Attributes contributing to the effective use of quality appraisal techniques by novice programmers

by

Guillaume Nel

Thesis submitted in fulfilment of the requirements for the degree

PHILOSOPHIAE DOCTOR
(Computer Information Systems)

in the Faculty of Natural and Agricultural Sciences

Department of Computer Science and Informatics

UNIVERSITY OF THE **FREE STATE**
UNIVERSITEIT VAN DIE **VRYSTAAT**
YUNIVESITHI YA **FREISTATA**

Bloemfontein - South Africa

January 2020

Promoter: Prof JC Cronje

Declaration

I, Guillaume Nel, hereby declare that the thesis titled "*Attributes contributing to the effective use of quality appraisal techniques by novice programmers*" is the result of my own independent investigation and that all the sources I have used or quoted have been indicated and acknowledged by means of complete references. I further declare that the work is submitted for the first time at this university/faculty towards the *Philosophiae Doctor degree in Computer Information Systems* and that it has never been submitted to any other university/faculty for the purpose to obtain a degree. I also declare that I am aware that the copyright of this thesis is vested in the University of the Free Sate.

[illegible] 27 January 2020

……………………………….... ……………………………….....

Signature **Date**

Acknowledgements

I would like to express my most sincere gratitude and appreciation to the following individuals:

- My wife, Liezel, for her endless love, patience and support.
- My three four-legged girls for providing welcome distractions during difficult times.
- All members of my family for their love, prayers and support.
- My fellow PSP instructors from the Carnegie Mellon Software Engineering Institute (SEI) and the Johannesburg Centre for Software Engineering (JCSE) for introducing me to the world of measured software quality improvements.
- My colleagues at the Central University of Technology, Free State for their constant interest and support.
- All the students who have participated in the various research activities that formed part of this research study.
- Suezette Opperman for the language editing of this thesis report.
- Prof. Johannes Cronje for his wisdom, positive energy, insightful comments and suggestions.
- Above all, to God my creator, for his care, grace and wisdom.

Guillaume Nel

Bloemfontein
January 2020

Table of Contents

Table of Figures

Table of Tables

Summary

Over the past 40 years, numerous studies have been conducted to evaluate the programming performance of undergraduate Computer Science (CS) students (often referred to as ‘novice programmers’) and to identify possible reasons for the low quality of programs these novices developed. Humphrey (1999) proposes that CS educators must shift their focus from the students’ programs to the data of the processes the students use. Hilburn and Towhidnejad (2000) suggest that the quality of student programs could be improved if instructors taught their students “a software development process that emphasise quality techniques, methods, and analysis”. They also propose the Personal Software Process (PSP) as a good candidate to contribute towards the implementation of a curriculum that focuses on software quality. A number of CS researchers have reported on attempts to incorporate PSP principles and strategies in order to improve the quality of novice programmers’ software development practices. Based on numerous implementation and adoption challenges identified by these researchers - especially with regard to students’ use of quality appraisal techniques (i.e. designs, code reviews, design reviews and quality measures) - there have been various calls for further investigations into attributes that might influence students’ use of PSP principles and strategies.

In response to these calls, the overall aim of this research study was to explore the attributes that could potentially influence novice programmers’ effective use of quality appraisal techniques in an educational context.

In addressing the stated aim, this research study followed a mixed-method approach based on Plowright’s (2011) Framework of Integrated Methodologies (FraIM). This study was divided into three phases in order to distinguish between the three main sources of data (cases). Based on the emergent nature of the research design followed in this study, the data, evidence and conclusions of each research phase were used to make final decisions regarding the structure and focus of the next phase.

In Phase 1, a survey approach was followed to evaluate the quality of the typical software development processes followed by novice programmers. By using the PSP quality improvement framework as an evaluation tool, the results of Phase 1 revealed that the novice programmers typically used a code-and-fix development strategy, their designs were almost non-existent, they mostly relied on testing to remove defects and they did not make use of measurements to gain insight into their development processes.

In a follow-up investigation, an integrated experimental case study approach was followed in Phase 2 to form a better understanding of the differences between novice programmers' perceived and actual development processes [including their use of quality appraisal techniques (QATs)] through the collection of both actual process measurement data and narrative data. The collected data was also used to identify nine attributes that could potentially influence novice programmer's use of QATs: understanding of development phases, technical programming skills, accuracy of measurement data, ability to find and fix defects, design skills, design review and code review skills, value of process measurement data, motivation orientation, and achievement goal orientation. It was, however, concluded that to ensure effective use of QATs, programmers first need to make the decision to change their current software development processes.

This issue was addressed in Phase 3 where a case study approach was followed to identify factors that could influence novice programmers' intent to adopt QATs. It was revealed that novices' intentions to adopt QATs are driven by six factors: ease of use, compatibility, usefulness, result demonstrability, subjective norm and career consequences.

By combining the results of Phase 2 and Phase 3, this research study therefore identified 15 attributes that could potentially influence novice programmers' effective use of QATs.

Keywords:
Software engineering, personal software process, software design, code review, design review, process measurement, early defect removal, defect prevention, novice programmers, Computer Science education

Chapter 1: Introduction

Software quality in its most common form can be described as the lack of defects in the end-product that evidently leads to conformance to requirements from the customer’s perspective (Kan et al., 1994). Software quality improvement efforts need to consider both product quality and process quality (Kan et al., 1994). Defects can have a major impact on both of these quality dimensions (Humphrey, 2005). The Personal Software Process (PSP) is a self-improvement framework based on the quality principles of Total Quality Management (TQM). The PSP was specifically developed to assist individual software developers in continuously improving their personal software development processes (Humphrey, 1995) through the use of process measurements (Deming, 1986). The PSP process improvement strategies are based on the following principles: developers must use a defined process and measurement data to improve their performance; and every developer is unique and must therefore use personal data to plan his/her work. The PSP quality management strategy is based on the following principles: personal responsibility for individual quality, early defect removal, and defect prevention. These principles are addressed through the use of specific strategies, namely design reviews, code reviews, design templates and quality measures. These strategies can therefore be collectively referred to as quality appraisal techniques (QATs). Since the introduction of PSP, various industrial success stories have been published. However, the use of these quality improvement strategies by higher education students need to be explored in more detail.

