomena (§2.4.2, Fig. 5.10, Fig. 6.5, Havenga et al., 2008).

Triangulation was applied, whereby both quantitative and qualitative methods were used to analyse participants' programs and their written thinking processes (see Section 5.5).

5.2 Analysis of participants' computer programs and thinking processes

After explaining the measurement criteria and outlining the general analysis and scoring of the computer programs and thinking processes, this section highlights two particular cases, namely a poor program and an excellent program respectively. The analyses of these two examples were conducted in the same way as those of all the other participants' programs and thinking processes (Table 5.2, §5.2.2, §5.2.3, §5.2.4).

Following this, statistical analysis on the raw scores of all the participants is presented (Table 5.2), investigating the constructs of cognition, metacognition and problem solving and their relationship to OOP. Section 5.2 culminates by considering what particular knowledge, skills and strategies are used by successful programmers (§5.2.7).

Analysis strategy used in subsections §5.2.1 to §5.2.4:

- *All participants'* computer programs and thinking processes were analysed using the measurement criteria that emerged from Chapters 3 and 4 (Table 5.1, Table 5.2);
- An example of a poor program was selected, along with the relevant participant's thinking processes. This represents the detailed knowledge, skills and strategies (or the lack thereof) of the participant and is discussed in detail (Table 5.2, §5.2.2, Programs 5.1 and 5.2, Appendix F). The selection was made according to low scores for the measurement criteria;
- An example of an *excellent* program was selected, along with the participant's thinking processes. This represents the detailed knowledge, skills and strategies of the participant and is discussed in detail (Table 5.2, §5.2.3, Programs 5.3 and 5.4, Appendix G). The selection was made according to high scores for the measurement criteria.

Analysis strategy used in subsections §5.2.5 to §5.2.7:

- A detailed description of certain statistical measurements is given in Subsection 5.2.5.
- All participants' computer programs and thinking processes were analysed using these statistical measurements, including confirmatory factor analysis, sample adequacy, reliability testing, descriptive statistics, such as the mean value and standard deviation, practical significance with reference to effect size and the relationship between two variables as represented by the correlation coefficient (Table 5.2, §5.2.6).
- Specific knowledge, skills and strategies used by successful participants are discussed (§5.2.7, Table 5.2, Table 5.18).

5.2.1 Measurement criteria

This subsection describes how the performances of the 48 students in the *Date class* and *Test class* computer programs and their thinking processes were analysed, according to the measurement criteria in Table 5.1. The raw scores obtained are given in Table 5.2. The criteria were generated from the theoretical literature studies of Chapters 3 and 4 and originated from, among others, the following subsections:

- Cognitive knowledge and skills (§3.3);
- Metacognitive strategies (§4.4);
- Problem-solving strategies (§4.5);
- OOP knowledge and skills (§3.2).

Several of the criteria that refer to aspects of programming performance were obtained from Sebesta (2004). Table 5.1 categorises each criterion and indicates its origin. Scores were allocated as follows: for each subcategory in Table 5.1, participants obtained a mark (score) out of 4, except for the problem-solving category where they could use more than one strategy. As a result, 8 marks were allocated for this section. However, participants who used the trial-and-error strategy obtained zero (0) for this section, since it was not considered an acceptable problem-solving strategy (§4.5.1.5).

Each participant obtained a total mark in the form of a percentage. The detailed mark allocations as well as the totals are shown in Table 5.2.

Subsections 5.2.2 and 5.2.3 present two detailed examples relating to a poor program and an excellent program respectively. The associated analyses were conducted in the same way as the analyses of all the other participants' data.

Table 5.1: Measurement criteria for the analysis of Delphi and Java programs and thinking processes

Measurement criteria	Categories: Knowledge, skills and strategies	Reference: Section (§) or Table	Maximum marks
	Cognitive knowledge and skills		
Evidence of knowledge of the programming language	Knowledge	§3.3.2 Table 3.1 Table 3.2	4
Interpretation of the problem	Comprehension		4
Application of prior knowledge in a new program	Application		4
Analysis of the problem – breaking it down in steps	Analysis		4
Design of a new program	Synthesis		4
Evaluation of the solution	Evaluation		4
	Metacognitive strategies		
Evidence of planning during programming	Planning	§4.4.1- §4.4.3	4
Evidence of monitoring tasks during programming	Monitoring		4
Evidence of regulation or modification to correct flaws during programming	Regulation		4
	Problem-solving strategies		
Application of problem-solving strategies during programming	Bottom-up, top-down, integrated, as-needed	§4.5	8

Table 5.1 (continued): Measurement criteria for the analysis of Delphi and Java programs and thinking processes

Measurement criteria	Category: OOP knowledge and skills	Reference: Section (§) or Table	Maximum marks
Analysis of the program requirements	Program requirements analysis	§3.5.2.1	4
*Programming techniques used: indentation, readability, variable names and declaration	Programming techniques	Sebesta (2004:193-195)	4
*Application of the correct use of programming statements	Programming statements	Sebesta (2004:293-313, 321-337)	4
Application of user-friendliness and usability	User-friendliness	–	4
Design of classes and instantiation of objects	Classes and objects	§3.2.3.1, §3.2.3.2	4
Application of methods, such as constructors, mutators and accessors	Method application	§3.2.3.3	4
*Decision on the accessibility: public, private	Access control	Sebesta (2004:459)	4
*Application of parameter passing: number, order, type of variables	Parameter passing	Sebesta (2004:356-367)	4
Application of reasoning skills in OOP	Reasoning	§3.3.1.2	4
*Application of exception handling	Exception handling	Sebesta (2004:542-544)	4
*Application of program structure and scope	Program structure and scope	Sebesta (2004:211-215)	4
Actual solution to the problem	Solution of problem	§3.5, §3.5.2	4
Evaluation of the *Date class* and *Test class*	Program evaluation	§3.5.2, §3.3.2	4
Evidence of correct program output and test data used	Correctness of output	–	4
Total			**100 (%)**

*Criteria were selected to reflect on general characteristics of programming as applied in the *Date class* and *Test class* programs of this study (Sebesta, 2004:8). All the participants' thinking processes and computer programs were analysed using these measurement criteria. Their scores are shown in Table 5.2. Certain criteria refer to information in the thinking processes and others relate to performance in the programming task.

Table 5.2: Analysis of Delphi and Java programs and thinking processes of all the participants (n=48)

Category	**BEd 2005 (n_1 = 11)**											**BSc 2005 (n_2 = 17 – continued on next page)**													
Participant number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
Cognitive knowledge, skills																									
Knowledge	4	2	2	2	4	4	4	3	4	3	3	4	4	4	4	3	4	4	4	4	4	4	4	3	4
Comprehension	3	2	2	3	3	4	4	3	4	3	3	4	4	3	4	3	4	4	4	3	4	4	4	3	4
Application	2	3	2	2	3	4	4	3	3	3	3	4	4	3	4	2	4	3	4	3	4	4	4	3	4
Analysis	3	2	2	2	3	4	3	3	3	2	2	4	3	3	3	2	4	3	3	3	4	3	4	2	3
Synthesis	2	2	1	1	2	4	3	2	3	2	2	4	3	2	3	1	4	2	3	2	4	3	4	2	3
Evaluation	2	1	1	1	2	3	3	2	3	1	1	4	3	2	3	0	3	0	2	0	3	2	3	2	2
Metacognitive strategies																									
Planning	2	3	3	3	3	4	4	3	4	3	3	4	4	3	4	2	4	4	4	3	4	3	4	3	4
Monitoring	1	1	0	2	3	3	3	3	4	2	2	4	3	2	3	2	4	3	2	0	4	0	4	1	3
Regulation	2	1	0	1	2	3	3	3	3	1	1	4	2	2	2	0	3	1	1	0	3	0	3	1	2
***Problem-solving strategies**	8	8	0	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	0	8
OOP knowledge and skills																									
Proper requirements analysis	2	2	1	2	4	4	4	3	3	3	3	4	3	3	4	3	4	4	4	3	4	3	4	2	3
Programming techniques	2	2	1	2	4	4	4	2	4	3	3	4	4	3	4	2	4	1	4	3	4	3	4	2	3
Programming statements	3	3	1	3	3	4	4	3	4	3	3	4	4	3	4	1	4	0	4	3	4	3	4	3	3
User-friendliness	1	2	1	2	4	3	4	3	4	2	3	3	2	0	3	0	0	0	1	0	2	0	2	0	0
Classes and objects	2	3	0	3	3	4	4	3	3	2	2	4	4	3	3	2	4	0	4	3	4	3	4	2	3
Method application	2	3	1	3	3	4	3	3	3	2	2	4	4	3	3	2	4	0	3	2	4	2	4	2	3
Access control	3	3	0	3	4	4	4	4	4	3	3	4	4	4	4	3	4	0	4	3	4	3	4	3	3
Parameter passing	3	3	0	3	4	4	4	4	4	3	3	4	4	3	4	2	4	0	4	3	4	3	4	3	3
Reasoning	3	3	1	3	3	3	4	3	3	2	2	4	3	3	3	2	4	2	3	2	4	3	4	2	3
Exception handling	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	1	0	0	0	3	0	0
Program structure and scope	3	3	1	3	3	4	4	3	4	3	3	3	4	3	3	1	3	0	3	2	3	3	4	3	3
Solution of problem	2	1	0	2	2	3	3	3	2	2	2	4	3	2	3	0	3	0	3	2	4	2	4	2	3
Program evaluation	1	1	0	2	2	3	3	3	2	1	1	3	3	2	3	0	3	0	2	0	3	2	4	1	2
Correctness of output	0	0	0	0	0	0	1	0	0	0	0	3	0	0	3	0	0	0	0	0	0	0	3	0	0
TOTAL (%)	56	54	20	56	72	85	85	70	79	57	58	95	80	64	82	41	83	39	75	52	86	61	94	45	69
*Problem-solving strategy	BU	BU	–	BU	BU	BU	BU	BU	BU	BU	BU	BU	BU	BU	TD	TD	BU	BU	BU	BU	BU	IG	IG	–	IG

* BU = Bottom-up, TD = Top-down, IG = Integrated strategy

Table 5.2 (continued): Analysis of Delphi and Java programs and thinking processes of all the participants (n=48)

Category	BSc 2005 cont			BEd 2006(n_3=3)			BSc 2006 (n_4 = 17)																
Participant number	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**
Cognitive knowledge and skills																							
Knowledge	4	4	4	4	4	**2**	**4**	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Comprehension	4	4	4	4	4	**2**	**4**	4	4	4	4	4	4	3	4	4	4	4	4	4	4	4	4
Application	4	4	4	4	3	**1**	**4**	4	4	3	4	4	4	3	4	4	4	4	4	4	3	4	4
Analysis	4	3	4	4	3	**1**	**4**	4	4	4	4	4	4	3	4	4	4	3	4	4	3	4	3
Synthesis	3	3	4	4	3	**1**	**4**	3	3	3	4	4	4	2	3	4	4	2	4	3	2	4	3
Evaluation	3	2	4	4	3	**1**	**4**	2	3	3	3	3	3	2	3	3	4	2	4	3	1	3	3
Metacognitive strategies																							
Planning	3	4	4	4	3	**0**	**4**	4	3	3	3	3	4	2	4	3	3	4	4	4	4	4	4
Monitoring	0	3	4	4	3	**1**	**4**	3	3	3	2	1	2	2	3	2	2	3	3	2	2	3	3
Regulation	0	2	3	3	2	**0**	**4**	3	3	3	2	1	2	1	2	1	2	3	3	2	1	2	3
***Problem-solving strategies**	8	8	8	8	8	**0**	**8**	8	0	8	8	8	8	8	8	8	8	8	8	8	8	8	8
OOP knowledge and skills																							
Programming requirements analysis	3	4	4	4	4	**1**	**4**	4	4	4	4	4	4	3	4	4	4	3	4	4	3	4	4
Programming techniques	3	4	4	4	4	**1**	**4**	3	3	4	4	4	4	3	4	3	4	4	4	4	3	4	4
Programming statements	3	3	4	4	3	**0**	**4**	3	3	4	4	4	4	3	4	3	4	4	4	4	3	4	3
User-friendliness	0	0	3	4	3	**2**	**4**	3	1	2	1	2	3	1	2	1	2	2	4	3	2	3	3
Classes and objects	3	3	4	4	3	**0**	**4**	3	4	4	4	4	4	3	4	4	4	3	4	4	3	4	3
Method application	3	3	4	4	3	**0**	**4**	3	3	4	3	3	3	3	3	3	4	3	4	3	2	3	3
Access control	3	4	4	3	3	**0**	**4**	3	3	3	4	4	4	3	4	4	4	3	4	4	4	3	4
Parameter passing	3	4	4	4	3	**0**	**4**	4	3	4	4	4	4	3	4	4	4	4	4	4	4	4	4
Reasoning	3	3	4	4	3	**1**	**4**	3	3	4	4	4	4	2	3	3	4	3	4	4	2	4	3
Exception handling	0	0	3	3	1	**0**	**4**	2	1	3	0	1	2	1	1	0	3	1	4	1	2	2	2
Program structure and scope	3	3	4	4	3	**2**	**4**	3	3	3	3	3	4	3	4	4	4	3	4	3	3	3	3
Solution of problem	3	3	4	4	3	**0**	**4**	3	3	3	4	4	4	2	3	3	4	3	4	3	2	4	3
Program evaluation	2	2	4	4	2	**0**	**4**	3	1	3	3	3	4	2	3	2	3	3	4	3	2	4	3
Correctness of output	0	0	3	4	2	**0**	**4**	1	0	2	2	2	3	0	3	0	3	1	3	2	0	0	3
TOTAL (%)	65	73	96	97	76	**16**	**100**	80	66	85	82	82	90	62	85	75	90	77	97	84	67	86	84
*Problem-solving strategy	BU	BU	BU	IG	BU	**TE**	**TD**	BU	TE	IG	BU	BU	BU	BU	BU	BU	BU	TD	BU	BU	BU	TD	BU

* BU = Bottom-up, TD = Top-down, IG = Integrated strategy, TE = Trial-and-error. P31 and P32's data are highlighted as an indication of a poor and an excellent program respectively.

Subsections 5.2.2 and 5.2.3 present detailed examples of a poor program and an excellent program, done by Participants 31 and 32 respectively. In these discussions:

- Occurrences of concepts referred to by the measurement criteria in Table 5.1 were extracted by selecting specific words of the *thinking processes* that represented particular criteria (Table 5.3, central column);
- Examples are given of associated *computer statements from these programs* (Table 5.3, third column);
- The way participants implemented each category of measurement criteria is discussed.

This approach highlights the relationship between participants' thinking processes and their associated programs. Note that, prior to this detailed analysis of the two selected participants, marks were allocated to all the participants according to the measurement criteria (see Tables 5.1, 5.2 and 5.11).

5.2.2 Example 1: A poor program

This subsection discusses the investigation of a poor program in DELPHI in which Participant 31 (P31) had problems in applying different knowledge, skills and strategies during the *Date class* programming task. P31 submitted two separate incomplete programs in an attempt to solve the *Date class* task, namely Programs 5.1 and 5.2 (Appendix F). Tables 5.3 – 5.5 display the application of the measurement criteria given in Table 5.1 (first column of Tables 5.3 – 5.5), with reference to P31's corresponding thinking processes (central column) and/or Delphi computer program segments (third column). In most situations, the central column contains italicised statements extracted from P31's written thinking processes. Appendix F, the data of Participant 31, contains P31's programs and written thinking processes.

5.2.2.1 Cognitive knowledge and skills

A discussion follows regarding P31's *Date class* programs and thinking processes in the cognitive domain (§3.3.2, Table 3.1, Table 3.2, Table 5.1). Table 5.3 presents examples of data in the cognitive domain in association with the measurement criteria.

Table 5.3: Examples of P31's cognitive knowledge and skills (or the lack thereof)

Cognitive knowledge and skills		
Measurement criteria	**Thinking processes**	**Evidence in the program**
Evidence of knowledge of the programming language **Mark: 2**	**Knowledge** *A form is created.* *Add the components on the Delphi form.*	Components added on the Delphi form: Label, GroupBox, EditBox and Button. LblOutput1 := **…** [Incomplete statements]
Interpretation of the problem **Mark: 2**	**Comprehension** *I had problems to interpret leap years.*	Leapyear := 1904 or 1936 or 2004; [Years named, not calculated]
Application of prior knowledge in a new program **Mark: 1**	**Application** *I think that GroupBoxes will be a better choice.*	There was very little evidence in P31's program to illustrate application
Analysis of the problem – breaking it down in steps **Mark: 1**	**Analysis** *I have asked when it is a leap year.*	**If** edtLeapyear = Leapyear; **then** **…** [Incomplete statements]
Design of a new program **Mark: 1**	**Synthesis** *An if-statement was necessary for leap years.*	**If** edtInput2 := February **then** lblOutput:= '28 or 29 Days'; [Incomplete calculations]
Evaluation of the solution **Mark: 1**	**Evaluation** *My program does not work.*	There was no evidence of program execution.

- ***Knowledge, comprehension and application skills***

  Table 5.3 shows examples of cognitive skills, or rather the lack of such, as displayed by P31. The participant did not understand the programming problem and used the textbook to access additional information. Furthermore, P31 lacked knowledge of the syntax of basic programming statements and made unnecessary errors. One example

is: LblOutput1.**caption ...** where he omitted to add the *caption* property to the *Label* component – see row 3 of Table 5.3.

Concerning comprehension, P31 initially had problems understanding when a year is a leap year. He did not realise that programmers had to personally program this calculation! The following assignment statement was incomplete: *Leapyear := 1904 or 1936 or 2004,* where the participant omitted to add 'Leapyear' before each 'or' boolean operator (Program 5.2, Table 5.3). In addition, this participant displayed limited knowledge of OOP and found it difficult to apply his knowledge in the *Date class* programs (§3.2). For example, he should have decided on a *testDate( )* method (function or procedure) to determine the number of days in each month.

- ***Analysis, synthesis and evaluation skills***

  Table 5.3, rows 6 and 7 refers to programming lines in which P31 attempted to determine whether a year was a leap year. However, programming statements were entered without any calculations: *if edtLeapyear = Leapyear; then ...* The programmer should have determined the specific number of days for each month. For example, February could have 28 or 29 days depending on whether a year was a leap year. Moreover, it was difficult for P31 to use synthesis skills (§3.3.2, Table 3.1, Table 3.2). This participant clearly found it difficult to complete the *Date class* program.

  Program self-evaluation is a continuous process during programming. It includes the evaluation of program statements, program segments and the evaluation of the complete program. P31 should have applied his knowledge and skills to evaluate the correctness of the program, and to diagnose the problems. Instead, he made the statement: *My program does not work*. Practical skills are required during program testing to debug a computer program and to understand error messages. Furthermore, P31 should have used test data to determine whether the program was producing the correct output.

5.2.2.2 Metacognitive strategies

A discussion follows on P31's *Date class* task and thinking processes in the metacognitive domain (§3.3.2, Table 3.1).

Table 5.4: Examples of P31's metacognitive strategies (or the lack thereof)

Metacognitive strategies		
Measurement criteria	**Thinking processes**	**Evidence in the program**
Evidence of planning during programming **Mark: 0**	**Planning** No evidence of planning was found.	No evidence of planning.
Evidence of monitoring tasks during programming **Mark: 1**	**Monitoring** *I still have problems. My program does not work.* *My program did not show me errors.*	Incomplete evidence of monitoring. P31 should monitor his own performance to identify errors.
Evidence of regulation or modification to correct flaws during programming **Mark: 0**	**Regulation** *I don't know if it is correct.*	No evidence of regulation; instead, ignorance on this aspect was evident.

- ***Planning, monitoring and regulation strategies***

  No evidence of any planning strategy was found. P31 mentioned that he encountered problems with the program and tried to go back to study the textbook (§3.4, §4.4, Table 5.4, Appendix F). He could not identify the programming errors. P31 was not able to monitor and regulate his programming task and could not solve the problem. This implies that he could not make a diagnosis nor make the necessary changes or correct any errors: *I don't know if it is correct*.

5.2.2.3 Problem-solving strategy

P31 used the trial-and-error strategy during the programming process. This strategy is applied when a learner attempts to reach a solution without using any explicit planning strategy (§4.5.1.5, Table 5.5). The following statement indicates this strategy: *I have typed all the things that I thought should be in the program*.

Table 5.5: Examples of P31's problem-solving strategies (or the lack thereof)

Problem-solving strategy		
Measurement criteria	**Thinking processes**	**Evidence in the program**
Application of problem-solving strategies during programming **Mark: 0**	**Trial-and-error** *I have typed all the things that I thought should be in the program*	Not applicable

5.2.2.4 Application of measurement criteria in the Delphi program

P31's performance in the OOP computer program was analysed to determine whether he was able to apply his knowledge and skills in writing the program. P31 submitted two separate incomplete programs in an attempt to solve the *Date class* task, namely Program 5.1 and Program 5.2 respectively (see Appendix F for the full attempts). In Programs 5.1 and 5.2 immediately following, which are extracts from Appendix F, very few program segments occur that could be highlighted for measurement of criteria in OOP, since the correct approach was clearly absent. Moreover, very few cognitive, metacognitive and problem-solving knowledge, skills and strategies were used. The result was incomplete computer programs.

Note that the highlighted program segments in Programs 5.1 and 5.2 following are associated with certain of the measurement criteria from Table 5.1. The relevant categories of the specific measurement criteria are included in framed labels. For example, in the programming statement in Program 5.1,

```
lblNaam: TLabel;
```

the category, User-friendliness, inserted by the researcher, refers to the measurement criterion: Application of user-friendliness and usability. The researcher's comments in Program 5.1 and Program 5.2 following are included in square parentheses.

```
unit Datum_u;  // [First application program saved as Datum_u]

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;

type
  TFrmDatums = class(TForm)
    lblNaam: TLabel;                            User-friendliness
    gpbSkrikkeljare: TGroupBox;
                ...

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmDatums: TFrmDatums;

implementation
                ...
procedure TFrmDatums.btnSkrikkeljaarClick(Sender: TObject);

begin;
end;

procedure TFrmDatums.btnMaandClick(Sender: TObject);
begin
If edtInvoer2 := Januarie, Maart, Mei, Julie, Augustus, Oktober, Desember then
lblUitvoer:= '31 Dae';
 If edtInvoer2 := Februarie                Programming statements   // [incomplete and wrong]
then
lblUitvoer:= '28 of 29 Dae';               // [incomplete and wrong]
  If edtInvoer2 := April, Junie, September, November then
lblUitvoer:= '30 Dae';
end;

procedure TFrmDatums.btnOKClick(Sender: TObject);
Dae:=Integer;
begin
If radDaeVerloop := checked                Programming statements   // [incomplete and wrong]
then
Dae:=edtDatum2-edtDatum1;
lblUitvoer3:= ''Dae'';
  If radVerskilTussenDatums := checked
then
Dae:=edtDatum2-edtDatum1;                  Programming techniques   // [incomplete and wrong]
lblUitvoer3:= ''Dae'';
end;
```

Program 5.1: Delphi program segment from the *Date class* program of P31 (First attempt)

```
unit Datum_u; [Second application program saved as Datum_u.~pas]

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TFrmDatums = class(TForm)
    lblNaam: TLabel;                      User-friendliness
    gpbSkrikkeljare: TGroupBox;
    lblInvoer1: TLabel;
    edtSkrikkeljaar: TEdit;
    lblUitvoer1: TLabel;                  User-friendliness
    btnSkrikkeljaar: TButton;
    procedure btnSkrikkeljaarClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  FrmDatums: TFrmDatums;

implementation
{$R *.DFM}

procedure TFrmDatums.btnSkrikkeljaarClick(Sender: TObject);

var Skrikkeljaar:Integer;
NieSkrikkeljaar:string;

begin
Skrikkeljaar := 1904 or 1936 or 2004;    Reasoning   // [incomplete and wrong]
If edtSkrikkeljaar = Skrikkeljaar ;
                                          Programming statements   // [incomplete and wrong]
then
 lblUitvoer1 := 'skrikkeljaar';           Programming statements   // [incomplete and wrong]
end;

end.                                      //   [This program cannot be compiled]
```

Program 5.2: Delphi program segment from the *Date class* program of P31 (Second attempt)

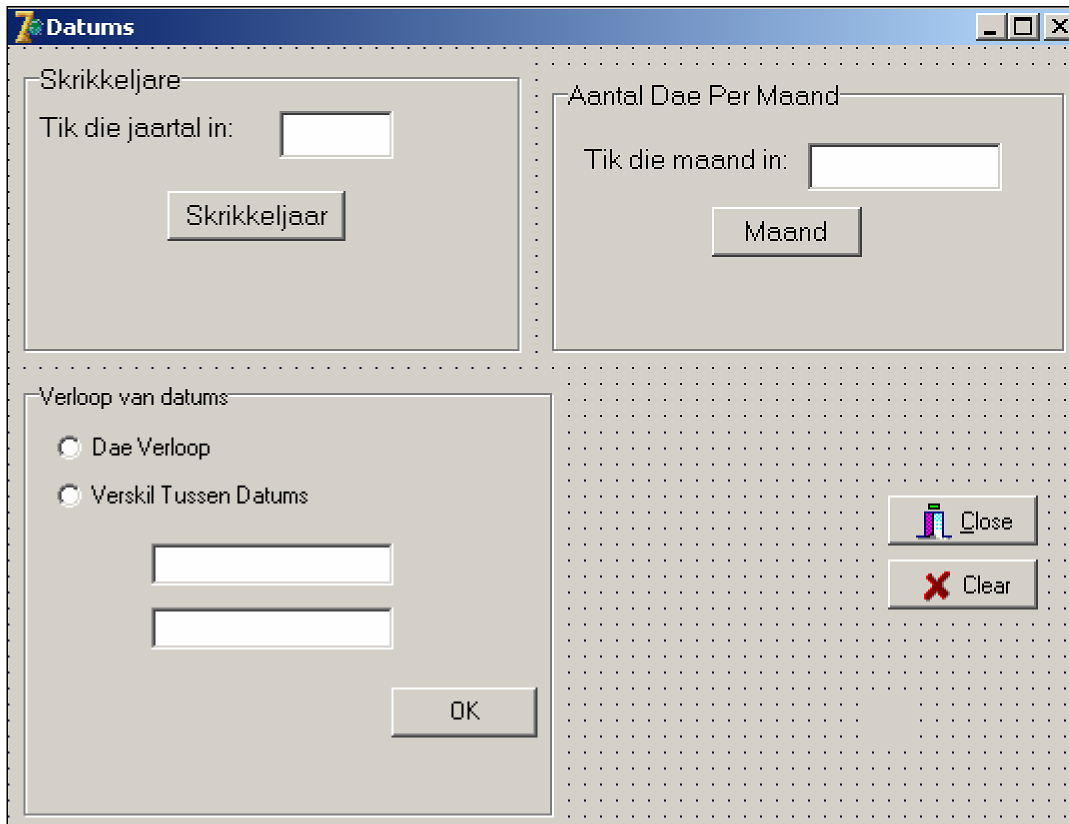


Figure 5.1: Design form of the first application program of P31

Program 5.1 showed that P31 was able to use various buttons when designing a form for the user (Fig. 5.1), namely: *editBox*, *label*, *groupBox*, *radiobuttons* and *bitmapButtons*. However, he experienced difficulties in writing the required functionality for each button. For example, he could not program the calculations for the LeapYear button ('Skrikkeljaar' button) at all. Furthermore, he had problems with the programming required to calculate the difference between two dates. In compiling P31's program, numerous errors were indicated by the compilation shown in Fig. 5.2:

```
[Error] Datum_u.pas(56): Undeclared identifier: 'edtInvoer2'
[Error] Datum_u.pas(56): 'THEN' expected but ':=' found
[Error] Datum_u.pas(58): 'THEN' expected but ':=' found
[Error] Datum_u.pas(61): 'THEN' expected but ':=' found
[Error] Datum_u.pas(68): Type of expression must be BOOLEAN
[Error] Datum_u.pas(72): Type of expression must be BOOLEAN
[Error] Datum_u.pas(83): Statement expected but 'PROCEDURE' found
[Error] Datum_u.pas(89): Undeclared identifier: 'radVerloopVanDae'
[Error] Datum_u.pas(89): Missing operator or semicolon
[Error] Datum_u.pas(33): Unsatisfied forward or external declaration: 'TFrmDatums.btnOKClick'
[Error] Datum_u.pas(34): Unsatisfied forward or external declaration: 'TFrmDatums.bmbCloseClick'
[Fatal Error] Datum_p.dpr(5): Could not compile used unit 'Datum_u.pas'
Build
```

Figure 5.2: Compilation of P31's program showing numerous errors

Table 5.6 displays the marks allocated to P31 for the OOP section. No program output was possible from either Program 5.1 or Program 5.2.

Table 5.6: Marks allocated to *P31 for OOP

Measurement criteria for OOP	Marks (4)
Analysis of the program requirements	1
Programming techniques used: indentation, readability, variable names and declaration	1
Application of the correct use of programming statements	0
Application of user-friendliness and usability	2
Design of classes and instantiation of objects	0
Application of methods, such as constructors, mutators and accessors	0
Decision on the accessibility: public, private	0
Application of parameter passing: number, order, type of variables	0
Application of reasoning skills in OOP	1
Application of exception handling	0
Application of program structure and scope	2
Actual solution to the problem	0
Evaluation of the *Date class* and *Test class*	0
Evidence of correct program output and test data used	0

* P31's programs are included in Appendix F.

The final marks for all sections, as well as the overall percentage are shown in Table 5.11 in Subsection 5.2.4, where they are compared with those of P32.

5.2.3 Example 2: An excellent program

Example 2 relates to an excellent program written in JAVA in which P32 applied insightful knowledge, skills and strategies in the programming of the *Date class* task. P32 submitted two separate complete programs for the *Date class* and *Test class* respectively (see Appendix G). Segments of these programs are shown in Programs 5.3 and 5.4 in §5.2.3.4. These programs are excellent examples of well-designed, accurate code. Tables 5.7 – 5.9 show application of Table 5.1's measuring criteria (first column of Tables 5.7 – 5.9), with reference to P32's corresponding thinking processes (central column) and/or Java computer program segments (third column). In most situations, the central column contains italicised statements extracted from P32's recorded thinking processes.

For the full text of P32's excellent thinking processes, see the second part of Appendix G. Effective use of supportive knowledge, skills and strategies was clearly in evidence.

Note: the % sign or 'modulus' refers to the remainder when a specific number is divided by 100 or 400 to determine whether a year is a leap year (§2.7.2). Marks were allocated to the participant according to the measurement criteria (Table 5.1, Table 5.2, Table 5.11).

5.2.3.1 Cognitive knowledge and skills

This subsection investigates the cognitive knowledge and skills that P32 used in his thinking processes and/or computer program (§3.3.2, Table 3.1, Table 3.2).

Table 5.7: Examples of cognitive knowledge and skills used by P32

Cognitive knowledge and skills		
Measurement criteria	**Thinking processes**	**Evidence in the program**

Evidence of knowledge of the programming language
 	
Mark: 4
	Knowledge
	Create a constructor for the Date.java class: the constructor should receive a date.
	You need to import the BufferedReader, InputStreamReader … use the while and Boolean.
		BufferedReader console = **new** BufferedReader (**new** InputStreamReader(System.in));
		...
		public Datum() **throws** IOException
		...
		while (trueDate == **false**)
		{
		System.out.print("Wat is vandag se datum (DD Month YYYY): ");
		...
		}

Interpretation of the problem
 	
Mark: 4
	Comprehension
	I should create the testDate method to test the dates. This method will receive a date and must test its correctness according to the calendar, for example: 45 Wednesday 1203 is an invalid date.
		public boolean testDate(**int** dayTemp, String monthTemp, **int** yearTemp)
		{
		int monthNum;
		int numDays = 0;
		boolean testMonth = **false**;
		boolean testDay = **false**;
		boolean testYear = **false**;
		...
		}

Application of prior knowledge in a new program
 	
Mark: 4
	Application
	Test century years – year mod (%) 100. If this equals 0, then it is a century year.
	Century leap year: year mod 400, if it equals 0, then it is a century leap year. Not a century leap year: year mod 4, if this equals 0 then it is a leap year.
		{
		if (yearTemp % 400 == 0)
		yearSkrik = **true**;
		...
		}

Table 5.7 (continued): Examples of cognitive knowledge and skills used by P32

Measurement criteria	Thinking processes	Evidence in the program

Analysis of the problem – breaking it down in steps
 	
Mark: 4
	Analysis
	Which calculations are needed to determine a leap year and what are the return values?
	Purpose? … test correctness
	Parameters? dayTemp, monthTemp, yearTemp
	Input, Output? return boolean
	Variables needed? … testMonth, testDay, testYear
	Calculations? test year, leapYear, months
	Return? Return true value if boolean is true else return false
		{
		if (yearTemp % 100 == 0)
		...
		if (yearTemp % 400 == 0)
		yearSkrik = **true**;
		else if (yearTemp % 4 == 0)
		yearSkrik = **true**;
		testYear = **true**;
		}

Design of a new program
 	
Mark: 4
	Synthesis
	Write DateDifference( ) method:
	Subtract: calculate the largest date.
	Use boolean to determine the largest e.g. firstBiggest. ...compare years and thereafter months and then days.
		if (year1 > year2) firstBiggest = **true**;
		else if (year2 > year1) firstBiggest = **false**;
		else if (year1 == year2)
		{
		if (month1Num>month2Num)
		firstBiggest = **true**;
		else if (month2Num>month1Num)
		firstBiggest = **false**;
		...
		}

Evaluation of the solution
 	
Mark: 4
	Evaluation
	Problems? Many!! The method was difficult … and I should include many exceptions for leap years.
	The biggest problem was the difference between days.
	I also have a few ArrayOutOfBound exceptions. This was solved with diagrams.
		public void datumsVerskil() **throws** IOException
		...
		for (**int** i = 0; i < numYears1; i++)
		{
		if ((1800 + i) % 100 == 0)
		if ((1800 + i) % 400 == 0)
		numSkrikYears1++;
		else
		numNieSkrikYears1++;
		else if ((1800 + i) % 4 == 0)
		numSkrikYears1++;
		else
		numNieSkrikYears1++;
		}

- ***Knowledge***

  P32 was able to apply his knowledge when writing a computer program. Table 5.7 shows examples of this as he refers to *constructor*, *class*, *while*-iteration, *boolean* and various ways of input: *BufferedReader*, *InputStreamReader*. He also used knowledge to test whether a condition is true by means of the *while*-statement. This displays procedural knowledge e.g., knowledge of how to perform specific activities during programming (§3.1).

- ***Comprehension skills***

  P32 comprehended what he had read about the programming problem, evaluated his understanding of the problem, was able to extract the important concepts and could determine the distinctive features of the problem. (Table 3.1, Table 5.7). This participant mentioned that only valid dates were acceptable. He comprehended the concept 'valid date' and suggested that input dates should be tested. This shows that he understood when a date is invalid and what a leap year meant.

- ***Application skills***

  P32 applied his knowledge and skills in the new programming problem and used it to calculate leap years. He used prior knowledge and previously-written programs in new problems, by applying his acquired knowledge, facts, techniques and rules in a different way (§3.2.5, Table 3.1, Table 5.7). This participant determined correctly that leap years should be divided by 4 and by 100. Century leap years should also be divided by 400, using modulus or mod (***%)*** in the calculation. If the remainder equals 0 then the year is a leap year (§2.7.2, Appendix G).

- ***Analysis skills***

  Table 5.7 includes examples of specific questions (Purpose? Parameters? etc.) that P32 posed to analyse the problem (§3.3.2): *Which calculations are needed to determine a leap year and what are the return values*? During decision making, different options were considered and a selection was made according to these options.

  He worked out how to calculate the difference between two dates, for example, between 18 October 2006 and 09 May 1984 (see Fig. 5.3). He also established how to calculate leap years. The *if*-statement was required to achieve this (Table 5.7).

- ***Synthesis skills***

  P32 had the ability to organise and combine different programming statements and incorporated various methods, such as *DateDifference( )* and *testLeapYear( )* to form a new application (§3.3.2, Program 5.3, Program 5.4). He combined the *Date class* and the *Test* class in such a way that new objects were created in the *Test class* that were instantiations of the *Date class* (Appendix G). Correct decisions were made to complete the program successfully.

- ***Evaluation skills***

  A computer program should be judged and evaluated (§3.3.2, Table 3.1). Table 5.7 gives examples of problems that occurred during the programming of the *Date class* task. P32 was able to articulate and solve his own problems. For example, he mentioned that the most difficult part of the program was determining the difference between two dates. However, he was able to use knowledge and various skills to solve the problems.

  Other examples of sound evaluation are demonstrated by the way this participant solved the problem with the *ArrayOutOfBound exception* and how he used test data to determine that the output of his program was correct (Fig. 5.3). He also mentioned that certain exception-handling techniques should be included to test for leap years.

5.2.3.2 Metacognitive strategies

P32's thinking processes and computer programs display examples of metacognitive strategies including: planning (§4.4.1), monitoring (§4.4.2) and regulation (§4.4.3). Table 5.8 shows the scores assigned using Table 5.1's measuring criteria with reference to P32's corresponding thinking processes and/or computer program segments. The central column contains statements extracted from P32's thinking processes and the column on the right displays associated programming statements from the Java computer program (Programs 5.3 and 5.4).

Table 5.8: Examples of metacognitive strategies used by P32

Metacognitive strategies		
Measurement criteria	**Thinking processes**	**Evidence in the program**
Evidence of planning during programming **Mark: 4**	**Planning** • *Create framework for Date and Test class* • *Create a constructor* • *Create the testDate method* • *Test with the Test class* • *Test for leap years* • *Update the test program, test the testLeapYear( ) method* • *Write the dateDifference( ) method* • *Update the test program.*	**public** Datum() **throws** IOException **...** **public boolean** testDate(…) **...** **public void** toetsSkrikkelJaar() **throws** IOException **...** **public void** datumsVerskil() **throws** IOException **...**

Evidence of monitoring tasks during programming
 	
Mark: 4
	Monitoring
	Create the constructor.
	I have problems reading in the date in the correct format.
	The dateDifference( ) method is difficult and I cannot think of a way immediately.
	It is difficult to determine how to count from the first date up to the second date. I cannot think of a way to do it right now.
		System.out.print("Wat is vandag se datum (DD Month YYYY): ");
		...
		yearTemp = Integer.parseInt(yearStr);
		monthTemp = input.substring(3,(input.length()-5));
		String dayStr = input.substring(0,2);
		dayTemp = Integer.parseInt(dayStr);
		
		trueDate = testDate(dayTemp, monthTemp, yearTemp);

Evidence of regulation or modification to correct flaws during programming
 	
Mark: 4
	Regulation
	I have problems determining a specific date format and decided on the format: DD Month YYYY e.g., 16 October 2006.
	In the dateDifference( ) method I can add the days from 1 January 1800 up to date1. This method is difficult and I should provide for many exceptions, especially for leap years
		{
		if ((1800 + i) % 100 == 0)
		if ((1800 + i) % 400 == 0)
		numSkrikYears1++;
		else
		...
		}

- ***Planning strategy***

  P32 used well-structured planning strategies to set goals and to analyse the programming task. For example: 'Create a framework', 'Create a constructor', and 'Create the testDate method' (Table 5.8). During this strategy, an appropriate sequence of actions was followed to plan the new program. P32's thinking processes included multiple steps of explicit objectives, goals and strategies for the programming

task and he stated clearly what he intended to achieve as indicated in Table 5.8. By using a comprehensive planning strategy and sticking to the plan, P32 enhanced successful task completion and produced a quality solution.

- ***Monitoring strategy***

  P32 indicated that he had problems with the correct date format and the difference between two dates (Table 5.8): *It is difficult to determine how to count from the first date up to the second date. I cannot think of a way to do it right now.* He reflected on his own cognitive processes and recognised the situations that he did not understand; he paused, reconsidered the problem and then decided what to do (§4.4.2). This strategy allowed effective management of the programming process and enabled him to guide the process to a satisfactory solution.

- ***Regulation strategy***

  P32 was continuously able to modify his own cognitive activity (§4.4.3). He identified various errors during monitoring and made appropriate changes to regulate his own cognitive processes. For example, he had problems with the date format and decided to use the following fixed format: DD Month YYYY (Table 5.8). It was also necessary to include certain exceptions to handle leap years. This participant was able to regulate his progress, build on his own strengths, exploit possible solutions and eventually to resolve the programming problems.

5.2.3.3 Problem-solving strategy

Participants were allowed to use more than one problem-solving strategy during programming (§4.5). However, as indicated in Table 5.9, P32 used the top-down strategy only, and with great success. This strategy entails the type of understanding that progresses from the general to the more specific (§4.5.1.2). This strategy is explicitly indicated by P32's actions when he starts 'with the framework for the *Date class* and *Test class'*, with the method headings only. The top-down strategy requires high-level planning and understanding of the overall system, without initially going into low-level details.