1.1 Research Problem and Question

Over the past 40 years, numerous studies have been conducted (Lister et al. 2004; McCracken et al., 2001; Soloway et al., 1982; Utting et al., 2013) to evaluate the programming performance of undergraduate Computer Science (CS) students (often referred to as ‘novice programmers’) and to identify possible reasons for the low quality of programs developed by these novices. Humphrey (1999) proposes that Computer Science educators must shift their focus from the programs that the students create to the data of the processes the students use. Hu (2016) also suggests that software development courses should have a stronger focus on process knowledge. Hilburn and Towhidnejad (2000) suggest that the quality of student programs could be improved if instructors taught their students “a software development process that emphasises quality techniques, methods, and analysis” (p. 171). They also propose the PSP as a good candidate to contribute towards the implementation of a curriculum that focuses on software quality. A number of CS researchers have reported on attempts to incorporate PSP principles and strategies in order to improve the quality of novice programmers’ software development practices (Börstler et al., 2002; Bullers, 2004; Carrington et al., 2001; Grove, 1998; Hou & Tomayko, 1998; Jenkins & Ademoye, 2012; Prechelt, 2001; Prechelt & Unger, 2001; Rong et al., 2012; Rong et al., 2016; Runeson, 2001; Towhidnejad & Salimi, 1996; Williams, 1997). In some studies, the novice programmers struggled to capture accurate and reliable data (Carrington et al., 2001; Grove, 1998; Prechelt, 2001; Towhidnejad & Salimi; 1996). There were also cases where the novices completely abandoned the use of PSP practices (Börstler et al., 2002; Bullers, 2004; Carrington et al., 2001; Hou & Tomayko, 1998; Towhidnejad & Salimi, 1996; Williams, 1997). In this regard, Prechelt and Unger (2001) note that students are unlikely to realise the potential benefits of PSP if they have to motivate themselves to adopt these principles as part of their natural process. Consequently, Prechelt and Unger (2001) call for further investigations into the technical, social and organisational attributes (beyond the level of training and infrastructure provided) that might influence students’ use of PSP principles and strategies. Rong et al. (2012) also call for future investigations into other factors that might influence the use of code reviews.

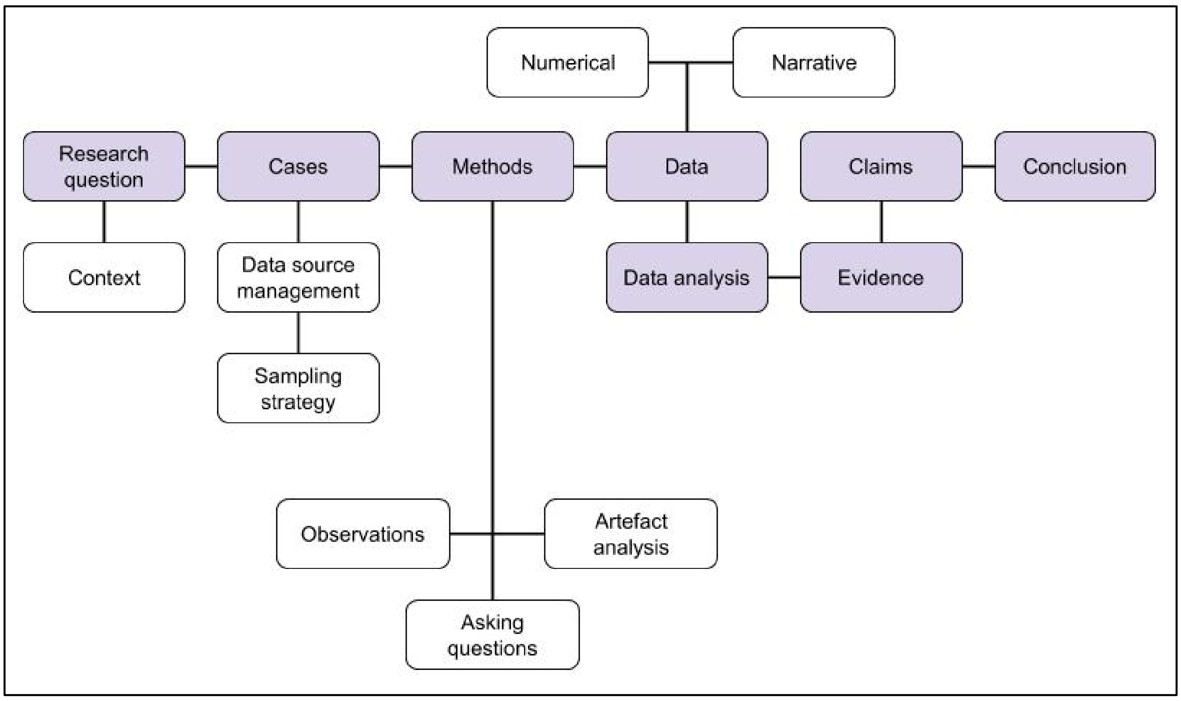
In response to the calls by Prechelt and Unger (2001) and Rong et al. (2012), the aim of the research study described in this thesis was to explore the attributes that could potentially influence novice programmers’ effective use of quality appraisal techniques in an educational context. This study was directed by the following main research question:

> *What are the attributes that could potentially influence novice programmers’ effective use of quality appraisal techniques?*

In the context of this study, an attribute refers to any feature, skill, quality or characteristic that could affect the effective use of quality appraisal techniques by novice programmers for the purpose of personal software development process improvement. These attributes can be personal, behavioural, technical and/or environmental in nature.

1.2 Research Design

In answering the stated main research question, this research study followed a mixed-method approach based on the Framework of Integrated Methodologies (FraIM) as suggested by Plowright (2011). The FraIM provides a basic structure (as illustrated in Figure 1-1) for the execution of a research project and guides the researcher in making various methodological decisions. This framework is of particular relevance in any study of social and educational phenomena that requires the integration of various research processes into a “coherent whole”. It is therefore possible to combine different research approaches into a single study. In following the FraIM, the researcher is also not required to take a philosophical stance at the beginning of the study. Consequently, the researcher is encouraged to have a “more responsive, flexible and open-minded attitude” (Plowright, 2011) in answering the stated research question(s). This fits in perfectly with the exploratory nature of this study.



(Source: Adapted from Plowright, 2011, p. 9)

Figure 1-1: Plowright's FraIM structure as used in this study

The following important differences between the FraIM and traditional mixed-method research approaches should be noted:

- The main research question of the study is formulated within the relevant contexts that influenced the choice of research topic.
- Although the FraIM can be used as a linear process, it is also possible to use a process of iteration where the researcher moves back and forth between the different stages as the research study progresses and research plans are adapted.
- The term ‘cases’ is used to refer to the sources that will provide the data for the research activity. In situations where these data sources are individuals, they can also be referred to as participants.
- The data source management strategy describes the approach that will be used to manage the sources of data. The choice of data source management strategy is influenced by three criteria: the number of cases, the degree of control that the researcher has in allocating cases to different groups, and the degree of naturalness of these groupings.
- The choice of data source management strategy will influence sampling decisions regarding which cases to include as part of the research activity.
- Methods describe the ways in which data will be generated and collected from the selected cases (i.e. the ‘data collection methods’). The FraIM classifies three methods of data collection: observations, asking questions and artefact analysis.
- The FraIM rejects any use of the “Q” words - qualitative and quantitative. In referencing the resulting data, the terms ‘numerical’ and ‘narrative’ are used instead. Each of the three methods (observations, asking questions and artefact analysis) can be used to collect both numerical and narrative data.

The design of this research study can also be described as emergent since the researcher did not present a fixed research plan at the beginning of the study - a characteristic that is typical of traditional qualitative designs (Creswell, 2014). An emergent design allows the researcher to adapt the research processes at any stage (conceptualisation, data collection, data analysis or composition) of the study based on the discovery of new ideas, concepts or findings (Palithorpe, 2017). Such a design encourages the researcher to react to unforeseen nuances in the data and often results in a richer set of data.

This study was divided into three phases in order to distinguish between the three main sources of data (cases) (Plowright, 2011). Based on the emergent nature of the research design followed in this study, the data, evidence and conclusions of each research phase were used to make final decisions regarding the structure and focus of the next phase. Consequently, the following research questions were addressed in each of the phases:

Phase 1: Evaluating the quality of novice programmers' typical software development processes

RQ1.1: *What is the quality of the typical software development processes followed by novice programmers?*

Phase 2: Understanding novice programmers' actual development processes and use of QATs

RQ2.1: *How do novice programmers' perceived software development processes (including their use of QATs) differ from their actual processes?*

RQ2.2: *What are the attributes that could potentially influence novice programmers' use of QATs?*

Phase 3: Factors influencing novice programmers' intent to adopt QATs

RQ3.1: *What are the factors that could influence novice programmers' intent to adopt QATs?*

These research questions are subsidiary to the main research question (as stated in Section 1.1). Since each of these phases focused on a separate research activity, the methodological details of each phase are reported in the actual phase descriptions (as presented in Chapters 3, 4 and 5 of this report).

1.2.1 Ethical considerations

Ethical approval for this study was obtained from the relevant institutional ethics committees (Reference number: UFS-HSD2015/0115). Although the procedures of this research study presented a relatively low-risk of emotional or physical harm to participants, specific methods were employed to mitigate the potential risks. Throughout all research activities, the participants were informed of the purpose of the activity and assured of confidentiality and anonymity of all their responses. Participants were also assured that their participation is voluntary and that there would be no academic implications if they chose to withdraw from the research activity. In cases where participants reported any specific problems regarding the procedures or nature of the interventions, steps were taken to responsibly manage such situations in a caring and fair way that placed the needs of the participant(s) first (McMillan & Schumacher, 2006). The specific participant information sheets and forms of consent used are referenced as part of the discussion of each of the three research phases.

1.2.2 Validity and reliability

The following was done to enhance the validity of the findings (Creswell, 2014; McMillan & Schumacher, 2006):

- Multiple methods of data collection (e.g. asking questions and analysing artefacts) were used where relevant to allow the triangulation of data from different sources.
- In the description of narrative data, rich and thick descriptions were used to convey the findings.
- Low inference descriptors were used in questionnaires to ensure uniform interpretation of questions.
- An existing software application (Process Dashboard©) was used to collect “mechanically recorded” student process data.
- During analysis of numeric data, outlier data was identified from the collected data to ensure that the sample contained no negative or discrepant data.

The following measures were taken to ensure the reliability of the findings (Creswell, 2014; Saunders et al. 2016):

- Transcripts of the focus group discussion (Phase 2) were checked and re-checked to ensure that it did not contain any mistakes.
- The researcher reported the particulars of each research activity in as much detail as possible to ensure that others would be able to replicate the activities in a similar way if they wanted to do so.

The researcher also enhanced reflexivity (McMillan & Schumacher, 2006) by keeping a reflection journal in which he recorded all decisions made during the research study and the rationale behind them.

1.3 Research Context

In order to form a deeper understanding of the origin of the main research question (as stated in Section 1.1) and as a starting point for following the FraIM (see Section 1.2), it is necessary to describe the various contexts that influenced the choice of research topic for this particular study.

1.3.1 Professional context

After working in industry as a network technician and programmer, I started my lecturing career in 1998 in the Department of Information Technology at the Central University of Technology, Free State (CUT – formerly known as the Free State Technikon). For the first few years, I was mostly responsible for presenting programming courses to engineering students on 1st, 2nd and 3rd year level. Later, I became more involved by teaching programming to CS students enrolled in the Web Development stream as well as presenting the 4th year Software Engineering course. I am currently presenting the introductory programming course (CS1) to a large group of students (300 students in 2019). While I was presenting the Software Engineering course for the first time, I realised that even after having completed three years of programming courses, the students were generally unable to successfully combine the knowledge they have gained from three years of study in order to design and develop fully functional programs.

In 2009, as part of an initiative of the Johannesburg Centre for Software Engineering (JCSE), I was selected to join a group of programmers who enrolled for the basic Personal Software Process (PSP) course. The course was presented by staff members of Carnegie Mellon’s Software Engineering Institute (SEI), and was my first introduction to Watts Humphrey’s PSP framework. For the first time in my career I realised what actually determines the quality of a software product, as well as the significant impact that early defect removal could have on software quality. After having completed the PSP course, I also did the PSP Instructor course and presented numerous PSP courses to programmers from one of the major commercial banks in South Africa (on behalf of the JCSE). In working with these industry programmers, I noticed that while some of them easily adapted their personal development processes to fit in with the PSP framework, others really struggled to do so. These “struggling” programmers had numerous reasons (or excuses) as to why they were unable to follow the prescribed development processes. The eye-opener was in 2011 when the JCSE provided funding for the PSP training of 20 CS graduates from CUT. During the training I became aware that these students not only struggled to use the PSP principles (similarly to what I have observed with the industry programmers), but also lacked basic programming skills. Based on my PSP experiences, I also started to incorporate some of the basic PSP principles in my Software Engineering course in an attempt to foster quality management and process awareness among the students.

From a personal perspective, I was compelled to undertake this study as I wanted to know why students are finding it so difficult to use good software development processes and PSP principles. I hoped that this research study would bring me one step closer to my ultimate goal in life - to convince students to use QATs to improve the quality of their software programs.