Table 5.9: The problem-solving strategy used by P32

Problem-solving strategy		
Measurement criteria	**Thinking processes**	**Evidence in the program**
Application of problem-solving strategies during programming **Mark: 8**	**Top-down strategy** *I will start with the framework for the Date class and Test class, headings, import given methods, etc.*	Not applicable

5.2.3.4 Application of the measurement criteria in the Java program

The computer programs were analysed to determine whether participants applied their knowledge and skills during programming the *Date class* and *Test class*. Program 5.3 following is an extract from P32's *Date class* program (Appendix G). Program 5.4 is a segment of the driver or *Test class* program determining the correct output (Appendix G). The highlighted segments in Programs 5.3 and 5.4 are associated with certain of the measurement criteria in Table 5.1. The corresponding categories of the specific measurement criteria, shown in framed labels, were added by the researcher.

```
import java.io.BufferedReader;                          Programming statements
import java.io.InputStreamReader;
import java.io.IOException;

public class Datum
{
BufferedReader console = new BufferedReader (new InputStreamReader(System.in) );
                                                        Programming statements

        String[] months31 = { "Januarie", "Maart", "Mei", "Julie", "Augustus", "Oktober",
        "Desember" };                                   Programming statements
        String[] months30 = { "April", "Junie", "September", "November" };
        String[] months = { "Januarie", "Febuarie", "Maart", "April", "Mei", "Junie",
        "Julie", "Augustus", "September", "Oktober", "November", "Desember" };

        private int day, year;                          Access control
        private String month;

        boolean trueDate = false;                       Programming statements

        public Datum() throws IOException               Exception handling
                //Konstruktor van die Datum klas
        {                                               Method application
```

Program 5.3: Java program segment from the *Date class* program of P32

```
        // Lees vandag se datum in wanneer objek gemaak word
        String input = "";                          Programming techniques
        // Toets dan die datum
        int dayTemp, yearTemp;
        String monthTemp;

        while ( trueDate == false )                 Programming statements
        {                                           Reasoning
          System.out.print("Wat is vandag se datum (DD Month YYYY): ");
          // Lees datum in
          input = console.readLine();
          String yearStr = input.substring( (input.length()-4),input.length());
          yearTemp = Integer.parseInt(yearStr);     Programming statements
          monthTemp = input.substring(3,(input.length()-5));
          String dayStr = input.substring(0,2);
          dayTemp = Integer.parseInt(dayStr);

          trueDate = testDate(dayTemp, monthTemp, yearTemp);
          // Toets datum                            Parameter passing
          if ( trueDate == false )
            System.out.println("Datum was inkorrek ingevoer, doen asb weer");
            // Indien inkorrek, herhaal die vraag   User-friendliness
            else
            {
            year = yearTemp;                        Reasoning
            // Stel waardes van ingeleesde datum gelyk aan globale veranderlikes

            for ( int i = 0; i < months.length ; i++) Programming statements

            if ( monthTemp.equalsIgnoreCase(months[i]) )
                {month = months[i];                 Programming statements
                day = dayTemp;}
            }
        }
                                                    Access control   Parameter passing
        public boolean testDate(int dayTemp, String monthTemp, int yearTemp)
        // Metode wat ek geskryf het om die datums wat ingelees word, te toets
        {                                           Method application
                int monthNum;
                int numDays = 0;
                boolean testMonth = false;          Programming techniques
                boolean testDay = false;
                boolean testYear = false;
                boolean yearSkrik = false;

                if ( yearTemp >= 1800 )
                // Toets of jaartal groter is as 1800, soos aangewys
                {
                if ( yearTemp % 100 == 0 )          Programming statements
                // Toets vir skrikkeljare, bedoel vir Februarie met sy verskil in dae
                if ( yearTemp % 400 == 0 )
                        yearSkrik = true;           Programming statements
                }
...} // Complete program in Appendix G
```

Program 5.3 (continued): Java program segment from the *Date class* program of P32

```
public class Toetsklas
{
        public static void main (String[] args) throws IOException
        {
                BufferedReader console = new BufferedReader( new
                InputStreamReader(System.in) );
                String input = "";
                Datum datum = new Datum();
            ...
        } ...
}// Complete program in Appendix G
```

Classes and objects

Exception handling

Classes and objects

Program 5.4: Java program segment from the *Test class* program of P32

C:\WINDOWS\system32\cmd.exe - java Toetsklas

```
E:\proj>java Toetsklas
Wat is vandag se datum (DD Month YYYY): 18 Oktober 2006
```

What is today's date (DD Month YYYY): 18 October 2006

C:\WINDOWS\system32\cmd.exe - java Toetsklas

```
Vandag se datum is: 18 Oktober 2006
##################################################################
#                    DATUM KLAS - TOETS PROGRAM                  #
#                                                                #
#  1. Toets of jaargetal 'n skrikkeljaar is of nie               #
#  2. Bereken die verkil tussen twee datums                      #
#  3. Exit                                                       #
#                                                                #
##################################################################
Jou Keuse: 1
Tik in 'n jaar getal(Enige jaar vanaf 1800): 1984
Die jaar 1984 is 'n skrikkeljaar
```

Your Choice: 1
Type in the year number (Any year from 1800): 1984
The year 1984 is a leap year

C:\WINDOWS\system32\cmd.exe - java Toetsklas

```
Vandag se datum is: 18 Oktober 2006
##################################################################
#                    DATUM KLAS - TOETS PROGRAM                  #
#                                                                #
#  1. Toets of jaargetal 'n skrikkeljaar is of nie               #
#  2. Bereken die verkil tussen twee datums                      #
#  3. Exit                                                       #
#                                                                #
##################################################################
Jou Keuse: 2
Eerste datum (DD Month YYYY of 'vandag se datum'): vandag se datum
Tweede datum (DD Month YYYY of 'vandag se datum'): 09 Mei 1984
Die verskil tussen hierdie datums is: 8197 dae
```

Your Choice: 2
First date (DD Month YYYY of 'today's date'): today's date [18 October 2006]
Second date (DD Month YYYY of 'today's date'): 09 May 1984
The difference between these dates is: 8197 days.

Measurement criteria:

Solution of problem

Program evaluation

Correctness of output

Figure 5.3: Java output from the *Date class* and *Test class* programs of P32

In Programs 5.3 and 5.4 and in the program output given in Fig. 5.3, certain portions are highlighted to indicate examples of the OOP measuring criteria from Table 5.1. Table 5.10 shows the marks allocated to P32 for the OOP section. As in the previous categories, P32 obtained full marks for all the criteria.

Table 5.10: Marks allocated to *P32 for OOP

Measurement criteria for OOP	Marks (4)
Analysis of the program requirements	4
Programming techniques used: indentation, readability, variable names and declaration	4
Application of the correct use of programming statements	4
Application of user-friendliness and usability	4
Design of classes and instantiation of objects	4
Application of methods, such as constructors, mutators and accessors	4
Decision on the accessibility: public, private	4
Application of parameter passing: number, order, type of variables	4
Application of reasoning skills in OOP	4
Application of exception handling	4
Application of program structure and scope	4
Actual solution to the problem	4
Evaluation of the *Date class* and *Test class*	4
Evidence of correct program output and test data used	4

* P32's complete programs are included in Appendix G.

The final marks for all sections, as well as the overall percentage, are shown in Table 5.11, where they are compared with those of P31. Analysis of P32's thinking processes and computer programs revealed sound examples of cognitive, metacognitive and problem-solving knowledge, skills and strategies, which were used successfully to complete the *Date class* and *Test class* programs. Due to the length of the complete program (Appendix G), segments of code were selected and are displayed in Programs 5.3 and 5.4. These illustrate P32's approach to the aspects addressed by the measurement criteria relevant to this example.

5.2.4 Evaluation of the computer programs: Participants 31 and 32

P31 had problems in applying the various skills of Bloom's taxonomy during computer programming, especially the higher-order skills required during problem solving (analysis, synthesis and evaluation). These findings are in line with those by Zant (2005), who claims that it is difficult for a novice programmer to become an expert without progressing through each of the six levels of Bloom's taxonomy (§3.3.2).

P31 also found the different metacognitive strategies challenging. No evidence of planning or regulation was given and little evidence of monitoring was included during the programming task (Table 5.4). P31 used the trial-and-error strategy and typed programming code without any explicit planning, as indicated in Table 5.5: *I have typed all the things that I thought should be in the program.* These findings are in line with those by Détienne (2003:21), who emphasises that the lowering of the level of control may result in using the trial-and-error strategy. This participant did not use the skills required for OOP, as shown in Programs 5.1 and 5.2.

P31 could not interpret and judge the program. He did not have a concept of the big picture and tried to program without fully understanding the programming question and without applying knowledge, skills and strategies to solve the problem successfully. For example, he had problems using the correct programming statements. This is also evident in his inability to apply procedural knowledge to determine the problem's solution (§3.1).

P31 is an example of an **unsuccessful programmer** (§3.5.3, §5.2.7) who displays limited knowledge and thinking skills in the programming domain. He could not demonstrate an understanding of OOP basics, such as defining classes, using the correct programming statements, creating and/or using methods and applying error handling. Furthermore, he could not solve the actual problem nor could he provide evidence of program output (Fig. 5.2). He showed deficiencies in cognitive, metacognitive and problem-solving knowledge, skills and strategies. This is supported by the examples given in §3.3.3, §3.4.3, §3.5.4, and §4.4. Table 5.11 shows that P31 obtained a final mark of only 16%.

By contrast, P32 did very well and completed the programming task with a final mark of 100% as shown in Table 5.11. He used correct programming statements and applied procedural knowledge to solve the problem (§3.1). Furthermore, P32 used all the skills of Bloom's taxonomy and applied various metacognitive strategies during the programming

process (§3.3.2). He clearly used a top-down approach during problem solving and program comprehension and applied his OOP skills very effectively.

P32 is an example of a **successful programmer** (§3.5.3, Table 3.5), who showed evidence of a well-organised personal knowledge structure. This is clearly illustrated in the different examples given in Tables 5.7 to 5.9, Programs 5.3 and 5.4, program output in Fig. 5.3, Appendix G and §5.2.7. He perceived large meaningful patterns, which guided his thinking and he worked through different levels of abstraction during the programming process. This participant could decompose the *Date class* program into smaller problem-solving units and he could determine which calculations were necessary. Moreover, he used detailed planning, monitoring and regulation strategies to reflect on the programming task.

P32 used a top-down strategy and applied different OOP skills very effectively to solve the problem. He started with goals and specific plans to solve the problem, which he succeeded in doing, and he completed the *Date class* and *Test class*. He used test data effectively to determine whether the program's output was correct (Fig. 5.3). P32 appropriately orchestrated all the knowledge, skills and strategies to solve the problem completely.

The marks allocated to Participants 31 and 32 are shown in Table 5.11, indicating the scores they obtained for each criterion, as well as the final percentage each obtained.

Possible solutions to the *Date class* programming task are provided, one in Java and one in Delphi. They are included on the CD inside the back cover of this thesis.

Table 5.11: Summary and comparison of the marks allocated to P31 and P32

Category	P31	P32
Cognitive knowledge and skills		
Evidence of knowledge of the programming language	2	4
Interpretation of the problem	2	4
Application of prior knowledge in a new program	1	4
Analysis of the problem – breaking it down in steps	1	4
Design of a new program	1	4
Evaluation of the solution	1	4
Metacognitive strategies		
Evidence of planning during programming	0	4
Evidence of monitoring tasks during programming	1	4
Evidence of regulation or modification to correct flaws during programming	0	4
***Problem-solving strategies** [8]	0	8
*Bottom-up (BU) / Top-down (TD) / Integrated (IG) / Trial-and-error (TE)	TE	TD
OOP knowledge and skills		
Analysis of the program requirements	1	4
Programming techniques used: indentation, readability, variable names and declaration	1	4
Application of the correct use of programming statements	0	4
Application of user-friendliness and usability	2	4
Design of classes and instantiation of objects	0	4
Application of methods, such as constructors, mutators and accessors	0	4
Decision on the accessibility: public, private	0	4
Application of parameter passing: number, order, type of variables	0	4
Application of reasoning skills in OOP	1	4
Application of exception handling	0	4
Application of program structure and scope	2	4
Actual solution to the problem	0	4
Evaluation of the *Date class* and *Test class*	0	4
Evidence of correct program output and test data used	0	4
TOTAL 100 (%)	**16**	**100**

5.2.5 Quantitative analysis – statistical methods

Various statistical methods are applied in this study, including confirmatory factor analysis, reliability testing, descriptive statistics and practical significance calculations. These methods, which are explained in this subsection, are used to analyse the raw data in Table 5.2, and the findings are outlined in the following subsection (§5.2.6).

- ***Confirmatory factor analysis***

  The purpose of factor analysis is to discover simple patterns of relationships between variables and to determine whether the observed variables can be explained largely or entirely in terms of a much smaller number of variables called factors. Confirmatory factor analysis was used to validate the constructs formed i.e., cognition, metacognition, problem solving and OOP. According to Johnson and Wichen (1992:396), if variables can be grouped by their correlations, then each group of variables represents a single underlying construct or factor. Factor analysis was based on the correlation matrix. The method of Kaiser was used to determine the number of factors extracted i.e., all the factors with eigenvalues greater than 1 were extracted.

  In addition, the Kaiser-Meyer-Olkin measure (KMO) of sampling adequacy (Kaiser, 1970) was used. Kaiser (1974) recommends the following interpretation concerning the study population based on the KMO measure:

  a) values below 0.5: unacceptable;
  b) between 0.5 and 0.7: moderate;
  c) between 0.7 and 0.8: good;
  d) between 0.8 and 0.9: great; and
  e) above 0.9: superb.

- ***Reliability testing***

  The reliability of the scale of each construct was also tested. According to Field (2005:666), another way of looking at reliability is to say that two people who are the same in terms of the construct being measured should get the same score. The Cronbach-alpha measure is utilised for this purpose. The Cronbach-alpha coefficient (α) measures reliability and has a value between 0 and 1 (Gliem & Gliem, 2003). Kline (1999) mentions that a value above 0.7 is good. However, after consulting the statistical consultation services at the institution where the research was done, it was decided that a value above 0.5 is also acceptable, since this is a high correlation of effect size (Ellis & Steyn, 2003:52).

- ***Descriptive statistics***

  The following descriptive statistics are included:

  - the mean value as a measure of central location ($\bar{x}$);
  - standard deviations (s) as a measure of spreading; and
  - Spearman rank correlation (*r*) as a measure of a monotone relationship between variables where:
    a) *r* = 0.1 implies a small effect;
    b) *r* = 0.3 implies a medium effect;
    c) *r* = 0.5 implies a large effect.

  Data with an *r*-value of 0.5 or higher, is considered as practically significant (Ellis & Steyn, 2003:52; Steyn, 2002).

- ***Practical significance: effect size calculations and correlation***

  Effect size

  The Cohen's d value was used to calculate the effect size for the difference between means. This value is given by: $d = \frac{|\bar{x}_1 - \bar{x}_2|}{s_{\max}}$, where $\bar{x}_1$ and $\bar{x}_2$ represent the mean values of the groups to be compared respectively and $s_{\max}$ is the maximum standard deviation (Ellis & Steyn, 2003). Cohen (1988) gives the following guidelines for interpretation of the effect size:

  a) *d* = 0.2 implies a small effect;
  b) *d* = 0.5 implies a medium effect;
  c) *d* = 0.8 implies a large effect.

  Correlation

  The Cohen's (1988) d value for relationships between two variables is given by $d = |r|$ where *r* represents the correlation coefficient. Guidelines for the interpretation of the effect size (Steyn, 2002) are:

  a) *d* = 0.1 implies a small effect;
  b) *d* = 0.3 implies a medium effect;
  c) *d* = 0.5 implies a large effect.

5.2.6 Evaluation of all participants' computer programs and thinking processes

Table 5.2 (which has already been discussed with regard to the performance of P31 and P32) illustrates the submarks and final percentage allocated to each participant. The scale was from 1 to 4, where 1 implies a low test score and 4 a high test score. However, before giving an overview of the overall scoring, various categories of marks are addressed. The values for each category namely, cognitive knowledge and skills, metacognitive strategies, problem-solving strategies and object-oriented programming, and the success rate in terms of correct program output are discussed. In addition, the correlation between various constructs is outlined.

5.2.6.1 Cognitive knowledge and skills

In this subsection, the following statistical measures are used: sample adequacy, factor analysis and reliability, mean values and standard deviations of cognitive knowledge and skills. Table 5.2 provides the raw data.

- *Sample adequacy, factor analysis and reliability*
  The KMO measure of information adequacy is 0.87, which indicates a suitable ('great') amount of data to perform a factor analysis (see Subsection 5.2.5) i.e., the size of the sample is more than adequate. The Kaiser method extracts one factor that shows that all the variables within the cognitive category – knowledge, comprehension, application, analysis, synthesis and evaluation – are highly correlated amongst themselves, hence the construct (cognitive knowledge and skills) is valid. The extracted factor explains 79.52% of the total variation. The scale of the construct is found to be reliable with a Cronbach-alpha value of 0.95, which implies that the cognitive category as a whole forms a reliable construct.

- *Mean values and standard deviation*
  Table 5.12 shows the mean values ($\bar{x}$) as well as the standard deviations (s) for all the participants (n=48) for each cognitive subcategory. The maximum score used for each section was 4. The mean values range from $\bar{x} = 3.73$ (knowledge) to $\bar{x} = 2.40$ (evaluation). The notable decreasing tendency in the $\bar{x}$ columns shows that participants experienced problems in using the higher-order skills (§3.3.2), especially synthesis and evaluation of computer programs. This indicates that participants had some difficulty in combining parts of a problem to form a new program (synthesis) and in evaluating their program. The mean value, namely 3.23, for the complete construct (cognitive knowledge and skills) is shown in the last row of Table 5.12.

Table 5.12: Mean values and standard deviations for cognitive knowledge and skills

Cognitive knowledge and skills	$\bar{x}$	s
Knowledge	3.73	0.61
Comprehension	3.65	0.60
Application	3.48	0.74
Analysis	3.25	0.79
Synthesis	2.88	0.96
Evaluation	2.40	1.09
Construct	**3.23**	**0.71**

5.2.6.2 Metacognitive strategies

The statistical measures applied in this subsection are: sample adequacy, factor analysis and reliability; mean values and standard deviations of metacognitive strategies. These methods were introduced in §5.2.5 and, once again, Table 5.2 provides the raw data.

- *Sample adequacy, factor analysis and reliability*

  For this construct (metacognitive strategies), the KMO measure of information adequacy is 0.67, which indicates a moderate amount of data to perform a factor analysis (§5.2.5). The Kaiser method extracted one factor, which shows that all the variables – planning, monitoring and regulation – are correlated amongst themselves. The extracted factor explained 78.09% of the total variation. The scale of this construct is found to be reliable with a Cronbach-alpha value of 0.86.

- *Mean values and standard deviation*

  During the programming process, participants struggled more with the monitoring ($\bar{x} = 2.44$) (§4.4.2) and regulation strategies ($\bar{x} = 1.92$) (§4.4.3) than with the planning strategy ($\bar{x} = 3.40$) (§4.4.1), as shown by the mean values and construct (metacognitive strategies) in Table 5.13. The mean value, namely 2.58, for the complete construct is shown in the last row. Participants could not easily reflect on their programming task and found it difficult to manage their thinking during programming (§4.4.4).

Table 5.13: Mean values and standard deviations for metacognitive strategies

Metacognitive strategy	$\overline{x}$	s
Planning	3.40	0.79
Monitoring	2.44	1.15
Regulation	1.92	1.11
Construct	**2.58**	**0.91**

5.2.6.3 Problem-solving strategies

Table 5.14 contains frequencies of the problem-solving strategies selected by various participants, using the raw data of Table 5.2. As shown in Table 5.14, only two participants used no strategy at all. The great majority (34) of participants used the bottom-up strategy (§4.5.1.1), whereas some participants (5) used the top-down (§4.5.1.2) and five participants used an integrated strategy (§4.5.1.3). Two participants used trial-and-error (§4.5.1.5). The majority of participants used the bottom-up strategy, probably because they experienced problems with planning a framework of the overall program and found it easier to start with the details of a class or a method.

Table 5.14: Frequencies for selected problem-solving strategies

Problem-solving strategy	Participants (n=48)
Bottom-up	34
Top-down	5
Integrated	5
Trial-and-error	2
None	2

5.2.6.4 Values allocated for object-oriented programming

In this subsection, the following statistical measures are applied to the raw data from Table 5.2: sample adequacy, factor analysis and reliability, mean values and standard deviations of OOP.

- *Sample adequacy, factor analysis and reliability*

  The KMO measure of information adequacy is 0.90, which indicates a highly suitable amount of data to perform a factor analysis (§5.2.5). The Kaiser method extracted two factors, namely OOP ability and Program execution, which are shown in the central and last columns of Table 5.15. The table below shows various OOP variables that contribute to one or both factors and that measure different aspects of the OOP knowledge and skills construct.

Table 5.15: Summary statistics for OOP knowledge and skills

OOP knowledge and skills	Factor 1 (OOP ability)	Factor 2 (Program execution)
Analysis of the program requirements	0.62	0.42
Programming techniques used: indentation, readability, variable names and declaration	0.83	0.37
Application of the correct use of programming statements	0.91	
Application of user-friendliness and usability		0.67
Design of classes and instantiation of objects	0.92	
Application of methods, such as constructors, mutators and accessors	0.85	0.34
Decision on the accessibility: public, private	0.92	
Application of parameter passing: number, order, type of variables	0.94	
Application of reasoning skills in OOP	0.75	0.48
Application of exception handling		0.88
Application of program structure and scope	0.78	
Actual solution to the problem	0.80	0.48
Evaluation of the *Date class* and *Test class*	0.72	0.60
Evidence of correct program output and test data used		0.88

The first factor measures participants' OOP ability whilst the second measures program execution in the OOP context. Hence, there are two groups of variables measuring different aspects of OOP. OOP ability refers to participants' ability to apply various OOP knowledge and skills such as the correct use of programming statements, design of classes and instantiation of objects, accessibility of methods, parameter passing and program structure and scope. The second factor, Program execution, refers to the application of various knowledge and skills to ensure correct program execution, for example, user-friendliness, exception handling and correct program output.

Since both factors measure different aspects of OOP, this construct (OOP knowledge and skills) will be treated as a whole. The two factors extracted, explained 79.83% of the total variation. The scale of this construct is found to be reliable with a Cronbach-alpha value of 0.96 (§5.2.5).

Complex variables are variables that contribute to both factors i.e., those that appear in both factor columns in Table 5.15 and are indicated with an asterisk below.

OOP ability: variables loading high on factor 1

- Application of parameter passing: number, order, type of variables;
- Decision on the accessibility: public, private;
- Design of classes and instantiation of objects;
- Application of the correct use of programming statements;
- Application of program structure and scope;
- *Analysis of the program requirements;
- *Programming techniques used: indentation, readability, variable names and declaration;
- *Application of methods, such as constructors, mutators and accessors;
- *Application of reasoning skills in OOP;
- *Actual solution to the problem; and
- *Evaluation of the *Date class* and *Test class.*

Program execution: variables loading high on factor 2

- Application of exception handling;
- Evidence of correct program output and test data used;
- Application of user-friendliness and usability.

Reliability was also tested for these factors. The Cronbach-alpha values (§5.2.5) for these two factors are $\alpha = 0.96$ and $\alpha = 0.81$ respectively, meaning that all the OOP variables within one construct are found to be reliable i.e., the OOP variables are the same in terms of the construct being measured.

- *Mean values and standard deviation*

  Table 5.16 illustrates mean values and standard deviations of the various OOP knowledge and skills that comprise the object-oriented programming category in this study, listing them in ascending order of $\bar{x}$. The last row shows the mean and standard deviation of the OOP construct as a whole ($\bar{x} = 2.71$).

Table 5.16: Mean values and standard deviations for OOP knowledge and skills

OOP knowledge and skills	$\bar{x}$	s
Application of exception handling	0.92	1.26
Evidence of correct program output and test data used	1.00	1.37
Application of user-friendliness and usability	1.94	1.33
Evaluation of the *Date class* and *Test class*	2.35	1.18
Actual solution to the problem	2.71	1.13
Application of methods, such as constructors, mutators and accessors	2.92	0.94
Application of reasoning skills in OOP	3.08	0.82
Application of program structure and scope	3.06	0.8
Design of classes and instantiation of objects	3.17	1.06
Application of the correct use of programming statements	3.27	0.98
Programming techniques used: indentation, readability, variable names and declaration	3.31	0.93
Decision on the accessibility: public, private	3.35	1.00
Application of parameter passing: number, order, type of variables	3.42	1.03
Analysis of the program requirements	3.42	0.82
Construct	**2.71**	**0.85**

Very few participants included exception handling in their programs ($\bar{x}$ = 0.92). A possible explanation is that they did not know how to apply exception handling. The mean value for correct program output and test data is $\bar{x}$ = 1.00, indicating that only a few participants were able to design, write and test the complete program successfully (§3.2).

Although user-friendliness has a relatively low mean value of $\bar{x}$ = 1.94, a contributing factor is that many participants conducted the programming in Java, a language that requires highly detailed programming. Considerably more time would be required to program a graphical user interface with buttons, labels, etc. (depending on the Java

Application Programming Interface being used). By contrast, it is simpler in Delphi, where it is possible to use an available graphical user interface, in which different messages can be used to support user-friendliness and usability. The mean value for solving the actual programming problem is only $\bar{x} = 2.71$, and the mean value for evaluating the whole problem with both classes (*Date class* and *Test class*) included, is $\bar{x} = 2.35$.

High mean values occurred for certain criteria. For example, the participants did not experience problems in analysing and determining the program requirements. They applied programming techniques such as indentation and readability satisfactorily. They used descriptive variable names and could declare the variables correctly. Participants could distinguish between different ways of method accessibility. Overall, they experienced problems with user-friendliness and usability, exception handling and in obtaining the correct output from their programs, indicating that many of them were making semantic errors in their code.

5.2.6.5 Correlations between various constructs

There is an indication that possible correlations exist between participants' expertise in cognition, metacognition and OOP knowledge and skills. The Spearman correlations between pairs of these variables (§5.2.5) are shown in Table 5.17. This test is also aimed at testing for a monotone relationship between variables to indicate the practical significance. In all the constructs measured, correlations are larger than 0.5 and therefore relevant in practice (Ellis & Steyn, 2003:52; Steyn, 2002).

Table 5.17: Correlations between cognition, metacognition and OOP knowledge and skills

Construct	*r*
Cognition Metacognition	0.63**
Cognition OOP	0.89**
Metacognition OOP	0.73**

** Practically significant (Steyn, 2002).

The high correlation between cognition and OOP (r = 0.89) implies that certain predictions can be made about successful programming in cases where a programmer makes effective use of all his cognitive knowledge and skills (Table 5.17). This shows that the higher-order thinking skills, such as analysis, synthesis and evaluation, are required during a programming task. This was possibly the reason why P31 experienced problems completing and solving the programming problem, as he did not use all the forms of cognitive knowledge and skills (Table 5.3 – Table 5.5).

Note that the correlation between cognition and metacognition is lower (r = 0.63) than the correlation between cognition and OOP (r = 0.89). The correlation between metacognition and OOP is r = 0.73. This indicates that the use of metacognition and reflection can support problem-solving performance in OOP. If the participants had used all the metacognitive strategies they would have been able to manage their own programming performance better and guide themselves towards finding the solution.

5.2.7 Knowledge, skills and strategies used by successful programmers

Subsections 5.2.2 and 5.2.3 were detailed studies of the performance and approach of two *particular* programmers, one who produced a poor program and another who wrote an extremely successful program, respectively. This section takes a more *general* view of the characteristics of a number of 'unsuccessful' and 'successful' programmers, as defined by the present researcher. With reference to the measurement criteria in Table 5.1 it was decided to classify successful and unsuccessful programmers according to their application of knowledge, skills and strategies during problem solving in OOP.

Table 5.18 was extracted from Table 5.2 by selecting all participants who scored 3 or 4 for the measurement criterion 'Correctness of output'. A successful performance cannot be determined solely by the final percentage obtained. Note in Table 5.2 that some participants obtained a total of more than 80%, but did not achieve correct program output (e.g. Participants 6 and 7). P7, for example, obtained a final mark of 85% but obtained only 1 out of 4 for correct program output. It was therefore decided that the factor of correct program output, which indicates whether or not a participant's program(s) executed successfully, is a better indication of successful programming than the final percentage. Furthermore, in Table 5.16 the mean value for the criterion of 'Correctness of output and test data used' is only 1.00. With this background, the measure of success for program output on a scale of 4 was defined as: **x ≥ 3**, leading to the generation of Table 5.18 out of Table 5.2 to show only the 'successful participants'. The success measure is indicated by a shaded block.

Table 5.18: Allocated values of successful programmers

Values allocated to successful programmers											
Participant number	**12**	**15**	**23**	**28**	**29**	**32**	**38**	**40**	**42**	**44**	**48**
Category											
Cognitive knowledge and skills											
Knowledge	4	4	4	4	4	4	4	4	4	4	4
Comprehension	4	4	4	4	4	4	4	4	4	4	4
Application	4	4	4	4	4	4	4	4	4	4	4
Analysis	4	3	4	4	4	4	4	4	4	4	3
Synthesis	4	3	4	4	4	4	4	3	4	4	3
Evaluation	4	3	3	4	4	4	3	3	4	4	3
Metacognitive strategies											
Planning	4	4	4	4	4	4	4	4	3	4	4
Monitoring	4	3	4	4	4	4	2	3	2	3	3
Regulation	4	2	3	3	3	4	2	2	2	3	3
***Problem-solving strategies**	8	8	8	8	8	8	8	8	8	8	8
OOP knowledge and skills											
Program requirements analysis	4	4	4	4	4	4	4	4	4	4	4
Programming techniques	4	4	4	4	4	4	4	4	4	4	4
Programming statements	4	4	4	4	4	4	4	4	4	4	3
User-friendliness	3	3	2	3	4	4	3	2	2	4	3
Classes and objects	4	3	4	4	4	4	4	4	4	4	3
Method application	4	3	4	4	4	4	3	3	4	4	3
Access control	4	4	4	4	3	4	4	4	4	4	4
Parameter passing	4	4	4	4	4	4	4	4	4	4	4
Reasoning	4	3	4	4	4	4	4	3	4	4	3
Exception handling	3	0	3	3	3	4	2	1	3	4	2
Program structure and scope	3	3	4	4	4	4	4	4	4	4	3
Solution of problem	4	3	4	4	4	4	4	3	4	4	3
Program evaluation	3	3	4	4	4	4	4	3	3	4	3
Correctness of output	**3**	**3**	**3**	**3**	**4**	**4**	**3**	**3**	**3**	**3**	**3**
TOTAL (%)	95	82	94	96	97	100	90	85	90	97	84
*Problem-solving strategy	BU	TD	IG	BU	IG	TD	BU	BU	BU	BU	BU

* BU = Bottom-up, TD = Top-down, IG = Integrated strategy

The proportion of successful participants from the Delphi programmers (i.e., those who produced the correct output) was only 1 out of 14, or 7.14%. The corresponding proportion for Java programmers was 10 out of 34 participants, or 29.41%. A possible explanation for the better performance of Java programmers is the fact that they were BSc students who received instruction in programming from their first year of study, while the BEd students only started learning to program in their second year. Another factor is that the BEd students were more focused on the teaching of programming in schools, whereas the BSc students focus on formal programming tasks as applied in industry. In total, only 11 out of 48 participants**,** or 23%, were successful. It should be remembered that the participants completed the instructional component of the course two or three months before doing the

Date class task, due to other information technology modules being interspersed with programming in their third year studies. Table 5.19 shows the means and standard deviations for unsuccessful and successful participants for subcategories of the three major categories.

Table 5.19: Means, standard deviations and practical significance of unsuccessful and successful programmers

Category	Unsuccessful programmers (37)		Successful programmers (11)		Practical significance
	$\overline{x}$	s	$\overline{x}$	s	d
Cognitive knowledge and skills	**3.05**	**0.71**	**3.85**	**0.20**	**1.13***
Knowledge	3.65	0.68	4.00	0.00	0.51
Comprehension	3.54	0.65	4.00	0.00	0.71
Application	3.32	0.78	4.00	0.00	0.87*
Analysis	3.08	0.80	3.82	0.40	0.93*
Synthesis	2.62	0.92	3.73	0.47	1.21*
Evaluation	2.05	0.97	3.55	0.52	1.55*
Metacognitive strategies	**2.36**	**0.88**	**3.33**	**0.54**	**1.10***
Planning	3.24	0.83	3.91	0.30	0.81*
Monitoring	2.19	1.13	3.27	0.79	0.96*
Regulation	1.65	1.06	2.82	0.75	1.10*
OOP knowledge and skills	**2.44**	**0.77**	**3.62**	**0.29**	**1.53***
Program requirements analysis	3.24	0.86	4.00	0.00	0.88*
Programming techniques	3.11	0.97	4.00	0.00	0.92*
Programming statements	3.08	1.04	3.91	0.30	0.80*
User-friendliness	1.62	1.30	3.00	0.77	1.06*
Classes and objects	2.97	1.12	3.82	0.40	0.76
Method application	2.70	0.94	3.64	0.50	1.00*
Access control	3.19	1.08	3.91	0.30	0.67
Parameter passing	3.24	1.12	4.00	0.00	0.68
Reasoning	2.89	0.81	3.73	0.47	1.04*
Exception handling	0.46	0.80	2.55	1.21	1.73*
Program structure and scope	2.86	0.88	3.73	0.47	0.99*
Solution of problem	2.41	1.09	3.73	0.47	1.21*
Program evaluation	2.00	1.08	3.55	0.52	1.44*
Correctness of output	0.35	0.72	3.18	0.40	3.93*

* d = 0.8, large effect size; d = 0.5, medium effect size (Ellis & Steyn, 2003:51).

Practical significant differences with a large effect size were found between successful and unsuccessful participants within all subcategories except for knowledge, comprehension, classes and objects, access control and parameter passing, where practical significant differences of a medium effect size occurred. A discussion follows on the knowledge, skills and strategies that unsuccessful and successful participants used during their programming task.

- *Cognitive knowledge and skills*

  Unsuccessful participants could not readily apply higher-order thinking skills. Consequently, they had problems in interpreting their errors, they could not complete the program, and many did not obtain output.

  The successful programmers achieved mean values of more than 3.5 on a 4-point scale for the higher-order thinking skills (application, analysis and synthesis) necessary during programming. This indicates that these participants explicitly used cognitive knowledge and skills while programming. It was noted that the successful participants used all the levels of Bloom's taxonomy given in the cognition section (Table 3.1, Table 5.19). Their performances illustrate that programmers should understand a problem precisely, and be able to interpret and evaluate their programming solutions. These findings correspond with Zant's (2005) results.

- *Metacognitive strategies*

  Unsuccessful participants found it difficult to apply metacognitive activities during programming. They encountered problems in monitoring and regulating ($\bar{x}$ = 1.65) their cognitive resources. Very few of them applied any form of regulatory strategy.

  Successful participants made extensive use of metacognitive strategies that included planning ($\bar{x}$ = 3.91) and monitoring ($\bar{x}$ = 3.27). However the mean value of their regulation strategy is slightly lower than 3 ($\bar{x}$ = 2.82), which implies that they could improve on their use of regulation strategies during programming. These findings correspond with findings by Hertzog and Robinson (2005:110, 111), who mention that monitoring plays a vital role in the cognitive performance of complex problem solving, since it can be used to guide the process of finding a solution.

- *Problem-solving strategies*

  Although most of the unsuccessful participants used a bottom-up strategy (27), some wrote that they worked without using any specific problem-solving strategies (2). Two used trial-and-error, three used a top-down strategy, and three used the integrated strategy.

  As shown in Table 5.18, seven successful participants used the bottom-up strategy, two used top-down, and two participants used the integrated strategy during program comprehension. Note that no successful participant used the trial-and-error strategy. This also implies that using trial-and-error is less successful than other problem-solving strategies in OOP.

- *Knowledge, skills and strategies in OOP*

  Unsuccessful participants did not obtain the required program output. Instead, they spent time iterating through their programming code to address errors, without understanding which sections were incorrect and how to rectify them. Such participants were much less accurate in their efforts to reach an appropriate solution.

  Successful participants systematically applied different knowledge, skills and strategies during programming. With the exception of 'Exception handling' ($\bar{x}$ = 2.55), all the other mean values are above 3 on a 4-point scale (Table 5.19). Participants should use more explicit exception-handling techniques in their programs to prevent errors during execution. Raising an exception is a useful way to signal that the routine could not execute normally, for example, when its input values are invalid (string instead of an integer type).

- *Mean values for cognition, metacognition and OOP*

  The unsuccessful participants have a mean value of $\bar{x}$ = 3.05 and successful participants have a mean of $\bar{x}$ = 3.85 for cognitive knowledge and skills. There are also differences for metacognition, where the unsuccessful participants have a mean of $\bar{x}$ = 2.36 and for successful participants $\bar{x}$ = 3.33. For OOP the unsuccessful participants have a mean value of $\bar{x}$ = 2.44 and for successful participants $\bar{x}$ = 3.62 on the 4-point scale. This data implies that successful participants used various knowledge and skills in Bloom's taxonomy, they reflected on their programming tasks, they were in control of their problem-solving activities and they had the ability to complete their programs successfully (Table 5.19).

5.2.8 Application of interpretivism and positivism in Section 5.2

- *Interpretivism*

  Principle 1 of Klein and Myers (1999; Table 2.1, Table 2.2) relates to the hermeneutic circle, advocating iteration between parts and the whole that they form. The researcher adhered to this principle as she analysed each participant's program as well as the detailed programming statements that comprise it, to determine which knowledge, skills and strategies were applied during problem solving (Table 5.2).

  In addition, the researcher applied Principle 6 (Klein & Myers, 1999), which requires sensitivity to different interpretations by participants of the same computer program. For example, she noted that P31 had problems interpreting leap years (§5.2.2.1, Table 5.3) whereas P32 experienced no problems in interpreting and implementing leap years (§5.2.3.1, Table 5.7).

- *Positivism*

  Positivism is applied in this section as an approach that verifies and confirms empirical observations by means of measurable ways to ensure reliability and validity of data (§2.6.2). The statistics are based on the raw scores obtained and given in Table 5.2, and the statistical methods include the following: factor analysis and sample adequacy, reliability testing, descriptive statistics (mean values and standard deviations), and practical significance in Subsection 5.2.7.

To summarise this section, it appears that use of all the levels of Blooms' taxonomy is required, as well as reflection on the programming task – including planning, monitoring and regulation. Successful participants used problem-solving strategies efficiently during programming. It can thus be said that application of certain knowledge, skills and strategies is required in order to complete an object-oriented program effectively and to obtain the correct program output.

5.3 Qualitative analysis of participants' thinking processes using *Atlas.ti* software

This section discusses the qualitative aspects of the empirical study, for which the groundwork was laid in Chapter 2 in Subsections 2.9.1 and 2.9.2. All the participants' thinking processes were analysed with the support of *Atlas.ti* software. Since there was a large amount of textual data in this study, it was decided to use *Atlas.ti,* which is an excellent software package to support such analysis.

Atlas.ti is a powerful tool for qualitative analysis. It provided support to the researcher during the analytical process, organisation and interpretation of participants' thinking processes and in the practical application of grounded theory (§2.4). Two examples are presented in Subsections 5.3.1.1, 5.3.1.2 and 5.3.2 to illustrate and explain the detailed process of data analysis. Various themes that emerged from the data are discussed in Subsections 5.3.3 to 5.3.7. Subsection 5.3.8 discusses the application of interpretivism and grounded theory and the themes that emerged, which are shown in Fig. 5.10 and further elaborated in Fig. 6.5 in Chapter 6.

The analysis strategy in this section was addressed as follows:

- Participants' written thinking processes were typed in Microsoft Office (Microsoft Word) and saved as a Rich Text Format (**.**rtf) file with each participant's number being used as the file name;
- A new hermeneutic unit (HU) (Fig. 2.4, Fig. 2.5) was created and saved under the name DATE_CLASS in *Atlas.ti.* Each participant's **.**rtf file was saved within this single HU as a primary document (PD);
- As stated above, each primary document was assigned to the hermeneutic unit. The PDs were opened, one at a time, to enable specific text to be coded (Fig. 2.5, Fig. 2.6);
- Each participant's thinking processes were read on the screen and particular words, sentences or paragraphs that represented explicit ideas, units of meaning or thinking processes in OOP were selected. A specific code was awarded to the selected text by labelling the written text in the PD (Fig. 2.7). Open coding was used during this task (§2.5.2). The process was repeated iteratively, selecting items of text that were an indication of notable concepts during the programming process. Some memos were also created during the coding process to record the researcher's comments (§2.9.1).

- This coding process continued until saturation of data was achieved (§5.3.2);
- Different codes were organised into coded families in *Atlas.ti* (§2.9), which might represent a category. Axial and selective coding were used during this process (§2.5.2);
- Three months later, further analyses were repeated on the same data, to refine the coded families and to enhance reliability. Each code was highlighted with a specific colour to indicate that it might belong to a specific family. Thereafter, each code considered as a candidate for a specific family was checked according to that family's correct description (Fig. 2.11). A list of all the codes is printed in Appendix D;
- Finally, specific themes were identified from the coded families (§2.9.1). Each theme could be used 'as the basis for an argument' and themes were organised to describe the research results as a whole (Henning et al., 2004:106-107) (Fig. 5.10). The strategy is summarised and illustrated in Fig. 2.4.

The two principle modes of working with *Atlas.ti* were applied in this study. Firstly, the *textual level* was used which refers to all activities of coding text from the written thinking processes, and secondly, the *conceptual level* was used in the form of model building by linking codes to form networks (§2.9).

5.3.1 The coding process of two detailed examples in *Atlas.ti*

It is not possible to discuss all the coding processes, however, two illustrations of data analysis are given in this subsection.

Subsection 5.2.7 proposed a definition for 'successful participants' and presented their scores in Table 5.18. The coding processes of two of these successful participants (P29 and P32) are explained in detail in the next subsections. P29 used Delphi and P32 used Java as the programming language. The thinking processes of all participants (n=48) were analysed in exactly the same way as the two examples in this subsection. Subsections 5.3.1.1 and 5.3.1.2 describe in detail how the data of two participants was coded, while 5.3.2 explains how the coded data of all participants was integrated by organisation into families and themes.

5.3.1.1 The coding process of Participant 29's thinking processes

The steps in analysing P29's thinking processes with *Atlas.ti* follow:

- **Thinking processes were collected**

  P29's PD was opened to enable specific text to be coded. Fig. 5.4 shows this participant's document while Fig. 5.5 demonstrates an extract from the coding process of a segment from the primary document (P29 point 2).