From a practical perspective, this research had to be conducted because novice programmers tend to develop software applications of low quality (full of defects). There is therefore a need to understand (1) the software development processes followed by these novices, and (2) the problems they experience when following quality improvement principles.

1.3.2 Organisational context

The context of this study was the Information Technology department at a selected South African University of Technology (UoT). At the time when this study was conducted, the department offered a three-year diploma course in Information Technology with specialisation in either software development or web development. In the first year of study, all students took the same modules, which included one year-long programming module (OPG1) with C# as implementation language within the Microsoft Visual Studio integrated development environment (IDE). This introduction to programming module could be regarded as a typical CS1 module - as designated by the Association for Computing Machinery (ACM[1]). In their second and third years of study, the software development students continued with more advanced C# programming modules (OPG2 and OPG3). These students also had to enrol for two technical programming modules (TPG2 and TPG3) that focused on JAVA programming and mobile application development with Android.

After completion of their first year, the web development students did not take further C#-specific programming modules. Instead, they registered for modules that focused on basic web page development using HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) (in WEB2 and INP2), and Internet programming using the ASP.NET framework with C# (in WEB3 and INP3).

All students, regardless of their chosen speciality stream, also had to take an Information Systems module in each of their three years of study. The first-year Information Systems module (SYS1) covered basic computer literacy skills and the working of computers. The second-year module (SYS2) included an introduction to database concepts as well as an introduction to Software Engineering principles (focusing mostly on basic analysis and design strategies). The third-year Information Systems module (SYS3) covered Project Management principles. The students received no additional training that focused specifically on the implementation of Software Engineering principles other than this basic introduction. The students in both streams also took several other modules of which the content were not directly related to the context of this study. Although we commonly refer to our students as ‘IT

[1] http://www.acm.org

students', the overall syllabus is more closely aligned with the ACM Computer Science curriculum. In the context of this research study it would therefore be more appropriate to refer to them as 'CS students'. These students can also be regarded as novice programmers since they do not have any industry-related software development experience.

1.3.3 National context

Since this research study was conducted in the context of a South African University of Technology (UoT), some background regarding the structuring of Higher Education institutions in South Africa are relevant here. South Africa has three types of institutions where students can study after having completed high school (Grade 12), namely universities, UoTs, and Technical and Vocational Education and Training (TVET) colleges.

Universities mostly offer bachelor's degree courses that take three to four years to complete. Although entry requirements for courses vary considerably, university requirements tend to be very specific and generally require higher achievements in Grade 12 than the other two types of institutions. Most university courses focus on providing students with theoretical training in a specialised field. After completion of their undergraduate degrees, students can continue with postgraduate studies up to doctoral level.

UoTs offer mainly certificate and diploma courses, but there are also options for further studies towards bachelor's degrees and postgraduate degrees. After completion of a three-year diploma course students can enrol for a one-year BTech degree course[2]. It would therefore take students at least four years to complete an undergraduate degree. Since students initially enrol for certificate or diploma courses, the entry requirements at UoTs are generally lower than those of the traditional universities. The UoTs are typically more focused on presenting career-directed courses and

[2] From 2020, qualifications at all South African Higher Education Institutions must be aligned with the Revised Higher Education Qualifications Sub-Framework (HEQSF). The purpose of this framework is to ensure alignment of qualification levels across all institutions. This will allow students to move more easily between qualifications and institutions if needed. Consequently, all BTech degrees are being phased out and replaced with advanced diplomas. Under the new framework, advanced diplomas and Bachelor's degrees will be on the same qualification level.

conducting applied research. Traditionally, these institutions were also more focused on community engagement.

TVET colleges provide various types of training courses, from a few months to three years in duration, with students receiving a certificate at the successful completion of the course. The focus of these institutions is to provide students with the necessary education and training to work in technical or vocational fields. In some cases, students can continue their studies at a UoT after obtaining a TVET certificate.

1.3.4 Theoretical context

Total Quality Management (TQM) is an industrial quality management system based on the philosophies of five quality gurus: Deming, Juran, Feigenbaum, Ishikawa and Crosby (Neyestani, 2017). The aims of TQM are to obtain total customer satisfaction, to continuously enhance product quality through variation control, to create a companywide quality culture, and to manage all quality improvements by means of a goal-oriented measurement system (Kan et al., 1994). Many of the TQM principles can be related to software quality. Software quality in its most common form can be described as the lack of defects in the end-product which evidently leads to conformance to requirements from the user’s perspective (Kan et al., 1994). The Software Engineering discipline was established with the aim “to create defect-free software, delivered on time and within budget, that satisfy the client’s needs” (Schach, 2011, p. 4). Within this discipline, various best practices (e.g. development process models, methods, approaches, tools, technologies, measures, metrics, and quality parameters) are recommended for achieving quality in software projects (Kan et al., 1994; Pressman, 2005; Schach, 2011; Sommerville, 2004). These practices are typically taught to CS students as part of their undergraduate studies.

The TQM philosophy also formed the foundation for the development of numerous software quality improvement frameworks (Kan, 2003). One such framework is the Capability Maturity Model (CMM) that was developed to address software process improvement, software process assessment and software capability evaluations within both large and small organisations (Paulk et al., 1993). Building on the principles of CMM, the PSP framework was specifically created as a self-improvement process to guide individual software developers in following good development practices

(Humphrey, 1995). Various researchers have reported on their own experiences with the incorporation of PSP principles in educational environments in an attempt to improve the quality of their students' development processes (Börstler et al., 2002; Jenkins & Ademoye, 2012; Towhidnejad & Salimi, 1996; Williams, 1997). Since PSP is a process improvement framework, none of these studies have recognised the potential value of the PSP framework as an evaluation model to assess the quality of individual development processes. However, several attempts have been made to develop models for the evaluation of the quality of students' software designs (Chen et al., 2005; Eckerdal et al., 2006a; Eckerdal et al., 2006b; Hu, 2016; Loftus et al., 2011).

Humphrey (1999) claims that one of the biggest challenges in software development is to persuade software developers to use effective methods. Software developers tend to stick to a personal process that they have developed from the first small program they have written, and it is difficult to convince them to adopt better practices. Various researchers have reported on the challenges they experienced in motivating their students to adopt PSP practices (Börstler et al., 2002; Bullers, 2004; Carrington et al., 2001; Hou & Tomayako, 1998; Towhidnejad & Salimi, 1996; Williams, 1997). There also have been numerous calls for further investigations into the factors (other than training) that might influence the adoption of PSP methods (Prechelt & Unger, 2001; Rong et al., 2012). A number of authors have also proposed the use of narrative data to gain better insight into students' development processes (Hu, 2016; McCracken et al., 2001). There are also numerous theoretical models that can be used to examine individual intentions to adopt technology tools and development methodologies (Davis, 1989; Moore & Benbasat, 1991; Riemenschneider et al., 2002; Venkatesh & Davis, 2000; Thompson et al., 1991).

The conceptual framework of this research study, based on the various research contexts discussed above, is illustrated in Figure 1-2.

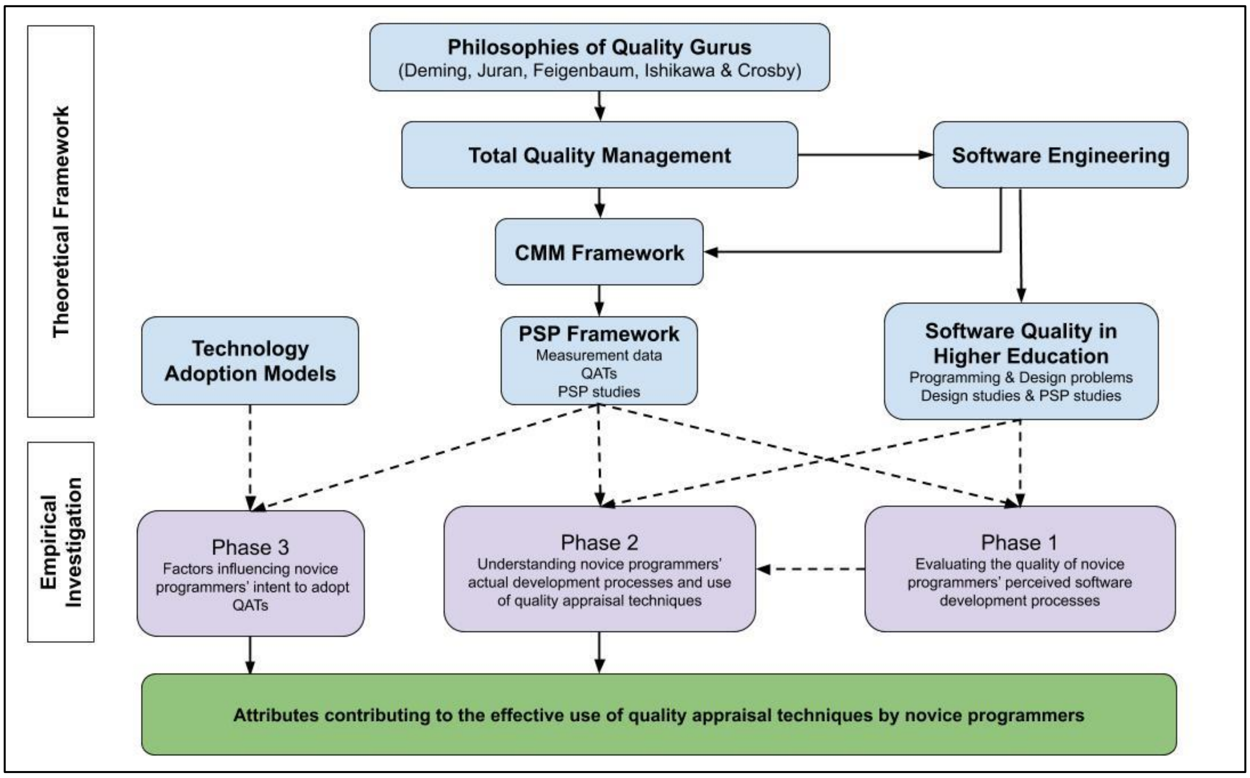

Theoretical Framework
Empirical Investigation
Philosophies of Quality Gurus
(Deming, Juran, Feigenbaum, Ishikawa & Crosby)
Total Quality Management
Software Engineering
CMM Framework
Technology Adoption Models
PSP Framework
Measurement data
QATs
PSP studies
Software Quality in Higher Education
Programming & Design problems
Design studies & PSP studies
Phase 3
Factors influencing novice programmers' intent to adopt QATs
Phase 2
Understanding novice programmers' actual development processes and use of quality appraisal techniques
Phase 1
Evaluating the quality of novice programmers' perceived software development processes
Attributes contributing to the effective use of quality appraisal techniques by novice programmers

Figure 1-2: Conceptual framework for this study

1.4 Outline of the Thesis

This thesis report comprises six chapters.

Chapter One explains the outline of the research. This chapter includes a brief explanation of the research problem and states the aim as well as the main research question that guided the research activities of this study. Moreover, an explanation of the integrated mixed-method design followed in this study is provided. The various contexts that influenced the choice of research topic are also described. Finally, a conceptual framework of the study is presented.

Chapter Two provides a literature overview and background information on the following: software quality management principles and practices, the nature of software quality problems in higher education, the use of PSP principles to improve the quality of student software programs, and technology adoption models.

Chapter Three describes the Phase 1 research activity. The aim of this research activity was to assess the quality of novice programmers' software development processes by using the PSP framework as an evaluation model. As part of the Phase 1 description, the case selection, methods, data collection and data analysis stages are outlined. Results in the form of evidence and claims are also presented.

Chapter Four describes the Phase 2 research activity. The aims of this research activity was twofold: (1) to form a better understanding of the differences between novice programmers' perceived and actual development processes (including their use of QATs) through the use of actual process measurement data (as prescribed by the PSP framework) supplemented by narrative data; and (2) to identify attributes that could potentially influence novice programmers' use of QATs. As part of the Phase 2 description, the case selection, methods, data collection and data analysis stages are outlined. Results in the form of evidence and claims are also presented.

Chapter Five describes the Phase 3 research activity. The aim of this activity was to identify factors that could influence novice programmers' intent to adopt QATs. As part of the Phase 3 description, the case selection, methods, data collection and data analysis stages are outlined. Results in the form of evidence and claims are also presented.

Chapter Six concludes the work by synthesising the empirical findings and outlining implications for research and practice. This chapter also describes the limitations of this study and makes recommendations for future research.

Chapter 2: Literature Review

In order to provide a conceptual and theoretical basis upon which the remainder of this thesis builds, this chapter presents a brief overview focusing on four key areas. Firstly, relevant theoretical background knowledge regarding the software quality management principles and practices that are of particular relevance to this study is presented. Secondly, the nature of software quality problems in higher education is examined. Thirdly, the outcomes of a number of studies in which higher education students have attempted to use PSP principles as part of their software development processes are reviewed. Finally, various theoretical adoption models are described and the applicability of these models to methodology adoption is explored.