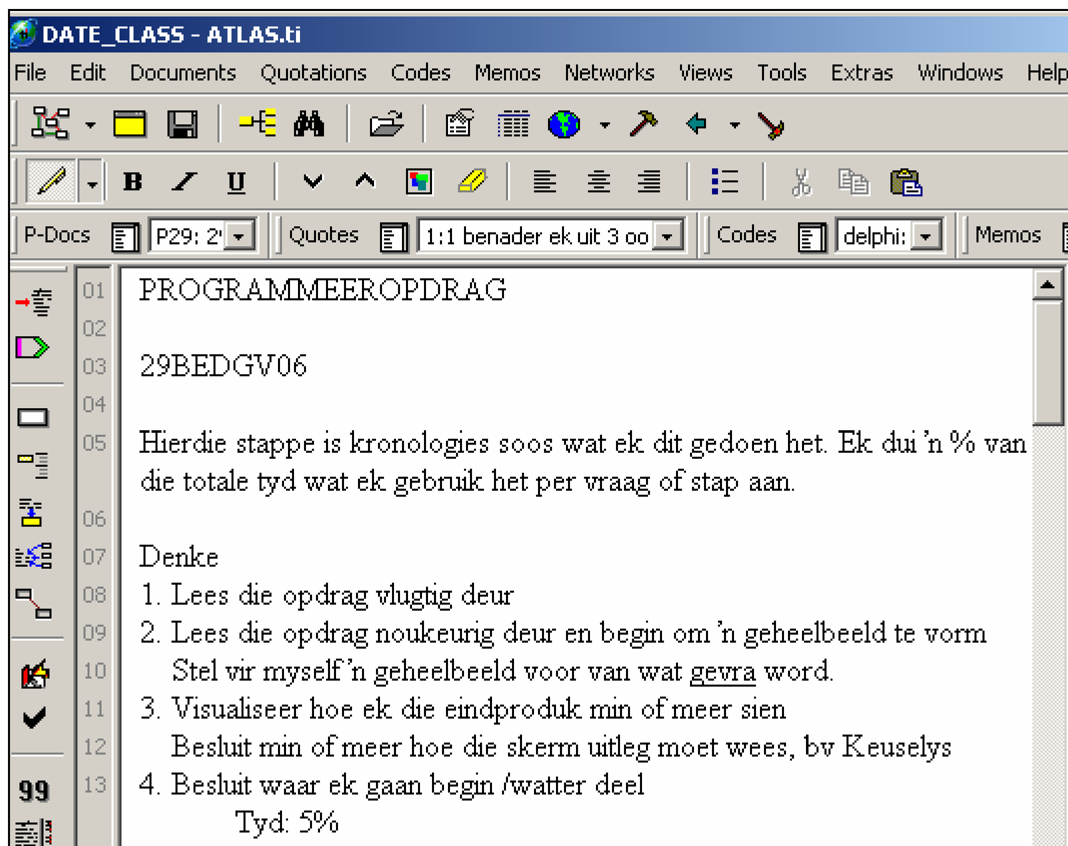


Figure 5.4: P29's data is assigned to the DATE_CLASS hermeneutic unit for analysis within the primary document

- **Codes were assigned to specific text**

  P29's thinking processes were read on the screen and particular words, sentences or a paragraph that represented explicit ideas, units of meaning or thinking processes in OOP were selected. A specific code was awarded to the selected text by labelling the written text in the PD. Open coding was used during this task (§2.5.2). The process was repeated iteratively, selecting items of text that indicated the important ideas of P29. The highlighted text in Fig. 5.5 is a translation from the Afrikaans language with the following thinking processes.

read the assignment with precision and get the big picture. Present the big picture of what is asked.

The code that was assigned to this selected segment is:

delphi:assignment:read with precision:determine big picture.

Henning et al. (2004:105) emphasise that it is important to resist the temptation to repeat codes, therefore the names of codes were selected to describe the meanings of the selected words or text as closely as possible. This enhances the accuracy of assignment of text to specific codes. Consolidation in the form of combining similar codes occurs in subsequent steps. The numbers in parentheses (9:10) after the code refer to the row number where the highlighted text associated with a specific code starts, and the row where that text ends (§2.9.1). The left column of Fig. 5.5 shows how point 2 of Fig. 5.4 was highlighted and the right column gives the code that was allocated to the selected text:

Figure 5.5: An example of the coding process and highlighted text of P29

5.3.1.2 The coding process of Participant 32's thinking processes

The steps in analysing P32's thinking processes with *Atlas.ti* follow:

- **Thinking processes were collected**

  P32's PD was opened to enable specific text to be coded. Fig. 5.6 shows the primary document while Fig. 5.7 demonstrates an extract of the coding process.

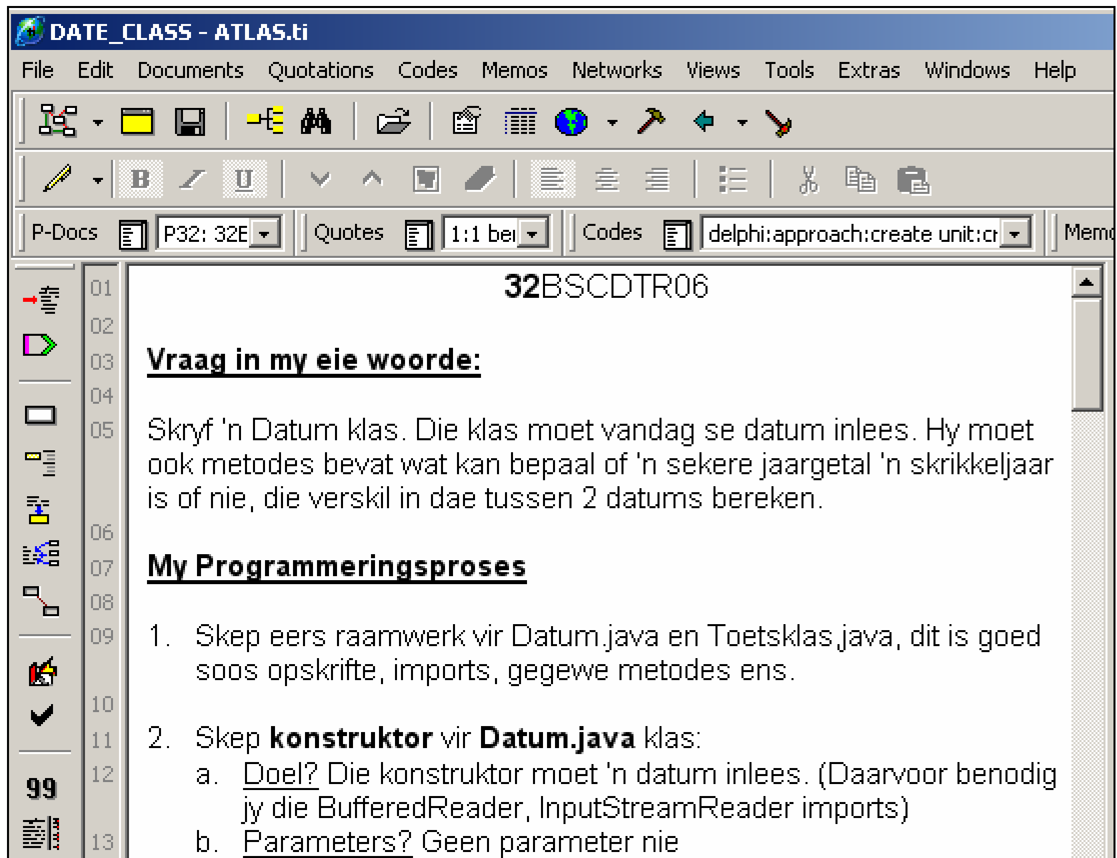


Figure 5.6: P32's data is assigned to the DATE_CLASS hermeneutic unit for analysis within the primary document

- **Codes were assigned to specific text**

  P32's thinking processes were read on the screen and particular words, sentences or a paragraph were selected, as explained for P29. Specific codes were awarded to selected text by labelling it in the PD, for example, the highlighted text in Fig. 5.7 is a translation of P32's point 1 in Fig. 5.6:

  Initially create the framework for the Date.java and Testclass.java, such as headings, imports, given methods etc.

  It was assigned to the code:

  java:assignment:framework of Date and Test class.

  In addition, the memo (ME – 06/12/29 [4]) (Fig. 5.7) refers to the note by the researcher:

  Approach:Top-down! – general to more specific programming details

  This comment indicates a top-down programming approach.

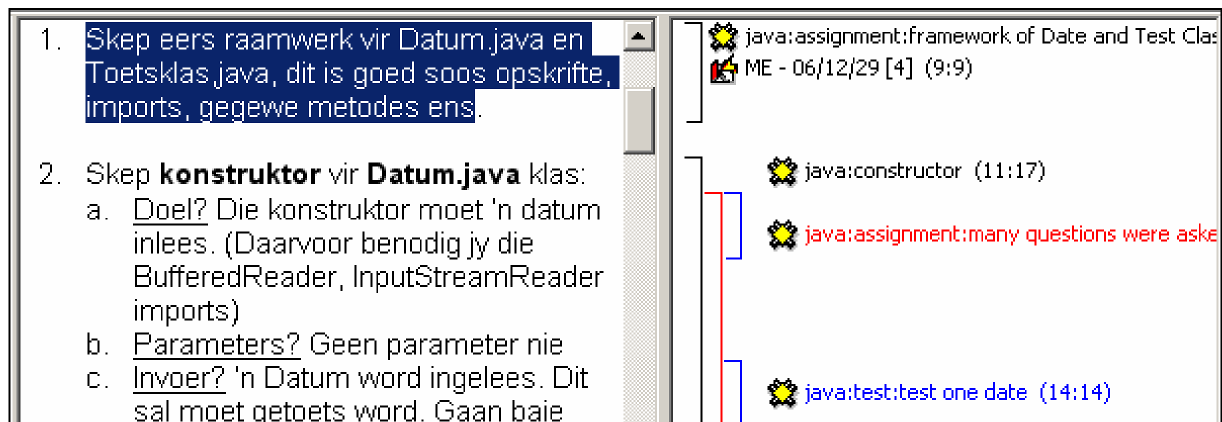


Figure 5.7: An example of the coding process and highlighted text of P32

5.3.2 The organisation of codes into families and identification of themes

The coding process continued until saturation of data was achieved (§2.9.2). Saturation occurs when no distinctly new codes emerged. Saturation did not occur until near the very end, demonstrating the value of having 48 participants. However, it was decided to continue the coding process until the thinking processes of all the participants had been analysed.

- **Different codes were organised into families**

  All the codes in the hermeneutic unit, which comprised all of the 48 PDs, were organised into possible coded families. Groups of related codes can be classified into subsets or families (§2.9.1). Partitioning various associated codes into families reduces the number of "chunks" requiring the researcher's attention (Muhr, 2004:191, 192) and further supports the analysis process. A segment of coding is shown in Fig. 5.8 with possible coded families indicated by a specific colour. The name of a possible coded family is indicated after each code. The selection of a name for each coded family was inductively guided by the data (§2.9). For example, the code:

  delphi:assignment:ask questions

  is an indication of the metacognition family; and

  delphi:assignment:determine leap years

  is an indication of the cognition family, as indicated on the right-hand side of Fig. 5.8.

Code-Filter: All

HU: DATE_CLASS
File: [C:\Documents and Settings\Administrator\My Documents\Scientific Software\ATLASti\Te...\DATE_CLASS.hpr5]
Edited by: Super
Date/Time: 07/01/26 07:12:26 PM

---

delphi:assignment:ask questions // metacognition (blue)
delphi:assignment:cannot apply problem // problem/error (yellow)
delphi:assignment:confused with procedures and functions // problem/error (yellow)
delphi:assignment:determine difference between two dates // cognition (red)
delphi:assignment:determine leap years // cognition (red)
delphi:assignment:difficult:not clear guidelines // problem/error (yellow)

Figure 5.8: Grouping of codes into the possible coded 'families'

All the coded families were identified in the same way as in Fig. 5.8. Once again, this was done by the grouping of various related codes and integrating them into families (Fig. 5.9). The researcher organised the codes into families as a manual process and selected appropriate names for the families, guided inductively by the data.

The codes on the bottom left in Fig. 5.9 represent various examples of the problem-solving family. The left side of the list below the family list (top pane), displays the codes that are already assigned to the selected problem-solving family. The right side of the list displays all the codes that are not currently assigned to this selected family.

List of selected families

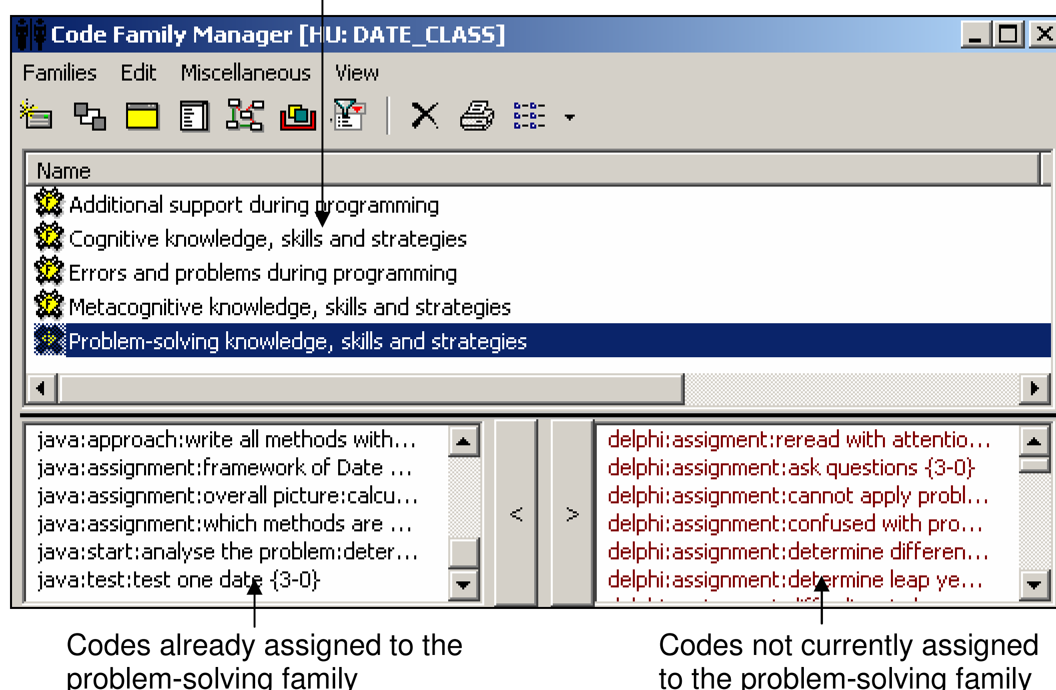


Figure 5.9: Selected codes in the problem-solving coded family

- **Inductive refinement of categories**

  Grounded theory analysis entails the inductive refinement of categories to more abstract levels and the integration of categories in a coherent whole that could explain various processes (§2.9.1). Using the terminology of *Atlas.ti,* a 'family' represents a 'category'. Following the generation of families from the codes, various themes were identified from the families in this study, with five major themes emerging:

  1. Cognitive knowledge, skills and strategies (in short, termed Cognition)
  2. Metacognitive knowledge, skills and strategies (Metacognition)
  3. Problem-solving knowledge, skills and strategies (Problem solving)
  4. Errors and problems during programming (Errors and problems)
  5. Additional support during programming (Additional support).

It is notable that the first three of these themes, while emerging naturally from the empirical analysis in this study, correspond closely with the major aspects investigated in the literature review of Chapters 3 and 4. It was equally clear that many participants made inadequate use of the skills and strategies embodied in these themes. The themes will be discussed in Subsections 5.3.3 to 5.3.8.

5.3.3 Theme 1: Cognitive knowledge, skills and strategies

In this subsection, the first of the five emergent themes, namely *cognitive knowledge, skills and strategies* is discussed. Table 5.20 presents Theme 1, where the various codes defined in *Atlas.ti* that are relevant to cognition, are shown along with associated quotations from the thinking processes.

The relevant participant's number appears in parentheses in the second column. It can be seen that the theme consists of various subthemes. Table 5.20 is followed by a discussion of each subtheme.

Table 5.20: Theme 1: Cognitive knowledge, skills and strategies – codes in *Atlas.ti* with associated quotations from participants' thinking processes

Theme 1: Cognitive knowledge, skills and strategies	
Codes in *Atlas.ti*	**Associated quotations from participants' recorded thinking processes**
Subtheme 1.1: Knowledge and comprehension skills	
- delphi : assignment : determine leap years - delphi : OOP uses objects in the program - java : assignment : determine requirements - java : assignment : must understand basic principles in programming - java : test : dates valid	- *I find out when it is a leap year* [P31] - *OOP uses objects in the program* [P7] - *First determine requirements* [P20] - *A programmer should understand basic principles* [P15] - *You should understand whether a date is valid or not* [P20]
Subtheme 1.2: Application and analysis skills	
- delphi : programming : determine variables required - delphi : planning : determine scope : private or public - delphi : planning : think about class structure - delphi : buttons : consider the screen layout - java : dates : compare : calculate days - java : assignment : which methods are necessary in the class - java : assignment : what instance variables should be declared? - java : date : separate day, month, year	- *Which variables do I need?* [P30] - *Insert private and public* [P11] - *Firstly I thought about the class structure* [P10] - *Think about the screen layout* [P29] - *I have decided to compare two dates* [P25] - *Which methods should be in the class?* [P21] - *Which instance variables are needed?* [P18] - *I receive the date as a string and separate it into days, months and year* [P40]
Subtheme 1.3: Synthesis and evaluation skills	
- delphi : programming : implementation : procedures and functions - java : dates : convert dates to days: subtract : convert - java : test : dates valid - java : test : leap years - java : programming : compile, change errors - java : program : program execute correctly : 100% - java : test class : test output	- *Now write functions for each date, month and year* [P9] - *I need a method for conversion to number of days* [P36] - *I also need a method to test for valid dates* [P23] - *There should be a function that tests for leap years* [P19] - *I have compiled and corrected errors* [P41] - *The program is working 100%* [P40] - *I tested the constructor to determine if it received the correct data* [P32]
Subtheme 1.4: Rehearsal, elaboration and organisation-and-integration strategies	
- java : class design : determine general and specific cases	- *When designing the class, I ask myself about the general and special cases in each situation* [P23]

- **Subtheme 1.1: Knowledge and comprehension skills**

  As indicated in the examples, participants mentioned the importance of specific knowledge (§3.3.2). P7 had knowledge about OOP and mentioned the requirement of using objects in a program. Participants 15 and 20 referred to the understanding of program requirements, which includes all the required functionality as well as the constraints applicable to the *Date class* program. Knowledge and comprehension skills are also required to determine whether a date is valid [P20] and when a year is a leap year [P31]. Most of the participants knew about variables, methods and programming statements such as *if*, *while* and *for*. In general, these examples illustrate that most participants remembered facts and interpreted the programming problem. With reference to Zant (2005), these results confirm the importance of interpretation of the programming problem.

- **Subtheme 1.2: Application and analysis skills**

  Application refers to the use of previously learned material in new situations, while analysis refers to the break-down of material into subparts (§3.3.2, Table 3.1). P29 used application skills when he considered, and later designed, a user-friendly screen layout to determine leap years and when he decided on the input and output involved. Whilst analysing the programming problem, decisions should be made about instance variables [P18, P30] and methods. P21 mentioned that specific methods should be included in the program. For example, various calculations were necessary in the *getDay( )*-method. Such calculations demand analytical reasoning and computational skills to provide the correct number of days for each month. P40 mentioned that he started with the analysis of the date e.g., 10 April 2006 and decided to separate the day, month and year. Examples emerged from participants' thinking processes where they thought about the implementation of the programming problem. This implies that various decisions were made on the programming details such as the screen layout, variables and methods, and the required calculations to determine leap years. However, some participants did not use detailed application and analysis skills and could not solve the problem i.e., P31 (Table 5.3).

- **Subtheme 1.3: Synthesis and evaluation skills**

  The combining of parts to form a new whole relates to synthesis, while evaluation is the activity of judging the value of material (Table 3.1). During synthesis, participants proposed a variety of different statements to implement the problem in a program, for example functions to determine the days, months and year [P9] and to convert the number of days [P36]. P36 tested the loop value and added the required number of

days. In addition, he used a flag to test for leap years. After creating an object, P23 called the *getTest( )* method and assigned a boolean value to the variable, depending on the correctness of the date. These thinking processes reflect on some examples of synthesis skills. However, the indication was that not all participants mentioned synthesis skills or they could not apply these skills in their programs. Some participants could not complete the program, and many did not obtain output i.e., P3, P24 and P31 (Table 5.2).

Evaluation is the highest level on Bloom's taxonomy and is used to determine whether the complete program works. Participant 40 noted that he made the necessary changes and that his programs executed correctly thereafter (Table 5.20, Table 5.2). A program can still be incorrect semantically, even when all the syntax errors are corrected. It is of concern that only a few participants mentioned the use of test data in their thinking processes. For example, P32 used a subsequent process of testing. He mentioned that he had tested the constructor to determine whether it received the correct data (Appendix G). This was followed with the testing of the *testLeapYear()* method, and finally the test program was updated to determine whether the *dateDifference()* method was working. Furthermore, P32 provided for many exceptions in the calculations for leap years. Emerging from the data was the ability of only a few participants to evaluate the correctness of their programs and to ensure the correct output.

Subthemes 1.1 to 1.3 indicate that only some participants applied all the skills of Bloom's taxonomy – knowledge, comprehension, application, analysis, synthesis and evaluation – to solve the problem successfully. These findings are in line with Carbone et al. (2002:2), who mention that programming is 'extremely cumulative', therefore previous knowledge and skills are used in each successive programming task.

- **Subtheme 1.4: Rehearsal, elaboration, organisation-and-integration strategies**

  These strategies are used to support attention, to help integrate new information with prior knowledge, and to organise knowledge and skills with a whole solution in mind (§4.3). It was not possible to identify clear examples of rehearsal, elaboration and organisation-and-integration strategies since participants did not mention them explicitly. However, an implicit example of the elaboration strategy (§4.3.2) was mentioned when P23 asked whether provision should be made *for general and specific cases for each situation*. Possible reasons for the unsuccessful use of cognitive strategies could also be that participants did not verbalise knowledge about these

strategies, they did not use cognitive strategies or they did not know how to apply such strategies during programming.

Participants' examples of knowledge, skills and strategies in the cognitive domain were discussed in this subsection. Theme 2 gives an overview of various metacognitive knowledge, skills and strategies.

5.3.4 Theme 2: Metacognitive knowledge, skills and strategies

Various examples of metacognitive knowledge, skills (§3.4) and strategies (§4.4) are shown in Table 5.21 and, as was the case with Theme 1, various subthemes emerged and are discussed.

Table 5.21: Theme 2: Metacognitive knowledge, skills and strategies – codes in *Atlas.ti* with associated quotations from participants' thinking processes

Theme 2: Metacognitive knowledge, skills and strategies	
Codes in *Atlas.ti*	**Associated quotations from participants' recorded thinking processes**
Subtheme 2.1: Metacognitive knowledge and skills	
- delphi : assignment : ask questions	- *I read the question carefully and determined what was being asked? What are the specifications?* [P29]
- delphi : assignment : cannot apply problem	- *I have the correct idea, but cannot apply it* [P5]
- java : approach : application of trial-and-error : difference between dates	- *I don't have a plan, but will try to code by means of trial-and-error* [P34]
Subtheme 2.2: Metacognitive strategies	
- java : time management : 3 to 5 hours	- *Time spent on this assignment was 3 to 5 hours* [P41]
- java : assignment : framework of Date and Test Class	- *Create framework for Date and Test class ...* [P32]
- java : error : calculate days for leap year incorrectly : must add 1 day	- *I determined the difference in days but was incorrect with 1 day* [P39]
- java : approach : reread assignment, write thinking down	- *I reread the question with attention and insight* [P30]
- java : reflection : should send the date to the constructor	- *I could send the date to the constructor* [P33]
- java : test : difference between dates	- *I recall a function to determine the difference – to decide on addition or subtraction* [P23]

- **Subtheme 2.1: Metacognitive knowledge and skills**

  Prior knowledge acquired from previous programming tasks can ease planning for a new task. In this regard, P29 asked many questions to set the context and to support his comprehension of the new task, for example: *What are the specifications?* P5 mentioned that he had the correct idea but that he could not apply it in a programming problem. This implies that he had an awareness of himself and his inability to complete the task. P34 mentioned his knowledge of the trial-and-error strategy that he had used during other programming experiences: *I don't have a plan but will try to code by means of trial-and-error.* Some examples emerged from this analysis where participants indicated knowledge about the self-as-learner programmer, as well as knowledge about programming (§3.4.2).

- **Subtheme 2.2: Metacognitive strategies**

  The participants mentioned only a few metacognitive strategies (§4.4, Table 5.21). P32 asked focused questions to elicit knowledge about the task and thus to direct his own thinking. He was goal-directed and planned his own programming task (§4.4.1). P32 asked questions about the framework for the *Date* and *Test class* (Table 5.8, Fig. 5.6). Very few participants monitored their own programming tasks (§4.4.2). P39 had problems calculating the difference between days, and P33 mentioned that he could initialise the constructor. The other participants did not mention their monitoring activities or were not aware of this strategy i.e., P3, P31 (Table 5.2, Table 5.4).

  Some participants used the regulation strategy when they modified their own programs (§4.4.3). P32 went back to determine whether the problem was solved (Table 5.8). P33 regulated his own programming when he realised that he could send the date to the constructor, and P23 recalled a function and decided on addition or subtraction to determine the difference (Table 5.21).

  It should be mentioned that application of the regulation strategy entails more than the interpretation of error messages only. It is a complex process that requires insight into the entire program and continuous modification of one's cognitive activity is needed to determine whether the programming problem has been successfully solved (§4.4.3). There is an indication that most of the participants applied no regulation strategies during programming of the *Date class*.

  Reflection includes planning, monitoring and regulation (§4.4.4). The *Reflection Assistant Model* (RA) of Gama (2004:668-677) refers to pre-task reflection and post-

task reflection. An example of pre-task reflection is: *I use previous Java experiences, and the Java textbook* [P35]. An example of post-task reflection is: *I could send the date to the constructor* [P33]. Analysis from the data indicates that participants did not explicitly apply reflective thinking during programming.

5.3.5 Theme 3: Problem-solving knowledge, skills and strategies

Examples of problem-solving knowledge, skills and strategies that participants used during programming are displayed in Table 5.22 and are discussed in this subsection.

Table 5.22: Theme 3: Problem-solving knowledge, skills and strategies – codes in *Atlas.ti* with associated quotations from participants' thinking processes

Theme 3: Problem-solving knowledge, skills and strategies	
Codes in *Atlas.ti*	**Associated quotations from participants' recorded thinking processes**
Subtheme 3.1: Problem-solving knowledge and skills	
- delphi : steps : think about problem : how to solve problem - java : assignment : overall picture : calculate dates - java : assignment : which methods are necessary in the class - java : approach : determine input, interface, calculations, test input	- *Think about the problem. How will I solve this problem?* [P5] - *When starting the class, keep the overall picture in mind* [P16] - *The question that I asked myself in the beginning was: which methods should be in the class and how can I calculate them?* [P21] - *determine input, design the interface and basic components, process and test the input* [P44]
Subtheme 3.2: Problem-solving strategies	
- delphi : strategy : complete all the detail program code of specific component before continuing with the next - java : approach : date class : empty methods - java : approach : application of trial-and-error : difference between dates	- *I will complete the code for a specific component before continuing with the next component* [P6] - *I will start with empty methods in my Date class as well as a constructor* [P43] - *I don't have a plan and will code in trial-and-error and hope to succeed* [P34]

- **Subtheme 3.1: Problem-solving knowledge and skills**

  Participants thought about the problem. P5 analysed the problem, while P16 determined the overall picture. It is necessary to decide which methods and calculations should be included in a class [P21]. P44 mentioned the following problem-solving steps during programming*: determining input, designing the interface and basic components, processing and testing the input*. In contrast, P31 did not follow any specific sequence of steps during problem solving (Table 5.5). The data analysis indicated that only some participants mentioned explicit problem-solving steps in their written thinking processes. (Some problems during programming are outlined in Theme 4 in §5.3.6).

- **Subtheme 3.2: Problem-solving strategies**

  In most cases, participants did not clearly mention which strategy they used. Only P34 mentioned explicitly that he used the trial-and-error strategy: *I don't have a plan and will code in trial-and-error and hope to succeed.* Other participants described the steps they used during the problem-solving process, and inferences could therefore be made from their descriptions to determine which problem-solving strategy they had used during program comprehension i.e., top-down, bottom-up, integrated, as-needed or trial-and-error strategy (§4.5.1).

  P6 explained that he finishes all the programming code that is associated with a specific component *before continuing with the next component*. This implies that he proceeded from the specific to the general, indicating the bottom-up strategy. Implicit examples of the top-down approach were described by P43, who started with *empty methods* and *a constructor* (Table 5.22), and P32 who reported, *I will start with the framework for the Date class and Test class, headings, import given methods, etc.* (Table 5.9). However, Lin (2001:23) emphasises that programmers should explicitly learn problem-solving strategies and should apply them systematically during programming, but this did not appear to take place with the participants in this study.

With reference to the *Serendipity* principle of *Atlas.ti* (§2.9), two additional themes were found without being searched for, namely Errors and problems, and Additional support during programming. A discussion of these two themes follows, as Theme 4 and Theme 5, respectively.

5.3.6 Theme 4: Errors and problems during programming

Some examples of errors and problems that occurred during programming appear in Table 5.23, followed by a discussion of this theme.

Table 5.23: Theme 4: Errors and problems during programming – codes in *Atlas.ti* with associated quotations from participants' thinking processes

Theme 4: Errors and problems during programming	
Codes in *Atlas.ti*	**Associated quotations from participants' recorded thinking processes**
- delphi : error messages : not displayed : don't know what is problem - java : error : forgot main method : public static main - java : exception handling : complex - java : error : difficult to compile Java on computer - java : error message : cannot diagnose the problem - java : method : don't know how to copy part of a string	- *The program didn't show me errors* [P31] -*I forgot to insert the statement: public static main* [P43] - *Exception handling is complicated* [P33] - *The program cannot associate with the Date class and cannot compile* [P41*]* - *my program didn't show errors and I don't know what the problem is* [P34] - *I cannot remember how to copy 'IZ' from 'ELIZNA'* [P39]

Participants made elementary mistakes such as the following:

> *The program cannot associate with the Date class and cannot compile* [P41]
> *I forgot to insert the 'public static main' statement* [P43]
> *It is complicated to do the exception handling* [P33]
> *I cannot remember how to copy 'IZ' from 'ELIZNA'* [P39]
> *The program didn't show me errors* [P31].

Some students could not apply exception handling [P34] and/or interpret errors [P31]. These errors were not due to a lack of time, because participants had a full week to complete the task. Possible reasons why some of these participants did not succeed were investigated. Some had not used the correct syntax [P39] and could not compile the program [P41]. In addition, P33 could not apply exception handling.

5.3.7 Theme 5: Additional support during programming

Various examples of supplementary support used by students during their OOP experiences are shown in Table 5.24, followed by a discussion of this theme.

Table 5.24: Theme 5: Additional support during programming – codes in *Atlas.ti* with associated quotations from participants' thinking processes

Theme 5: Additional support during programming	
Codes in *Atlas.ti*	**Associated quotations from participants' recorded thinking processes**
- delphi : bibliography : Internet - java : bibliography : internet websites - delphi : bibliography : study guide - java : bibliography : previous Java code - delphi : bibliography : Delphi textbook - java : bibliography : C# text book - java : bibliography : previous Java assignments - java : approach : lecturer : motivation : passion	- *Wikipedia.com for the requirements of leap years* [P29] - *Using Google.com [P35]* - *Used computer science study guide for classes and objects [P29]* - *Used sources: Big Java and previous code* [P48] - *Used the Delphi textbook* [P30] - *Used C#: How to program* [P37] - *I have used previous Java assignments* [P44] - *It makes a big difference if the lecturer cares and has a passion for what she is doing – it affects both* [the student and lecturer] [P25].

Many participants consulted their textbooks: *Big Java* [P48], the Delphi textbook [P30] or *C#: How to program* [P37]. Others used previous Java assignments [P44], their Computer Science study guide for the theory of classes and objects [P29], the Internet (*Wikipedia)* for the requirements of leap years [P29] and *Google.com* [P35]. One participant also mentioned the importance of a motivated lecturer who displays passion for lecturing OOP [P25].

5.3.8 Application of interpretivism and grounded theory in Section 5.3 and the generation of themes

- *Interpretivism*

  The researcher applied Principle 5 of Klein and Myers (1999; Table 2.1, Table 2.2) during the analysis of each participant's complete program. This principle advises that the researcher should be sensitive to possible contradictions between theoretical foundations and the data that emerges through the research process. Moreover, the researcher should make the research process as transparent as possible to the reader. Such contradictions occurred in this study. From the theoretical chapters it is clear that programmers should use various cognitive and metacognitive strategies (§4.3, §4.4)

during their programming process, however only a very few examples of use of cognitive strategies and metacognitive strategies (with specific reference to regulation), emerged from the empirical data. Subsections 5.3.1.1 and 5.3.1.2 gave a detailed and transparent overview of the analysis process using *Atlas.ti.* Furthermore, detailed examples of participants' thinking processes were given in Tables 5.20 to 5.24.

In addition, the researcher applied Principle 7 (Klein & Myers, 1999), which requires sensitivity to biases and distortions identified in the written thinking processes of participants. Furthermore, the researcher had to explicitly avoid bias in her own interpretation of the qualitative data of the *Date class* task (Table 5.20). This had to be applied in the study, particularly where deficiencies were identified in unsuccessful programmers. This led to the conviction that such deficiencies can be addressed by applying supportive measures, as outlined in Chapter 6 (Fig. 6.5).

- *Grounded theory*

  It is important to mention that grounded theory (which is one of the major research methodologies selected for this study) had a strong influence on the original design of the qualitative analysis software, *Atlas.ti* (§2.5.2), which was used in this research. Grounded theory is an approach where theory is generated inductively from the analysis of the data as concepts are formulated into a logical, systematic and explanatory scheme (§2.9, Fig. 5.10, Fig. 6.5).

  The qualitative analysis in this study involved interpretation of the students' thinking processes. Different steps of grounded theory – open coding, axial and selective coding – were applied in *Atlas.ti,* as explained in §2.5.2. Furthermore, selections from participants' recorded thinking processes were assigned to codes, which were organised into coded families within *Atlas.ti*.

  Saturation did not occur until near the very end, demonstrating the value of having 48 participants and also confirming the utility of having programmers representing two variants of OOP, namely Java and Delphi. Finally, themes are identified and theory is generated inductively from the analysis of data and further elaborated in an explanatory scheme in Chapter 6.

  To code data into themes, the researcher must manually recognise possible themes from the data and identify the emerging concepts. According to Neuman (2002:421),

this process rests on four abilities that were applied in this study, and which were relevant:

- recognising patterns in the data, for example, recognition of the *Cognitive knowledge, skills and strategies* pattern in Subsection 5.3.3;
- thinking in terms of systems and concepts, for example, Fig. 5.8 in Subsection 5.3.2 shows examples where various skills required in OOP were identified;
- having in-depth background knowledge, for example, knowledge about OOP as outlined in Section 3.2; and
- possessing relevant information, for example, the importance of cognitive support in the learning of OOP was overviewed in Subsections 3.3.3, 4.3.1, 4.3.2, 4.3.3 and 5.3.3.

Each theme should capture qualitative richness to explain specific phenomena such as OOP. During selective coding, the following five themes emerged from the foregoing analysis and were identified by the researcher. The section or table of relevance is given in parentheses in the list below:

- Cognitive knowledge, skills and strategies (§5.3.3, Table 5.20)
- Metacognitive knowledge, skills and strategies (§5.3.4, Table 5.21)
- Problem-solving knowledge, skills and strategies (§5.3.5, Table 5.22)
- Errors and problems during programming (§5.3.6, Table 5.23)
- Additional support during programming (§5.3.7, Table 5.24).

These themes are integrated in a schematic representation given in Fig. 5.10. Some themes are more important and can be described as the foundation and the pillars of the framework. They are dynamic and interactive (§6.3) and are indicated by the grey background in Fig. 5.10, namely:

- Cognitive knowledge, skills and strategies
- Metacognitive knowledge, skills and strategies
- Problem-solving knowledge, skills and strategies.

It became clear that many participants' OOP experiences was characterised by Theme 4, Errors and problems, which confounded the process, resulting in incomplete programs and lack of correct output. Fortunately, the discerning use of Theme 5, Additional support, can assist programmers to address and overcome these problems and errors that occur during programming.

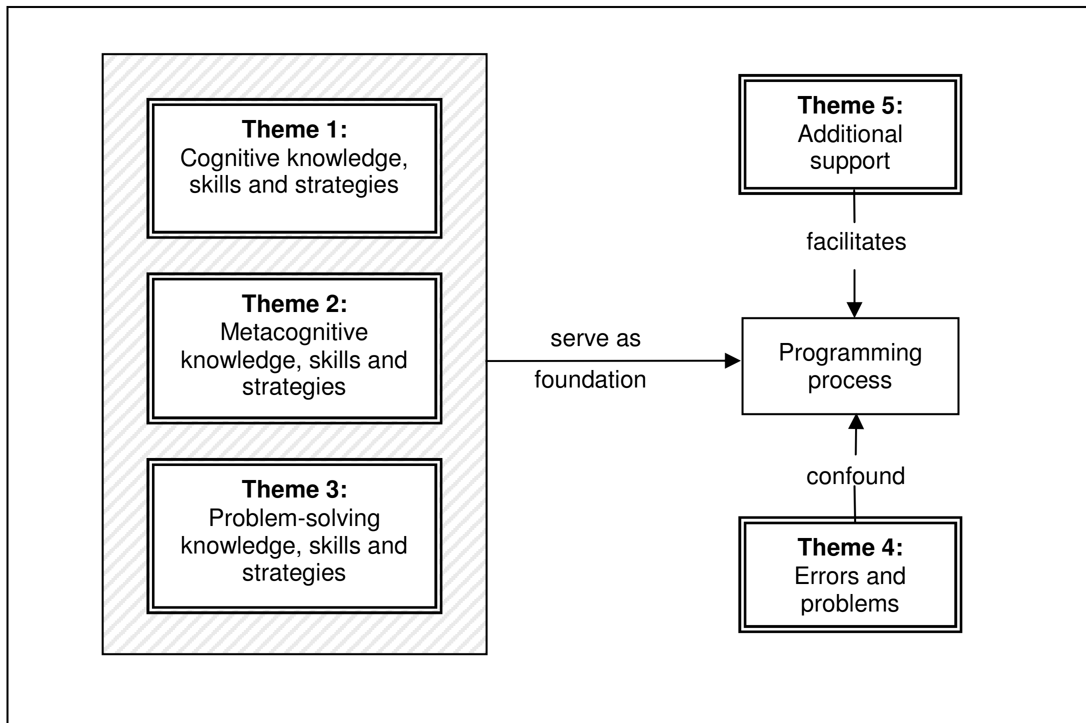


Figure 5.10: An integrated representation of the themes that emerged from participants' thinking processes

Glaser and Strauss (1967:3-5) cite four criteria for a well-constructed grounded theory, all of which are applied in this section (§2.4):

- *Fit*

  The categories and properties fit the realities being studied, which includes various knowledge, skills and strategies during OOP. The various codes used for the *Cognitive knowledge, skills and strategies* category, are an example of 'fit' (Fig. 5.11).

Cognitive knowledge, skills and strategies	
delphi:assignment:determine leap years {3-0}	knowledge
java:assignment:determine requirements {5-1}	skills [comprehension]
delphi:buttons:consider the screen layout {12-0}	skills [application]
java:assignment:methods: which methods are necessary {5-1}	skills [analysis]
delphi:programming:implementation:procedures and functions {2-0}	skills [synthesis]
java:program:program execute correctly: 100% {1-0}	skills [evaluation]
java:class design:determine general and specific cases {1-0}	strategies [organisation-and-integration]
...	

Figure 5.11: Extraction from the Cognitive knowledge, skills and strategies category

- *Work*

  In order to work, the theory should explain variations in behaviour. This relates to variations between students' thinking processes during programming, which have been

discussed comprehensively in this chapter. For example, a detailed analysis of the thinking processes of P29 and P32 was presented in Subsections 5.3.1.1 and 5.3.1.2. Subsection 5.3.2 relates to data from all the participants, as the codes were organised and refined into families. In this process, mention was made of knowledge, skills and strategies used in OOP, as well as problems that occurred – further illustrations of addressing variations in behaviours.

- *Relevance*

  This is achieved when a grounded theory both fits and works, which was the case in this study, as shown in the previous two points.

- *Modifiability*

  The emerging theory, which will be further elaborated in Chapter 6, is open to adaptation as new data is integrated.

Fig. 5.10 will be extended in Chapter 6 (Fig. 6.5) in the learning repertoire of OOP where the importance of cognitive, metacognitive and problem-solving knowledge, skills and strategies is emphasised and applied in a framework to support the learning of OOP.

5.4 Statistical analysis – questionnaire

To extend the research, the participants of **2006** completed a questionnaire about their knowledge, skills and strategies in programming (§2.7.4). Twenty participants completed this questionnaire. Since no ideal questionnaire was available for this purpose, a customised questionnaire was designed (see Appendix E).

Details about the participants were given in Subsection 2.7.1, and are elaborated in Subsection 5.4.1, using information obtained from the first paragraph of the questionnaire. Biographical information is given, after which various examples of significant constructs are discussed. In Subsections 5.4.2 and 5.4.3, the closed-ended and open-ended questions are addressed respectively.

The analysis strategy in this section was addressed as follows:

- Closed-ended questions: the purpose was to determine participants' knowledge, skills and strategies used during the OOP process (§5.4.2). The use of cognitive knowledge and skills, metacognitive strategies and problem-solving strategies as given in the questionnaire was analysed using the following statistical measurements: sample adequacy, factor analysis, reliability testing and descriptive statistics (mean values and standard deviations) (§5.2.5).
- Open-ended questions: four questions were asked to elaborate on the questionnaire responses and to collect additional information on students' behaviour during programming, for example, to determine whether the students viewed themselves as successful in programming (§5.4.3). The open-ended responses were analysed by comparing the answers of successful participants (5.2.7, Table 5.18) to unsuccessful participants and discussing the differences between them.

5.4.1 Biographical information

Biographical information provides details such as the participants' average age, gender, programming experience, qualification for which enrolled, and preferred programming language. Only 20 students participated in 2006 and completed the questionnaire.

Average age and gender

Participants' average age ranged between 19 and 26 years. Most were male (18), with only two students being female.

Degree and programming experience

Most of the 2006 students were enrolled for the BSc degree (17) and only three were studying towards a BEd. Except for four students, all the others were in their third year. Five had no prior experience in the form of higher-grade or standard-grade Computer Studies at school. Many students had prior experience in different programming languages, summarised in Table 5.25. Note that some students knew more than one programming language.

Table 5.25: Programming language currently used

Programming language/s currently used	Number of students
Java	16
Delphi	3
C++	0
C#	15
Visual Basic	5
Other	6

5.4.2 Closed-ended questions

The questionnaire (Appendix E) was divided into categories for A: Cognitive knowledge and skills, B: Metacognitive strategies and C: Problem-solving strategies. In order to enhance reliability, the questions were deliberately not grouped according to these three categories. For each category, three related questions were presented on each issue (i.e., subcategory) within it. All three questions were designed to measure the same construct. For example, the following questions refer to *evaluation* within the *cognitive knowledge and skills* category:

Question 16: *I can explain the use of specific programming statements in my solution.*
Question 30: *I find it difficult to evaluate my programming solution to determine if I have solved the problem correctly.*
Question 38: *If two different solutions of the same problem are given to me, I can select the best solution*

Note that Question 30 is a negative statement. This was taken into account in the analysis process.