2.1 Software Quality Management Background

In this section, a theoretical background regarding the software quality management principles and practices that are of particular relevance to this study is presented. The discussion is divided into four sub-sections that will focus on the following: defining the meaning of quality in the context of software development; defining TQM and discussing the underlying philosophies that contributed to the forming of the TQM movement; defining software engineering and discussing the suitability of TQM principles for the Software Engineering discipline; and providing an overview of software quality improvement frameworks that are built on the TQM philosophy. As part of the final sub-section, a detailed discussion of the PSP framework is provided in order to create conceptual understanding regarding specific principles and practices that are referenced in the remainder of this report.

2.1.1 Software quality

The American National Standards Institute (ANSI) generally regards quality as a “subjective term” where the exact definition depends on the person who is defining it and the context it relates to. According to the *Cambridge English Dictionary online*, the term “quality” (n.d.) is typically used to describe how good or bad something is by referring to its “degree of excellence”. Quality management (according to the ISO 9001:2015 standard) can therefore be described as “the act of overseeing all activities and tasks needed to maintain a desired level of excellence” (ISO, 2015).

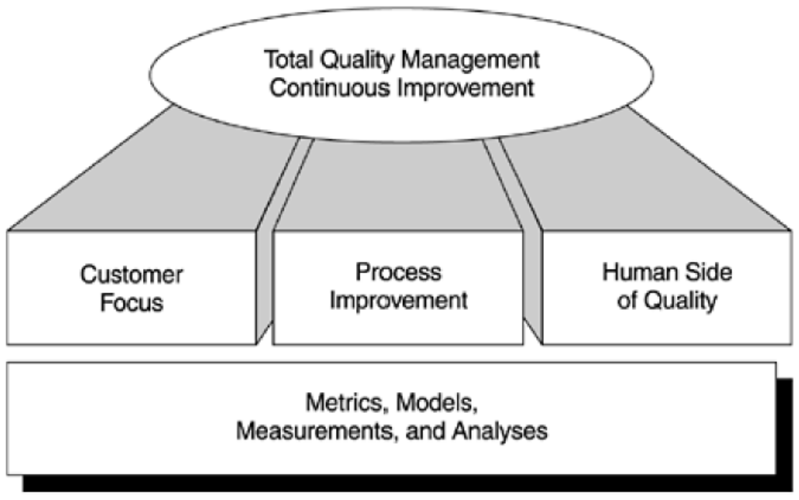
Software quality in its most common form can be described as the lack of defects in the end-product that evidently leads to conformance to requirements from the user's perspective (Kan et al., 1994). Humphrey (2005) acknowledges that defects in software do not always have the highest priority, but argues that if software contains defects that cause inconsistent product performance, "the user will not use it regardless of its other attributes" (p. 136). From a customer satisfaction viewpoint, software products are usually rated on what Juran (1999) describes as "fitness for use" (p. 2.2) parameters. These end-product quality attributes include "capability, usability, performance, reliability, installability, maintainability, documentation and availability" (Kan et al., 1994, p. 6). To increase customer satisfaction, these end-product quality attributes need to be considered throughout the entire software development process. Software developers tend to spend most of their time finding and fixing defects and therefore tend to neglect the attributes that matter to end-users (Humphrey, 2005). Humphrey (2005) therefore argues that even though defects are not the top priority, defect management is essential to product quality. It is, however, necessary to differentiate between end-product quality and process quality (Kan et al., 1994). Software is created through a complex development process that involves various stages. Kan et al. (1994) argue that each stage in this process can be regarded as an "internal user" (p. 6) of the previous stage. From this viewpoint, Crosby's (1979, p. 9) "conformance to [user] requirements" definition of quality also relates to process quality. Humphrey (2005) further claims that "product and process quality goes hand in hand" (p. 136).

Software quality can therefore be divided into two categories: product quality and process quality. Defects can have a major impact on both of these quality categories. The underlying philosophies of the TQM framework have been used successfully to improve the quality of various manufacturing industries (Crosby, 1979; Deming, 1986; Feigenbaum, 1983; Ishikawa, 1989; Juran, 1992). Kan et al. (1994) argue that these TQM philosophies could also be used to improve quality in the software development industry. The next section provides a closer look at the key elements and underlying philosophies of TQM.

2.1.2 The TQM philosophy

In 1985, the Naval Air Systems Command defined its Japanese-style management approach as total quality management (TQM) (Kan et al., 1994). The main purpose of TQM is to implement a long term, companywide process with improvement initiatives at all levels (Elshennawy et al., 1991). The early development of the total quality movement was influenced by several significant role players or so-called “quality gurus” (Neyestani, 2017, p. 2): Philip B. Crosby, W. Edwards Deming, Armand V. Feigenbaum, Kaoru Ishikawa and Joseph M. Juran. Despite the different interpretations of TQM, the key elements of this philosophy can be described as follows (Kan et al., 1994) (also see Figure 2-1):

- *Customer focus:* The objective is to obtain total customer satisfaction through analysing needs and requirements, and by measuring and managing customer satisfaction.
- *Process:* The objective is to enhance product quality by reducing process variations and continuously improving business and product development processes.
- *Human side of quality:* The objective is to create a companywide quality culture by focusing on leadership, management commitment, total participation, employee empowerment, and other social, psychological and human factors.
- *Measurement and analysis:* The objective is to manage all continuous quality improvements through a goal-oriented measurement system.



(Source: Kan 2003, p. 1.4)

Figure 2-1: Key elements of total quality management

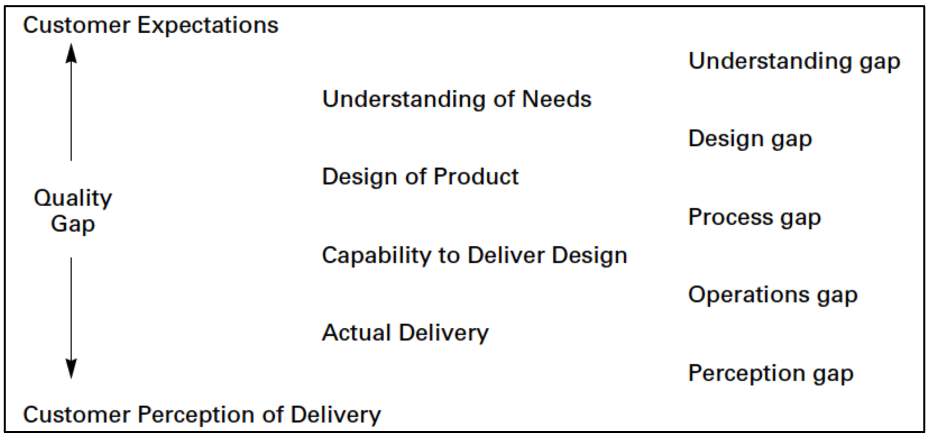
The following discussions will look more closely at the contributions of the “quality gurus” in relation to the key elements of TQM.

2.1.2.1 Customer focus

In the first of his four absolutes for quality management, Crosby (1979) states that “the definition of quality is conformance to requirements” (p. 9). He argues that his “doing it right the first time (DRIFT)” (Crosby, 1984, p. 59) principle can only be achieved through clearly defined requirements that are understood by both customers and suppliers. In this regard, Ishikawa (1989) includes the customer as part of the development process. According to Juran (1999), quality can be regarded as the “features of products” (p. 2.1) that provide customer satisfaction. Juran (1999), however, warns that “conformance to specification” (p. 2.2) would not necessarily equate to customer satisfaction. Satisfaction comes from customer expectations in a product, while dissatisfaction comes from deficiencies in the product. In order to manage quality, Juran (1999) introduces a trilogy consisting of three basic managerial processes: quality planning, quality control and quality improvement. Quality planning is a structured process that leads to the development of products that ultimately satisfy customer needs (Early & Coletti, 1999). The biggest problem with quality planning lies in the “quality gap” (Early & Coletti, 1999, p. 3.2) between the customer’s expectations and the customer’s perception. This gap (see Figure 2-2) is caused by smaller component gaps, namely lack of understanding customer needs, lack of design, lack of a process to consistently create products conforming to a design, lack in means by which processes are operated and controlled, and the customer’s perceived perception. Juran’s quality planning process describes very specific processes, methods, tools and techniques to minimise these component gaps (Early & Coletti, 1999).

2.1.2.2 Process

Deming argues that most production problems originate with the process - something that can be controlled by using statistics (Gitlow & Gitlow, 1987). His quality management philosophy focuses on the continuous improvement of all processes. If the process is performed correctly, there will be less need for rework and quality goods can be produced at a lower cost (Gitlow & Gitlow, 1987). In his second quality absolute, Crosby (1984) states that “the system of quality is prevention” (p. 66). He



(Source: Early & Coletti, 1999, p. 3.2)

Figure 2-2: The quality gap and its constituent gaps

argues that appraisal in the form of inspection and testing is the most visible expense of quality practice and that errors should be eliminated through prevention. Juran (1999) defines quality as "freedom from deficiencies" (p. 2.2). In this regard, higher quality processes would not only reduce error rates, shorten production time, and lead to a reduction in waste, rework, inspection and test, but also reduce aftersales comebacks, warranty charges and customer dissatisfaction. From this viewpoint, higher quality will cost less in the end. Quality control, the second process in Juran's trilogy, is used to evaluate actual performance, compare actual performance to goals, and act on the difference to restore process stability (Juran & Godfrey, 1999). Feigenbaum (1983) believes that one of the basic elements of total quality control is sound technological methods that will help to increase customer satisfaction (Stevens, 1994) through the continuous improvement of all processes in the company (Jancikova & Brycht, 2009). Feigenbaum (1983) regards quality as something that "must be designed and built into a product; it cannot be exhorted or inspected into it" (p. 77). His approach to total quality control is a prevention-based system that places emphasis on product, service and process design as well as the streamlining of source activities.

2.1.2.3 Human side of quality

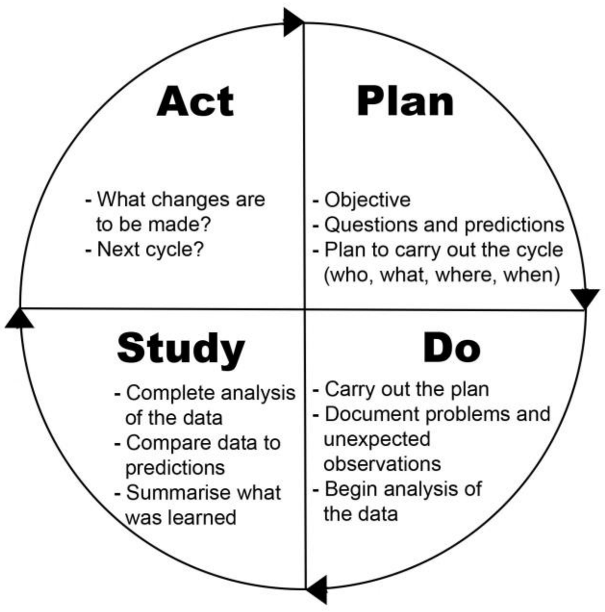
Feigenbaum regards effective human relations that focus on "building up employee responsibility for, and interest in, product quality" (Feigenbaum, 1983, p. 6) as another basic element of total quality control. He believes that total quality control lies in the "human hands" responsible for creating the products and that these individuals need to be “guided in a skilled, conscientious, and quality minded fashion” (Feigenbaum, 1983, p. 6). In this regard, Ishikawa introduces the concept of Quality Circles (QC) - quality control workgroups that discuss and improve their work processes on a regular basis (Krüger, 2001). This technique requires every member of the workforce to be committed to quality. He also believes that TQM “begins with education and ends with education” (Ishikawa, 1989, p. 68). The third process in Juran’s trilogy, quality improvement, focuses on the actual actions necessary to adjust the level of performance (Juran & Godfrey, 1999). Juran (1992) claims that humans follow a similar process to adjust their individual performance when they apply the concept of self-control. In this regard, Crosby (1984) states in his third absolute that “the performance standard is zero defects” (p. 74). He claims that a performance standard is a device that helps individuals to recognise the importance of all parts of a company coming together to make everything turn out well. He further argues that people make mistakes because of two factors: lack of knowledge and lack of attention. While knowledge can be measured and deficiencies corrected, lack of attention is an attitude problem. Individuals need commitment to monitor themselves in every detail to avoid errors, which will enable them to move closer to zero defects.