As stated, these sets of related questions were not explicitly listed together within one of the structured categories, A, B or C, but were distributed throughout the questionnaire. The mark sheet (see Appendix E), however, groups the three consistent questions for each issue, indicating related action verbs such as the following: justify, evaluate and compare in the case of the *Evaluation* subcategory. Tables 5.26 – 5.28 show the questions grouped by categories, along with various statistics.

The questionnaire utilised the following Likert scale on a continuum: 'never', 'seldom', 'often' and 'always', represented respectively by scores 1, 2, 3 and 4. The decision to use a 4-point scale was taken to avoid middle options.

5.4.2.1 Cognitive knowledge and skills

The raw data from the questionnaire was analysed using the following statistical measures: sample adequacy, factor analysis and reliability, mean values and standard deviations of cognitive knowledge and skills.

- *Sample adequacy, factor analysis and reliability*
  The KMO measure of sample adequacy of the raw data from the closed-ended questions is 0.43. This indicates that the amount of data is inadequate to perform a factor analysis. However, the scale reliability measure (Cronbach alpha) was calculated and is shown in Table 5.26. For the subcategories (e.g., knowledge), where the scale is not reliable, no summary statistics are provided, but the questions are discussed individually after the table. In addition, statistics for the cognitive knowledge and skills category are shown in the cases of reliable subcategories.

- *Mean values, standard deviation and reliability*
  Table 5.26 depicts all the questions in the cognition section (§3.3). The mean values with standard deviations and Cronbach-alpha values for each subcategory are also included. Only four out of six subcategories are reliable constructs and have a Cronbach-alpha value greater than 0.5 (§5.2.5). These are: comprehension, application, synthesis and evaluation.

Table 5.26: Questions about cognitive knowledge and skills
(Knowledge, comprehension, application and analysis)

Cognitive knowledge and skills						
Subcategory	$\bar{x}$	**s**	**Question No**	**Statements**	$\bar{x}$	**s**
Knowledge $\alpha< 0.5$	–	–	24	When I program, I try to remember what the lecturer had said or what I had read in the textbook that is relevant to the problem in hand	3.15	0.67
			33	In the preparing for a test, I make sure that I can define or describe a new programming concept	3.00	0.73
			36	I find it difficult to know what the program, as required by the problem description, is supposed to do	3.30	0.57
Comprehen-sion $\alpha = 0.66$**	3.18	0.52	*1	The first time I learn a new programming concept, I make sure that I understand it	3.15	0.75
			12	I can predict what the output of a program will be	3.20	0.70
			14	I find it difficult to interpret a programming question	3.15	0.49
Application $\alpha = 0.68$**	2.92	0.62	2	When I write a new program, I know which programming statements to apply	2.70	0.66
			10	I can complete the programming code of a given incomplete program	2.85	0.67
			22	I can easily classify different types of methods, such as a constructor, destructor, mutator and accessor	3.20	1.01
Analysis $\alpha < 0.5$	–	–	3	I find it difficult to analyse a given programming problem	3.05	0.51
			8	It is hard for me to break down a problem into smaller parts	3.40	0.60
			31	When I read a programming question, I can easily distinguish between the necessary and unnecessary parts of the description	3.10	0.79

* Comprehension construct reliable if question 1 is excluded.
**Reliable constructs ($\alpha > 0.5$), (Ellis & Steyn, 2003).
Underlined question numbers indicate negative statements (Appendix E). This negativity was taken into account in the analysis process.

Table 5.26 (continued): Questions about cognitive knowledge and skills
(Synthesis and evaluation)

Subcategory	$\overline{x}$	s	Question No	Statements	$\overline{x}$	s
Synthesis α = 0.80**	3.12	0.60	4	I can create test data for a new program	3.40	0.68
			6	I can easily design a solution for a new programming problem	2.85	0.81
			27	I can combine the necessary programming statements successfully in a new program	3.10	0.64
Evaluation α = 0.80**	3.43	0.66	16	I can explain the use of specific programming statements in my solution	3.70	0.47
			30	I find it difficult to evaluate my programming solution to determine if I have solved the problem correctly	3.30	0.86
			38	If two different solutions of the same problem are given to me, I can select the best solution	3.30	0.92

**Reliable constructs ($\alpha > 0.5$), (Ellis & Steyn, 2003).
Underlined question numbers indicate negative statements (Appendix E)

- ***Knowledge***

This subcategory, for which the relevant statements in the questionnaire are 24, 33 and 36, has a Cronbach-alpha value < 0.5, and thus did not form a reliable construct.

Statements 24 and 33 relate to basic aspects of learning such as remembering what was taught in class, details they had read, and what core programming constructs they have mastered in preparation for a test. The respective means were 3.15 and 3.00, indicating that participants coped well with remembering these basic concepts of knowledge.

However, in statement 36, a mean value of $\bar{x}$ = 3.30 in this negative statement shows that participants found it difficult to understand the requirements of problem descriptions. This implies that they had problems determining what is expected from them in a new program. This refers mainly to students' inability to interpret a given problem description, although it could, on occasions, be due to inadequate problem descriptions.

- ***Comprehension***

Statements 1, 12 and 14 (Table 5.26, Table 3.1) relate to comprehension in the context of extracting key points from the problem statement. With statement 1 excluded, a reliable construct was formed with statements 12 and 14 (α = 0.66). Understanding is crucial to goal setting and in planning how to proceed. Participants indicated that they could indeed predict what the output would be. Although some participants found interpretation difficult, they were able to use comprehension skills during programming.

- ***Application***

Students should be able to apply programming statements in a new program, to complete code within a given incomplete program and to classify different methods, as indicated by statements 2, 10 and 22 (α = 0.68) (Table 5.26, Table 3.1, Table 3.2, §3.3.2). Participants found it difficult to determine whether their knowledge and skills are appropriate in new situations. The mean value for the application subcategory was $\bar{x}$ = 2.92, which is slightly lower than the mean value for comprehension ($\bar{x}$ = 3.18). These results show that participants found it difficult to apply their knowledge and skills in a new program.

- ***Analysis***

Statements 3, 8, and 31 refer to situations where participants had to differentiate between parts of the program. Some participants found it difficult to analyse a given programming problem, to decompose a problem into smaller parts and to distinguish between the necessary and unnecessary parts of the description. During analysis, participants should determine which methods are necessary in a particular class and which programming statements should be used. For example, is an ***if-...*** or a ***switch***-statement necessary to determine the specific days of each month and what is the difference between those two statements?

- ***Synthesis***

Synthesis is required when designing and combining specific programming statements in a coherent way in order to develop a new solution. In Table 5.26, statements 4, 6 and 27 formed a reliable construct regarding synthesis ($\alpha = 0.80$). The mean value for the synthesis subcategory ($\bar{x} = 3.12$) is higher than the mean value for application. Students are sometimes confronted with a new programming problem that they find difficult to solve. In such cases some of the participants tried to code the new program by combining various programming statements without using the previous categories of application and analysis. Such students may not succeed in the programming task.

- ***Evaluation***

Evaluation and testing of a complete program should also be part of the programming process. Statements 16, 30 and 38 in Table 5.26 relate to the use of evaluation skills during programming. Some of the evaluation may occur during the programming process and sometimes it may occur after completion of the task. The mean value ($\bar{x} = 3.43$) is higher than the mean in any of the previous subcategories. This result probably occurred because the questionnaire is an indication of participants' perceptions of a programming task. However, when relating these results to their actual performances in the *Date class* task, it became clear that most did not use evaluation skills. Another reason is that participants probably used evaluation without applying the previous subcategories.

5.4.2.2 Metacognitive strategies

The following statistical measures are used in this subsection: sample adequacy, factor analysis and reliability, mean values and standard deviation of metacognitive strategies. (The questionnaire provides the raw data).

- *Sample adequacy, factor analysis and reliability*
  There was an inadequate amount of data to perform a factor analysis, as indicated by the KMO measure of 0.43. However, the reliability (Cronbach alpha) was calculated and is shown in Table 5.27. No summary statistics are provided for those subcategories where the scale was found not reliable (e.g., monitoring). For these subcategories, the questions are discussed individually. For the subcategories with reliable scales, individual questions as well as the whole construct (metacognitive strategies) are summarised statistically.

- *Mean values, standard deviation and reliability*
  Table 5.27 shows examples of statements in the planning, monitoring and regulation subcategories (§4.4). The mean values and standard deviations were calculated. Two of the three subcategories are reliable constructs, with a Cronbach-alpha value larger than 0.5, namely, planning and regulation. The table is followed by discussion of each construct.

Table 5.27: Questions about metacognitive strategies

Metacognitive strategies						
Subcategory	$\bar{x}$	**s**	**Question No**	**Statements**	$\bar{x}$	**s**
Planning $\alpha = 0.53$**	2.95	0.59	9	I plan the solution of my program to achieve the goal	3.20	0.70
			18	I write down plans to direct my thinking in programming	2.30	1.08
			20	I think about what I should do first to solve a new programming problem	3.35	0.59
Monitoring $\alpha < 0.5$	–	–	15	When I program, I stop once in a while and go over what I have already programmed	2.75	1.07
			37	I ask myself questions to make sure that I understand a difficult programming statement when I use it in a program	2.85	0.93
			40	When I program, I trace the program's execution with a trace table	1.45	0.60
Regulation $\alpha = 0.68$**	3.37	0.48	7	Even when the program is difficult to write, I go back and modify it until the problem is solved successfully	3.32	0.67
			25	I take the programming statements that have errors in them and adjust them until I have solved the problem successfully	3.25	0.55
			28	I reread the description of a difficult problem to make sure that I have understood it correctly and that it is correctly programmed	3.53	0.61

** Reliable constructs ($\alpha > 0.5$), (Ellis & Steyn, 2003).

- ***Planning***

  Statements 9, 18 and 20 were used to determine whether students plan their programming as whole tasks in the metacognitive domain (Table 5.27). The mean value for planning is $\bar{x}$ = 2.95. Although participants felt that they planned their programs upfront, responses to statement 18 show that only a few participants actually wrote down the plans to direct their thinking, and the mean value is $\bar{x}$ = 2.30.

- ***Monitoring***

  This subcategory was not a reliable construct, with the Cronbach $\alpha < 0.5$ (Table 5.27), therefore each statement is detailed separately (15, 37 and 40). Participants indicated via statement 15 that they stopped once in a while to review what they had already programmed ($\bar{x}$ = 2.75). In response to statement 37, some of them agreed that they had asked themselves questions to ensure that they understood a difficult programming statement. This is shown by a mean of 2.85. However, the means indicate that participants could use monitoring more frequently. Furthermore, participants rarely use a trace table (statement 40, §3.5.4) ($\bar{x}$ = 1.45). A possible reason for this is that it takes time, and it is sometimes difficult to complete a trace table of a programming application with more than one class.

- ***Regulation***

  Statements 7, 25 and 28 were used to determine whether students view regulation of the programming process as being important. These statements formed a reliable construct ($\alpha = 0.68$) (§4.4.3). The statements include the following actions that should be employed during programming: rereading the problem description of a difficult program to understand it, going back and modifying the program until it is solved, and adjusting incorrect programming statements. The mean value for the regulation category is $\bar{x}$ = 3.37. Such a high mean value was not expected for regulation, as the means for the previous categories, planning and monitoring were lower. A possible explanation could be that the participants tried to solve the problem by continuously modifying cognitive activity and self-evaluation to determine whether they had solved the problem successfully (§4.4.3). However, much of the regulation conducted, relates to the modification of erroneous programming that was identified during monitoring. It appears that some participants' unsuccessful attempts to modify a specific program segment were due to the fact that they could not identify the real problem.

5.4.2.3 Problem-solving strategies

Sample adequacy, factor analysis and reliability, mean values and standard deviations are various statistics discussed in this subsection. The calculations are based on data obtained from the questionnaire.

- *Sample adequacy, factor analysis and reliability*

  Once again, the amount of data was inadequate to perform a factor analysis (0.51). However, the Cronbach-alpha values, which indicate reliability, are shown in Table 5.28. For all the subcategories where the scale was found not to be reliable, the questions are discussed individually (e.g., as-needed). For the subcategories with reliable scales, individual questions as well as the whole construct (problem-solving strategies) are summarised statistically.

- *Mean values, standard deviation and reliability*

  Table 5.28 reveals different examples of problem-solving strategies. The subcategories that formed reliable constructs with a Cronbach-alpha value greater than 0.5, are bottom-up (§4.5.1.1), top-down (§4.5.1.2), integrated (§4.5.1.3) and trial-and-error strategies (§4.5.1.5). The mean value for the different problem-solving strategies varies from $\bar{x}$ = 2.57 to $\bar{x}$ = 2.89. This implies that participants may have been unsure about the use of problem-solving strategies and did not know the difference between them or could not apply them correctly. A more detailed discussion of each strategy follows after the table.

Table 5.28: Questions about problem-solving strategies

Problem-solving strategies						
Subcategory	$\bar{x}$	**s**	**Question No**	**Statements**	$\bar{x}$	**s**
Bottom-up α = 0.80**	2.77	0.71	23	When I program, I start with **all the details of a method** before proceeding with the next method	2.65	0.93
			29	During programming, **I start with the details,** such as variables of a specific class before programming the details of the next class	2.85	0.75
			39	When I program, I **complete the programming code of one class** before proceeding with the programming code of the next class	2.80	0.83
Top-down α = 0.61**	2.89	0.64	5	Before programming, I **consider the whole solution before** going into the **details** of the solution	3.40	0.60
			13	When I program, I start with the ***declaration*** of all the **methods of one class**, before proceeding with the ***detailed*** programming of each method	2.63	1.01
			32	When I program, I start with the ***declaration of all the classes*** before proceeding with the ***details*** of ***each class***	2.60	0.88
Integrated α = 0.87**	2.57	0.82	19	When I program, I start with the ***declaration and details*** of the first class and methods before proceeding with the next class	2.70	0.80
			35	When I program, I start with the declaration of a framework for a certain class and proceed with ***all the methods of the same class*** before starting with the framework and details of the next class	2.70	0.92
			42	I program the **whole class with its details** and complete it before proceeding with the next class	2.30	1.03
As-needed α < 0.5	–	–	11	I can alter specific parts of a programming solution when the requirements have changed	3.30	0.57
			17	I only make changes to a specific method when required due to errors in my program	3.15	0.59
			26	During modification of a given program, I expand a specific section only	2.70	0.47
Trial-and-error α = 0.75**	2.67	0.58	21	I try to program a possible solution and hope that it will work	2.30	0.80
			34	I do not know where to start with the programming of a new problem	3.25	0.55
			41	During programming, I use any possible solution that would work, but not necessarily the very best solution	2.45	0.76

** Reliable constructs (α > 0.5), (Ellis & Steyn, 2003). Underlined question numbers indicate negative statements (Appendix E)

- ***Bottom-up***

  Statements 23, 29 and 39 were intended to determine whether participants used the bottom-up strategy in programming. The indication was that participants used this strategy when starting with the details of a class, such as variables and the implementation of methods, before proceeding to more general programming with higher levels of abstraction (§4.5.1.1). The mean value is $\bar{x}$ = 2.77 on a 4-point Likert scale, representing participants' perceptions of using this strategy during a task. In actual fact, this value indicates that participants had difficulty in identifying exactly which problem-solving strategy they had applied during programming.

- ***Top-down***

  The top-down strategy is represented by statements 5, 13 and 32. A mean value of $\bar{x}$ = 2.89 is obtained. Participants indicated that they had used this strategy when they first completed a framework of methods, for example *getDay( ), getMonth( ), getYear( )* within a class, before proceeding to programming the details of each method (§4.5.1.2). Only a few participants mentioned that they had used the top-down strategy during the actual programming process.

- ***Integrated strategy***

  In Table 5.28, statements 19, 35 and 42 refer to descriptions of the integrated strategy. This is a strategy in which a programmer uses both the bottom-up and top-down strategies during computer programming, and switches between them may occur (§4.5.1.3). The mean value for the integrated strategy is $\bar{x}$ = 2.57. Participants who used this strategy started with a framework of a class or methods and completed their details before proceeding with a next class. This low value illustrates that participants were not sure which problem-solving strategy they were using in a programming task.

- ***As-needed strategy***

  The Cronbach alpha was < 0.5 and therefore this strategy is not a significant construct (Table 5.28). Statements 11, 17 and 26 refer to changes and modifications of a given program in which only a specific section is expanded. This strategy is used mainly during maintenance of software programs when a specific part of the program has to be modified. It is therefore more relevant to updates and changes within existing programs (§4.5.1.4).

- ***Trial-and-error strategy***

  A mean value of $\bar{x}$ = 2.67 (Table 5.28) reveals that certain participants indicated that they used this strategy. These students tried to program a possible solution and hoped that it would work (Statement 21). The participants who took this approach had difficulty in completing their programs. However, this can ultimately lead to a gradual form of program development (§4.5.1.5).

5.4.2.4 Mean values, standard deviations and correlation of various constructs

Although the constructs (cognition, metacognition and problem solving) could not be validated (inadequate amount of data), mean values along with standard deviations have been calculated and are shown in Table 5.29

Table 5.29: Mean values and standard deviations for cognition, metacognition and problem-solving sections

Construct	$\bar{x}$	s
Cognition	3.16	0.50
Metacognition	3.15	0.43
Problem solving	2.72	0.38

The highest mean value occurs in the cognitive domain with $\bar{x}$ = 3.16 on a Likert scale of 1 to 4. This indicates that most questionnaire participants made use of cognitive knowledge, and skills during the programming task. There was also reasonable evidence of metacognitive activities in programming, as indicated by a mean of $\bar{x}$ = 3.15. The lowest mean is for problem-solving strategies with a value of $\bar{x}$ = 2.72. Possible reasons for this result are that they lacked knowledge of specific problem-solving strategies or were unable to apply these approaches in a programming task. In many cases, it is expected that novice programmers would use an implicit approach, rather than develop their own strategies as they learn to program. Next, the correlations between the constructs will be discussed.

Table 5.30: Correlations between constructs

Construct	*r*
Cognition Metacognition	0.31
Cognition Problem solving	0.33
Metacognition Problem solving	0.45

The indication is that there was a positive correlation between different constructs, which means that cognitive, metacognitive and problem-solving knowledge, skills and strategies may support the learning of OOP (Table 5.30).

5.4.3 Open-ended questions

The questionnaire also included four specific open-ended questions about programming, which are shown in Table 5.31. These were analysed manually in a qualitative way and are discussed after the table.

Table 5.31: Open-ended questions in the questionnaire

Question No	Open-ended question
43	Would you describe yourself as **successful** or **unsuccessful** in computer programming? Please motivate
44	Do you make use of any special strategies, plans or useful 'tricks' when you write a computer program? If so, please give all the details
45	During programming of a ***new class***, I use the following sequence of general steps to solve a problem (please specify your sequence in detail)
46	Do you make use of any supportive memory representation techniques **during** your programming task? If you do, give a **diagram** or a **description** of all the details please

- ***Question 43***

  Participants were required to classify themselves as successful or unsuccessful in the programming domain. Note that this section is only applicable to the participants of 2006 who were required to complete the questionnaire (Participants 29 to 48) (Table

5.2). According to their descriptions, only six participants rated themselves as successful, namely Participants 32, 36, 40, 41, 44 and 47. Furthermore, when comparing the number of self-rated successful participants in this question to the information in §5.2.7 (Table 5.18), it is interesting to note that according to the successful output of the *Date class* program (Program output: $x \geq 3$ on a scale of 4), Participants 32, 40 and 44 were indeed successful, obtaining marks of 100%, 85% and 97% respectively.

With reference to Table 5.2, Participants 36, 41 and 47 also described themselves as successful. P36 obtained 82%, P41 obtained 75% and P47 obtained 86%, although they did not provide the correct program output (Program output: $x < 3$ on a scale of 4). These participants probably require additional knowledge, skills and strategies to become truly successful programmers.

Although Participants 29, 38, 42 and 48 obtained satisfactory program output and scored 3 or 4 on a 4-point scale for successful programming, they did not describe themselves as successful programmers. Yet, they obtained high marks for the cognitive, metacognitive, problem-solving and OOP categories in Table 5.18, namely 97%, 90%, 90% and 84% respectively.

The **successful participants**, as indicated from their program output in Table 5.18, and who completed the questionnaire in 2006, are P29, P32, P38, P40, P42, P44 and P48, four of whom did not view themselves as successful in programming (P29, P38, P42 and P48).

- ***Question 44***

  It is important to understand how programmers think and, in this question, the participants indicated whether they used any special strategies, plans or useful ‘tricks’ during programming. Three successful participants mentioned that they do not employ any such problem-solving strategies (P38, P42 and P48). Eight participants responded that they use certain strategies, plans or ‘tricks’, and these are shown in Table 5.32. Note that P30 mentioned that he had studied existing examples to support him in the programming process i.e., he applied the knowledge and comprehension skills of Bloom’s taxonomy (§3.2.2). By contrast, P29, P32 and P40 successfully applied knowledge and skills in analysing and synthesising a new program. For example, P29 visualised the whole program; P32 used a *point-by-point* strategy as illustrated in the

detailed analysis in Appendix G, and P40 decomposed the problem into subproblems. These are higher-order skills that can enhance the programming process.

Both P41 and P44 used the test and reason out strategy. However, P41 was unsuccessful, whereas P44 was successful with regard to program output. As shown in Table 5.2, P44 obtained 3 marks for the application of regulation strategies, and P41 obtained 1. Furthermore, P41 experienced difficulties with the correct programming techniques, statements, application of various methods, reasoning skills in OOP and exception handling (Table 5.2). P41 obtained 75% and P44 obtained 97%. However, both participants used the bottom-up approach.

Table 5.32: Strategies, plans or useful 'tricks' that participants used when writing a program

Question 44: Strategies, plans and useful 'tricks' used by participants	
Unsuccessful participants	**Successful participants**
- *Study examples* [P30] - *Determine general and specific cases* [P34] - *Build the same as Lego-blocks* [P36] - *Test and reason out* [P41]	- *Visualise the whole program* [P29] - *Working point by point* [P32] - *Try to break down the problem into subparts* [P40] - *Test and reason out* [P44]

- ***Question 45***

  The purpose of this question was to determine whether participants used a specific sequence of general steps during programming a *new* class.

Table 5.33: Problem-solving steps used by participants in a programming task

Question 45: Problem-solving steps used by participants	
Unsuccessful participants	**Successful participants**
- *Use an example, plan and extend* [P30] - *Read question, use text book, try to program* [P31] - *Design GUI, methods, output* [P33] - *Design methods, test methods and class, modify* [P34] - *Design class, methods, modify if necessary* [P35] - *Design class, determine scope, complete* [P36] - *Write constructor, initialise variables, write methods, test errors* [P37] - *Write variables, constructor, methods, detail* [P41] - *Write class, methods, use variables* [P46]	- *Visualise, use components, methods, test* [P29] *Determine purpose, parameters, input, output variables required, calculations, return values and problems* [P32] (see Appendix G) - *Methods, constructors, complete detail* [P38] - *Methods, declare variables, test methods* [P40] - *Write variables, constructor, methods, test program, use tests data* [P42] - *Start with the class, extend the class* [P44] - *Determine methods, variables, constructor and complete the detail* [P48]

Based on Table 5.33, specific problem-solving steps used by successful programmers can be inferred:

Step 1: Determine the purpose of the program.

Step 2: Make decisions about the framework of the program, input, output and variables, calculations, return values required.

Step 3: Complete the details of the constructor and all other methods.

Step 4: Test the program using test data and solve the problem

Note that these steps are similar to the detailed steps during OOP shown in Table 3.2, which refers to understanding, designing, coding and testing of a program (§3.3.2) and applies all the levels of Bloom's taxonomy in each step.

- ***Question 46***

  This question set out to determine whether or not participants used supportive memory representation techniques during programming. Examples are extracted and given in Table 5.34.

Table 5.34: Supportive memory representation techniques that participants used during a programming task

Question 46: Supportive memory representation techniques used by participants	
Unsuccessful participants	**Successful participants**
- *Underline important information* [P30] - *Try to remember what the lecturer explained* [P31] - *Math functions, graphs, tables and notes* [P36] - *Use decision trees for if-statements* [P41]	- *I make use of a diagram* [P29] - *I show the program flow* [P40]

Two of the successful programmers mentioned supportive memory representations. The other successful programmers made no explicit use of specific memory representations.

The unsuccessful programmer, P30, mentioned *underlining of important information*, that emphasises important concepts in the problem statement. P31 tried to remember what the lecturer had explained. This is an example of knowledge and comprehension skills as outlined in Subsection 3.2.2 in Bloom's taxonomy (Table 3.1). P36 referred to the use of functions, graphs, tables and notes to support him in the programming process. He obtained 82%, but could not provide correct program output (Table 5.2). P41 used decision trees for *if*-statements, however, he also encountered problems in providing correct output. These examples of approaches used by unsuccessful programmers emphasise the need for explicit

teaching of metacognitive approaches and problem-solving skills and strategies to support effective programming.

5.4.4 Application of positivism and interpretivism in Section 5.4

- *Positivism*

  Positivism was applied in Subsections 5.4.2.1 to 5.4.2.4 for the verification and confirmation of empirical observations by using statistical measurements to ensure reliability and validity of data (§2.6.2). The statistics are based on the raw scores given in Tables 5.26 to 5.30 of the closed-ended questions and include the following: descriptive statistics (mean values and standard deviations), correlations between constructs and reliability testing.

- *Interpretivism*

  Interpretivism was used to investigate participants' subjective and reflective interpretations (§5.4.3) regarding the issue of whether or not they consider themselves as successful programmers, what special strategies they used while programming, which sequence of steps they follow during problem solving, and whether they used supportive memory representation techniques during a programming task (§5.4.3). The following principles laid down by Klein and Myers (1999) (Table 2.2) were considered in the analysis of participants' thinking processes in the open-ended questions:

  Principle 4, Abstraction and Generalisation, suggests that the specific interpretation of data and idiographic findings of the study should be related to general concepts. This study does so by relating the students' specific thinking processes and programming statements to generic problem solving in object-oriented programming. This may explain differences between unsuccessful and successful participants as shown in Question 43. Furthermore, generic problem-solving steps may be identified from participants' thinking processes, as shown in Table 5.33 (Question 45) and the subsequent text.

Principle 7, Suspicion, was also applied. The researcher acknowledges that identifying biases and distortions in the written thinking processes of participants requires sensitivity and she deliberately did it as objectively as possible. In addition, the researcher explicitly avoided bias in her interpretation of the qualitative data of the open-ended responses.

5.5 Triangulation between different analysis methods

Triangulation is a method used by qualitative researchers to check and establish validity in their studies (§2.5.3). Methodological triangulation establishes validity between different analysis methods (Guion, 2002:2). This section takes triangulation further by describing the relationship between the quantitative (§5.2, §5.2.5) and qualitative (§2.9, §5.3) analysis methods by referring to the participants' cognitive, metacognitive, problem-solving and OOP activities. This is done by comparing statistical data, extracted from Table 5.2, with associated details from the *Atlas.ti* records (Section 5.3). Note that analysis of participants' computer programs and thinking processes is an indication of their *actual performance* during the programming process whereas the analysis of the questionnaire (the closed-ended questions, and to a lesser extent, the open-ended questions) indicates participants' *perceptions* about themselves and about their programming behaviour.

Certain participants were selected as explained in the second set of bullets below, and their scores from particular aspects of the computer programs were tabulated against extracts from their thinking processes:

- Measurement criteria from Table 5.1 were selected. The marks obtained for those criteria (Table 5.2) are presented in the first column of each of the tables following. Although values for only analysis, synthesis and evaluation, in the case of cognition, are presented, the mean value for the *complete* construct is given.
- Examples of participants' associated thinking processes are given in the central column.
- The relationship between these two analysis methods is summarised in the third column by referring to participants' cognitive, metacognitive, problem-solving and OOP activities.

Tables 5.35 to 5.40 are presented in ascending order of total scores obtained by participants (see Table 5.2). Data of the following participants is presented:

- two unsuccessful participants, P31 and P24, who used Delphi and Java respectively. The weakest participant and a participant obtaining a mark near 50% were selected (Tables 5.35 and 5.36 – participants' total scores are given in the header row);
- eight average participants: P1, P4, P10, P11 (Delphi), P14, P22, P26, P39 (Java), obtaining scores between 55% and 65% (total scores given in second-last row of Tables 5.37 and average score in the header row in Table 5.38); and
- two successful participants: P29 (Delphi) and P32 (Java), who obtained 97% and 100% respectively (Tables 5.39 and 5.40 – total scores are given in the header row).

Table 5.35: Triangulation between different analysis methods: P31's data

Participant 31 (Delphi – unsuccessful programmer: 16%)		
Quantitative analysis (Table 5.2)	**Qualitative analysis: *Atlas.ti* (Section 5.3)**	**Triangulation**
Cognition * $\bar{x}$ = 1.33 Analysis: 1 Synthesis: 1 Evaluation: 1	*I find out when it is a leap year* (Table 5.20, Sub-theme 1.1). *I had problems to interpret leap years.* *An if statement was necessary for leap years* (Analysis). *I still have problems* (Synthesis). *My program does not work (Evaluation)* (Table 5.3).	The thinking processes reflect the problems that P31 experienced during programming. The mark that he obtained for cognition ($\bar{x}$ = 1.33), as well as the detailed marks for Analysis, Synthesis and Evaluation confirm that he was unable to apply cognitive activities in the program.
Metacognition $\bar{x}$ = 0.33 Planning: 0 Monitoring: 1 Regulation: 0	*My program does not work. My program did not show me errors* (Monitoring). *I don't know if it is correct* (Regulation) (Table 5.4.).	No evidence was found of any planning or regulation strategy as indicated by both the thinking processes and the marks for these subcategories. The mean value for metacognition ($\bar{x}$ = 0.33) further emphasises that P31 could not diagnose the errors nor make the necessary changes or correct them.
Problem solving $\bar{x}$ = 0 Application of problem-solving strategies during programming	*I have typed all the things that I thought should be in the program* (Table 5.5).	P31 used a trial-and-error strategy during the programming process, as implicitly indicated in his thinking processes. This is reflected where zero marks were allocated for trial-and-error, since it was not considered an acceptable strategy (§5.2.1).
OOP knowledge and skills $\bar{x}$ = 0.5 Application of various OOP knowledge and skills. Program 5.1, Program 5.2, Fig 5.2	Very few examples of programming knowledge and skills were included. *I have problems with the programming of classes* (see Appendix F).	This participant had deficiencies in OOP and could not solve the problem as indicated by the thinking processes. Compilation of P31's program in Fig. 5.2 shows numerous errors. This is also reflected in the mark that he obtained for OOP ($\bar{x}$ = 0.5).

*The mean value for the complete cognition construct of P31, was converted to a mark out of 4 (Table 5.2)

Table 5.35 relates to P31, a very weak programmer with a final score of 16%. He had problems in applying the various skills of Bloom's taxonomy ($\overline{x}$ = 1.33). P31 could not interpret and judge the program, and lacked the concept of a holistic view. No evidence of regulation was given and little evidence of monitoring emerged. Compilation showed many errors, but the participant was unable to monitor or regulate his work by using compilation to evaluate it. In contrast, he stated that the program did not show errors.

P31 used the trial-and-error strategy and typed programming code without any explicit planning. He displayed limited knowledge and thinking skills in the programming domain (Programs 5.1 and 5.2). Furthermore, he could not solve the actual problem nor could he produce program output (Fig. 5.2). He had deficiencies in various domains as indicated in both the computer program and thinking processes. The programming process was ineffective and the final product was non-functional.

Table 5.36 following also relates to an unsuccessful programmer, P24 with a score of 45%.

Table 5.36: Triangulation between different analysis methods: P24's data

Participant 24 (Java – unsuccessful programmer: 45%)		
Quantitative analysis (Table 5.2)	**Qualitative analysis: *Atlas.ti* (Section 5.3)**	**Triangulation**
Cognition $\bar{x}$ = 2.5 Analysis: 2 Synthesis: 2 Evaluation: 2	*Subtract the date variables (day, month, year) from the original date to determine the difference* (Analysis). *I should use: if, for, while to solve the problem* (Synthesis). *Does the method execute according to what was expected?* (Evaluation).	Some analysis, synthesis and evaluation skills were demonstrated in P24's thinking processes. However, he had problems in applying his knowledge and skills in the context of a new program. This is reflected in the mark obtained for the cognition construct ($\bar{x}$ = 2.5).
Metacognition $\bar{x}$ = 1.67 Planning: 3 Monitoring: 1 Regulation: 1	Planning: - *Describe class* - *Determine types of variables, required methods* - *Describe steps to determine the difference between two dates* *The method should return a value* (Monitoring) *Write a test program to test the class* (Regulation).	Although this participant aimed to use some appropriate steps in planning the program as shown in the second column, he could not follow them through by using monitoring and regulation strategies efficiently. This is confirmed by 1.67 for metacognition.
Problem solving $\bar{x}$ = 0 Application of problem-solving strategies during programming	A bottom-up approach was demonstrated in the thinking processes where P24 referred to details of the *Date class e.g., Describe class … required methods*.	P24's thinking processes demonstrated a bottom-up approach. In contrast, his incomplete program showed the framework of methods, which is an indication of a top-down approach. As a result of this discrepancy, he obtained zero marks for this section.
OOP knowledge and skills $\bar{x}$ = 1.79 Application of various OOP knowledge and skills	Only a few examples of programming knowledge and skills were included e.g., *The method should return a value … use an if, for, while to solve the problem … write a test program to test all methods*	The program was incomplete without evidence of detailed programming. He could not solve the problem and did not produce correct program output as confirmed by a mean of 1.79 for the OOP construct. The final score of 45 indicates potential but inability to bring the task to closure.

P24, the second ‘unsuccessful programmer’, showed potential, as he mentioned some valid examples of analysis, synthesis, evaluation, and planning. He obtained mean values of 2.5, 1.67 and 1.79 for cognitive, metacognitive and OOP activities respectively (Table 5.36). Unfortunately he demonstrated little evidence of monitoring and regulation. Note that, although P24’s thinking processes indicated a bottom-up approach, the incomplete program, by contrast, showed the framework of methods, which indicates a top-down approach. This indicates a degree of confusion, and this is a clear case where explicit teaching of skills and strategies might have made a difference. He obtained zero for the problem-solving category. He could not use the formulae to calculate leap years and the difference between two dates. Furthermore, P24 was unable to solve the actual problem and could not provide evidence of program output.

When comparing P24 to P31 (Tables 5.36 and 5.35), some indication of progress is in evidence. P24 used more cognitive knowledge and skills and obtained a mean of 2.5. By contrast, P31 obtained only 1.33. Furthermore, P24 was able to plan his task (3) and indicate the use of regulation (1); whereas P31 did not show any evidence of planning or regulation strategies at all (he obtained 0 for both). Both participants obtained zero marks for the problem-solving strategy. However, P24 used more OOP knowledge and skills (1.79) than P31 (0.5).

The next part of the discussion considers eight so-called ‘average participants’. Table 5.37 was extracted from Table 5.2 by selecting participants who obtained overall percentages between *55%* and *65%*. The table shows their detailed scores and the group means of these average participants for all subcategories and major categories.

Table 5.37: Allocated values of average participants

Average participants (55%-65%)									
	Delphi				Java				
Participant number	**1**	**4**	**10**	**11**	**14**	**22**	**26**	**39**	$\overline{x}$
Cognitive knowledge, skills									**3.14**
Knowledge	4	2	3	3	4	4	4	4	**3.50**
Comprehension	3	3	3	3	3	4	4	3	**3.25**
Application	2	2	3	3	3	4	4	3	**3.00**
Analysis	3	2	2	2	3	3	4	3	**2.75**
Synthesis	2	1	2	2	2	3	3	2	**2.13**
Evaluation	2	1	1	1	2	2	3	2	**1.75**
Metacognitive strategies									**2.53**
Planning	2	3	3	3	3	3	3	2	**2.75**
Monitoring	1	2	2	2	2	0	0	2	**1.38**
Regulation	2	1	1	1	2	0	0	1	**1.00**
***Problem-solving strategies**	8	8	8	8	8	8	8	8	**8.00**
OOP knowledge and skills									**2.60**
Proper requirements analysis	2	2	3	3	3	3	3	3	**2.75**
Programming techniques	2	2	3	3	3	3	3	3	**2.75**
Programming statements	3	3	3	3	3	3	3	3	**3.00**
User-friendliness	1	2	2	3	0	0	0	1	**1.13**
Classes and objects	2	3	2	2	3	3	3	3	**2.63**
Method application	2	3	2	2	3	2	3	3	**2.50**
Access control	3	3	3	3	4	3	3	3	**3.13**
Parameter passing	3	3	3	3	3	3	3	3	**3.00**
Reasoning	3	3	2	2	3	3	3	2	**2.63**
Exception handling	0	0	0	0	0	0	0	1	**0.13**
Program structure and scope	3	3	3	3	3	3	3	3	**3.00**
Solution of problem	2	2	2	2	2	2	3	2	**2.13**
Program evaluation	1	2	1	1	2	2	2	2	**1.63**
Correctness of output	0	0	0	0	0	0	0	0	**0.00**
TOTAL (%)	56	56	57	58	64	61	65	62	**59.88**
***Problem-solving strategy**	BU	BU	BU	BU	BU	IG	BU	BU	

Table 5.38 is based on Table 5.37, summarising and integrating the data of the average participants by giving the mean scores obtained. In addition, certain stereotypical examples of their thinking processes are given to support triangulation.

Table 5.38: Triangulation between different analysis methods: average participants' data

Participants P1, P4, P10, P11, P14, P22, P26, P39 (Average partcipants: 60%)		
Quantitative analysis (Table 5.2)	**Qualitative analysis: *Atlas.ti* (Section 5.3)**	**Triangulation**
Cognition $\overline{x}$ = 3.14 Analysis: $\overline{x}$ = 2.75 Synthesis: $\overline{x}$ = 2.13 Evaluation: $\overline{x}$ = 1.75	- *A leap year is a year that can be divided by 4 without a remainder* [P1] (Analysis). - *Use the procedure Add(value:integer)* [P11] (Synthesis). - *The program cannot be compiled* [P39] (Evaluation).	These participants could not fully apply higher-order thinking skills, although there was evidence of awareness of such skills. They battled with the calculations. Only a few evaluated their programs and this is confirmed by the average mark that they obtained for the evaluation subsection ($\overline{x}$ = 1.75).
Metacognition $\overline{x}$ = 2.53 Planning: $\overline{x}$ = 2.75 Monitoring: $\overline{x}$ = 1.38 Regulation: $\overline{x}$ = 1.00	- *Create new unit, use various procedures … functions, create new application, program event handlers* [P11] (Planning). - *I should convert the string to an integer* [P14] *(*Monitoring). - *I determined the difference in days but was incorrect with 1 day* [P39] (Regulation).	The average participants found it difficult to apply metacognitive activities during programming. In particular, they struggled with monitoring and regulation of cognitive resources, as shown by the general lack of such in the thinking processes. They obtained $\overline{x}$ = 1.38 for monitoring and $\overline{x}$ = 1.00 for regulation strategies.
Problem solving $\overline{x}$ = 8 Application of problem-solving strategies during programming.	- *I use two dates with the year, month and day fields* [P1]. - *I will start with the Data class unit and use various procedures and functions [*P11]. - *The program requires classes with various methods* [P22]. - *What is the date? Which format is required? [*P26]. - *I start with the number of days for each month* [P39].	The second column shows some examples of the participants' detailed descriptions with reference to variables and methods required in the *Date class*. Seven participants used the bottom-up strategy and one used the integrated strategy [P22]. All participants obtained 8 for this section.
OOP knowledge and skills $\overline{x}$ = 2.60 Application of various OOP knowledge and skills	Participants' thinking processes indicated discernment, but insufficient to complete the task correctly. Some examples: - *… insert procedure add, procedure subtract* [P11]. - *… use parseInt to convert string values to integer* [P22].	Participants had problems with user-friendliness, exception handling and program evaluation. None of them scored marks for correctness of output. The mean for OOP is $\overline{x}$ = 2.60. This is confirmed by their thinking processes.

Some fairly good examples of programming knowledge and skills were evidenced in these programs. However, the average participants should improve on the use of higher-order thinking skills. They had problems in transferring their knowledge and skills to a new program in a context with which they were not familiar.

These participants found it difficult to apply metacognitive activities, such as self-efficacy and self-judgement, during programming. They encountered particular problems in monitoring ($\overline{x}$ = 1.38) and regulating ($\overline{x}$ = 1.00) their cognitive resources. Very few of them applied any form of regulatory strategy and this hindered refinement and finalisation of the programs.

With regard to problem-solving skills, the scores allocated were high, due to clear evidence of use of strategies. Seven used the bottom-up approach and one the integrated strategy (P22). This indicates that most of them commenced with the details of a method or unit and then proceeded to higher levels of abstraction (§4.5.1.1). Many students used strategies implicitly not explicitly, which is not the same as having an optimal approach, and it is of concern that the 'average' student was unable to complete the task successfully.

Participants obtained 53% and more for various programming sections, indicating a grasp of syntax and semantics. However, they encountered problems in programming user-friendliness, exception handling and program evaluation. None of them obtained any marks for the 'correctness of output' criterion. Overall, they encountered problems with synthesis, evaluation, regulation, and application of certain aspects of OOP knowledge and skills during the programming process.

To consolidate, these findings indicate, firstly, that the participants were ineffective in overall problem solving and, secondly, that they were unable to integrate fine-grained generic reasoning processes with their technical programming knowledge. To address the first, they used problem-solving skills adequately in the process of designing and coding a program, but they were not able to use problem-solving skills and mathematical cognition effectively to determine the complex formulae for calculations with dates. The thinking processes played an important role in pinpointing this problem. With regard to the second issue, participants had satisfactory knowledge and comprehension of basic programming syntax and semantics, but because many of them applied coding processes to implement flawed reasoning, their programs were doomed to failure. In addition, creativity and metacognition were not in evidence when handling the specialised aspects such as usability and exception handling. The discussion now moves on to 'successful programmers' who scored 97% and 100%.

Table 5.39: Triangulation between analysis methods: P29's data

Participant 29 (Delphi – successful programmer: 97%)		
Quantitative analysis (Table 5.2)	**Qualitative analysis: *Atlas.ti* (Section 5.3)**	**Triangulation**
Cognition $\overline{x}$ = 4 Analysis: 4 Synthesis: 4 Evaluation: 4	*I determine the requirements and visualise the final result. How will I proceed to solve the problem? I consider the screen layout. Make changes to the form and test the program.*	P29 is a successful participant and obtained 97%. He had insight into the problem and used all the skills of Bloom's taxonomy effectively. He made the required changes to the program and tested the output. These thinking processes are confirmed by an average of 4 for the cognitive section.
Metacognition $\overline{x}$ = 3.67 Planning: 4 Monitoring: 4 Regulation: 3	Planning *I reread the problem with attention; design application form; create new Date unit; design framework for procedures; include calculations within procedures and functions; test the program.* *I should make a few changes to the program e.g. set boundaries (months = 12) … insert a few error messages.* (Monitoring and Regulation).	P29 is goal-oriented and applied various metacognitive strategies to monitor his programming performance. His control of cognitive resources, as mentioned in his thinking processes, is reflected in the marks that he obtained for planning and monitoring (4). However, he could improve on the use of regulation strategies (3).
Problem solving $\overline{x}$ = 8 Application of problem-solving strategies during programming	*I read the assignment with attention and determine the big picture. I start to solve the problem. I design a new form. I create a new Date class unit and complete the details of that unit.*	An integrated approach was followed, where P29 referred to both the big picture and to details of a specific unit. For this section, he obtained 8.
OOP knowledge and skills $\overline{x}$ = 3.86 Application of various OOP knowledge and skills (Fig 4.3)	Sound examples were given of programming knowledge and skills e.g., *create the framework for the Date class ... calculate the difference between dates … use a procedure or function.*	P29 solved the problem. A mean value of $\overline{x}$ = 3.86, was awarded for this section. This is confirmed by various examples of programming knowledge and skills.

P29 had a solid knowledge base from which to work. His thinking processes indicate that he spent more time determining how to proceed and how to solve the problem. This is confirmed with scores of 4 for analysis and synthesis. Furthermore, he had holistic insight into the programming problem and applied cognitive, metacognitive, problem-solving and OOP activities to solve the problem successfully (Table 5.39). He used planning strategies to direct his thinking and carefully monitored his own thinking processes.

An integrated problem-solving approach was followed, as P29 referred to both the big picture and the details of a specific unit. He solved the problem, however, he could improve on exception-handling techniques to prevent possible run-time errors.

Table 5.40 following relates to the final selected programmer, P32, another successful programmer – the best of the entire group – who obtained a score of 100%.

Table 5.40: Triangulation between different analysis methods: P32's data

Participant 32 (Java – successful programmer: 100%)		
Quantitative analysis (Table 5.2)	**Qualitative analysis: *Atlas.ti* (Section 5.3)**	**Triangulation**
Cognition $\overline{x}$ = 4 Analysis: 4 Synthesis: 4 Evaluation: 4	*Which calculations are needed … Purpose? Parameters? Input, Output? Calculations?* (Analysis) (Fig. 5.7). *Write DateDifference( ) method: Subtract: calculate the largest date … compare years and thereafter months and then days* (Synthesis). *The biggest problem was the difference between days* (Evaluation) (Table 5.7, Appendix G).	P32 used all the skills of Bloom's taxonomy. He worked through different levels of abstraction and could determine which calculations were required. His detailed thinking processes, shown in Appendix G, are confirmed by a mean of 4 for cognition.
Metacognition $\overline{x}$ = 4 Planning: 4 Monitoring: 4 Regulation: 4	*Create framework for Date and Test class, ... Create a constructor …* (Planning). *The dateDifference( ) method is difficult and I cannot think of a way immediately* (Monitoring). *In the dateDifference( ) method I can add the days from 1 January 1800 up to date1. This method is difficult and I should provide for many exceptions, especially for leap years* (Regulation) (Table 5.8).	He applied various metacognitive strategies during the programming process. P32 used planning strategies to set goals and to analyse the programming task. He identified various errors during monitoring and made appropriate changes to regulate his own cognitive processes. This is reflected in the mark he obtained (4) for this section.
Problem solving $\overline{x}$ = 8 Application of problem-solving strategies during programming	*I will start with the framework for the Date class and Test class, headings, import given methods, etc.* (Table 5.9, Fig. 5.7).	P32 clearly used a top-down approach during problem solving and program comprehension and applied his OOP skills very effectively. A mark of 8 was awarded to the problem-solving construct.
OOP knowledge and skills $\overline{x}$ = 4 Application of various OOP knowledge and skills Programs 5.3 and 5.4, program output (Fig. 5.3).	Excellent examples of programming knowledge and skills were included in the thinking processes (Tables 5.7 to 5.9, Fig. 5.7, §5.2.7 and Appendix G).	Sound examples of OOP were illustrated in both the computer program ($\overline{x}$ = 4) and the detailed discussion in the thinking processes.

P32 has a well-organised personal knowledge structure and worked through different levels of abstraction during the programming process (Table 5.40). As indicated by his thinking processes, for example, *I cannot think of a way immediately*, P32 put the problem aside for a while without consciously reflecting on it. As time went by, he obtained new perspectives on the problem. This is an example of incubation (Sternberg, 2006:419). P32 used a more systematic approach and his detailed analysis skills (Table 5.7, Appendix G) are evidence of the fact that he spent much time on thinking about the problem. He further showed high accuracy in reaching the appropriate solution. P32 used a top-down strategy and applied different OOP skills very effectively to solve the problem. He used test data to determine whether the program's output was correct (Programs 5.3 and 5.4, program output Fig. 5.3).

In conclusion, P32 deliberately focused on the programming problem and integrated all the required knowledge, skills and strategies to succeed. He compared his performance with the goal and produced a convergence solution where various cognitive, metacognitive, problem-solving and OOP activities were integrated to solve the problem with aplomb. These actions are reflected in a final mark of 100%, clearly showing P32 to be the best programmer in the study.

P32 was better than P29 where he (P32) more effectively regulated his programming task and applied all OOP knowledge and skills such as exception handling and access control (Tables 5.39, 5.40 and 5.2)

To summarise this section, methodological triangulation was applied to establish validity between different analysis methods. It is clearly illustrated in the different examples in Tables 5.35 to 5.40 that the participants' thinking processes were reflected in their scores and in their performances with relation to the kind of programs they produced.

Furthermore, unsuccessful, average and successful participants demonstrated different kinds of thinking patterns. This is confirmed by the statistical measurements in the first column of each table.

Unsuccessful participants did not have a plan during programming. They had deficiencies in various domains and consequently could not provide evidence of program output. The average participants had problems with synthesis, evaluation, monitoring and regulation during programming. They preferred to start their programming at the level of details and then proceed to higher levels of abstraction.

Average participants could handle the basic aspects of programming, but had problems in the programming of user-friendliness, exception handling and program evaluation. Although they obtained $\bar{x}$ = 2.60 for OOP, they scored zero for the correctness of output.

By contrast, successful participants orchestrated various knowledge, skills and strategies in a comprehensive way. They spent much time on thinking about the problem. In addition, they selectively combined various programming statements in such a way that they could solve the problem and give evidence of correct output.

The results from Tables 5.35 to 5.40 indicate a close relationship between the quantitative and qualitative analysis methods by referring to participants' thinking processes and computer programs. Each of these tables covers various activities and together they address which kinds of knowledge, skills and strategies participants actually use during OOP and their impact on the effectiveness of the programming process.

5.6 Measures to ensure rigour and quality of data

Chapter 2 (§2.5.3, §2.6.2) refers to various measures to ensure the rigour and quality of data. The qualitative and quantitative measures used in this study are discussed below.

5.6.1 Qualitative measures

Data was triangulated so as to investigate data analysed by more than one method. Students provided detailed and descriptive thinking processes, along with their computer programs, to outline the various knowledge, skills and strategies that they applied during the programming task. In addition, a questionnaire was completed by some of the students. The following qualitative measures were used:

- Detailed descriptions were obtained of students' thinking processes (§5.3). A CD containing the complete analysis process is provided with this thesis – see pocket on inside of back cover.
- Methodological triangulation was applied as the researcher used quantitative and qualitative analysis methods to support a main theory (§5.2, §5.3 and §5.4, §5.5).
- The grounded theory data analysis was aimed towards a process of saturation. However, saturation did not occur until near the very end, on account of the fact that two programming languages, Delphi (P1 to P11; P29 to P31) and Java (P12 to P28, P32 to

P48; Table 5.2) were used. However, the researcher decided to continue the analysis process until all participants' thinking processes had been analysed (§5.3).

- The researcher moved forward from the codes to a coded family, and backwards, from the coded family to codes. This bidirectional iteration supported reconsideration of the correctness of families (§5.3) and the consequent emerged themes.
- The researcher's efforts in striving for coherent interpretations of different resources (§5.2, §5.3) and in triangulating various analysis methods (§5.5) contribute to validity.
- Various of the Klein and Myers' principles (1999, Table 2.1, Table 2.2) were applied to further ensure reliability in Sections 5.2 to 5.4.
- An overview of the themes and the emerging grounded theory was outlined in Subsection 5.3.8.

5.6.2 Quantitative measures

The following quantitative measures were used to ensure validity and reliability (§2.5.2):

- Sample adequacy and factor analysis (Table 5.15) were conducted to validate the scales of the constructs (§5.2.6).
- Cronbach-alpha statistics were used as a measure of internal consistency and reliability (Table 5.26 – 5.28).
- Descriptive statistics were used to show the means and standard deviations (Table 5.12, Table 5.13, Table 5.16, Table 5.19, Table 5.26 – 5.29);
- Correlations (r) between various constructs were investigated (Table 5.17 and Table 5.30);
- The practical significance (effect size) between successful and unsuccessful programmers was determined in terms of various constructs (Table 5.19).

5.7 Overview of the research findings

In this study, students were asked to program a *Date class* and a *Test class*. Whilst it is impossible to identify ideal knowledge, skills and strategies for every programming problem, some inferences can be made about general prerequisites for problem solving in the OOP process. With this objective, different constructs were addressed in this chapter and findings are provided from the following analyses:

- analysis of computer programs and thinking processes (§5.2);
- qualitative analysis of participants' thinking processes with the support of *Atlas.ti* (§5.3);
- statistical and qualitative analysis of the questionnaire (§5.4); and
- triangulation between different analysis methods (§5.5).

The methods were complementary, as each presented a different perspective and covered various facets of the research process (§2.2). The various constructs are reviewed below:

5.7.1 Cognitive knowledge, skills and strategies

- *Computer programs and thinking processes*

  The mean scores showed a downward tendency from the first subcategory of Bloom's taxonomy (§3.2.2) to the last. This implies that many participants could not effectively analyse, synthesise and evaluate their computer programs (Table 5.12). It was notable that all of the eleven successful participants used all of Bloom's subcategories (Table 5.18, Table 5.19). There was however, a slight downward tendency among these successful participants from a score of 4.00 (knowledge, comprehension and application) to 3.55 (evaluation).

  Regarding the themes that emerged from the analysis by means of *Atlas.ti* (Subthemes 1.1 – 1.3), participants' thinking processes illustrate that they did indeed comprehend the core concepts and applied their knowledge and skills in a program. However, certain participants indicated that completing and evaluating a new object-oriented program is difficult (Table 5.20). Only a single reference was made to the explicit use of cognitive strategies (Subtheme 1.4, Table 5.20). Participants had problems with the organisation-and-integration strategy, and it was clear that they found it difficult to structure the various programming statements and methods into a coherent whole.

  To indicate the relationship between participants' computer programs and thinking processes, various examples were extracted and discussed. Triangulation was conducted between different analysis methods of participants' data, as outlined in Section 5.5.

- *Questionnaire*

  By contrast, analysis of the questionnaire data – which related to students' perception and not to their actual performances – presented a slight increase in synthesis ($\overline{x}$ = 3.12) and evaluation ($\overline{x}$ = 3.43) skills (see Table 5.26). It was expected that the mean values would decrease as the implementation of cognitive skills becomes more complex on the higher levels of Bloom's taxonomy (Table 3.1). This interesting discrepancy shows that participants over-estimated their cognitive skills. This probably occurred because they were conscious of their explicit aims to accurately synthesise a program and evaluate it. This finding is in line with the finding by Edwards (2004:27), which suggests that most novices focus on the development of programs and use synthesis skills to write a program, but they should first master the basic comprehension and analysis skills. Zant (2005) also claims that it is difficult for a novice programmer to become an expert without progressing through each of the six levels of Bloom's taxonomy.

5.7.2 Metacognitive knowledge, skills and strategies

- *Computer programs and thinking processes*

  The analysis of computer programs as shown in Table 5.13, revealed a downward tendency from planning ($\overline{x}$ = 3.40) to regulation ($\overline{x}$ = 1.92) in the analysis of computer programs. Most participants could plan the programming process, set goals, and analyse tasks, but had problems managing (monitoring) and regulating their own activities. Subtheme 2.2 (Table 5.21) from *Atlas.ti* analysis showed examples of problems where participants could not reflect on their own programming task:

  > *I have the correct idea but cannot apply it* [P5].

  Examples of monitoring and regulation in Theme 2.2 were:

  > *I determined the difference in days but was incorrect with 1 day* [P39].
  > *I could [have] sent the date to the constructor* [P33].

- *Questionnaire*

  As was the case in Subsection 5.7.1, where a decrease was expected in line with the decrease in performance, the values of perceptions in the questionnaire actually increase. For example the mean value for regulation ($\overline{x}$ = 3.37) was significantly higher than the mean value for planning ($\overline{x}$ = 2.95) (Table 5.27). When programmers do not monitor their performance, this may force them to use regulation strategies as

they attempt to modify and/or correct programming problems, usually unsuccessfully. Here too, participants' perceptions of themselves did not correspond with the actual scores assigned to their programming of the *Date class*. This occurred because they did not know which part of the code was wrong and had not monitored the programming process adequately. Much of the regulation conducted is enforced regulation relating to the iterative modification of specific program segments, without success (Table 5.2).

Although the successful participants used planning, monitoring and regulation strategies (Subtheme 2.2, Table 5.21), they could have applied regulation more explicitly. The programmer should understand the task and should have self-knowledge as well as knowledge regarding appropriate problem-solving strategies.

These results are in line with other researchers' findings. Programmers should use monitoring to observe and reflect on their programming experiences (Schwartz & Perfect, 2002:4, 5) and regulation of their cognitive activity by continuous modification during programming. Bergin et al. (2005:81) mention that students who perform well in programming use more metacognitive management strategies than lower-performing students. Programmers should reflect on their programming, improve the accuracy of judgment and refine their insight into the task (Bergin et al., 2005:82; Roberts & Newton, 2005:132,154; Kapa, 2001:320).

5.7.3 Problem-solving knowledge, skills and strategies

- *Computer program and thinking processes*

  Certain participants did not have the necessary problem-solving knowledge and skills and asked questions such as (Theme 3, Table 5.22):

  > *How will I solve this problem?* [P5]

  Other participants, by contrast, used the required knowledge and skills to solve a problem:

  > *When starting the class, keep the overall picture in mind* [P16],
  > *The question that I asked myself in the beginning was: which methods should be in the class and how can I calculate them?* [P21]

  Most participants described using the bottom-up strategy in their thinking processes. However, it appears that they lacked explicit knowledge of appropriate problem-solving strategies in program comprehension. With reference to Table 5.14, 34 participants

used the bottom-up strategy, while only five used top-down, five used the integrated strategy and two used trial-and-error. Détienne (2003:21) mentions that the lowering of the level of control may result in using the trial-and-error strategy. Two participants did not use any specific strategy (Table 5.14, Table 5.22, Subtheme 3.2). Cañas et al. (2005:96) claim that the learning of appropriate problem-solving strategies reduces cognitive demands and accelerates performance.

- *Questionnaire*

  The use of strategies as mentioned in the questionnaire was low. Mean values ranged from $\bar{x}$ = 2.57 to $\bar{x}$ = 2.89 (Table 5.28). All the mean values were less than 3 on the Likert scale of 1 to 4. In this case, participants' perceptions of their abilities, as indicated by their questionnaire responses, correspond with the actual scores assigned to their performance during programming of the *Date class* task. All the forms of data indicate that they experienced problems in using explicit problem-solving strategies.

5.7.4 Application of knowledge and skills in object-oriented programming

The mean value for solving the actual programming problem was only $\bar{x}$ = 2.71 (Table 5.16). Moreover, participants experienced overall problems with user-friendliness and usability, exception handling and in obtaining the correct output from their programs. Many of these problems arose from logic errors in the code and a lack of insight into the semantics of the programming language. Only some participants were able to design, write and test the program successfully.

As had been stated previously, the measure of success for program output was defined as: $x \geq 3$, on a scale of 4 (Table 5.18, §5.2.7). Using this benchmark, only 11 out of 48 participants, 23%, were successful. The successful participants applied a variety of knowledge, skills and strategies during programming.

5.8 Chapter conclusion

This chapter addressed the empirical research conducted for this study. The focus was on describing knowledge, skills and strategies used by students during computer programming. Both qualitative and quantitative research practices were used. The participants' object-oriented computer programs and written documents were analysed by means of program

analysis and *Atlas.ti* software. A questionnaire was designed to determine which cognitive knowledge and skills, as well as metacognitive and problem-solving strategies students use during OOP. Responses to closed-ended questions were statistically analysed and responses to open-ended questions were discussed.

Grounded theory had a strong influence on the qualitative analysis software of *Atlas.ti* (§2.5.2), which was applied in Section 5.3. Different steps of grounded theory – open coding, axial and selective coding – were used in *Atlas.ti,* as explained in §2.5.2. Finally, theory in the form of themes was generated inductively from the analysis of data. This will be further elaborated in an explanatory scheme in Chapter 6.

Students can use knowledge, skills and strategies to help them to reach specific goals. Chapter 6 will focus on a learning repertoire, which represents the role of cognitive, metacognitive and problem-solving activities during OOP (Fig. 6.5). Furthermore, certain implications for the teaching of OOP will be outlined.

6 Discussion and conclusion

Grounded theories...are likely to offer insight, enhance understanding, and provide a meaningful guide to action (Strauss & Corbin, 1998:12)

6.1 Introduction

The purpose of this study was to investigate *which knowledge, skills and strategies are used during problem solving in object-oriented programming* (§1.3, Tables 1.1 and 6.1). The underlying research ethos of the study is constructivist problem solving, which refers to the students' active construction of computer programs and their application of programming constructs such as classes and objects. It also relates to the researcher's construction of a body of knowledge regarding the students' programming processes, and her interpretation of and reflection on those programming experiences.

Both qualitative and quantitative research was used (Fig. 2.1) and the findings are therefore based on a dual research approach (§2.2, §2.3, §2.6). When applying grounded theory (§2.5.2) in this study, the aim was to inductively generate theory from concepts that emerged out of the literature chapters as well as from empirical data analysis. Qualitative and quantitative methodologies associated with interpretivism and positivism were applied to combine the interpretivist approach with statistically significant effects for further clarification (Fig 2.1).

This chapter focuses on the underlying concepts from the literature and the empirical research findings (§6.2) and proposes a *learning repertoire,* which can serve as a framework to support the learning of object-oriented programming (Fig. 6.5). In addition, suggestions are made regarding how the specific knowledge, skills and strategies used by successful participants can be applied in the practices of teaching. The learning repertoire is presented in Section 6.3, while Section 6.4 applies the findings by suggesting ways in which facilitators/educators can implement them in the teaching and learning of OOP.

6.2 Discussion of the findings of this study

Discussion follows of findings from the literature and from empirical research with reference to the research questions and subquestions in Tables 1.1 and 6.1. The discussion focuses more explicitly on the empirical findings than on the literature, because the literature studied (Chapters 3 and 4) was used as a foundation for the criteria used in data collection and analysis. The literature is therefore implicitly and intrinsically part of the study and its findings. The questions and subquestions in Table 6.1 are not answered singly in sequence. Rather, the discussion is structured by combining questions from the same domains e.g., Questions 1.1 and 2.1 from the cognitive domain, Questions 1.2 and 2.2 from the metacognitive domain, and Questions 1.3 and 2.3 with regard to problem-solving. Question 3.1 is answered in Subsection 6.2.4, based on a discussion regarding successful and unsuccessful participants. Finally, Question 3.2 is addressed in Section 6.4 by contributing to the practice of teaching knowledge, skills and strategies that are used by successful OOP programmers.

Table 6.1 Research questions and subquestions

	Which knowledge, skills and strategies are used during problem solving in object-oriented programming?
1.	**Which *knowledge and skills* are used during problem solving in object-oriented programming?**
1.1	Which cognitive knowledge and skills are used in OOP?
1.2	Which metacognitive knowledge and skills are used in OOP?
1.3	Which problem-solving knowledge and skills are used in OOP?
2.	**Which *strategies* are used during problem solving in object-oriented programming?**
2.1	Which cognitive strategies are used in OOP?
2.2	Which metacognitive strategies are used in OOP?
2.3	Which problem-solving strategies are used in OOP?
3.	**What are the differences between the ways in which unsuccessful and successful programmers apply supportive knowledge, skills and strategies in OOP?**
3.1	What are the differences between the ways in which unsuccessful and successful programmers apply cognitive, metacognitive and problem-solving knowledge, skills and strategies in OOP?
3.2	What contribution can be made to the practices of teaching and learning OOP by applying the knowledge, skills and strategies used by successful programmers?

6.2.1 Response to Subquestions 1.1 and 2.1: Cognitive knowledge, skills and strategies

This subsection briefly summarises specific responses with reference to the questions in Table 1.1 and 6.1, to determine *which cognitive knowledge, skills (Question 1.1) and strategies (Question 2.1) are used in OOP*.

Results from the literature study indicate that Bloom's cognitive categories of learning are appropriate norms for evaluating the range of cognitive abilities (§3.3.2, Table 3.1). Analysis of participants' computer programs and thinking processes according to Bloom's six levels revealed that participants could not readily apply higher-order thinking skills (Table 5.2, Table 5.12, Tables 5.35 and 5.36). Some students lacked important application and analysis skills or tried to synthesise without using all the preceding levels in Bloom's taxonomy (Table 5.20, Subtheme 1.2, Subtheme 1.3, Table 5.26, §5.7.1).

A cognitive strategy is a plan for orchestrating cognitive resources efficiently and helps us to remember, select and organise information within memory (§4.3). Many students found it difficult to organise and integrate various programming statements in a coherent way and did not know how to apply cognitive strategies during programming (Table 5.20, Subtheme 1.4).

The *Date class* task is an example of an open question. This is a critical dimension, since there are many choices and a programmer should make specific decisions. It is essential for students to focus and maintain attention on the problem and the programming process, and to organise their cognitive processes in ways that direct their thinking. In addition, they should learn explicit strategies to support such organisation of their thinking processes while they tackle a programming problem.

6.2.2 Response to Subquestions 1.2 and 2.2: Metacognitive knowledge, skills and strategies

Responses relating to metacognitive activities used during OOP are consolidated in this subsection. The following questions are answered: *Which metacognitive knowledge, skills* (Question 1.2) *and strategies are used in OOP?* (Question 2.2) (Tables 1.1 and 6.1).

Metacognition includes the awareness by learners of the strengths and weaknesses of their abilities and the management of their own cognitive processes. In addition, reflection includes actions during the planning, monitoring and regulation of a task (§4.4.4).

The results of this study show that some participants did not indicate any use at all of specific metacognitive knowledge and skills. Some mentioned awareness of their weaknesses and their inability to complete the programming task (§3.4.1, Table 5.21, Subtheme 2.1, Table 5.23). It was clearly found that programmers who applied metacognition performed better in this programming task (e.g., P32).

Metacognitive strategies are required to direct, monitor and support cognitive processes. Explicit metacognition can improve activities such as selective attention, error detection and control. Results showed that most participants were able to plan a computer program but had problems in managing their own programming activities (monitoring) and in regulating their performance (Table 5.13, Table 5.21: Subtheme 2.2). In some cases students repetitively tried to correct errors but failed to do so. Possible reasons are:

- they could not identify the errors in the program;
- they made incorrect interpretations of error messages;
- they lacked the expertise to diagnose the cause or to address their errors; and/or
- they had problems monitoring their progress.

In programming, help-seeking can be done by consulting a textbook or by accessing on-line help (§4.4.2). A few students used help-seeking strategies to regulate and promote programming e.g., reading manuals or books, studying previous assignments, asking the lecturer and searching relevant websites (Table 5.24, Theme 5). In such situations, students should know how to transfer previous programming experiences and how to contextualise these successfully in new situations.

6.2.3 Response to Subquestions 1.3 and 2.3: Problem-solving knowledge, skills and strategies

The responses regarding problem-solving knowledge, skills and strategies are outlined in this subsection (Table 1.1 and Table 6.1), namely *Which problem-solving knowledge, skills* (Question 1.3) and *strategies* (Question 2.3) *are used in OOP?*

During problem solving, various possible solutions must be identified in order to select the best one to achieve a goal (§3.5). The empirical research showed that many participants could not apply appropriate problem-solving steps in their programs, as outlined in Tables 5.2, 5.22 (Theme 3) and Table 5.23 (Theme 4). Furthermore, they battled to solve and test

the actual program. Even although participants probably had knowledge of problem-solving steps, some could not apply them.

In order to gain a deeper understanding of the programming process, students' strategies during object-oriented program development were investigated (§4.5). It is difficult to identify specific evidence of strategic use and program comprehension, and students are sometimes unaware of the strategies that they use. Various strategies may be used in combination in order to perform different programming tasks. However, learners tend to choose strategies that they believe will result in the most effective performance (Roberts & Newton, 2005:132, §4.2).

The majority of participants used the bottom-up strategy (34) as shown in Table 5.14. Only one participant mentioned explicitly that he used the trial-and-error strategy. It seems that participants used a strategy implicitly without knowing how to direct their thinking during problem solving. Results from the questionnaire show that students were not able to distinguish clearly between the various strategies. They did not have explicit knowledge about different problem-solving strategies as confirmed by the results (Table 5.2, Table 5.28).

From reflection on the findings of the last three subsections, it became clear that cases occurred in this study (as well as in the researcher's general experience of teaching programming) where students had a reasonable grasp of coding and programming constructs – on occasions even sufficient to earn a reasonable mark (score) – but they failed to master the semantics of programming. Furthermore, many such students did not have a holistic grasp of the problem and were over-involved in details. Their inadequate problem-solving skills and/or the inability to comprehend the semantics resulted in non-functional computer programs (e.g. P20, obtained 52%, Table 5.2).

6.2.4 Response to Subquestion 3.1: Differences between unsuccessful and successful programmers

This subsection addresses responses to Subquestion 3.1 where successful and unsuccessful participants apply/do not apply knowledge, skills and strategies during programming (Tables 1.1 and 6.1). *What are the differences between the ways in which unsuccessful and successful programmers apply cognitive, metacognitive and problem-solving knowledge, skills and strategies in OOP?*

Cognitive knowledge, skills and strategies

Unsuccessful participants battled to decompose the problem scenario and to relate subparts to the overall structure. With regard to actual programming (§5.2, Table 5.2), they could not readily apply higher-order thinking skills. Although they used knowledge and comprehension skills, their programs indicate that they debugged and evaluated the code without using detailed application and analysis skills. As a result, they had problems in interpreting their errors, they could not complete the program, and many did not obtain output.

For the higher-order thinking skills (analysis, synthesis and evaluation) required for programming, the successful participants received a mean value of more than 3.5 on a 4-point scale (Table 5.19). Their ability to apply all the levels of Bloom's taxonomy in a task was clear and they achieved a high level of accuracy in solving the problem (Table 5.18, Table 5.19, Table 5.20, Subtheme 1.3). It is notable that they spent more time on the analysis phase and on differentiating how parts are inter-related within the complete program. Their performances illustrate that programmers should understand the problem precisely and they should interpret and evaluate their programming solutions. These findings are in line with Carbone et al., (2002:2) who mention that programming is 'extremely cumulative', and that previous knowledge and skills are therefore used in each successive programming task.

Although cognitive knowledge was clearly evident, only one successful participant explicitly mentioned a cognitive strategy that was used during programming (Table 5.20, Subtheme 1.4). Possible reasons could be that participants did not use cognitive strategies, or they did not realise that they were applying such strategies, or they did not know how to do so in programming. In this regard, Bergin et al. (2005:85) show that cognitive strategies are not as useful in the learning of introductory OOP as they are in other domains.

Metacognitive knowledge, skills and strategies

Unsuccessful participants found it difficult to apply metacognitive activities during programming as they encountered problems in monitoring and regulating their cognitive resources. Very few of them applied any form of regulatory strategy. They could not easily reflect on the task and their own understanding of it, and found it difficult to manage their thinking and reasoning (Subtheme 2.2, Table 5.21).

By using detailed planning strategies, successful participants were able to complete their tasks and to produce high-quality solutions (Table 5.19). Most participants monitored their progress and effectively managed their cognitive resources in the process of finding a

solution (Table 5.18). The regulation strategy of successful participants was slightly lower than 3 ($\bar{x}$ = 2.82, Table 5.19), which implies that they could improve further on regulatory strategies during programming. These findings correspond with Hertzog and Robinson (2005:110, 111) who suggest that monitoring plays a vital role in cognitive performance of complex problem solving and that it guides the process of finding a solution.

Problem-solving knowledge, skills and strategies

Unsuccessful participants did not obtain the required program output. Some encountered problems in systematically applying problem-solving strategies. Instead, they spent time iterating through their programming code to address errors without understanding which sections were incorrect and how to rectify them. Such participants were much less accurate in their efforts to reach an appropriate solution. Although most of the unsuccessful participants used a bottom-up strategy (27), some wrote that they worked without using any specific problem-solving strategies (2). Two used trial-and-error, three used a top-down strategy, and three used the integrated strategy (Table 5.2).

Successful participants had considerable domain knowledge and highly efficient problem-solving skills, which they were able to apply successfully in the task. During program comprehension, seven of them used the bottom-up strategy (§4.5.1.1), two the top-down (§4.5.1.2), and two the integrated strategy (§4.5.1.3). None of the successful participants used the trial-and-error strategy (§4.5.1.5). This appears to indicate that it is not a successful approach in OOP, whereas all the other problem-solving strategies were used effectively (Table 5.2, Table 5.18).

6.2.5 Performance patterns of unsuccessful and successful participants

It is clear from the research results that unsuccessful and successful participants differ in the way they apply various cognitive, metacognitive and problem-solving activities. In order to facilitate a view of these differences, the relationship between these constructs is portrayed visually by a performance profile that represents the differences between the activities used by unsuccessful and successful participants.

6.2.5.1 Imbalances between the constructs

To provide a clearer picture of the role of cognition, metacognition and problem solving, certain differences between unsuccessful and successful programmers are elaborated in a way that indicates various imbalances between the constructs. If cognitive, metacognitive

and problem-solving knowledge, skills and strategies are applied in a balanced way, only small differences should occur between the values obtained for each construct.

With regard to the unsuccessful participants (who did not obtain program output, Table 5.2), the diagram in Fig. 6.1 emphasises imbalances between their cognitive (C), metacognitive (M) and problem-solving (P) abilities. The mean values for each construct (cognition and metacognition), as obtained from **Table 5.19,** are displayed in parentheses. In the case of the problem-solving construct, this was initially scored out of 8 (Section 5.2, **Table 5.2**) and the mark obtained is converted so that all values are presented on a scale of 4.

The success rate (§5.2.7) shows that if the values obtained are below 3 (on a 4-point scale), there are deficiencies in the performance of that specific construct. To be viewed as successful in the various activities, the requirements are:

- to obtain a value ≥ 3 on a 4-point scale for each construct; and
- to minimise the differences between the various constructs.

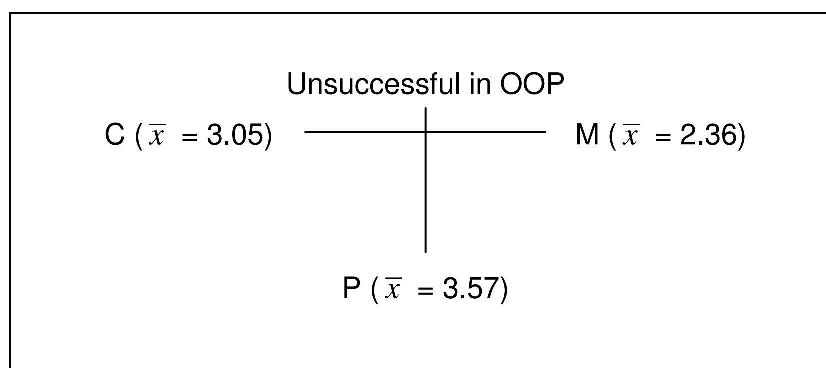


Figure 6.1: Possible imbalances in unsuccessful participants' thinking

The imbalances between cognition, metacognition and problem solving indicate areas where there are some deficiencies. For example, Fig. 6.1 shows that unsuccessful participants obtained 3.05 (cognition), 2.36 (metacognition) and 3.57 (problem solving) respectively. They could improve on the use of cognitive and metacognitive activities. Note that the high mean value obtained for problem solving refers to the explicit or implicit use of specific strategies during the programming task. Only two unsuccessful participants did not use any strategy at all and two used trial-and-error (Table 5.2).

With reference to successful participants, Fig. 6.2 shows that they obtained 3.85 (cognition), 3.33 (metacognition) and 4 (problem-solving) for each construct respectively. This represents greater uniformity in performance and, furthermore, all the values are greater than

3. These participants could however improve on the use of metacognitive knowledge, skills and strategies.

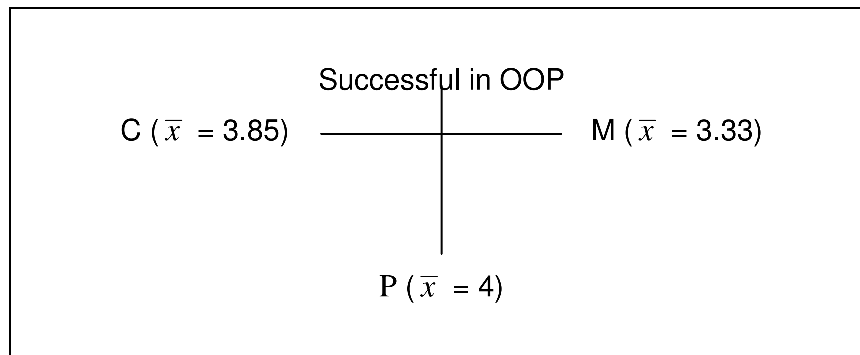


Figure 6.2: Possible imbalances in successful participants' thinking

To further represent such imbalances between the three constructs, a performance profile is applied, which concisely presents deficiencies in specific subconstructs of each domain (e.g. evaluation in the *Cognitive* domain). This is outlined in the subsection below.

6.2.5.2 Performance profile of unsuccessful and successful participants

Participants displayed specific thinking patterns during the programming process. These are summarised by means of a performance profile (PP) representing the detailed differences between unsuccessful and successful participants that emerged from Tables 5.19 and 5.2. The performance profile may display an *overall* thinking pattern of a group of participants during a programming task. In addition, problem areas in unsuccessful participants' thinking processes can be identified and then addressed by applying corrective measures such as those presented in Fig. 6.5 in the next section.

In Tables 5.18 and 5.19 and the subsequent discussion, general profiles of unsuccessful (PP_{un}) and successful participants (PP_{suc}) are shown and explained. The performance profiles of unsuccessful and successful programmers are shown in Fig. 6.3 and Fig. 6.4 respectively, followed by an explanation of the variables in the profile.

The representation shown by the profile in Fig. 6.3 indicates whether participants were unsuccessful or successful in the cognitive (C), metacognitive (M) and problem-solving (P) constructs. The three major constructs – cognition, metacognition and problem solving – appear in *capital* letters, while the detailed subconstructs of each (e.g. knowledge) are in

lower case letters. Where participants displayed various *deficiencies* ($\bar{x}$ < 3), these subconstructs are displayed in *square parentheses,* otherwise *round parentheses* are used ($\bar{x}$ ≥ 3, on a 4-point scale).

Unsuccessful participants

Figure 6.3 shows the performance profile of unsuccessful participants. Round parentheses indicate the areas where they obtained success and square parentheses the areas where they had deficiencies:

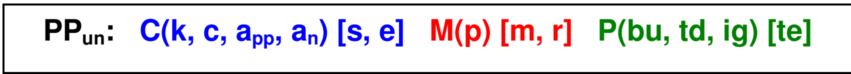


Figure 6.3: Performance profile of unsuccessful participants

Where PP indicates the performance profile

C	=	cognition construct
M	=	metacognition construct
P	=	problem-solving construct
k	=	knowledge
c	=	comprehension
a_{pp}	=	application
a_n	=	analysis
s	=	synthesis
e	=	evaluation
p	=	planning
m	=	monitoring
r	=	regulation
bu	=	bottom-up
td	=	top-down
ig	=	integrated strategy
te	=	trial-and-error, not considered an acceptable problem-solving strategy (§5.2.1)

Fig. 6.3 indicates that unsuccessful programmers experienced problems in synthesising ('s') and evaluating ('e') their programming problem (indicated by square brackets). In addition, they could not apply monitoring and regulation strategies successfully. Most participants obtained marks for problem-solving strategies, however, some used trial-and-error ('[te]' in the profile) and did not obtain any marks.

Successful participants

Fig. 6.4 shows the profile of successful participants, namely, those who obtained program output (Table 5.18, Table 5.19).

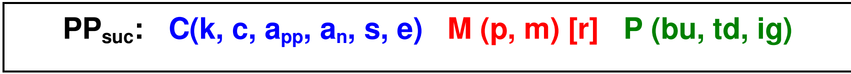


Figure 6.4: Performance profile of successful participants

Although these participants were successful, there is scope for them to improve on the use of metacognitive strategies. Fig. 6.4 indicates that successful programmers competently applied all the cognitive subconstructs (round parentheses). However, they could improve on the use of metacognitive regulation strategies ('r') in OOP. All successful participants obtained marks for problem-solving strategies and none of them used trial-and-error.

By comparing figures 6.3 and 6.4, three main differences emerged between unsuccessful and successful participants:

- Successful participants applied synthesis and evaluation skills during programming, while the unsuccessful did not;
- Monitoring was used by most successful programmers, but not by the unsuccessful; and
- Some unsuccessful programmers used the trial-and-error strategy.

To summarise this subsection, the performance profile is an integrated representation that concisely presents the performance of participants in OOP. The differences between Fig. 6.3 and Fig 6.4 indicate clearly that unsuccessful programmers displayed considerably more deficiencies than successful programmers.

The performance profile is used in this study to represent the difference between the unsuccessful and successful participants, but such a profile may equally well be used to represent the performance of an individual with regard to cognitive, metacognitive and problem-solving thinking patterns during a programming process.

The information presented in Sections 5.2, 5.3 and 5.4; and the general ethos of this study indicates that various techniques are required to support and enhance the object-oriented programming process. Some of the imbalances in performance can be addressed by applying techniques and activities, for example, those used by *successful* participants (Table 5.19). Suggestions are made in the following sections, in particular, the proposal in §6.3 of a learning framework or *learning repertoire* to support students in the learning of OOP.

6.3 A learning repertoire of knowledge, skills and strategies for object-oriented programming

6.3.1 Research methodology applied

The constructivist theory claims that knowledge is actively constructed by a learner (§2.1). The process of OOP is constructivist-driven as programmers continuously make high-level decisions in 'constructing a programming solution' and in solving the problem successfully. Students must be actively involved in selecting, constructing, reflecting and applying their knowledge, and skills in OOP.

Grounded theory had a strong influence on the qualitative analysis of this study as indicated in Section 5.3. A grounded theory is generated inductively from the analysis of the data, as concepts are formulated into a logical, systematic and explanatory scheme (§2.9), as shown in Fig. 5.10. Different steps of grounded theory were used along with the analytical tool, *Atlas.ti,* as explained in §2.5.2. Various themes emerged from this process, each of which captures qualitative richness to explain specific phenomena in OOP (§5.3.3 – §5.3.8; Table 5.20 – Table 5.24, Fig. 5.10). From the 'families' identified, five major themes were generated:

- Cognitive knowledge, skills and strategies;
- Metacognitive knowledge, skills and strategies;
- Problem-solving knowledge, skills and strategies;
- Errors and problems during programming; and
- Additional support during programming.

The first three themes correspond with the major aspects investigated in the literature review of Chapters 3 and 4, while *Errors and problems*; and *Additional support* emerged naturally from the empirical analysis.

6.3.2 A proposed learning repertoire for the effective learning of OOP

Programmers must apply the sum of their knowledge, skills and strategies in a programming task (§4.2). Since programming is highly complex, a *learning repertoire* is proposed to facilitate and support students in learning OOP and in active, holistic involvement in the programming process. Its content is drawn from the underlying literature and from the

empirical research of this study, highlighting ways in which **successful** participants solved the programming problem.

This subsection proposes the learning repertoire, which represents cognitive, metacognitive and problem-solving knowledge, skills and strategies in OOP and their interrelationships. It is shown as an integrated framework in Fig. 6.5 and is based upon observation and scientific study that emphasises how successful students construct their own learning and understanding during programming and how they reflect on those experiences by means of setting goals, monitoring their performance and regulating their progress (Havenga et al., 2008, Appendix H).

Various dimensions are integrated in the repertoire, which explicitly distinguishes between knowledge and skills on the one hand, and strategies on the other. Knowledge and skills form the core. Cognitive knowledge and skills on all levels of Bloom's taxonomy are required for the understanding, designing, coding and testing of a programming problem. Specific emphasis is placed on the higher-order thinking skills. Setting of goals, a high level of motivation, and knowledge about specific tasks are required in the metacognitive domain. In addition, adequate programming knowledge and skills are essential to the ability to complete a new program successfully.

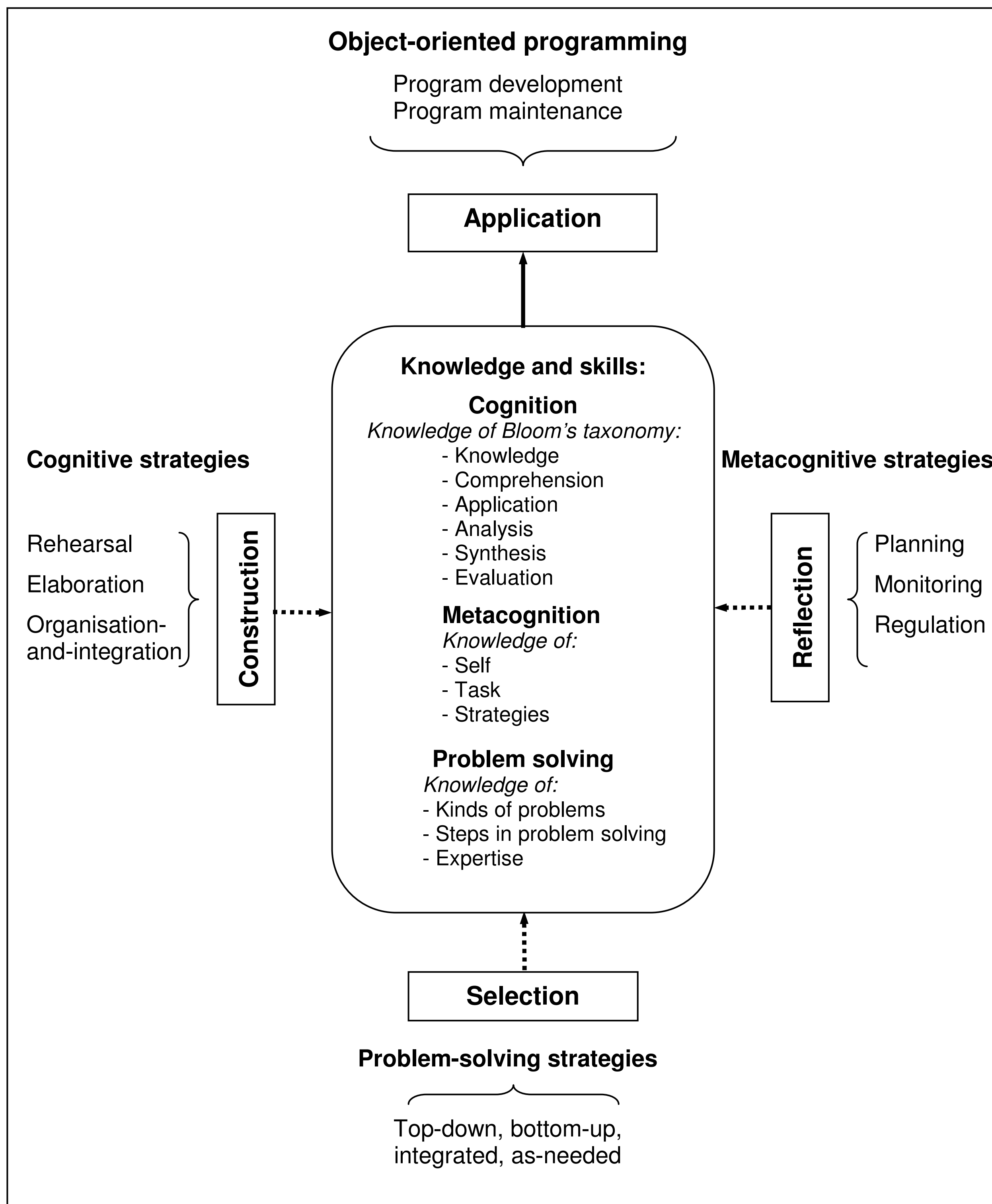


Figure 6.5: A learning repertoire of cognitive, metacognitive and problem-solving knowledge, skills and strategies in an OOP task

Dynamic interaction, indicated by the arrows in Fig. 6.5, occurs as the specific cognitive, metacognitive and problem-solving activities support the various core knowledge and skills and are applied to the milieu of OOP. As an example, successful object-oriented

programming requires the *application* of skills from Bloom's taxonomy, particularly analysis, synthesis and evaluation to determine whether a program is correct, and to rectify it if not.

Use of the strategies lying outside the core can enhance the knowledge and skills, and are used within the processes of *Construction*, *Reflection* and *Selection* in OOP, shown in blocks in Fig. 6.5 and elaborated below. The three dashed arrows on the left, the right and below the core indicate the dynamic and continuous use of cognitive, metacognitive and problem-solving strategies in the first three processes, while the bold arrow above the core relates to the *Application* (see block) of these activities in designing new programs and maintaining existing ones.

- *Construction*

  The use of cognitive strategies can enhance acquisition of the knowledge and skills in Bloom's taxonomy. Rehearsal supports the learning of facts about OOP (knowledge) and the grasping of programming content (comprehension). Elaboration can facilitate the use of previously learned material in new situations (application) and the decomposition of a problem into subproblems (analysis). The organisation-and-integration strategy can support programmers in combining objects, methods and attributes in a class (synthesis), and in testing the correct solution (evaluation). Object-oriented programmers should be actively involved in their tasks, using prior knowledge and applying a repertoire of knowledge and skills to help them recall information and organise it in memory during the process of constructing a program.

- *Reflection*

  Students should reflect on their cognitive processes during OOP by conducting deliberate planning, monitoring and regulation. They should question themselves, discover misconceptions, identify errors and continuously modify their programs in order to succeed. Such reflection places them in control of the programming task as they explicitly query the correctness of their code and reflect on their prior thinking to identify errors and correct flaws. Appropriate responses to feedback and the continuous improvement of code help to optimise the solution and to achieve the required outcomes.

- *Selection*

  The ability to make discerning selections helps students to choose a suitable problem-solving strategy for a given problem. They may select and apply one or more problem-solving strategies during program comprehension to help them reach specific goals.

For example, effective use of a top-down strategy demonstrates that a student has holistically conceptualised the entire program involving multiple classes, instances and methods.

- *Application*

  Finally and, in consolidation, the construction, reflection and selection of knowledge, skills and strategies have to be applied in OOP tasks to develop new programs and maintain existing ones. Learning to program is an active process of knowledge construction, reflection, and selection of appropriate activities to ensure successful programming. It is not the intention that every strategy should be applied in every situation. Different forms of knowledge, skills and strategies are relevant to different content and contexts. Relevant, customised subsets of the repertoire can be used as required.

Learning OOP requires a balanced approach of all the different activities involved. This implies, for example, that the application of Bloom's skills alone, without explicit reflection, or the application of strategies without applying analysis, synthesis and evaluation skills will not support successful completion of a new program. In such cases, students must explicitly query the correctness of their problem-solving and programming processes, and reflect on their prior thinking to identify errors and to correct flaws.

6.4 Application of this study to teaching and learning

This section practically applies the findings of this study by addressing Subquestion 3.2 – *What contribution can be made to the practices of teaching and learning OOP by applying the knowledge, skills and strategies used by successful programmers?*

The section briefly discusses areas where participants may improve on their thinking processes during programming. It also refers to teaching practices.

An in-depth analysis and plan on how to proceed are required in order to succeed, as did P32 and some others (Appendix G). Students should strengthen their abilities in the application of higher cognitive skills and they should apply such skills and strategies frequently. They should evaluate programming statements and segments during the programming process and at the end. This sometimes requires realignment of thoughts and

the application of additional skills and strategies to solve the problem and to determine whether the goal was achieved.

Where programmers have adequate knowledge combined with metacognition, they should be able to identify appropriate responses to correct the errors. Students should reflect on their programming, particularly on the semantics of their statements. They should pause periodically and check for errors to improve the accuracy of their reasoning and their programs. To foster this, students should be supported with various strategies that can help them to take corrective action in their programs and assess the output. Furthermore, students need explicit teaching on how to apply metacognitive strategies and the habits of reflection, which are essential in programming.

These activities should be addressed during the teaching and learning processes. The types of activities should be explained, demonstrating how they can be applied in a task (Table 6.3). In this regard, De Raadt et al. (2006:2) emphasises the need for explicit teaching of problem-solving strategies in programming. Students must learn how to investigate programming problems at a deeper level. A detailed test plan helps to ensure program correctness (Table 3.4). Practical skills are required for the processes of program testing, program debugging and the comprehension of error messages (§3.5).

Good teaching practices complement instruction on OOP content and constructs by also imparting information about valuable knowledge, skills and strategies (Fig. 6.5). However, these practices should be implemented gradually to prevent cognitive overload of students. The cognitive load of a task is related to the interactions between various elements within working memory (§3.3.1.1). However, cognitive load can be managed in a systematic way. The educator or facilitator should teach a few supportive activities at a time, provide various programming examples, give regular feedback and create a reflective environment in which students should learn the required qualities to succeed.

Examples of facilitator practices are given in Tables 6.2 – 6.4. These practices can be used to teach explicit knowledge, skills and strategies, which should contribute to deeper understanding and support students in overcoming programming difficulties in the complex multi-facetted domain of OOP.

Table 6.2: Facilitator practices in teaching cognitive knowledge, skills and strategies

Knowledge, skills and strategies	Facilitator practices
Rehearsal strategy, knowledge and comprehension	- Explain and clarify new concepts in detail - Focus students' attention when explaining concepts (§4.3.1) - Demonstrate practical ways of interpreting a textual programming problem (Table 3.2, §3.5.2) - Find evidence of students' knowledge of the programming language (Table 5.1) - Provide guidelines to support the programming process (Table 2.4) - Advise students to underline and select the main concepts in a programming problem (§4.3.3) - Discuss inherent requirements of the particular program e.g., what is a leap year? (Table 5.20, Theme 1)
Elaboration strategy, application and analysis	- Select items of content and ask directed questions that elaborate on previous knowledge (§4.3.2) - Demonstrate the use of memory diagrams to explain, for example, objects or arrays (§3.3.3) - Instruct students to study sections in a textbook to elaborate on specific content (Theme 5, §5.3.7) - Emphasise generative note-taking and integration of information (§4.3.2) - Guide students in working step-by-step during program analysis and in asking questions about the purpose, parameters, variables and return values (P32, Appendix G, Theme 1, Table 5.7) - Explain CRC cards for the identification of classes and their relationships (§3.2.4.3) - Illustrate the use of a semantic network for displaying different relationships e.g., an *is-a* relationship between concepts (§3.3.3)
Organisation-and-integration strategy, synthesis and evaluation	- Organise a class into constructor(s), mutators and accessors (Questionnaire, Question 22, Appendix E, Table 5.26) - Explain different types of programming problems, using several examples (Questionnaire, Question 38, Table 5.26) - Demonstrate how to integrate various programming statements in a new programming problem (Questionnaire, Question 27, Table 5.26) - Guide learners in setting their own programming questions and designing their solutions (§3.3.2) - Demonstrate and explain the use of specific statements in a program (Table 5.26, Question 16) - Evaluate and verify two programs that solve the same problem and explain how to select the best one (Table 5.26, Question 38) - Instruct students on how to generate appropriate test data for the program (Table 3.2, Table 5.1)

Table 6.3: Facilitator practices in teaching metacognitive knowledge, skills and strategies

Knowledge, skills and strategies	Facilitator practices
Planning	- Direct students' thinking in terms of goal setting (§3.4.3) - Deliberately plan a programming solution (§4.4.1) - Simulate a planning strategy for a specific programming problem (§4.4.1) - Motivate students to design planning schemes for their problem solving and programming
Monitoring	- Suggest that students use a journal or reflective diary to record the difficulties they experience (§3.4.3) - Motivate students to use help-seeking and self-questioning (§4.4.2) - Instruct students to monitor their own strategic use (§4.4.2) - Teach various debugging techniques and encourage students to apply specific ones, such as Watch and Trace (§4.4.3) - Discuss ways of determining how programming statements would behave and predict how a change in programming code would result in a change in program behaviour - Advise the use of trace tables to track behaviour of a well-functioning program
Regulation	- Provide meaningful feedback to students when they experience difficulties in their programming (§4.4.3) - Instruct students to re-read and to go back to the programming problem to ensure accuracy of their programs (§4.4.3) - Teach students how to implement corrective measures and how to react to feedback (§4.4.3) - Demonstrate how to interpret error messages and how to make the required changes to a program - Show students how to make predictions about the correctness of their programs (§4.4.3) - Motivate students to use the given rubrics to determine the correctness of their programs - Advise students to utilise metacognitive knowledge, skills and strategies from previous experiences and transfer these successfully into new programming situations (§4.4.4)
Reflection	- Guide students in developing a reflective approach towards the programming solution (§4.4.4) - Motivate students to ask questions, to self-question, to identify possible misconceptions, and to continuously modify their program in the process of achieving an optimal solution (§4.4.4) - Encourage students to explicitly investigate the correctness of their code and to reflect on their prior thinking in order to identify errors - Direct students in checking the accuracy of their judgement, in refining their personal insight, and in correcting programming errors (§4.4.4) - Provide regular opportunities for students to explicitly reflect on their own programming task

Table 6.4: Facilitator practices in teaching problem-solving knowledge, skills and strategies

Knowledge, skills and strategies	Facilitator practices
Problem-solving knowledge, skills and strategies	- Categorise problems into different types, where each type requires specific knowledge, skills and strategies (§3.5.1) - Explain and highlight the purpose of a program or program segments (§3.5.4) - Explain example programs and the reasoning that underlies the solutions of these (§3.5.4) - Require students to download various examples of similar problems and to explain the programming code (§3.5.4) - Require students to study programming examples from the *Help file* and to explain these examples to the group (§3.5.4) - Discuss trace tables in one or more classes and demonstrate how to use them to determine whether the problem was successfully solved - Teach how to use custom-developed test data (§3.5.4, Table 3.4) - Explicitly introduce collaborative learning, e.g. pair programming, where students can support each other in the programming process (§3.5.4) - Provide exposure to various types of programming problems e.g., structured and unstructured (open) problems - Teach specific problem-solving skills (§3.5.1) - Provide an incomplete program and ask students to complete specific sections e.g., methods (Questionnaire, Question 11, Table 5.28)
Expertise	- Teach students to answer open-ended questions (§4.5) - Instruct students to test program segments individually as well as to test the program as a whole (Table 5.7, §5.2.3.1) - Provide instruction in detailed problem representation skills in order to solve problems more efficiently (Table 3.5) - Explain how to understand multiple interpretations of the same problem (Table 3.5) - Teach a diagnostic approach to correcting flaws and errors in programming (Table 5.1)

This section contributes to the practice of teaching by listing examples of techniques that OOP educators can integrate into their instruction. The knowledge, skills and strategies in the preceding tables can be explicitly taught to support students in successful learning of OOP. The use of these approaches can lead to positive differences in the achievement and success of students. However, such an approach takes time. It requires more class time and possibly workshop sessions.

6.5 Recommendations and future research directions

To be successful in OOP, programmers require explicit learning both of programming content and higher-order mental activities. The findings of this study, which distinguish between successful and unsuccessful programmers, indicate the need for a framework to support novice programmers. This should address programming subject matter as well as cognitive, metacognitive and problem-solving knowledge, skills and strategies. Fostering awareness and application of the latter among learners sets a particular challenge to educators (lecturers) to identify creative and effective means of doing so.

Although this framework focuses mainly on OOP, it can also be applied to support students in other programming paradigms, such as procedural programming. However, due to the particular complexities of OOP, the framework focuses specifically on a holistic view where different kinds of decisions are required in programming one or more classes.

Future work will concentrate on the role of the lecturer or facilitator in conducting explicit teaching of the required knowledge, skills and strategies and supporting students by creating an educational environment in which the learning repertoire can be effectively applied. This may also require a search for appropriate strategic tools to facilitate the practices. The development of assessment criteria to test the effective application of the activities of the learning repertoire in an OOP task should further support the students.

6.6 Chapter conclusion

The purpose of this chapter was to discuss the findings of this study with the specific aim of answering the research question. Throughout the chapters, meticulous attention was paid to ethical aspects, thus strenghtening the integrity of the data obtained and also protecting

confidentiality of participants (Appendix A, Appendix B). Both qualitative and quantitative research was used (§2.2) and the findings are applied to combine the interpretivist approach with statistically significant effects for further clarification (§5.2, §5.3, §5.4). The grounded theory method was used as an analytic strategy to collect rich data from multiple sources, to define the properties of the categories and identify their relevant contexts. Grounded theory is generated inductively from the analysis of data as concepts are formulated into a logical systematic and explanatory scheme (§6.3). The positivist paradigm was applied to add another facet of analysis by 'measuring' data to ensure reliability and validity (Table 5.2, Tables 5.26 – 5.28). The statistics used in this study include the following: factor analysis, reliability testing, descriptive statistics and practical significance, all of which are outlined in Subsection 5.2.5 and applied in Sections 5.2 and 5.4.

Methodological triangulation was applied by describing the relationship between the quantitative and qualitative analysis methods by referring to the participants' cognitive, metacognitive, problem-solving and OOP activities. This was done by comparing statistical data (Table 5.2) with associated details from the *Atlas.ti* records. A questionnaire was administered to measure participants' perceptions.

Two major contributions of this study are the identification of specific types of knowledge, skills and strategies that are required during OOP, and the suggestion of guidelines and support in the learning of OOP that were generated as a result of this research. A detailed framework or learning repertoire was proposed, whereby various actions are integrated to support programmers in meaningfully constructing, and critically selecting, various knowledge, skills and strategies which support understanding, as well as explicit reflection on the activities involved in OOP. Implementation of the activities represented in Section 6.3 and the teaching practices outlined in Section 6.4, should help programmers in understanding, designing, coding and testing object-oriented programs.

References

Ala-Mutka, K. (2004). Problems in Learning and Teaching Programming – a literature study for developing visualizations in the Codewitz-Minerva project. Retrieved July, 2006, from http://www.cs.tut.fi/~edge/literature_study.pdf

Aleven, V., McLaren, B., Roll, I. & Koedinger, K. (2004). Toward tutoring help seeking. Applying cognitive modeling to meta-cognitive skills. Retrieved October, 2007, from www.pitt.edu/~bmclaren/HelpSeeking-ITS04.pdf

Alexander, C. (1996). The Origins of Pattern Theory. The Future of the Theory, and The Generation of a Living World. ACM Conference on Object-Oriented Programs, Systems, Languages and Applications (OOPSLA).

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. & Angel, S. (1977). A *Pattern Language. Towns, Buildings, Construction*. New York: Oxford University Press.

Anderson, L. W. & Krathwohl, D.R. (Ed.). (2001). *A Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York: Longman.

Baddeley, A. (2003). Working memory: looking back and looking forward. *Neuroscience*, 4:829-939.

Barrow, J., Miller, L., Malan, K. & Gelderblom, H. (2005). *Introducing Delphi Programming. Theory through practice* (4th ed.). Cape Town: Oxford University Press.

Beck, K. & Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. OOPSLA Conference Proceedings. *SIGPLAN Notices,* 24(10):1-6.

Bennedsen, J. & Caspersen, M.E. (2004). Teaching Object-Oriented Programming – Towards Teaching a Systematic Programming Process. Retrieved June, 2006, from www.cs.umu.se/~jubo/Meetings/ECOOP04/Submissions/BennedsenCaspersen.pdf

Bergin, S., Reilly, R. & Traynor, D. (2005). Examining the Role of Self-Regulated Learning on Introductory Programming Performance*. International Computing Education Research (ICER),* 2005:81-86.

Berntsen, K.E., Sampson, J. & Østerlie, T. (2004). Interpretive research methods in Computer Science. Retrieved March, 2006, from http://www.idi.ntnu.no/~thomasos/paper/interpretive.pdf

Bloom, B.S., Krathwohl, D.R. & Masia, B.B. (1973). *Taxonomy of Educational Objectives. Book 2: Affective Domain*. London: Longman Group.

Booch, G. (1991). *Object Oriented Design with applications*. New York: The Benjamin/Cummings Publishing Company.

Boy, G.A. (2005). Decision Making: A Cognitive Function Approach. Proceedings of the Seventh International NDM Conference. Amsterdam: The Nederlands:1-20.

Boychuk Duchscher, J.E. & Morgan, D. (2004). Grounded theory: reflections on the emergence vs. forcing debate. *Journal of Advanced Nursing,* 48(6):605-612.

Breed, E.A. (2006). 'n Analise van die reflektiewe vermoëns van effektiewe en oneffektiewe leerders in rekenaarprogrammering. MEd dissertation. Potchefstroom: North-West University.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies,* 18:543-554.

Cañas, J.J., Antolí, A., Fajardo, I. & Salmerón, L. (2005). Cognitive inflexibility and the development and use of strategies for solving complex dynamic problems: effects of different types of training. *Theoretical Issues in Ergonomics Science,* 6(1):95-108.

Carbone, A., Mitchell, I.J., Gunstone, R. & Hurst, A.J. (2002). Designing programming tasks to elicit self-management metacognitive behaviour. International Conference on Computers in Education*, (ICCE 2002),* Auckland, New Zealand.

Caspersen, M.E. & Kölling, M. (2006). A Novice's Process of Object-Oriented Programming. *OOPSLA.* Portland,1-9.

Charmaz, K. (2000). Grounded Theory. Objectivist and Constructivist Methods. In N.K. Denzin & Y.S. Lincoln (Ed.). *Handbook of Qualitative Research.* (2nd ed.). London: Sage Publications.

Chmura, G.A. (1998). What abilities are necessary for success in Computer Science? *SIGCSE Bulletin,* 30(4):55-58.

Cleenewerck, T. (2003). Lost in Object Space. Retrieved June, 2006, from http://prog.vub.ac.be/Publications/2003/vub-prog-tr-03-22.pdf

Cohen, J. (1988). *Statistical Power Analysis for the behavioural Sciences. (*2nd ed.). Hillsdale, NJ: Erlbaum.

Cohen, L., Manion, L. & Morrison, K. (2000). *Research methods in education* (5th ed.). London: RoutledgeFalmer.

Concise Oxford English Dictionary. (2004). Oxford: Oxford University Press.

Conklin, W.A. (2006). Bottom-Up meets Top-Down: A New paradigm for Software Engineering Instruction. Proceedings of the 10th Colloquium for Information Systems Security Education 2006:131-136.

Corritore, C.L. & Wiedenbeck, S. (2000). Direction and Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study. *IEEE Computer Society*,139-148.

Cupchik, G. (2001). Constructivist Realism: An Ontology that Encompasses Positivist and Constructivist Approaches to the Social Sciences. Retrieved October, 2007, from www.qualitative-research.net/fqs-texte/1-01/1-01cupchik-e.pdf

De Raadt, M., Watson, R. & Toleman, M. (2006). Chick Sexing and Novice Programmers: Explicit Instruction of Problem-Solving Strategies. 8th Australasian Computing Education Conference, 52:1-8.

De Villiers, M.R. (2005a). Three approaches as pillars for interpretive Information Systems research: development research, action research and grounded theory. In J. Bishop & D. Kourie. *Research for a Changing World: Proceedings of SAICSIT 2005*:142-151. ACM International Conference Proceedings Series.

De Villiers, M.R. (2005b). Interpretive research models for Informatics: action research, grounded theory, and the family of design- and development research. *Alternation*, 12(2):10-52.

Deek, F.P. (1999). A framework for an automated problem solving and program development environment. *Journal of Integrated Design and Process Science*, 3(3):1-13.

Deek, F.P., Turoff, M. & McHugh, J.A. (1999). A Common Model for Problem Solving and Program Development. *IEEE Transactions on Education*, 42(4):331-336.

Denzin, N.K. & Lincoln, Y.S. (Ed.). (2000). *Handbook of Qualitative Research.* (2nd ed.). London: Sage Publications.

Détienne, F. (1995). Design strategies and knowledge in object-oriented programming: Effects of experience. *Human-Computer Interaction,* 10:129-169.

Détienne, F. (2002). *Software Design – Cognitive Aspects*. London: Springer.

Détienne, F. (2003). Memory of past designs: distinctive roles in individual and collective design. *Journal of Cognitive Technology*, 1(8):16-24.

Du Plooy, G.M. (2001). *Communication Research: Techniques, Methods and Applications*. Lansdowne: Juta.

Dunsmore, A. (1998). Comprehension and Visualisation of Object-Oriented Code for Inspections. Retrieved June, 2006, from http://www.cis.strath.ac.uk/research/efocs/papers/EFoCS-33-98.pdf

Eckel, B. (2003). *Thinking in Java.* (3rd ed.). New Jersey: Prentice Hall.

Eckerdal, A. & Thuné, M. (2005). Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory. ITiCSE:89-93.

Edwards, S.H. (2004). Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education 2004:26-30.

Ellis, S.M. & Steyn, H.S. (2003). Practical significance (effect sizes) versus or in combination with statistical significance (p-values). *Management Dynamics*, 12(4):51-53.

Ertmer, P.A. & Newby, T.J. (1996). The expert learner: strategic, self-regulated, and reflective. *Instructional Science*, 24:1-24.

Fekete, A., Kay, J., Kingston, J. & Wimalaratne, K. (2000). Supporting reflection in introductory Computer Science. *SIGCSE,* 144-148.

Field, A.P. (2005). *Discovering Statistics using SPSS*. (2nd ed.). London: Sage Publications.

Filcher, C. & Miller, G. (2000). Learning strategies for distance education students. *Journal of Argricultural Education*, 41(1):60-68.

Flavell, J.H. (1979). Metacognition and cognitive monitoring. A new area of cognitive developmental inquiry. *American Psychologist*, 34(10):906-911.

Fowler, M. (2000). *UML Distilled. A Brief Guide to the Standard Object Modeling Language.* (2nd ed.). Boston: Addison-Wesley.

Gama, C. (2004). Metacognition in Interactive Learning Environments: The Reflection Assistant Model. Proceedings of the 7th International Conference on Intelligent Tutoring Systems. 2004:668-677.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley.

Garrido, J.M. (2003). *Object-Oriented Programming: From Problem Solving to Java*. Massachusetts: Charles River Media.

Garton, A. (2004). *Exploring Cognitive Development: The Child as Problem Solver.* Malden, M.A.: Blackwell Publishers.

Gilhooly, K.J. (2005). Working memory and strategies in reasoning. In M.J. Roberts & E.J. Newton (Eds.). *Methods of Thought: Individual Differences in Reasoning Strategies*. NY: Psychology Press.

Glaser, B.G. & Strauss, A.L. (1967). *The Discovery of Grounded Theory. Strategies for Qualitative Research*. London: Weidenfeld and Nicolson.

Glaser, R. (1999). Expert knowledge and processes of thinking. In R. McCormick, C. Paechter (Eds.). *Learning and Knowledge*. London: Paul Chapman.

Gliem, J.A. & Gliem, R.R. (2003). Calculating, Interpreting and Reporting Cronbach's Alpha Reliability Coefficient for Likert-Type Scales. Retrieved February, 2007, from www.alumni-osu.org/midwest/midwest%20papers/Gliem%20&%20Gliem--Done.pdf

Golafshani, N. (2003). Understanding reliability and validity in qualitative research. *The Qualitative Report*, 8(4):597-607.

Goulding, C. (1998). Grounded theory: the missing methodology on the interpretivist agenda. *Qualitative Market Research: An International Journal*, 1(1):50-57.

Govender, I. & Grayson, D. (2006). Learning to program and learning to teach programming: A closer look. ED-Media 2006 Proceedings:1687-1693.

Grant, N.S. (2003). A Study on Critical Thinking, Cognitive Learning Style, and Gender in Various Information Science Programming Classes. Proceedings of the 4th conference on Information Technology Curriculum. 2006:96-99. ACM International Conference Proceedings Series.

Gravill, J.I., Compeau, D.R. & Marcolin, B.L. (2002). Metacognition and IT: The influence of self-efficacy and self-awareness. Eighth Americas Conference on Information Systems:1055-1064.

Gu, P.Y. (2005). Learning Strategies: Prototypical Core and Dimensions of Variation. Retrieved August, 2006, from http://www.crie.org.nz/research_paper/Peter_Gu.pdf

Guba, E.G. & Lincoln, Y.S. (1989). *Fourth generation evaluation*. Newbury Park, California: Sage Publications.

Guion, L.A. (2002). Triangulation: Establishing the validity of qualitative studies. *Institute of Food and Argricultural Sciences,*1-3.

Hadar, I. & Leron, U. (2008). How intuitive is object-oriented design? Communications of the ACM. 51(5):41-46.

Haden, P. & Mann, S. (2003). The Trouble with Teaching Programming. Retrieved April, 2008, from www.naccq.ac.nz/conference04/proceedings_03/pdf/63.pdf

Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. *Integrating Technology into Computer Science Education,*171-174.

Hammouri, H.A.M. (2003). An investigation of undergraduates' transformational problem solving strategies: cognitive/metacognitive processes as predictors of holistic/analytic strategies. *Assessment & Evaluation in Higher Education*, 28(6):571-586.

Havenga, H.M., Mentz, E. & De Villiers, M.R. (2008). Knowledge, skills and strategies for successful object-oriented programming: a proposed learning repertoire. *South African Computer Journal* [in press].

Heines, J.M. & Schedlbauer, M.J. (2007). Teaching Object-Oriented Concepts Through GUI Programming. Retrieved August, 2008, from www.cs.umu.se/~jubo/Meetings/ECOOP07/Submissions/HeinesSchedlbauer.pdf

Henning, E., Van Rensburg, W. & Smit, B. (2004). *Finding your way in qualitative research*. Pretoria: Van Schaik Publishers.

Hertzog, C. & Robinson, A.E. (2005). Metacognition and Intelligence. In O. Wilhelm & R.W. Engle (Eds.). *Handbook of Understanding and Measuring Intelligence*. London: Sage Publications.

Holliday, M.A. & Luginbuhl, D. (2003). Using Memory Diagrams When Teaching a Java-Based CS1. Retrieved June, 2006, from http://sol.cs.wcu.edu/~holliday/papers/acmse03.pdf

Huan Keat, T. (2004). Semantic Networks. Retrieved June, 2006, from http://ww2.cs.fsu.edu/~huanktoh/project/Semantic%20Networks.pdf

Johnson, R.B. & Onwuegbuzie, A.J. (2004). Mixed method research: A research paradigm whose time has come. *Educational Researcher*, 33(7):14-26.

Johnson, R.A. & Wichen, D.W. (1992). *Applied Multivariance Statistical Analysis*. (3rd ed.). New Jersey: Prentice Hall.

Jonassen, D. (2003a). Using cognitive tools to represent problems. *Journal of Research on Technology in Education*, 35(3):362-381.

Jonassen, D. (2003b). What is problem-solving? Retrieved March, 2007, from http://media.wiley.com/product_data/excerpt/79/07879643/0787964379.pdf

Jonassen, D.H. (2004). *Learning to Solve Problems: An Instructional Design Guide*. San Francisco: Pfeiffer.

Jonides, J. & Nee, D.E. (2006). Brain mechanisms of proactive interference in working memory. *Neuroscience*, 139:181-193.

Kaiser, H.F. (1970). A second generation little jiffy. *Psychometrika*, 35(3):401-415.

Kaiser, H.F. (1974). An index of factorial simplicity. *Psychometrika*, 39(1):31-36.

Kapa, E. (2001). A metacognitive support during the process of problem solving in a computerized environment. *Educational Studies in Mathematics*, 47:317-336.

Kayashima, M., Inaba, A. & Mizoguchi, R. (2005). What Do You Mean by to Help Learning of Metacognition? Retrieved March, 2007, from www.ei.sanken.osaka-u.ac.jp/pub/documents/AIED05-Kaya.pdf

Keefe, K., Sheard, J. & Dick, M. (2006). Adopting XP Practices for Teaching Object Oriented Programming. Proceedings of the 8th Australasian Computing Education Conference. 52:10. ACM International Conference Proceeding Series.

Kirkwood, M. (2000). Infusing higher-order thinking and learning to learn into content instruction: a case study of secondary computing studies in Scotland. *Journal of Curriculum Studies*, 32(4):509-535.

Klein, H.K. & Myers, M.D. (1999). A set of principles for conducting and evaluating interpretive field studies in Information Systems. *MIS Quarterly*, 23(1):67-94.

Kline, P. (1999). *The handbook of psychological testing.* (2nd ed.). London: Routledge.

Kölling, M. (1999). The problem of teaching object-oriented programming. Part 1: Languages. *Journal of Object-Oriented Programming*, 11(8):8-15.

Koriat, A. (2002). Metacognition research: an interim report. In T.J. Perfect & B.L. Schwartz (Eds.). *Applied Metacognition*. UK: Cambridge University Press.

Leedy, P.D. & Ormrod, J.E. (2001). *Practical Research. Planning and Design.* (7th ed.). New Jersey: Merrill Prentice Hall.

Lemaire, P. & Fabre, L. (2005). Strategic aspects of human cognition: Implications for understanding reasoning. In M.J. Roberts & E.J. Newton (Eds.). *Methods of Thought. Individual Differences in Reasoning Strategies*. New York: Psychology Press.

Lewis, T.L., Pérez-Quiñones, M.A. & Rosson, M.B. (2004). *A Comprehensive Analysis of Object-Oriented Design: Towards A Measure of Assessing Design Ability.* ASEE/IEEE Frontiers in Education Conference. 16-21.

Lin, X. (2001). Designing metacognitive activities. *Educational Technology Research and Development*, 49(2):23-40.

Malik, D.S. & Nair P.S. (2003). *Java Programming: From Problem Analysis To Program Design.* United States: Thomson.

Matlin, M.W. (2002). *Cognition.* (5th ed.). Fort Worth: Harcourt College Publishers.

McDowell, C., Werner, L., Bullock, H. & Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. *SIGCSE Bulletin,* 38-42.

Mentz, E. (1998). 'n Ondersoek na Semantiese Nette in Probleemoplossing vir die Skoolvak Wiskunde. MSc dissertation. Potchefstroom: Potchefstroom University for CHE.

Morris, B.J. & Schunn, C.D. (2005). Rethinking logical reasoning skills from a strategy perspective. In M.J. Roberts & E.J. Newton. (Eds.). *Methods of Thought. Individual Differences in Reasoning Strategies.* New York: Psychology Press.

Muhr, T. (2004). User's Manual for ATLAS.ti 5.0. (2nd ed.). Retrieved May, 2007, from http://www.atlasti.com/downloads/atlman.pdf

Neath, I. & Surprenant, A.M. (2003). *Human Memory.* (2nd ed.). Australia: Thomson Wadsworth.

Neubauer, B.J. & Strong, D.D. (2002). The object-oriented paradigm: More natural or less familiar? *Journal of Computing Sciences in Colleges*, 18(1):280-289.

Neuman, W.L. (2002). *Social Research Methods. Qualitative and Quantitative Approaches.* (4th ed.). Boston: Allyn and Bacon.

Northrop, L.M. (1992). Finding an Educational Perspective for Object-Oriented Development. Proceedings of OOPSLA 1992:245-249.

Nurvitadhi, E., Leung, W. & Cook, C. (2003). Do class comments aid Java program understanding? 33rd IEEE Frontiers in Education Conference. 2003:13-17.

Oliver D., Dobele, T. Greber, M. & Roberts, T. (2004). This Course Has A Bloom Rating Of 3.9. In R. Lister & A. Young (Eds.). *Conferences in Research and Practice in Information Technology.* 6th Australasian Computing Education Conference. (ACE2004). 30:227-231.

Or-Bach, R. & Lavy, I. (2004). Cognitive activities of abstraction in object orientation: An empirical study. *The SIGCSE Bulletin*, 36(2):82-86.

Ormrod, J.E. (2003). *Educational Psychology. Developing Learners*. (4th ed.). Ohio: Merril Prentice Hall.

Pacione, M.J. (2004). *Evaluating a Model of Software Visualisation for Software Comprehension.* University of Strathclyde: Department of Computer and Information Sciences.

Panaoura, A. & Philippou, G. (2005). The measurement of young pupils' metacognitive ability in mathematics: The case of self-representation and self-evaluation. Retrieved May, 2007, from http://cerme4.crm.es/Papers%20definitius/2/panaoura.philippou.pdf

Pennington, N., Lee, A.Y. & Rehder, B. (1995). Cognitive Activities and Levels of Abstraction in Procedural and Object-Oriented Design. *Human-Computer Interaction*, 10:171-226.

Polya, G. (1981). *Mathematical Discovery. On Understanding, Learning, and Teaching Problem Solving.* New York: John Wiley.

Pozzebon, M. (2004). Conducting and Evaluating Critical Interpretive Research: Examining Criteria as a Key Component in Building a Research Tradition. In B. Kaplan, D.P. Truex, D. Wastell, A.T. Wood-Harper & J.I. DeGross (Eds.). *Information Systems Research: Relevant Theory and Informed Practice.* Norwell, MA: Kluwer Academic Publishers.

Prasad, A. & Prasad P. (2002). The coming of age of interpretive organizational research. *Organizational Research Methods*, 1(1):4-11.

Pressing, J. (1999). Cognitive complexity and the structure of musical patterns. Proceedings of the 4th Conference of the Australasian Cognitive Science Society. Australia: University of Newcastle.

Purao, S., Bush, A. & Rossi, M. (2001). Problem and Design Spaces during Object-Oriented Design: An Exploratory Study. Proceedings of the 34th Hawaii International Conference on System Sciences.

Ragonis, N. & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3):203-221.

Rajlich, V. & Wilde, N. (2002). The Role of Concepts in Program Comprehension. Proceedings of IWPC. 2002:271-278. CA: IEEE Computer Society Press.

Reed, A. (2003). Object-Oriented Programming and Objectivist Epistemology: Parallels and Implications. *The Journal of Ayn Rand Studies*, 4(2):251-284.

Roberts, M.J. & Newton, E.J. (2005). Strategy usage in a simple reasoning task: overview and implications. In M.J. Roberts & E.J. Newton (Eds.). *Methods of thought. Individual Differences in Reasoning Strategies.* New York: Psychology Press.

Robinson-Riegler, G. & Robinson-Riegler, B. (2004). *Cognitive Psychology. Applying the Science of the Mind*. Boston: Pearson.

Rosson, M.B. & Alpert, S.R. (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction*, 5:345-379.

Satzinger, J.W., Jackson, R.B. & Burd, S.D. (2004*). Systems Analysis and Design in a Changing World.* (3rd ed.). Boston: Thomson Course Technology.

Satzinger, J.W. & Ørvik, T.U. (2001). *The Object-Oriented Approach. Concepts*, *System Development, and Modeling with UML*. Boston: Course Technology.

Schach, S.R. (2005). *Object-Oriented and Classical Software Engineering. (*6th ed.). Boston: McGraw-Hill.

Schneider, W. & Lockl, K. (2002). The development of metacognitive knowledge in children and adolescents. In T.J. Perfect & B.L. Schwartz (Eds.). *Applied Metacognition*. UK: Cambridge University Press.

Schön, D. (1983). *The Reflecting Practitioner: How Professionals Think in Action*. London: Temple Smith.

Schulte, C. & Niere, J. (2002). Thinking in Object Structures: Teaching Modelling in Secondary Schools. 6th Workshop on Pedagogies and Tools for Learning Object Oriented Concepts. ECOOP.

Schunk, D. (2000). *Learning Theories. An Educational Perspective.* (3rd ed.). New Jersey: Prentice-Hall.

Schwartz, B.L. & Perfect, T.J. (2002). Introduction: Toward an applied metacognition. In T.J. Perfect & B.L. Schwartz (Eds.). *Applied Metacognition*. UK: Cambridge University Press.

Sebesta, R.W. (2004). *Concepts of Programming Languages.* (6th ed.). Boston: Pearson Addison Wesley.

Shaft, T.M. (1995). Helping programmers understand computer programs: The use of metacognition. *Data Base Advances*, 26(4):25-46.

Shalloway, A. & Trott, J. (2002). *Design Pattern Explained: A New Perspective on Object-Oriented Design.* Boston: Addison Wesley.

Shannon, C. (1999). *Developer's Guide to Delphi Troubleshooting*. Texas: Wordware Publishing.

Shimamura, A.P. (2000). Toward a cognitive neuroscience of metacognition. *Consciousness and Cognition*, 9:313-323.

Sicilia, M-A. (2006). Strategies for Teaching Object-Oriented Concepts with Java. *Computer Science Education*, 16(1):1-18.

Staats, W.J. & Blum, T. (1999). Enhancing an Object-Oriented Curriculum: Metacognitive Assessment and Training. ASEE/IEEE Frontiers in Education Conference 1999:13-19.

Stamouli, I. & Huggard, A. (2006). Object-oriented programming and program correctness: The student's perspectives. *International Computing Education Research,*109-118.

Sternberg, R.J. (2006). *Cognitive Psychology.* (4th ed.). United Kingdom: Thomson Wadsworth.

Steyn, H.S. (jr) (2002). Practically significant relationships between two variables. *SA Journal of Industrial Psychology,* 28(3):10-15.

Storey, M.-A.D., Wong, K. & Müller, H.A. (1997). How Do Program Understanding Tools Affect How Programmers Understand Programs? Retrieved April, 2006, from www.cs.uvic.ca/~mstorey/papers/scp.pdf

Strauss, A. & Corbin, J. (1998). *Basics of Qualitative Research. Techniques and Procedures for Developing Grounded Theory*. London: Sage Publications.

Stroustrup, B. (1995). *Why C++ is not only an object-oriented programming language.* OOPSLA 1995:1-13. Addendum to the Proceedings.

Tan, J., Biswas, G. & Schwartz, D.L. (2006). Feedback for Metacognitive Support in Learning by Teaching Environments. Retrieved May, 2007, from www.teachableagents.org/papers/p828-cogsci.pdf

Tan, S.C., Turgeon, A.J. & Jonassen, D.H. (2001). Develop Critical Thinking in Group Problem Solving through Computer-Supported Collaborative Argumentation: A Case Study. *Journal of National Resources and Life Science Education*, 30:97-103.

Thorsen, S. (2007). Why are leap years used? Time and Date AS. Time and Date.com. Retrieved October, 2007, from http://www.timeanddate.com/date/leapyear.html

Thramboulidis, K.C. (2003). A Constructivism-Based Approach to Teach Object-Oriented Programming. *Journal of Informatics Education and Research*, 5(1):1-14.

Traynor, D. & Gibson, P. (2004). Towards the development of a cognitive model of programming, a software engineering approach. Retrieved June, 2006, from http://www.cs.nuim.ie/~pgibson/Research/Publications/E-Copies/PPIG04.pdf

Vögele, E. & Wild, K. (2003). Task Understanding as Regulating Entity of Cognitive Learning Strategy use. Retrieved August, 2006, from http://www.ezw.uni-freiburg.de/uploads/media/03_earli_voegele_wild.pdf

Von Mayrhauser, A. & Vans, A.M. (1997). Program Understanding Behavior during Debugging of Large Scale Software. Seventh Workshop on Empirical Studies of Programmers,157-179.

Weber, R. (2004). Editor's Comments. The rhetoric of positivism versus interpretivism: A personal view. *MIS Quarterly*, 28(1):iii-xii.

Weinstein, C.E. & Meyer, D.K. (1991). Cognitive learning strategies and college teaching. *The Directions for Teaching and Learning*, 45:15-26.

Weisfeld, M. (2004). *The Object-Oriented Thought Process.* (2nd ed.). Indiana: Developer's Library.

Weiss, M.A. (2000). *Data Structures and Problem Solving using C++.* Tokyo: Addison-Wesley.

White, G.L. & Sivitanides, M.P. (2002). A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics. *Journal of Information Systems Education,* 13(1):59-66.

Wigglesworth, J. & Lumby, P. (2000). *Java Programming. Advanced Topics*. United States: Course Technology.

Williams, L., Wiebe, E., Yang, K., Ferzli, M. & Miller, C. (2002). In support of pair programming in the introductory Computer Science course. *Computer Science Education*, 12(3):197-212.

Williamson, K. (2006). Research in constructivist frameworks using ethnographic techniques. *Library Trends*, 55(1):83-101.

Xu, S. & Rajlich, V. (2004). Cognitive process during program debugging. IEEE International Conference on Cognitive Informatics. 2004:176-182.

Xun, G.E. & Land, S.M. (2004). A conceptual framework for scaffolding ill-structured problem-solving processes using question prompts and peer interactions. *Educational Technology Research and Development*, 52(2):5-22.

Young, P. (1996). Program Comprehension. Retrieved April, 2006, from http://vrg.dur.ac.uk/misc/PeterYoung/pages/work/documents/lit-survey/prog-comp/index.html

Yousoof, M., Sapiyan, M. & Kamaluddin, K. (2006). Reducing cognitive load in learning computer programming. *Transactions on Engineering, Computing and Technology*, 12:259-262.

Zant, R.F. (2005). Problem Analysis and Program Design: Using Subsystems and Strategies. Retrieved June, 2006, from http://isedj.org/isecon/2001/39b/ISECON.2001.Zant.pdf

Zhang, X. (2005). Analysis techniques for Program Comprehension. Retrieved July, 2006, from www.cs.uoregon.edu/~xzhang/documents/AreaExam-long%20version/position.pdf

Appendix A: Consent form

I, ____ (First name and surname) and student number_________________________, state that I have not been put under any pressure to participate in this evaluation exercise, and have willingly participated in it.

I realise that the findings of the evaluation will be used for research purposes and that findings will be published.

I am assured that all my personal information will be handled with confidentiality.

Signed ____

Date ____

Appendix B: Ethical approval

This study carries the approval of the Dean of the Faculty of Education and the Head of the School for Computer Science to conduct this research with students as participants. It also meets with the approval of the Ethical Committee and Research Director of the tertiary education institution where this research was conducted. However, it was required that no explicit reference should be made to a subject group, school, and faculty of the university where the study has been done and that the information regarding the participants should be confidential.

Appendix C: Programming assignment

ENGLISH

1. Programming assignment

1.1 Create a *Date class* that includes calculations with dates. Assume that there is no package available to do these calculations in the programming language you use. Use the included test data as well as the required methods as shown in Table 1 on the following page.

The *new Date class* must do the following:

- Calculate leap years. You must read in a year and determine if it is a leap year.
  Also determine the number of days for each month
  January, March, May, July, August, October, December: 31 days
  February: 28 or 29 days – depends on whether it is a leap year or not (p 3)
  April, June, September, November: 30 days
- Calculate the difference between two dates. You must read in two dates and determine the difference between the first and second date.

1.2 Now write a *Test class* that uses the *Date class* and give the necessary output.

2. Write down your thoughts *during* the writing of the *Date class* on a piece of paper

2.1 Write down the question in your own words and understanding.

2.2 Write on a paper *point by point* all your *thoughts, plans and steps down* ***during*** the programming of the *Date class*. If this is hard, write down all the questions you will ask yourself during programming.

2.3 It is important to mention how you will start the programming – what is your first step.

2.4 Write down a complete bibliography – e.g. study guide, textbook, Internet address.

2.5 The programming must be your **own work**, and other people must not do the programming for you! The value is that your own programming style can then be determined.

Write down the time you have used to complete this task

3. Printout and output

Submit the following printouts – even if your program is not working:

3.1 *Date class*

3.2 *Test class*

3.3 Program output (if possible)

4. Paper with thoughts

Also submit the page with your detailed thoughts while programming the *Date class*! Include the following and decide on additional methods if necessary:

Date class
Include the following: Variables Constructor Input: today's date
Use the following methods (Java); procedures and functions (Delphi): setTodaysDate (format: yyyymmdd) getDay( ); getMonth( ); getYear( ) isLeapYear( ) – test for leap years dateDifference( ) – calculate the difference between two dates
Application or *Test class:* Instantiate an object Decide which method of input will be used (files/streams/ components etc.) Decide what exception handling is necessary if any dates are incorrect

Table 1

Calculate leap years as follows

Leap years:

Leap years are years that are divided by 4.

Century years are years (e.g. 2**000**) that are divided by 100 and 400, and these are leap years as well.

Test data

Leap years: 1904, 1936, 2004

Not leap years: 1900, 2003, 1899

Difference between 2 dates: (must work with any two dates except future dates)

Test data 1

First date: 10 April 2006

Second date: 28 August 2006

The difference between these dates:… days

Test data 2

First date: 12 September 1899

Second date: today's date

The difference between these dates: …days

5. Submission

E-mail all program files to my e-mail address. Also submit the page with your detailed thoughts when we will complete the questionnaire.

6. Time frame

Please email your task before or on *Tuesday 16 October 2006.* You must complete a questionnaire on *Tuesday 17 October* the 5th period.

THANK YOU

M Havenga

Appendix D: Codes in *Atlas.ti*

Code-Filter: All

HU: DATE_CLASS
File: [C:\Documents and Settings\Administrator\My Documents\Scientific Software\ATLASti\Te...\DATE_CLASS.hpr5]
Edited by:Super
Date/Time: 08/06/04 08:29:09 nm

---

delphi:approach:create unit:create main application with buttons
delphi:approximate:input:processing:output
delphi:assignment:ask questions
delphi:assignment:cannot apply problem
delphi:assignment:confused with procedures and functions
delphi:assignment:determine difference between two dates
delphi:assignment:determine leap years
delphi:assignment:difficult:not clear guidelines
delphi:assignment:don't know if it is the right answer
delphi:assignment:errors in program
delphi:assignment:insert procedures and functions
delphi:assignment:must learn how to interpret error messages
delphi:assignment:program not working:discouraged:struggle
delphi:assignment:read with precision:determine big picture
delphi:assignment:reread with attention
delphi:assignment:think about the new class
delphi:assignment:very difficult: problems
delphi:bibliography:Delphi textbook
delphi:bibliography:Internet
delphi:bibliography:study guide COMP 312
delphi:buttons:consider the screen layout
delphi:class new:declare in interface
delphi:class:function:be visible
delphi:class:function:returnValue
delphi:class:function:test:logical errors:correct
delphi:class:theory
delphi:error messages: insert
delphi:error messages: not displayed: don't know what is problem
delphi:function:framework design
delphi:input date:format yyyymmdd
delphi:knowledge: don't know date calculations: only year uses
delphi:look for an example:didn't find example
delphi:object attributes:initialize
delphi:OOP uses objects in the program
delphi:output:NumberOfDays only:integer
delphi:plan what to do:difficulty with programming
delphi:planning
delphi:planning:button event handlers program
delphi:planning:create new unit
delphi:planning:determine scope:private or public
delphi:planning:think about class structure
delphi:procedure:framework
delphi:procedure:initialize variable to zero
delphi:processing:daysDifferences:subtraction
delphi:processing:leapyear:divide4
delphi:processing:monthdays:case statement
delphi:programming: program is working
delphi:programming:copy value:convert to integer
delphi:programming:days per month determine
delphi:programming:declare in interface type and scope
delphi:programming:determine days:if statement

delphi:programming:determine variables required
delphi:programming:errors:not understand
delphi:programming:event handler:subtract dates
delphi:programming:implementation:procedures and functions
delphi:programming:implementation:program detail code
delphi:programming:implementation:remember:add Tform before procedure and function
delphi:programming:interface:procedure clear
delphi:programming:month:if statement
delphi:programming:program interface:structure of procedures and functions
delphi:programming:return result
delphi:purpose:determine days of month
delphi:purpose:input 2 dates:calculate difference
delphi:question read carefully
delphi:read more about classes and dates
delphi:reread assignment many times:think
delphi:start again:calculate dates:staying in same home
delphi:steps: example uses:difference between two dates
delphi:steps:class design
delphi:steps:class:calculate dates
delphi:steps:dates add and subtract
delphi:steps:think about problem:how to solve problem
delphi:strategy:ask questions
delphi:strategy:change programming code
delphi:strategy:complete all the detail program code of specific component before continuing with next
delphi:strategy:first design form
delphi:strategy:more organized approach:enhance effective programming
delphi:test:range test:month
delphi:testing program
delphi:time management
deplhi:programming:date calculations
java:approach:'black box' programming
java:approach:1)write test class 2) write date class
java:approach:add days of each dates from year 0
java:approach:application of trail-and-error:difference between dates
java:approach:ask many questions during programming process
java:approach:ask questions:write all methods:test:correct:test
java:approach:class,constructor,methods,testclass,test
java:approach:date calculations
java:approach:date class:empty methods
java:approach:determine input,interface,calculations,test input
java:approach:find new ways to solve a problem
java:approach:lecturer:motivation:passion
java:approach:OOP:methods necessary for functionality
java:approach:programming difficult:start with an example to explain
java:approach:reread assignment, write thinking down
java:approach:write all methods without input, output
java:approach:write program with pencil before continue
java:assignment: don't have problems to determine leap years
java:assignment: was a challenge
java:assignment:2 classes:date class and test class
java:assignment:ask questions
java:assignment:determine difference between 2 dates
java:assignment:determine leap years
java:assignment:determine requirements
java:assignment:difficult programming:test days of month
java:assignment:difficult to plan
java:assignment:difficult to write down thinking during programming
java:assignment:direction needed to program
java:assignment:don't know how to calculate difference between 2 dates
java:assignment:framework of Date and Test Class with headings, imports and methods
java:assignment:glad to program again:also receive award
java:assignment:how to calculate necessary methods?
java:assignment:inheritance necessary?

java:assignment:knowlege that you can program
java:assignment:many questions were asked: purpose?, parameters?, input_output?, problems? calculations and variables?
java:assignment:methods: which methods are necessary?
java:assignment:must understand basic principles in programming
java:assignment:overall picture:calculate dates
java:assignment:problems:determine days, difference between dates
java:assignment:reread e-mail
java:assignment:think again
java:assignment:type code without knowing why:unnecessary code
java:assignment:very confused:did programming a long time ago
java:assignment:what instance variables should be declared?
java:assignment:which methods are necessary in the class
java:assumption:date format:ddmmjjjj
java:assumption:methods write without code
java:bibliography:C# textbook
java:bibliography:Internet websites
java:bibliography:Java study guide
java:bibliography:Java text book
java:bibliography:previous Java assignments
java:bibliography:previous Java code
java:calculations:use integer only:String not necessary
java:class design:determine general and specific cases
java:class:create
java:class:framework
java:class:scope
java:class:theory
java:constructor
java:constructor:default
java:constructor:initialize
java:convert integer number to date
java:date:separate day, month, year
java:dates:compare:calculate days
java:dates:convert dates to days:subtract:convert
java:dates:two dates needed
java:day:add:return value:convert to date
java:determine day of month
java:error message: cannot diagnose the problem
java:error:calcalate days for leap year incorrectly:must add 1 day
java:error:calculate days, months and years wrongly
java:error:change return value
java:error:didn't program error management and handling
java:error:differDates:must use 3 loops
java:error:difficult to compile java on computer
java:error:don't know how to change error
java:error:error found:change variable names
java:error:error handling not very good
java:error:error handling write
java:error:error in calculation:difference between dates
java:error:exception handling:reread about this
java:error:exceptions: handle:ArrayOutOfBounds exception
java:error:forgot main method:public static main
java:error:generate:wrong dates
java:error:make necessary changes
java:error:not 100% working
java:error:should use array
java:error:syntax errors:correct them
java:exception handling:complex
java:fact
java:input
java:input:2 dates as parameters
java:input:date format: jjjjmmdd
java:input:date:determine year,month:output:leap year,days of month
java:input:today's date:from test class

java:instance variables determine
java:Internet:search: for 'substring'
java:java:AND operator:search website
java:leapYear:divided by 4
java:leapYear:website available
java:method:accessor:return values
java:method:calculate days of month
java:method:calculation
java:method:convert to integer:parseInt()
java:method:don't know how to copy part of string
java:method:save as string and integer format
java:method:substring date:convert to integer
java:method:testDate:return boolean value
java:method:validity check:convert to days
java:methods: if dates equal:return difference 0
java:methods:call get methods
java:methods:conversion:to integer
java:methods:conversion:toString:return value
java:methods:conversion:years
java:methods:dates differ:return boolean:biggest
java:methods:declare
java:methods:determine difference between months
java:methods:determine difference between years
java:methods:determine year difference
java:methods:difference between dates
java:methods:get and set methods
java:methods:leap years calculate
java:methods:mutator accessor
java:methods:mutator:0/1/more values
java:methods:program all get methods
java:methods:return values
java:methods:scope:private
java:methods:set methods use
java:methods:start:empty constructor
java:methods:start:empty methods
java:methods:test
java:methods:toString
java:methods:void:no return value
java:methods:year difference:determine leap years:add difference
java:object:attributes assign
java:object:create:instantiation of Date class
java:object:object call
java:oop:description of oop
java:oop:theory
java:parameter:receive
java:parameters:day,month, year receive
java:polymorphism:same constructor receives two dates
java:program:program execute correctly : 100%
java:programming:array:present days of month
java:programming:arrays:has problems with arrays
java:programming:comments in brackets
java:programming:compile, change errors
java:programming:exception handling
java:programming:if statement:dates test
java:programming:nested if statements
java:programming:output change to correct format
java:programming:switch:to set dates
java:programming:test
java:purpose:calculations with dates
java:purpose:determine difference between 2 dates
java:reflection:should send the date to the constructor
java:reread
java:reread:exception handling
java:reread:how to write class and testclass

java:reward:motivate student
java:search:internet:difference between 2 dates
java:start: write test class
java:start:analyse the problem:determine requirements
java:test Difference between dates: output:difference
java:test leap years
java:test program:update
java:test: days:boolean
java:test:dates valid
java:test:days of month
java:test:difference between dates
java:test:leap year: update test program
java:test:leapYear:boolean
java:test:leapYear:century:uses flags to test
java:test:LeapYear:divide by 4 or 100 and 400
java:test:leapyears:century:another formula uses
java:test:LeapYears:uses mod
java:test:months:1-12
java:test:test days
java:test:test one date
java:test:test simple program:test structure
java:test:years
java:testclass:call Date class 2 times:send 2 values
java:testclass:test output
java:testclass:write the test class
java:textbook: Java: reread
java:textbook:reread constructors
java:textbook:reread how to create a class
java:textbook:reread JOptionPane
java:time management: 3 to 5 hours
java:time management:1h30min
java:time management:2 hours
java:time management:4 hours
java:time management:4.5 hours
java:time management:5 hours
java:variables:assign
java:variables:boolean
java:variables:day, month, year
java:variables:declare
java:variables:determine
java:variables:global variables uses
java:variables:instance variables:day, month, year
java:variables:scope
java:variables:static
java:variables:type:integer,string
java:variables:use global variable
java:variables:variable assign:number of days in each month

Appendix E: The questionnaire and mark sheet

Personal information

Official use only

1. Student number: ________________

2. Age: ____

3. Gender: __________

4. Degree you are registered for this year (mark applicable box with an **X)**:

BSc	BEd	BCom	Other

If other, please specify____

5. What is your highest qualification in Computer Science / Information Technology?

1st year	2nd year	3rd year

6. Did you take Computer Studies at school?

Yes	No

7. If yes, was it on HG (Higher grade) or SG (Standard grade)?

HG	SG

8. Did you have any prior programming experience before studying at the university? Please give details of any programming courses you have passed:

____

____

9. In which programming language/s do you currently program? You may mark more than one option:

Java	1
Delphi	2
C++	3
C#	4
Visual Basic	5
Other	6

If other, please specify____

All information will be treated confidentially

QUESTIONNAIRE

Knowledge, Skills and Strategies in Object-Oriented Programming Questionnaire

M Havenga

2006

INSTRUCTIONS

Read every statement and mark the statement that **describes you best**.

Scale

1. **Never:** the statement would never be true of you.
2. **Seldom:** the statement would seldom be true of you.
3. **Often:** the statement would often be true of you.
4. **Always:** the statement would be true of you all the time.

Cross out the number that describes you best.

Example

~~1~~	2	3	4

Try to answer according to *how **well** the statement describes you*, not how you think the answer should be. There are no right or wrong answers to these statements! Mark only **one** answer per statement.

Please be honest! Your impressions and opinions are really important to the success of this research.

	STATEMENT	Never 1	Seldom 2	Often 3	Always 4
1.	The first time I learn a new programming concept, I make sure that I understand it.	1	2	3	4
2.	When I write a new program, I know which programming statements to apply.	1	2	3	4
3.	I find it difficult to analyse a given programming problem.	1	2	3	4
4.	I can create test data for a new program.	1	2	3	4
5.	Before programming, I **consider the whole solution before** going into the **details** of the solution.	1	2	3	4
6.	I can easily design a solution for a new programming problem.	1	2	3	4
7.	Even when the program is difficult to write, I go back and modify it until the problem is solved successfully.	1	2	3	4
8.	It is hard for me to break down a problem into smaller parts.	1	2	3	4
9.	I plan the solution of my program to achieve the goal.	1	2	3	4
10.	I can complete the programming code of a given incomplete program.	1	2	3	4
11.	I can alter specific parts of a programming solution when the requirements have changed.	1	2	3	4
12.	I can predict what the output of a program will be.	1	2	3	4
13.	When I program, I start with the ***declaration*** of all the **methods of one class**, before proceeding with the ***detailed*** programming of each method.	1	2	3	4
14.	I find it difficult to interpret a programming question.	1	2	3	4
15.	When I program, I stop once in a while and go over what I have already programmed.	1	2	3	4

STATEMENT	Never 1	Seldom 2	Often 3	Always 4
16. I can explain the use of specific programming statements in my solution.	**1**	**2**	**3**	**4**
17. I only make changes to a specific method when required due to errors in my program.	**1**	**2**	**3**	**4**
18. I write down plans to direct my thinking in programming.	**1**	**2**	**3**	**4**
19. When I program, I start with the ***declaration and details*** of the first class and methods before proceeding with the next class.	**1**	**2**	**3**	**4**
20. I think about what I should do first to solve a new programming problem.	**1**	**2**	**3**	**4**
21. I try to program a possible solution and hope that it will work.	**1**	**2**	**3**	**4**
22. I can easily classify different types of methods, as a constructor, destructor, mutator and accessor.	**1**	**2**	**3**	**4**
23. When I program, I start with **all the details of a method**, before proceeding with the next method.	**1**	**2**	**3**	**4**
24. When I program, I try to remember what the lecturer had said or what I read in the textbook that is relevant to the problem in hand.	**1**	**2**	**3**	**4**
25. I take the programming statements that have errors in them and adjust them until I have solved the problem successfully.	**1**	**2**	**3**	**4**
26. During modification of a given program, I expand a specific section only.	**1**	**2**	**3**	**4**
27. I can combine the necessary programming statements successfully in a new program.	**1**	**2**	**3**	**4**
28. I reread the description of a difficult problem to make sure that I understood it correctly and that it is correctly programmed.	**1**	**2**	**3**	**4**
29. During programming **I start with the details,** such as variables of a specific class before programming the details of the next class.	**1**	**2**	**3**	**4**
30. I find it difficult to evaluate my programming solution to determine if I have solved the problem correctly.	**1**	**2**	**3**	**4**

STATEMENT	Never 1	Seldom 2	Often 3	Always 4
31. When I read a programming question, I can easily distinguish between the necessary and unnecessary parts of the description.	**1**	**2**	**3**	**4**
32. When I program, I start with the ***declaration of all the classes*** before proceeding with the ***details*** of ***each class.***	**1**	**2**	**3**	**4**
33. In the preparing for a test, I make sure that I can define or describe a new programming concept.	**1**	**2**	**3**	**4**
34. I do not know where to start with the programming of a new problem.	**1**	**2**	**3**	**4**
35. When I program, I start with the declaration of a framework for a certain class and proceed with ***all the methods of the same class*** before starting with the framework and details of the next class.	**1**	**2**	**3**	**4**
36. I find it difficult to know what the program, as required by the problem description, is supposed to do.	**1**	**2**	**3**	**4**
37. I ask myself questions to make sure that I understand a difficult programming statement when I use it in a program.	**1**	**2**	**3**	**4**
38. If two different solutions of the same problem are given to me, I can select the best solution.	**1**	**2**	**3**	**4**
39. When I program, I **complete the programming code of one class**, before proceeding with the programming code of the next class.	**1**	**2**	**3**	**4**
40. When I program, I trace the program's execution with a trace table.	**1**	**2**	**3**	**4**
41. During programming I use any possible solution that would work, but not necessarily the very best solution.	**1**	**2**	**3**	**4**
42. I program the **whole class with its details** and complete it before proceeding with the next class.	**1**	**2**	**3**	**4**

43. Would you describe yourself as **successful** or **unsuccessful** in computer programming? Please motivate.

____

____

____

____

44. Do you make use of any special strategies, plans or useful 'tricks' when you write a computer program? If so, please give all the details:

____

____

____

____

45. During programming of a ***new class***, I use the following <u>sequence</u> of general steps to solve a problem (please specify your sequence in detail):

____

____

____

____

46. Do you make use of any supportive memory representation techniques **during** your programming task? If you do, give a **diagram** or a **description** of all the details please.

Thank you for your participation!

Mark sheet of the questionnaire

1. **Scale values** 1 = 1, 2 = 2, 3 = 3, 4 = 4
2. **Grouping of items**

COGNITIVE SKILLS		**Action verbs / words**
Knowledge	24,33,**36**	Remember, define, know
Comprehension	1,12,**14**	Understand, predict, interpret
Application	2,10,22	Apply, complete, classify
Analysis	**3**,**8**,31	Analyse, break down, distinguish
Synthesis	4,6,27	Create, design, combine
Evaluation	16,**30**,38	Justify, evaluate, compare
METACOGNITIVE STRATEGIES		
Planning	9,18,20	Estimate performance, plan to direct, what should do first
Monitoring	15,37,40	Stop once and go over, ask myself questions, trace program execution
Regulation	7,25,28	Go back and modify, adjust wrong statements, reread problem
PROBLEM-SOLVING STRATEGIES		
Bottom-up	23,29,39	Details of each method, details e.g. variables, complete one class
Top-down	5,13,32	Overview whole solution, declare the methods of class, declare class
Integrated	19,35,42	Declarations and details of class, all methods of same class, whole class with details
As-needed	11,17,26	Alter specific part, changes specific method when needed, expand specific section
Trial-and-error	21,**34**,**41**	Hope it will work, confused and don't know how to start, find any possible solution not the best

Bold numbers indicate negative statements

Appendix F: Data of Participant 31, an unsuccessful programmer

Programming examples

P31 submitted two separate application programs and both are included.

DELPHI-program: Program 5.1 (first attempt)

```
unit Datum_u;  // [saved as Datum_u]

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;

type
  TFrmDatums = class(TForm)
    lblNaam: TLabel;
    gpbSkrikkeljare: TGroupBox;
    lblInvoer1: TLabel;
    edtSkrikkeljaar: TEdit;
    lblUitvoer1: TLabel;
    btnSkrikkeljaar: TButton;
    gpbAantalDaePerMaand: TGroupBox;
    lblInvoer2: TLabel;
    edtMaand: TEdit;
    btnMaand: TButton;
    lblUitvoer2: TLabel;
    gpbDatums: TGroupBox;
    radDaeVerloop: TRadioButton;
    radVerskilTussenDatums: TRadioButton;
    btnOK: TButton;
    lblUitvoer3: TLabel;
    edtDatum1: TEdit;
    edtDatum2: TEdit;
    bmbClose: TBitBtn;
    bmbReset: TBitBtn;
    procedure btnSkrikkeljaarClick(Sender: TObject);
    procedure btnMaandClick(Sender: TObject);
    procedure btnOKClick(Sender: TObject);
    procedure bmbCloseClick(Sender: TObject);
    procedure bmbResetClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmDatums: TFrmDatums;
```

```
implementation

{$R *.DFM}

procedure TFrmDatums.btnSkrikkeljaarClick(Sender: TObject);

begin;
end;

procedure TFrmDatums.btnMaandClick(Sender: TObject);
begin
If edtInvoer2 := Januarie,Maart,Mei,Julie,Augustus,Oktober,Desember then
lblUitvoer:= '31 Dae';
 If edtInvoer2 := Februarie
then
lblUitvoer:= '28 of 29 Dae';
  If edtInvoer2 := April, Junie,September,November then
lblUitvoer:= '30 Dae';
end;

procedure TFrmDatums.btnOKClick(Sender: TObject);
Dae:=Integer;
begin
If radDaeVerloop := checked
then
Dae:=edtDatum2-edtDatum1;
lblUitvoer3:= ''Dae'';
  If radVerskilTussenDatums := checked
then
Dae:=edtDatum2-edtDatum1;
lblUitvoer3:= ''Dae'';
end;

procedure TFrmDatums.bmbCloseClick(Sender: TObject);
begin
close;
end;

procedure TFrmDatums.bmbResetClick(Sender: TObject);
begin
 edtSkrikkeljaar.clear;
 edtMaand.clear;
 edtDatum1.clear;
 edtDatum2.clear;
 radVerloopVanDae.checked:= false;
 radVerskilTussenDatums.checked:= false;
 edtskrikkeljaar.Setfocus;
end;

end.
```

Participant 31

Program 5.2 (Second attempt)

This second application program of P31 was not correctly saved and could not be compiled.

```
unit Datum_u; // [saved as Datum_u.~pas]

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TFrmDatums = class(TForm)
    lblNaam: TLabel;
    gpbSkrikkeljare: TGroupBox;
    lblInvoer1: TLabel;
    edtSkrikkeljaar: TEdit;
    lblUitvoer1: TLabel;
    btnSkrikkeljaar: TButton;
    procedure btnSkrikkeljaarClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmDatums: TFrmDatums;

implementation

{$R *.DFM}

procedure TFrmDatums.btnSkrikkeljaarClick(Sender: TObject);

var Skrikkeljaar:integer;
NieSkrikkeljaar:string;

begin
Skrikkeljaar := 1904 or 1936 or 2004;
If edtSkrikkeljaar = Skrikkeljaar ;
then
 lblUitvoer1 := 'skrikkeljaar';
end;

end.
```

Thinking processes of Participant 31

Programming processes: P31

Ek het nou eers die hele 'vraag' deurgelees om te kyk wat alles van my gevra word.
Bietjie nagedink oor die klas wat ek gaan skep.
Terug gegaan na die Delphi boek want ek sukkel met klasse.
Die vorm geskep met die nodige buttons wat ek dink ek gaan nodig kry.
My naam in 'n label gesit.
Eerder Groupboxes ingesit want ek dink dit gaan beter werk.
Weer terug gegaan na 'n ander boek. Uitgevind by my ma wanneer dit 'n skrikkeljaar is.
'n if-stelling ingesit vir skrikkeljaar.
Ek sukkel nog steeds.
My program wil nie werk nie.
Ek is moedeloos.
Gaan aan. My program wys nie my foute vir my nie, bly net in die ding waar mens die program ontwerp.
Ek weet nie of my goed reg is nie.
Ek tik alles wat ek dink moet in wees.

Appendix G: Data of Participant 32, a successful programmer

Programming example and thinking processes

Program in JAVA

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Datum
{
BufferedReader console = new BufferedReader(new InputStreamReader(System.in) );


        String[] months31 = { "Januarie", "Maart", "Mei", "Julie", "Augustus", "Oktober",
        "Desember" };
        String[] months30 = { "April", "Junie", "September", "November" };
        String[] months = { "Januarie", "Febuarie", "Maart", "April", "Mei", "Junie",
        "Julie", "Augustus", "September", "Oktober", "November", "Desember" };

        private int day, year;
        private String month;

        boolean trueDate = false;

        public Datum() throws IOException
        // Konstruktor van die Datum klas
        {
        // Lees vandag se datum in wanneer objek gemaak word
        String input = "";
        // Toets dan die datum
        int dayTemp, yearTemp ;
        String monthTemp ;

        while ( trueDate == false )
        {
          System.out.print("Wat is vandag se datum (DD Month YYYY): ");
          // Lees datum in
          input = console.readLine();
          String yearStr = input.substring( (input.length()-4), input.length());
          yearTemp = Integer.parseInt(yearStr);
          monthTemp = input.substring(3,(input.length()-5));
          String dayStr = input.substring(0,2);
          dayTemp = Integer.parseInt(dayStr);

          trueDate = testDate(dayTemp, monthTemp, yearTemp);
          // Toets datum
          if ( trueDate == false )
            System.out.println("Datum was inkorrek ingevoer, doen asb weer");
            // Indien inkorrek, herhaal die vraag
```

```
        else
        {
        year = yearTemp;
        // Stel waardes van ingeleesde datum gelyk aan globale veranderlikes

        for ( int i = 0; i < months.length ; i++)
        if ( monthTemp.equalsIgnoreCase(months[i]) )
            {month = months[i];
            day = dayTemp;}
        }
    }

    public boolean testDate(int dayTemp, String monthTemp, int yearTemp)
    // Metode wat ek geskryf het om die datums wat ingelees word, te toets
    {
            int monthNum;
            int numDays = 0;
            boolean testMonth = false;
            boolean testDay = false;
            boolean testYear = false;
            boolean yearSkrik = false;

            if ( yearTemp >= 1800 )
            // Toets of jaartal groter is as 1800, soos aangewys
            {
            if ( yearTemp % 100 == 0 )
            // Toets vir skrikkeljare, bedoel vir Febuarie met sy verskil in dae
            if ( yearTemp % 400 == 0)
                    yearSkrik = true;
                    else if ( yearTemp % 4 == 0 )
                            yearSkrik = true;
                    testYear = true;
            }
            for ( int i = 0; i < months.length ; i++)
            // Gebruik arrays vir toets van korrekte aantal dae in die maande
          {
            if ( monthTemp.equalsIgnoreCase(months[i]) )
            {
            testMonth = true;
            for (int x1 = 0; x1 < months31.length; x1++)
                    if ( months[i].equalsIgnoreCase(months31[x1]) )
                            numDays = 31;
                    for (int x2 = 0; x2 < months30.length; x2++)
                    if ( months[i].equalsIgnoreCase(months30[x2]) )
                            numDays = 30;
            }
            if ( monthTemp.equalsIgnoreCase("Febuarie") )
            // Uitsondering gemaak vir Februarie
                    {
                            testMonth = true;
                            if ( yearSkrik == true )
                                    numDays = 29;
                            else
                                    numDays = 28;
                    }
          }
```

```
		if ( dayTemp <= numDays )
			testDay = true;

		if ( testYear == true && testMonth == true && testDay == true )
		// Om aan te dui of datum korrek is of nie
			return true;
		else
			return false;
	}

	public String kryDatum()
	// Metodes wat vereis was
	{
		return (day + " " + month + " " + year) ;
	}

	public String kryMaand()
	{
		return month;
	}

	public int kryJaar()
	{
		return year;
	}

	public void toetsSkrikkelJaar() throws IOException
	// Skrikkeljaar metode
	{
	// Lees in jaar ... bepaal dan of dit skrikkeljaar is
	int year = 0;
	int temp;
	while ( year < 1800 )
	// As verkeerd ingevoer word , herhaal die vraag
	{
		System.out.print("Tik in 'n jaar getal(Enige jaar vanaf 1800): ");
		String input = console.readLine();
		year = Integer.parseInt(input);
		// Lees in waarde van jaar
		if ( year < 1800 )
		// Toets die jaar se domein
		   System.out.println("Inkorrek jaar");
	}
		if ( year % 100 == 0 )
			// Bepaal of jaar 'n skrikkeljaar is
		{
		if ( year % 400 == 0 )
			System.out.println("Die jaar " + year + " is 'n skrikkeljaar");
		else
			System.out.println("Die jaar " + year + " is nie 'n skrikkeljaar nie");
		}
		else if ( year % 4 == 0  )
			System.out.println("Die jaar " + year + " is 'n skrikkeljaar");
		else
			System.out.println("Die jaar " + year + " is nie 'n skrikkeljaar nie");
}
```

```
public void datumsVerskil() throws IOException
// Om verskil tussen 2 datums te bepaal
{
// Agv limitasie op jaargetal het ek 'n snaakse metode gebruik
String input = "";
int dayTemp, yearTemp;
String monthTemp ;
boolean firstDate = false;
boolean secondDate = false;

int[] numDaysMonths = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
// Om aantal dae in spesifieke maand te bepaal
int day1 = 0;
int day2 = 0;
int year1 = 0;
int year2 = 0;
int month1Num = 0;
int month2Num = 0;
String month1 = "";
String month2 = "";

while ( firstDate == false )
// Invoer van eerste datum vind hier plaas
{
// volg selfde patroon as konstruktor, lees in en toets en stel dan gelyk aan
//veranderlikes
System.out.print("Eerste datum (DD Month YYYY of 'vandag se datum'): ");
input = console.readLine();
if ( input.equalsIgnoreCase("vandag se datum") )
        input = this.kryDatum();
        String yearStr = input.substring( (input.length()-4), input.length());
        yearTemp = Integer.parseInt(yearStr);
        monthTemp = input.substring(3,(input.length()-5));
        String dayStr = input.substring(0,2);
        dayTemp = Integer.parseInt(dayStr);

        firstDate = testDate(dayTemp, monthTemp, yearTemp);
        if ( firstDate == false )
                System.out.println("Datum was inkorrek ingevoer, doen asb
                weer");
        else
        {
                year1 = yearTemp;
                for ( int i = 0; i < months.length ; i++)
                if ( monthTemp.equalsIgnoreCase(months[i]) )
                                        month1 = months[i];
                day1 = dayTemp;
        }
}

while ( secondDate == false )
// Selfde as invoer van eerste datum
```

```
{
        System.out.print("Tweede datum (DD Month YYYY of 'vandag se
        datum'): ");
        input = console.readLine();
        if ( firstDate == true && secondDate == true )
        // As altwee datums korrek ingelees is
        {
                boolean firstBiggest = false;
                //Hierdie if …bepaal watter datum die grootste is
                for ( int i = 0; i < months.length ; i++)
                // Kyk hier na hoeveelste maand die betrokke datum is
                {
                if ( month1.equalsIgnoreCase(months[i]) )
                        month1Num = i + 1;
                        if ( month2.equalsIgnoreCase(months[i]) )
                        month2Num = i + 1;
                }

                if ( year1 > year2 )
                // Toets nou hierdie verskillle tussen die twee datums
                        firstBiggest = true;
                // Toets eers jaargetal, dan maand en dan dag
                else if (year2 > year1 )
                        firstBiggest = false;
                else if ( year1 == year2 )
                {
                        if ( month1Num > month2Num )
                                firstBiggest = true;
                        else if (month2Num > month1Num )
                                firstBiggest = false;
                        else if ( month1Num == month2Num )
                        {
                                if ( day1 > day2 )
                                        firstBiggest = true;
                                else if ( day2 > day1 )
                                        firstBiggest = false;
                                else if ( day1 == day2 )
                                // Indien die 2 datums dieselfde is
                                {
                                System.out.println("Die verskil tussen hierdie datums
                                is: 0 dae");
                                firstDate = false;
                                secondDate = false;
                                }
                        }
                }

        if ( firstBiggest == true )
        // As eerste datum die grootste is
        {
        // Bepaal die aantal dae vanaf 01 januarie 1800 tot en met die datum wat
        // ingevoer is nadat altwee bepaal is word hulle van mekaar afgetrek
        int numYears1 = year1 - 1800 ;
        int numSkrikYears1 = 0 ;
```

```
int numNieSkrikYears1 = 0 ;
int totDays1 = 0;
int numYears2 = year2 - 1800 ;
int numSkrikYears2 = 0 ;
int numNieSkrikYears2 = 0 ;
int totDays2 = 0;
// First Date
for ( int i = 0; i < numYears1; i++ )
//Aantal jare tussen 1800 en eerste datum wat skrikkeljare is of nie
{
if ( (1800 + i) % 100 == 0 )
        if ( (1800 + i) % 400 == 0)
                numSkrikYears1++;
        else
                numNieSkrikYears1++;
        else if ( (1800 + i) % 4 == 0 )
                numSkrikYears1++;
        else
                numNieSkrikYears1++;
}

for ( int i = 0; i < numSkrikYears1; i++ )
// Indien dit skrikkel jare is of nie, 366 dae of 365 dae ens
totDays1 += 366 ;
for ( int i = 0; i < numNieSkrikYears1; i++ )
        totDays1 += 365 ;

for ( int i = 1; i < month1Num; i++ )
// Dae in maande word bygetel vir die huidige jaar
        totDays1 += numDaysMonths[i-1];

if ( year1 % 100 == 0 && month1Num > 2)
// Plus 1 ekstra omdat Februarie ekstra dag het in 'n skrikkeljaar
{
if ( year1 % 400 == 0)
        totDays1 ++;
}
else if ( year1 % 4 == 0 && month1Num > 2 )
        totDays1 ++;
totDays1 += day1;
// Plus aantal dae vir daardie maand self

// Second Date - Selfde as Eerste Datum
for ( int i = 0; i < numYears2; i++ )
{
        if ( (1800 + i) % 100 == 0 )
        if ( (1800 + i) % 400 == 0)
                numSkrikYears2++;
        else
                numNieSkrikYears2++;
        else if ( (1800 + i) % 4 == 0 )
                numSkrikYears2++;
        else
                numNieSkrikYears2++;
}
```

```
            for ( int i = 0; i < numSkrikYears2; i++ )
                  totDays2 += 366 ;
            for ( int i = 0; i < numNieSkrikYears2; i++ )
                  totDays2 += 365 ;
            for ( int i = 1; i < month2Num; i++ )
                  totDays2 += numDaysMonths[i-1];
            if ( year2 % 100 == 0 )
            {
            if ( year2 % 400 == 0)
                  totDays2 ++;
            }
            else if ( year2 % 4 == 0 )
                  totDays2 ++;
            totDays2 += day2;

      // Trek totale van dae van mekaar af om verskil te bereken
      System.out.println("Die verskil tussen hierdie datums is: " + (totDays1-totDays2)
       " dae");
            }
            else
            // Selfde as boonste bracket, enigste verskil is in laaste reel
            {
                  int numYears1 = year1 - 1800 ;
                  int numSkrikYears1 = 0 ;
                  int numNieSkrikYears1 = 0 ;
                  int totDays1 = 0;

                  int numYears2 = year2 - 1800 ;
                  int numSkrikYears2 = 0 ;
                  int numNieSkrikYears2 = 0 ;
                  int totDays2 = 0;

                  // First Date
                  for ( int i = 0; i < numYears1; i++ )
                  {
                        if ( (1800 + i) % 100 == 0 )
                        if ( (1800 + i) % 400 == 0)
                              numSkrikYears1++;
                        else
                              numNieSkrikYears1++;
                        else if ( (1800 + i) % 4 == 0 )
                              numSkrikYears1++;
                        else
                              numNieSkrikYears1++;
                  }

                  for ( int i = 0; i < numSkrikYears1; i++ )
                        totDays1 += 366 ;
                  for ( int i = 0; i < numNieSkrikYears1; i++ )
                        totDays1 += 365 ;

                  for ( int i = 1; i < month1Num; i++ )
                        totDays1 += numDaysMonths[i-1];
```

```
                if ( year1 % 100 == 0 )
                {
                        if ( year1 % 400 == 0)
                                totDays1 ++;
                }
                else if ( year1 % 4 == 0 )
                        totDays1 ++;

                totDays1 += day1;

                // Second Date
                for ( int i = 0; i < numYears2; i++ )
                {
                        if ( (1800+i)%100 == 0 )
                        if ( (1800+i)%400 == 0)
                                numSkrikYears2++;
                        else
                                numNieSkrikYears2++;
                        else if ( (1800 + i) % 4 == 0 )
                                numSkrikYears2++;
                        else
                                numNieSkrikYears2++;
                        }

                        for ( int i = 0; i < numSkrikYears2; i++ )
                                totDays2 += 366 ;
                        for ( int i = 0; i < numNieSkrikYears2; i++ )
                                totDays2 += 365 ;
                        for ( int i = 1; i < month2Num; i++ )
                                totDays2 += numDaysMonths[i-1];
                        if ( year2%100 == 0 && month2Num > 2 )
                        {
                                if ( year2%400 == 0)
                                        totDays2 ++;
                        }
                        else if ( year2 % 4 == 0 && month2Num > 2)
                                totDays2 ++;
                        totDays2 += day2;


                // Is nou totDays2-totDays1 ipv andersom
                   System.out.println("Die verskil tussen hierdie datums is: " +
                   (totDays2-totDays1)        + " dae");
            }
        }
    }
}
```

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Toetsklas
{
        public static void main (String[] args) throws IOException
        {
                BufferedReader console = new BufferedReader( new
                InputStreamReader(System.in) );
                String input = "";
                Datum datum = new Datum();
                while ( !input.equalsIgnoreCase("3") )
                {
                        for ( int i = 0; i < 20; i++ )
                                System.out.println ("");
                        System.out.println ("Vandag se datum is: " + datum.kryDatum());
                System.out.println
                ("########################################");
                System.out.println ("#       DATUM KLAS - TOETS PROGRAM            #");
                System.out.println ("#                                             #");
                System.out.println ("#  1. Toets of jaargetal 'n skrikkeljaar is of nie #");
                System.out.println ("#  2. Bereken die verkil tussen twee datums    #");
                System.out.println ("#  3. Exit                                     #");
                System.out.println ("#                                             #");
                System.out.println ("################################");
                System.out.print ("Jou Keuse: ");
                input = console.readLine();
                if ( input.equalsIgnoreCase("1") )
                        datum.toetsSkrikkelJaar();
                else if ( input.equalsIgnoreCase("2") )
                        datum.datumsVerskil();
                        String temp = console.readLine();
                }
        }
}
```

Thinking processes of Participant 32

Vraag in my eie woorde:

Skryf ‘n Datumklas. Die klas moet vandag se datum inlees. Dit moet ook metodes bevat wat kan bepaal of ‘n sekere jaargetal ‘n skrikkeljaar is of nie. Die verskil in dae tussen 2 datums bereken.

My Programmeringsproses

1. Skep eers raamwerk vir Datum.java en Toetsklas.java, dit is goed soos opskrifte, imports, gegewe metodes ens.

2. Skep **konstruktor** vir **Datum.java** klas:
   a. Doel? Die konstruktor moet ‘n datum inlees. (Daarvoor benodig jy die BufferedReader, InputStreamReader imports)
   b. Parameters? Geen parameter nie
   c. Invoer? ‘n Datum word ingelees. Dit sal moet getoets word. Gaan baie keer datums getoets word? Ja, dus skryf ons ‘n metode vir datumstoets, nl. testDate (Verwys na punt 3)
   d. Uitvoer? Net as datum verkeerd ingevoer word. Gebruik ‘n while en Boolean om herhaling van die vraag reg te kry. Boolean waarde word verkry deur testDate metode. (Verwys na punt 3)
   e. Probleme? Het gesukkel om datum in te lees in goeie formaat. Het dus vir gebruiker ‘n vaste formaat gevra om te gebruik nl. DD Month YYYY, bv. 16 Oktober 2006.
   f. Waardes wat ingelees is, word gelyk aan globale veranderlike gestel. (Indien hulle later gebruik sou wou word)

3. Moet nou **testDate metode** skep, vir toets van datums:
   a. Doel? Hy moet ‘n datum wat ontvang is, toets of dit korrek is volgens ons kalender, m.a.w. moet kan sê dat bv. 45 Woensdag 1203 ‘n ongeldige datum is.
   b. Parameters? Sal moet datum wat getoets is ontvang, dus dayTemp, monthTemp, yearTemp.
   c. Invoer, Uitvoer? Daar is geen, het alle nodige data en sal net Boolean-waarde terug stuur.
   d. Veranderlikes betrokke?: Daar sal moet ‘n paar Boolean-veranderlikes wees nl. testMonth, testDay, testYear ens. Ook ‘n numDays integer veranderlike geskep, om te gebruik by dae toets
   e. Berekeninge:
      i Eerste jaar toets, moet groter wees as 1800. Toets dan of dit skrikkeljaar is of nie (vir Februarie maand).
      ii Dan die maande toets. Daarvoor skep arrays om te kan gebruik vir vergelyking. Gebruik dan for lus om deur almal te hardloop en te vergelyk (Probleme: Moes 3 arrays skep omdat sekere maande ander hoeveelheid dae het. Ek het dit agtergekom by dae toets. Moes ook ‘n uitsondering maak vir Februarie a.g.v. die skrikkeljaar probleem.)
      iii Nou dae toets. Het numDays veranderlike gebruik om dae te vegelyk. Kon dit so toets.
      iv As die toets positief was het dit die betrokke boolean verander na true
   f. Return? Sal ‘n true waarde return as alle booleans van toetsdata waar is anders return dit ‘n vals waarde.
   g. Probleme? Net met die maand se toets. Moes dus die arrays gebruik.

4. Noudat nodige metodes vir konstruktor werkend is, kan ons dit toets met die toetsprogram nl. Toetsklas.java. Om dit te kontroleer moet ons vinnig die gegewe metodes nl. kryDatum, kryMaand, kryJaar by die Datum klas byvoeg. Toetsprogram kan dan net vinnig datum.kryDatum() roep om te kontroleer.

5. Moet nou volgende metode vir Datum klas skryf, nl toetsSkrikkelJaar
   a. Doel? Moet jaargetal inlees en dan toets of dit skrikkeljaar is of nie
   b. Parameters? Geen, lees sy invoere, en return geen waardes nie
   c. Invoer, Uitvoer? Invoer: Moet net jaargetal inlees. Die getal moet groter wees as 1800. Anders geen probleme nie. Uitvoer: Moet net eenvoudige uitvoer gee: is dit 'n skrikkeljaar of nie.
   d. Veranderlikes? Net vir invoer.
   e. Berekeninge? Moet toets of dit groter is as 1800, indien nie, herhaal weer die vraag. Doen sommer altwee met 'n while (year < 1800). Dan net die eenvoudige toets vir 'n skrikkeljaar:
      i Toets eers of dit 'n eeujaar is met, year % 100. Indien dit gelyk is aan 0, dan is dit 'n eeujaar
      ii Eeujaar: year % 400, as dit gelyk is aan 0 dan is dit 'n skrikkeljaar, anders is dit nie. Nie-Eeujaar: year % 4, as dit gelyk is aan 0 dan is dit 'n skrikkeljaar anders nie.
      iii Gee uitvoer dan volgens wat afgelei is.
   f. Probleme? Nie regtig nie, sodra ek agtergekom het dat dit 'n nested-if sal benodig, was dit baie maklik.

6. Opdateer die toetsprogram om die toetsSkrikkeljaar metode te toets. D.w.s. las net by 'n roep instruksie nl datum.toetsSkrikkeljaar();

7. Nou datumsVerskil() metode skryf:
   a. Doel? Moet twee datums inlees en dan die verskil in dae tussen die twee bepaal.
   b. Parameters? Geen, gebruik eie invoer. Return ook geen waardes nie, want hy maak sy eie uitvoer.
   c. Invoer, Uitvoer? Invoer: Moet 2 datums inlees. Gebruik hier dieselfde manier as die konstruktor. Het net 2 stelle veranderlikes. Gebruik dus ook testDate metode. Uitvoer? Moet net die verskil in dae uitvoer tussen die twee datums, m.a.w. net 'n veranderlike aan die gebruiker vertoon.
   d. Veranderlikes? Die twee stelle vir datums. Twee booleans om aan te dui dat datums korrek is, volgens testDate metode. Moet ook 'n array skep om aan te dui hoeveel dae daar in elke maand is nl. numDaysMonths.
   e. Berekeninge? Moes dink aan 'n metode om dae te kan bepaal en dan verskil tussen die twee. Opsies was: 1. Om by een datum te begin en dan te tel totdat volgende datum bereik is. Klink of dit moeilik gaan wees en kan nie dadelik aan 'n duidelike manier dink nie. 2. A.g.v. die grens bepaal die jaargetal nl. 1800, het daar 'n ander moontlikheid voorgekom. Kan die dae tel vanaf 1 Januarie 1800 tot en met datum1. Doen dan dieselfde met datum2 en trek die 2 waardes van mekaar af. Het gevoel opsie 2 is meer prakties. Opsie 2:
      i Aftreksom. Moet dus bepaal watter datum is die grootste om aftreksom te kan reg doen. (Kon datums ook net aftrek en dan positief maak. Het nie vir my reg gevoel nie). Gebruik boolean om aan te dui watter is groter nl. firstBiggest. Het eers jare vergelyk dan maande en laastens dae. Het uitsondering gemaak indien hulle dieselfde is. Sal dan dadelik uitvoer gee dat verskil 0 dae is.
      ii As eerste datum groter is of nie maak amper geen verskil in bereken van aantal dae van 01 Januarie 1800 tot en met betrokke datum nie. Maak net verandering in aftreksom aan die einde. (Baie klein verandering)
      iii Bepaal eers hoeveelheid jare. Bepaal apart hoeveel is skrikkeljare en hoeveel nie. Dan plus korrekte aantal dae by veranderlike vir aantal dae nl. totDays1.

- iv Gebruik dan array van aantal dae in maande en plus betrokke dae by vir die huidige jaar se maande m.a.w. as dit die 5de maand is, plus aantal dae vir maande by tot en met 5de maand. Maak uitsondering vir skrikkeljaar d.w.s. een ekstra dag vir Februarie.
- v Plus dan aantal dae by bv. 9 Mei d.w.s. totDays1 + 9 dae.
- vi Bereken net so datum2 se totale dae nl. totDays2
- vii Trek dan die twee van mekaar af soos aangedui van watter is groter. Gee dan uitvoer oor verskil tussen dae

f. Probleme? Baie! Met die lees het die metode maklik gevoel, maar is in werklikheid nie. Moes baie uitsonderings maak, veral vir skrikkeljare ens. Grootste probleem was los hande die manier waarop die verskil in dae bepaal gaan word m.a.w. opsies om dae te tel. Het ook ‘n paar *ArrayOutOfBounds* exceptions gehad. Opgelos met diagramme.

8. Opdateer nou weer die toetsprogram om die datumsVerskil() metode te toets met datum.datumsVerskil();
9. Verander die toetsprogram in ‘n meer gebruikervriendlike omgewing deur ‘n “menu” in te sit ens.

Addisionele Notas

Die dokument was saamgestel uit ‘n klomp los stukke notas wat ek gebruik het terwyl ek geprogrammeer het. Die rede vir hierdie dokument is, omdat net ek self daardie notas sou kon verstaan. Dit bevat baie persoonlike afkorting en gekrap, en sal nie vir enige iemand sin maak nie. Die dokument is saamgestel sodat die leser kan sien op watter manier ek dink gedurende die programmerings proses en watter vrae ek my self vra.

BRONNE

HORSTMANN, C., 2002. Big Java: programming and practice.

Appendix H: Article accepted for the *South African Computer Journal*

Knowledge, skills and strategies for successful object-oriented programming: a proposed learning repertoire

HM (Marietjie) Havenga
E (Elsa) Mentz
MR (Ruth) de Villiers

Abstract

Third year Computer Science students were studied in order to determine which knowledge, skills and strategies they used during an object-oriented programming task. Quantitative and qualitative methods were used to analyse their computer programs and associated thinking processes. Successful programmers applied significantly more cognitive, metacognitive and problem-solving knowledge, skills and strategies, also using a greater variety, than the unsuccessful ones. Based on the approaches of the successful programmers, we propose a learning repertoire of integrated knowledge, skills and strategies, which can serve as a framework to support novices learning object-oriented programming (OOP).

Introduction

Learning and conducting object-oriented programming (OOP) is multidimensional and complex (Govender and Grayson, 2006:1687). OOP requires the use of specific knowledge, skills and strategies to solve problems and write the associated programs. Successful and unsuccessful programmers differ in the way they approach and solve programming problems. An unsuccessful programmer is a person who did not achieve the stated outcomes, while a successful programmer is one who did achieve them and who dealt efficiently with problems (Govender and Grayson, 2006:1689). Successful programmers possess a well-organised, carefully-learned knowledge structure (Ala-Mutka, 2004:2); they use self-regulatory processes

and monitor their problem-solving activities (Glaser, 1999:91-92) and they can solve a problem quickly, although they often appear to spend more time in problem representation (Sternberg, 2006:424). These are some examples of cognitive, metacognitive and problem-solving activities that are required in programming. However, these are not merely personal or isolated learning techniques, but rather distinct activities that should explicitly be integrated to address a programming problem and solve it successfully.

This paper considers the following research questions:

1. What are the differences between the ways that successful and unsuccessful programmers apply their knowledge, skills and strategies in an object-oriented programming task?
2. How can novices be supported in learning OOP?

The objective of the first question was an attempt to identify cognitive, metacognitive and problem-solving knowledge, skills and strategies used by successful and unsuccessful programmers in OOP. To answer the second, we attempted to integrate the approaches of successful programmers into a learning repertoire that can serve as a framework for novices learning OOP.

Literature survey

Computer programming involves a rich environment in which specific programming words, statements and constructs come together to be integrated in a tightly defined way to solve a problem efficiently. This requires high-level knowledge, skills and strategies. In general, the knowledge relates to information and skills acquired through experience or education. A skill refers to the ability to do a particular task, while a strategy is a designed plan to achieve a purpose and to solve a problem (Concise Oxford English Dictionary, 2004:789,1351,1425; Gu, 2005:1). It is often assumed that students implicitly and independently master the required high-level knowledge, skills and strategies, and that teaching should focus on programming content and coding structures only. However, to be successful in the complex domain of OOP, explicit learning of both facets is required. This survey briefly overviews some aspects and techniques that can support successful programming.

Cognition

The concept of cognition refers to the mental processes used in the acquisition, storage, transformation and application of knowledge (Sternberg, 2006:157). In this regard Bloom's taxonomy (1973) defines six types of learning, hierarchically ordered according to the level within the cognitive domain: knowledge; comprehension; application; analysis; synthesis; and evaluation. The way in which these concepts are used (or not used) can define the differences between successful and unsuccessful programmers (Zant, 2005:1), where the six associated skills are, respectively: knowledge of the programming language; interpretation of the programming problem; application of prior knowledge in a new program; analysis of the problem; design of a new program; and evaluation of the solution. Since programming is 'extremely cumulative', novices must progress through each of Bloom's six levels to become truly successful (Carbone *et al.,* 2002:2; Zant, 2005:1).

Recall of information can be improved by cognitive strategies (Schunk, 2000:139-144), such as rehearsal, elaboration and organisation (Bergin *et al.,* 2005:82). Rehearsal strategies, for example: focussing attention, structured recall, and distributed practice over a period of time; can support recollection and help to pinpoint important information within a context. In the programming context, programmers who repetitively sequence activities in a particular way 'preserve the effect', using less working capacity (Gu, 2005:9). Elaboration helps students to integrate new information with prior knowledge by, for example, generative note taking, asking questions, summarising, and creating analogies. The organisation strategy includes extraction of the main idea from text as well as integration of concepts (Bergin *et al.,* 2005:82) with the goal of achieving a holistic problem solution.

Metacognition (cognition about cognition)

Metacognitive knowledge is explicit knowledge of one's own cognitive strengths and weaknesses, beliefs and conditions that affect memory performance (Gravill *et al.,* 2002:1055; Koriat, 2002:267). Self-knowledge, task-knowledge and strategy knowledge are required in the metacognitive domain (Flavell, 1979). Metacognitive strategies include planning, monitoring and regulation. In programming, planning entails analysis of the problem and the identification of possible classes and methods to solve it, while monitoring guides the process of finding a solution by means of self-

testing (Bergin *et al.,* 2005:82). Regulation involves the continuous modification of one's cognitive activities to determine whether the problem is being solved successfully. Bergin *et al.* (2005:83) discuss self-regulated learning with regard to the performance of students in their third level of introductory OOP. They found that students with high levels of intrinsic motivation perform better and use more metacognitive-management strategies than lower performing students.

Problem solving

Different kinds of problems are solved in different ways and require different approaches. Students should understand how problems vary according to their structuredness, complexity, dynamicity and domain-specificity (Jonassen, 2004:3-9). In this regard, programming experience and exposure play roles and Sternberg (2006:426) suggests that experts develop sophisticated internal representations of certain kinds of problems, based on their structural similarities.

Standard problem-solving strategies are: bottom-up, top-down, integrated, as-needed and trial-and-error (Corritore and Wiedenbeck, 2000:139; Edwards, 2004:26; Zhang, 2005:7). Research shows that expert object-oriented programmers tend to use top-down strategies during the early phases of programming to understand systems holistically. In contrast, the same experts may use a bottom-up strategy when programming in an unfamiliar context or during program maintenance where individual parts are combined to form larger components (Corritore and Wiedenbeck, 2000:139-148).

Object-oriented programming

OOP is based on the object-oriented approach, where objects are models of real-world entities that have the responsibility of carrying out specific tasks to solve the problem (Garrido, 2003:26-27). OOP involves various knowledge and skills relating to data types, control structures, instantiation of objects, methods, GUI tools, exception handling, database connectivity (Jackson and Satzinger, 2003:3), input/output validation, performance correctness (Stamouli and Huggard, 2006:113), debugging and the development of test data. Due to the complexity of OOP, students have difficulty in applying the required activities successfully (Govender & Grayson, 2006:1693). Explicit

teaching and learning of high-level knowledge, skills and strategies may therefore be a requirement to support success in OOP.

Research design

The underlying research ethos of this study is constructivist problem solving, which refers to the students' active construction of computer programs and application of programming constructs such as classes and objects. It also relates to the researcher's construction of a body of knowledge regarding the students' programming constructs, as she interprets and reflects on those programming experiences. This implies a continuous process of interpretation and reflection.

In a mixed methodology, both quantitative and qualitative research methods were used to analyse participants' computer programs and the associated written thinking processes. Quantitative methods include statistical calculations such as descriptive statistics, practical significance and correlation. As a qualitative research practice, grounded theory was applied to guide the systematic collection of data and to generate a model inductively from the ongoing data collection and analysis to explain the specific phenomenon (De Villiers, 2005:24; Glaser and Strauss, 1967:1).

Data collection

The research was conducted over a period of two years. The participants (n = 48) came from two groups: the first group, namely 2005, consisted of 11 BEd and 17 BSc 3rd year students, and the second group, namely 2006, comprised three BEd and 17 BSc 3rd year students. Students from both groups took Computer Science as a major subject.

Each participant had to create an object-oriented program relating to leap years. It was an open-ended question and participants had to decide personally which calculations were necessary in the program. However, some requirements were included to direct the programming process. At the very least, the students should write a *Date class* program to calculate which years are leap years and the difference between any two dates in the range 1 January 1800 to a later date. A *Test class* program was also required to determine whether the output of the *Date class* was correct. The programs could be done in either Delphi or Java. During the major process of

programming the *Date class* task, participants were required to record their thinking and problem-solving processes in writing.

Data collection included both the computer programs and the recorded thinking processes. Triangulation was applied by investigating data from these two sources, i.e. the coded programs and the associated thinking processes written by participants as they considered the problem and coded their solutions. Finally, coherence between the different data sources was investigated to identify patterns of meaning and to describe the emerging theory that leads to the learning repertoire.

Data analysis and findings

Two approaches were followed. In the first approach, each program itself and the recorded thinking processes were evaluated, using as an instrument, a set of measurement criteria that had emerged from the literature review. The 24 criteria (or subcategories) shown in Table 1 originate from four major categories: cognitive knowledge and skills; metacognitive strategies; problem-solving strategies; and OOP knowledge and skills. Measurement of 23 of the criteria was scored on a 4-point scale where 1 indicates poor performance and 4 an excellent performance. For the problem-solving category with its single criterion, participants could use more than one strategy, so a maximum of 8 was allocated instead of 4. Participants who used the trial-and-error strategy received zero, since it was not considered an acceptable problem-solving strategy. The 24 criteria thus score a total of 100. As the indicator of 'successful' programming, participants had to obtain 3 or 4 for the 'Correctness of output' subcategory (last criterion in Table 1), relating to evidence of correct program output and the test data used. Based on this approach, there were 11 successful and 37 unsuccessful programmers.

The scores were analysed by descriptive statistics to determine the means and standard deviations of successful and unsuccessful participants for all criteria and for the overall categories. Practical significant differences (effect size) between successful and unsuccessful participants were determined for all criteria, as shown in Table 2. Guidelines for the interpretation of effect size are as follows: $d = 0.2$ small effect; $d = 0.5$ medium effect; $d = 0.8$ large effect (Cohen, 1988). Values ≥ 0.8 mean that the effect size of constructs is regarded as practically significant (Ellis and Steyn, 2003). However,

Thompson (2001:82-83) warns that researchers should avoid using these guidelines in an overly rigid way. In order to determine correlations between the cognitive, metacognitive and OOP constructs, the Spearman ranked correlation coefficient was used, as shown in Table 3. The correlation is interpreted as follows: $r = 0.1$ small effect; $r = 0.3$ medium effect; and $r = 0.5$ large effect (Cohen, 1988). Data with an r-value ≥ 0.5 is considered as practically significant (Ellis and Steyn, 2003:52; Steyn 2002).

The second analysis approach investigated the thinking processes of participants, using the qualitative analytical software package, *Atlas.ti*. The purpose was to identify various themes that emerged from the recorded thinking processes. The researcher allocated codes to particular segments in the typed textual data until sufficient similar patterns were identified, indicating that saturation had occurred. After the codes were grouped and categorised, various themes were identified.

Table 1: *Measurement criteria and associated categories*

Category		Criterion
Cognitive knowledge and skills		
Knowledge	(4)	Evidence of knowledge of the programming language
Comprehension	(4)	Interpretation of the problem
Application	(4)	Application of prior knowledge in a new program
Analysis	(4)	Analysis of the problem – breaking it down into steps
Synthesis	(4)	Designing a new program
Evaluation	(4)	Evaluation of the solution
Metacognitive strategies		
Planning	(4)	Evidence of planning during programming
Monitoring	(4)	Evidence of monitoring tasks during programming
Regulation	(4)	Evidence of regulation or modification to correct flaws during programming

Table 1: *Measurement criteria and associated categories continued*

Category	Criterion
Problem-solving strategies (8)	Application of problem-solving strategies: bottom-up, top-down, integrated, as-needed
OOP knowledge and skills	
Program requirements analysis (4)	Analysis of the program requirements
Programming techniques (4)	*Programming techniques used: indentation, readability, variable names and declaration
Programming statements (4)	*Application of the correct use of programming statements
User-friendliness (4)	Application of user-friendliness and usability
Classes and objects (4)	Designing of classes and instantiation of objects
Method application (4)	Application of methods such as constructors, mutators and accessors
Access control (4)	*Decision on the accessibility: public, private
Parameter passing (4)	*Application of parameter passing: number, order, type of variables
Reasoning (4)	Application of reasoning skills in OOP
Exception handling (4)	*Application of exception handling
Program structure, scope (4)	*Application of program structure and scope
Successful programming (4)	Actual solution to the problem
Program evaluation (4)	Evaluation of the *Date class* and *Test class*
<u>Correctness of output</u> (4)	Evidence of correct program output and test data used
TOTAL (%)	

* Criteria selected specifically to reflect on general characteristics of programming (Sebesta, 2004:8).

Quantitative findings re participants' programs and thinking processes

Table 2 summarises the measurement criteria for each category and its subcategories, giving the: mean values, standard deviations and effect size for

successful and unsuccessful participants, respectively. The means for cognition, metacognition and OOP are higher for successful participants than for the unsuccessful. Practical significant differences with a large effect size were found between successful and unsuccessful participants within all subcategories except for knowledge, comprehension, classes and objects, access control and parameter passing, where practical significant differences of a medium effect size occurred.

Table 2: *Means, standard deviations and practical significances for unsuccessful and successful participants*

Category	Unsuccessful participants (37)		Successful participants (11)		Practical significance (effect size)
	$\overline{x}$	s	$\overline{x}$	s	d
Cognition	**3.05**	**0.71**	**3.85**	**0.20**	**1.13***
Knowledge	3.65	0.68	4.00	0.00	0.51
Comprehension	3.54	0.65	4.00	0.00	0.71
Application	3.32	0.78	4.00	0.00	0.87*
Analysis	3.08	0.80	3.82	0.40	0.93*
Synthesis	2.62	0.92	3.73	0.47	1.21*
Evaluation	2.05	0.97	3.55	0.52	1.55*
Metacognition	**2.36**	**0.88**	**3.33**	**0.54**	**1.10***
Planning	3.24	0.83	3.91	0.30	0.81*
Monitoring	2.19	1.13	3.27	0.79	0.96*
Regulation	1.65	1.06	2.82	0.75	1.10*

Table 2: *Means, standard deviations and practical significances for unsuccessful and successful participants continued*

Category	Unsuccessful participants (37)		Successful participants (11)		Practical significance (effect size)
	$\overline{x}$	s	$\overline{x}$	s	d
OOP constructs	**2.44**	**0.77**	**3.62**	**0.29**	**1.53***
Program requirements analysis	3.24	0.86	4.00	0.00	0.88*
Programming techniques	3.11	0.97	4.00	0.00	0.92*
Programming statements	3.08	1.04	3.91	0.30	0.80*
User-friendliness	1.62	1.30	3.00	0.77	1.06*
Classes and objects	2.97	1.12	3.82	0.40	0.76
Method application	2.70	0.94	3.64	0.50	1.00*
Access control	3.19	1.08	3.91	0.30	0.67
Parameter passing	3.24	1.12	4.00	0.00	0.68
Reasoning	2.89	0.81	3.73	0.47	1.04*
Exception handling	0.46	0.80	2.55	1.21	1.73*
Program structure and scope	2.86	0.88	3.73	0.47	0.99*
Successful programming	2.41	1.09	3.73	0.47	1.21*
Program evaluation	2.00	1.08	3.55	0.52	1.44*
Correctness of output	0.35	0.72	3.18	0.40	3.93*

*d = 0.8, large effect size; d = 0.5, medium effect size (Ellis and Steyn, 2003:51)

There are possible correlations between participants' expertise in cognition, metacognition and OOP knowledge and skills. Table 3 shows Spearman correlations between pairs of these variables. In all the constructs measured, correlations were greater than 0.5 and therefore relevant in practice (Steyn, 2002). The high correlation between cognition and the OOP construct (r = 0.89) implies that certain predictions can be made regarding successful

programming in cases where participants make effective use of all the cognitive activities. The correlation between metacognition and OOP (r = 0.73) suggests that the use of metacognition and reflection can support problem-solving performance in OOP.

Table 3: *Correlations between cognition, metacognition and OOP constructs*

Construct	r
Cognition Metacognition	0.63**
Cognition OOP construct	0.89**
Metacognition OOP construct	0.73**

*** Practically significant (Steyn, 2002).*

Analysis of the thinking processes with Atlas.ti

Five main themes emerged in an inductive grounded-theory approach from the analysis of the participants' thinking processes in association with their programming of the *Date class*, namely: cognitive knowledge, skills and strategies; metacognitive knowledge, skills and strategies; problem-solving knowledge, skills and strategies; errors and problems in programming; and additional support in programming.

Theme 1: Cognitive knowledge, skills and strategies

The unsuccessful participants did not refer to explicit evaluation skills as in Bloom's taxonomy nor to cognitive strategies. Responses indicating that they used some of the skills in Bloom's taxonomy are: *I find out when it is a leap year [P31]*; I first determine the requirements [P20]; Which variables do I need? [P30]. Firstly, I thought about the class structure [P10]; Which methods should be in the class? [P21]; I need a method to convert the number of days [P36].* *[P31] refers to Participant 31, etc.

Successful participants applied the full set of skills from Bloom's taxonomy, some examples being: *A programmer should understand basic principles [P15]*; *I received the date as a string and separated it into days, months and years [P40].* During synthesis and evaluation, participants integrated various

methods in the class: *I also need a method to test for valid dates [P23].* Participant 40 referred to evaluation skills when he indicated that his program was *working 100%.* Participant 23 applied the elaboration strategy in the following statement: *When designing the class, I ask myself about the general and special cases in each situation*.

Theme 2: Metacognitive knowledge, skills and strategies
Unsuccessful participants reflected and acknowledged their programming weaknesses. Two examples are: *I have the correct idea but cannot apply it [P5]; I do not have a plan … [P34].* Some useful responses of unsuccessful participants about metacognitive strategies are: *I re-read the question with attention [P30]; I could send the date to the constructor [P33]; I forgot to insert close brackets [P41]; I have determined the difference in days but was incorrect with one day [P39].*

Successful participants applied a spectrum of metacognitive activities: *I read the question carefully and determined what was being asked? What are the specifications? [P29].* Participant 32 used planning, monitoring and regulation strategies: *Many questions were asked to determine the purpose, parameters, input, output, and problems of the programming task* (planning). He also reflected on the programming task: *Problems? Many! The method was difficult … and I should include many exceptions for leap years. The biggest problem was the difference between days. I have a few ArrayOutOfBounds exceptions. This was solved with diagrams* (monitoring and regulation).

Theme 3: Problem-solving knowledge, skills and strategies
Unsuccessful participants found it difficult to follow specific steps during problem solving: *I do not know if it is correct. I have typed all the things that I thought should be in the program [P31]. I … will try to code by means of trial-and-error [P34].* Participant 6 used the bottom-up strategy to solve the problem*: I will complete the code for a specific component before continuing with the next component.*

Successful participants described their systematic problem-solving steps in more detail. For example: *I determine the input, design the interface and basic components, process and then test the input [P44].* Participant 32 used

the top-down strategy when he indicated: *I will start with the framework for the Date and Test class, headings, import given methods, etc.*

Theme 4: Errors and problems in programming

This theme highlights examples of errors and problems, some of which also relate to a lack of metacognitive strategies. Unsuccessful participants pointed out: *I wonder why I typed some of this code, because I will not use it [P39]; …exception handling is complicated [P33].* Some participants could not apply exception handling or interpret errors [P31, P33]; others used incorrect syntax [P39] and could not compile the program.

Successful participants were able to diagnose and correct their errors. Two examples from P32: *I had problems determining a specific date format [P32]; …the Difference() method was difficult and I should provide for many exceptions… [P32].*

Theme 5: Additional support in programming

Both unsuccessful and successful participants referred to supplementary means of support during the programming process: *I used…previous code* [*P48]; textbooks [P30]; …previous…assignments [P44]; and Wikipedia.com for the requirements of leap years [P29].*

Research questions revisited

This section answers the first research question: *What are the differences between the ways that successful and unsuccessful programmers apply their knowledge, skills and strategies in an object-oriented programming task?* The answer relates to the three major themes that emerged from the grounded theory analysis: cognitive-, metacognitive- and problem-solving knowledge, skills and strategies that unsuccessful and successful participants apply/do not apply in the process of a programming task.

Cognitive knowledge, skills and strategies

Unsuccessful participants battled to decompose the problem scenario and to relate subparts to the overall structure. With regard to actual programming, they could not readily apply higher-order thinking skills. Although they used knowledge and comprehension skills, their programs indicate that they debugged and evaluated the code without using detailed application and

analysis skills. As a consequence, they had problems in interpreting their errors, they could not complete the program, and many did not obtain output.

For the higher-order thinking skills (analysis, synthesis and evaluation) required for programming, the successful participants received a mean value of more than 3.5 on a 4-point scale. Their ability to apply all the levels of Bloom's taxonomy in a task was clear and they achieved a high level of accuracy in solving the problem. It is notable that they spent more time on the analysis phase and differentiated how parts are inter-related in the complete program. Their performances illustrate that programmers should understand the problem precisely, interpret and evaluate their programming solutions.

Only one successful participant explicitly mentioned a cognitive strategy that was used during programming. Possible reasons could be that participants did not verbalise knowledge about these strategies, they did not use cognitive strategies, or they did not know how to apply such strategies in programming. In this regard, Bergin *et al.* (2005:85) show that cognitive strategies are not as useful in the learning of introductory OOP as they are in other domains.

Metacognitive knowledge, skills and strategies

Unsuccessful participants found it difficult to apply metacognitive activities during programming; they encountered problems in monitoring and regulating their cognitive resources. Very few of them applied any form of regulatory strategy. They could not easily reflect on the task and their own understanding of it, and found it difficult to manage their thinking and reasoning.

By using detailed planning strategies, successful participants were able to complete their tasks and produced high quality solutions. Most participants monitored their progress and effectively managed their cognitive resources in the process of finding a solution (Table 2). The regulation strategy of successful participants was slightly lower than 3 ($\bar{x}$ = 2.82), which implies that they could improve further on regulatory strategies during programming. These findings correspond with Hertzog and Robinson (2005:110, 111), who suggest that monitoring plays a vital role in cognitive performance of complex problem solving and guides the process of finding a solution.

Problem-solving knowledge, skills and strategies

Unsuccessful participants did not obtain the required program output. Some encountered problems in systematically applying problem-solving strategies. Instead, they spent time iterating through their programming code to address errors, without understanding which sections were incorrect and how to rectify them. Such participants were much less accurate in their efforts to reach an appropriate solution. Although most of the unsuccessful participants used a bottom-up strategy (27), some wrote that they worked without using any specific problem-solving strategies (2). Two used trial-and-error, three used a top-down strategy, and three used the integrated strategy.

Successful participants had considerable domain knowledge and highly efficient problem-solving skills, which they were able to apply successfully in the task. Seven of them used the bottom-up strategy, two the top-down, and two the integrated strategy during program comprehension. None of the successful participants used the trial-and-error strategy. This appears to indicate that it is not a successful approach in OOP, whereas all the other problem-solving strategies were used effectively.

The second research question is: *How can novices be supported in learning OOP?* It is answered by presenting a proposed learning repertoire.

Proposed learning repertoire

The constructivist problem-solving approach supports active involvement of students in constructing computer programs and applying constructs such as classes and objects. This paradigm also acknowledges the researcher's part in the construction of knowledge about the programming constructs of students, where action, interpretation and reflection are vital.

Educators need to play supportive roles that facilitate the acquisition of appropriate activities as students learn to apply the sum of their knowledge, skills and strategies in programming. OOP is a dynamic and constructive process involving various actions and dimensions. Since its complexity can be overwhelming, we propose a learning repertoire in Figure 1 to serve as an integrated framework to support novices in learning OOP. The content of the repertoire is drawn from the empirical research, which highlights ways in which successful participants solved the programming problem. Subsets of the repertoire can be selected and used for a particular context or task.

Various dimensions are integrated in the repertoire, which explicitly distinguishes between knowledge and skills on the one hand, and strategies on the other. Knowledge and skills form the core. Cognitive knowledge and skills on all levels of Bloom's taxonomy are required for the understanding, designing, coding and testing of a programming problem. Specific emphasis is placed on the higher-order thinking skills such as analysis, synthesis and evaluation. Setting of goals, a high level of motivation, and knowledge about specific tasks are required in the metacognitive domain. In addition, adequate programming knowledge and skills are essential to the ability to complete a new program successfully.

Dynamic interaction, indicated by the arrows in Figure 1, occurs between the core sections of cognitive, metacognitive and problem-solving activities. As an example, successful object-oriented programming requires the 'application' of skills from Bloom's taxonomy, particularly synthesis and evaluation to determine whether a program is correct and to rectify it if not.

The dimensions in Figure 1 are supported by strategies lying outside the core. Students can use these strategies to enhance the acquisition of knowledge and skills, and can apply them during the processes of Construction, Reflection, Selection and Application in OOP. The three dashed arrows on the left, the right and below the core indicate the dynamic and continuous use of cognitive, metacognitive and problem-solving strategies in the first three processes, while the bold arrow above the core relates to the application of these activities in designing new programs and maintaining existing ones.

- Construction
  The use of cognitive strategies can enhance acquisition of the knowledge and skills in Bloom's taxonomy. Rehearsal supports the learning of facts about OOP (knowledge) and the grasping of programming content (comprehension). Elaboration can facilitate the use of previously-learned material in new situations (application) and the decomposition of a problem into subproblems (analysis). The organisation-and-integration strategy can support programmers in combining objects, methods and attributes in a class (synthesis) to program and test the correct solution (evaluation). Object-oriented programmers should be actively involved in their tasks, using prior

knowledge and applying a repertoire of knowledge and skills to help them recall information and organise it in memory during the process of constructing a program.

- Reflection
  Students should reflect on their cognitive processes during OOP by conducting deliberate planning, monitoring and regulation. They should question themselves, discover misconceptions, identify errors and continuously modify their programs in order to succeed. Such reflection places them in control of the programming task as they explicitly query the correctness of their code and reflect on their prior thinking to identify errors and correct flaws. Appropriate responses to feedback and the continuous improvement of code help to optimise the solution and to achieve the required outcomes.

- Selection
  The ability to make discerning selections, helps students to choose a suitable problem-solving strategy for a given problem. They may select and apply one or more problem-solving strategies during program comprehension to help them to reach specific goals. For example, effective use of a top-down strategy demonstrates that a student has holistically conceptualised the entire program involving multiple classes, instances, and methods.

- Application
  Finally and, in consolidation, the construction, reflection, and selection of knowledge, skills and strategies must be applied in OOP tasks to develop new programs and maintain existing ones. It is not the intention that every strategy should be applied in every situation. The various forms of knowledge, skills and strategies are relevant to different contexts. Learning to program is an active process of knowledge construction, reflection, and selection of appropriate activities to ensure successful programming.

Learning OOP requires a balanced approach of all the different activities involved. This implies, for example, that the application of Bloom's skills without explicit reflection; or the application of strategies without any analysis, synthesis and evaluation skills will not support successful completion of a new program. In such cases, students must explicitly query

the correctness of their own code and reflect on their prior thinking to identify the errors and to correct flaws.

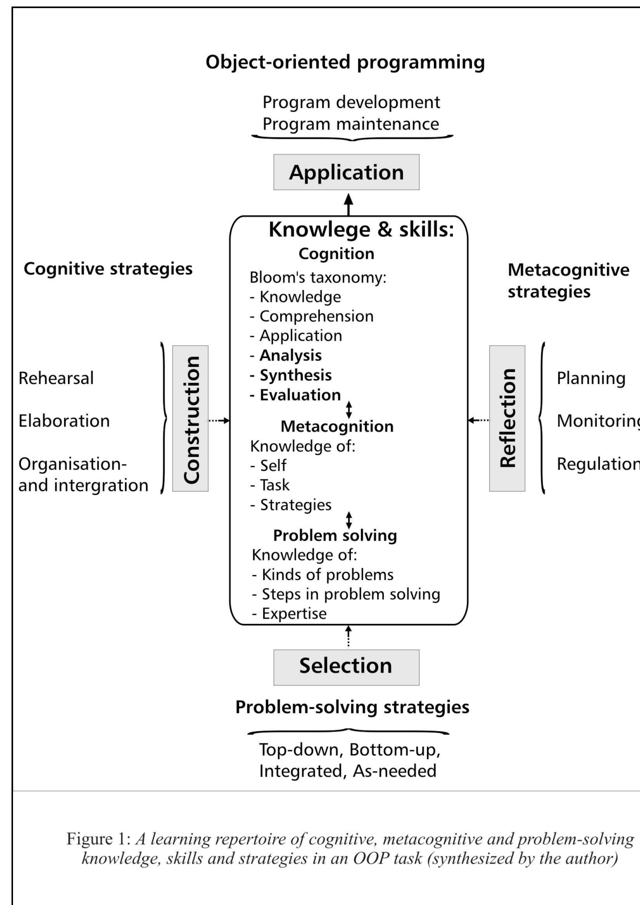


Figure 1: *A learning repertoire of cognitive, metacognitive and problem-solving knowledge, skills and strategies in an OOP task (synthesized by the author)*

Conclusion

To be successful in OOP, programmers require explicit learning both of programming content and higher-order mental activities. The findings of this research, which distinguishes between successful and unsuccessful programmers, indicate the need for a framework to support novice programmers. This should address programming subject matter as well as cognitive, metacognitive and problem-solving knowledge, skills and strategies. Fostering awareness and application of the latter among learners sets a particular challenge to educators (lecturers) to identify creative and effective means of doing so.

We propose a learning repertoire that includes knowledge, skills and strategies used by successful programmers. In order to apply this, various activities should occur during programming to meaningfully construct, explicitly reflect on, and critically select appropriate knowledge, skills and strategies to understand, design, code and test high quality programs. This involves the integration of specific cognitive, metacognitive and problem-solving techniques in a balanced manner. Although this framework focuses mainly on OOP, we believe that it can also be applied to support students in other programming paradigms, such as procedural programming. However, due to the particular complexities of OOP, the framework focuses specifically on a holistic view where various different decisions are required in programming one or more classes.

Future work will concentrate on the role of a lecturer or facilitator in the explicit teaching of the required knowledge, skills and strategies, supporting them in creating an educational environment in which the learning repertoire can be effectively applied. The development of assessment criteria to test the effective application of the activities of the learning repertoire in an OOP task should further support the students.

Glossary

Novice: a person who is inexperienced and new in a particular field
Expert: a knowledgeable person with superior skills in a particular field

References

Ala-Mutka, K 2004. Problems in Learning and Teaching Programming – a literature study for developing visualizations in the Codewitz-Minerva project. Retrieved July 2006, from http://www.cs.tut.fi/~edge/literature_study.pdf

Bergin, S, R Reilly & D Traynor 2005. Examining the Role of Self-Regulated Learning on Introductory Programming Performance. *ICER* 2005:81-86.

Bloom, BS, DR Krathwohl & BB Masia 1973. *Taxonomy of Educational Objectives. Book2: Affective Domain*. London:Longman Group.

Carbone, A, IJ Mitchell, R Gunstone & AJ Hurst 2002. Designing programming tasks to elicit self management metacognitive behaviour. *International Conference on Computers in Education, (ICCE 2002),* Conference Suite, North Harbour Stadium, Auckland, New Zealand.

Cohen, J 1988. *Statistical Power Analysis for the behavioural Sciences.* (2nd ed.). Hillsdale, NJ:Erlbaum.

Concise Oxford English Dictionary 2004. Oxford:Oxford University Press.

Corritore, CL & S Wiedenbeck 2000. Direction and Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study *IEEE Computer Society*. 139-148.

De Villiers, MR (Ruth). 2005. Interpretive research models for Informatics: action research, grounded theory, and the family of design- and development research. *Alternation* 12.2: 10-52.

Edwards, SH 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education:26-30.

Ellis, SM & HS Steyn 2003. Practical significance (effect sizes) versus or in combination with statistical significance (p-values). *Management Dynamics* 12.4:51-53.

Flavell, JH 1979. Metacognition and Cognitive Monitoring. A New Area of Cognitive Developmental Inquiry. *American Psychologist*. 34.10:906-911.

Garrido, JM 2003. *Object-Oriented Programming. From Problem Solving to Java*. Massachusetts:Charles River Media, Inc.

Glaser, BG & AL Strauss 1967. *The Discovery of Grounded Theory. Strategies for Qualitative Research.* London:Weidenfeld and Nicolson.

Glaser, R 1999. Expert knowledge and processes of thinking. In: McCormick, R, C Paechter (Eds.). *Learning and Knowledge*. London:Paul Chapman.

Govender, I & D Grayson 2006. Learning to program and learning to teach programming: A closer look. *ED-Media 2006 Proceedings*:1687-1693.

Gravill, JI, DR Compeau & BL Marcolin 2002. Metacognition and IT: The influence of Self-Efficacy and Self-Awareness. Eighth Americas Conference on Information Systems. 1055-1064.

Gu, PY 2005. Learning Strategies: Prototypical Core and Dimensions of Variation. Retrieved August 2006, from http://www.crie.org.nz/research_paper/Peter_Gu.pdf

Hertzog, C & AE Robinson 2005. Metacognition and Intelligence. In: Wilhelm, O & RW Engle (Ed.). *Handbook of Understanding and Measuring Intelligence*. London:Sage Publications.

Jackson RB & JW Satzinger 2003. Teaching the Complete Object-oriented Development Cycle, Including OOA and OOD, with UML and the UP. *EDSIG*:1-17

Jonassen, DH 2004. *Learning to solve problems: an instructional design guide*. San Francisco:Pfeiffer.

Koriat, A 2002. Metacognition research: an interim report. In: Perfect, TJ & BL Schwartz (Ed.). *Applied Metacognition*.UK: Cambridge University Press:261-268.

Schunk, DH 2000. Learning Theories. *An Educational Perspective*. (3rd ed.). New Jersey:Merrill Prentice-Hall.

Sebesta, RW 2004. *Concepts of Programming Languages* (6th ed.). Boston:Pearson Addison Wesley.

Stamouli, I & A Huggard 2006. Object-Oriented programming and Program Correctness: The Student's Perspectives. *ICER:*109-118.

Sternberg, RJ 2006. *Cognitive Psychology* (4th ed.). United Kingdom:Thomson Wadsworth.

Steyn, HS (jr) 2002. Practically significant relationships between two variables. *SA Journal of Industrial Psychology* 28.3:10-15.

Thompson, B. 2001. Significance, effect sizes, stepwise methods, and other issues: Strong arguments move the field. *Journal of Experimental Education* 70: 80-93.

Zant, RF 2005. Problem Analysis and Program Design: Using Subsystems and Strategies. Retrieved June 2006, from http://isedj.org/isecon/2001/39b/ISECON.2001.Zant.pdf

Zhang, X 2005. Analysis-based techniques for Program Comprehension. Retrieved July 2006, from www.cs.uoregon.edu/~xzhang/documents/AreaExam-long%20version/position.pdf