2.1.2.4 Measurement and analysis

As part of this fourth quality absolute, Crosby (1984) states that “the measurement of quality is the price of nonconformance” (p. 85). According to Crosby, the price of non-conformance is regarded as the cost of fixing things that were done incorrectly, while the price of conformance can be seen as the cost of doing things right the first time. Feigenbaum (1983) regards “unavailability of meaningful data” (p. 109) as the major factor leading to the misconception that higher quality involves higher cost. In this regard he suggests that the costs of control should be measured in two areas (Feigenbaum, 1983, p. 111):

- *Appraisal costs* (e.g. testing and inspections) for maintaining the quality level of the product; and
- *Prevention costs* (e.g. quality engineering and quality training of employees) to keep defects and nonconformities from occurring in the first place.

Similarly, cost of failure of control should be measured in terms of internal failure costs (e.g. defects and rework) and external failure costs (e.g. customer satisfaction) (Feigenbaum, 1983). To assist in measurement and analysis, Ishikawa created seven quality control tools: Pareto chart, Cause-and-Effect diagram, Stratification chart, Scatter diagram, Check sheet, Histogram and Control chart (Neyestani, 2017). Ishikawa (1985) believed that 95% of all quality problems in a company can be solved by using these tools. Deming is regarded as the *brainchild* of process improvement through statistical quality control (Sommerville, 2011, p. 706). Deming claims that “understanding and controlling variation can lead to the total achievement of quality” (Gitlow & Gitlow, 1987, p. 9). Based on this philosophy, Deming became famous for his Plan-Do-Study-Act (PDSA) cycle (see Figure 2-3), which is a systematic process for continual improvement of a product or process through measurement and variation control (Deming, 2000b).


Act
- What changes are to be made?
- Next cycle?
Plan
- Objective
- Questions and predictions
- Plan to carry out the cycle (who, what, where, when)
Study
- Complete analysis of the data
- Compare data to predictions
- Summarise what was learned
Do
- Carry out the plan
- Document problems and unexpected observations
- Begin analysis of the data

(Source: Adapted from Moen, 2009)

Figure 2-3: PDSA cycle

Summary

Even though all the quality gurus made significant contributions to the formulation of the key TQM elements (customer focus, process, human side of quality, and measurement and analysis), each one had their own unique views and procedures towards quality improvement. In considering the underlying philosophies of the quality gurus, the following insights came to light:

- Defects in the end-product are caused by the quality gap between customer expectations and customer perceptions. The existence of this gap can be attributed to lack of understanding customer needs, lack of design, lack of a process to consistently create products conforming to a design, lack in means by which processes are operated and controlled, and perceived perception of the customer.
- Defect prevention can be used to lower the cost of rework.
- Quality improvement efforts should be built-in throughout the whole process and not through testing and mass inspection only.
- The person who creates the product must take responsibility for its quality.
- Individuals must be committed to monitor themselves and have the ability to adjust their performance to prevent defects.
- Lack of commitment to quality can be regarded as an attitude problem.
- Cost reduction will not be possible without the availability of meaningful data.
- Quality can only be controlled through the use of relevant process measures such as appraisal costs, prevention costs and failure costs.
- Process quality improvement can be achieved by continuously using measurement data for statistical variation control.
- The understanding of measurement data is the key to process improvement.

Kan et al. (1994) regard process measurement as the foundation of quality improvement for any scientific or engineering discipline (also see Figure 2-1). They further suggest that the holistic approach of TQM should guide the software engineering industry to mature towards a true engineering discipline and propose that the use of quantitative approaches in software development should become an “ingrained” practice. In the next section, the suitability of TQM principles for the Software Engineering discipline is discussed.

2.1.3 Traditional Engineering vs Software Engineering

The term software engineering was first endorsed at a NATO conference held in 1968 to devise solutions to what was termed the “software crisis” (Naur & Randell, 1969).

The software crisis was caused by software projects that were running late, over budget and of a low-quality that did not meet the user requirements (Schach, 2011). The Software Engineering discipline was established with the aim "to create defect-free software, delivered on time and within budget, that satisfy the client's needs" (Schach, 2011, p. 4). The NATO group argued that software development can be treated similar to traditional engineering disciplines and that software engineering should use the same philosophies and paradigms to improve the quality of software (Naur & Randell, 1969).

Watts Humphrey, who based his software quality improvement philosophy on Deming's work, acknowledges the similarities between traditional engineering and software engineering by suggesting that the statistical process control concepts used by Deming in his work with the Japanese industry "are just as applicable to software as they are to automobiles, cameras, wristwatches and steel" (Humphrey, 1988, p. 74). Ian Sommerville, however, does not support this view. Sommerville (2004) states that in manufacturing, there exists a clear relationship between the quality of the development process and the quality of the end-product because the processes in manufacturing are easy to standardise and monitor. He (Sommerville, 2004) argues that software is creatively designed and not manufactured through a mechanical process. In this regard, Schach (2011) acknowledges that even though software production is similar to traditional engineering, it "has its own unique properties and problems" (p. 5). Sommerville (2004) further states that "software quality is not dependant on a manufacturing process but on a design process where individual human capabilities are significant" (p. 668). Although Sommerville (2004) does acknowledge that the quality of some types of products are determined by the process used, he argues that "for innovative applications in particular, the people involved may be more important than the process used" (p. 668). He does, however, recognise the strong relationship between process quality and software quality by stating that fewer defects can be achieved in the delivered software through process quality management and improvement. According to Sommerville (2004), the four main factors that influence the quality of design-created products such as software are development technology, process quality, people quality as well as cost, time and schedule. He also emphasises that the size and type of project will influence the impact of each of these factors. For example, in a large project the software process will have the biggest influence, while in a smaller project, the people quality will be more important. Good development technology, on the other hand, will be of particular importance for small teams working on small projects. Cost, time and schedule, however, remains the most critical factor regardless of product size. Since process quality depends highly on resources, only highly skilled individuals will be able to save a project if the resources and process are inadequate.

Similarly to the viewpoints of Deming (2000a) and Humphrey (2005), Pressman (2005) acknowledges that “variation control is the heart of quality control” (p. 745) and that measurement enables developers “to gain insight regarding the process and the project by providing a mechanism for objective evaluation” (p. 649). Sommerville (2004) argues that most companies do not use software measurement to assess software quality because of poorly defined processes, a lack of standards for metrics, limited tool support for data collection and analysis, and misinterpretation of system or process data. In this regard, Kan et al. (1994) suggest the following regarding software measurement:

- Poor data quality is an obstacle to quality improvement and should be addressed with good tracking systems;
- The amount of data could be overwhelming and therefore the number of metrics must be selected carefully; and
- The information extracted from the data has to be focused, accurate and useful.

Summary

Humphrey and Pressman both agree that statistical process control could be used for software process improvement. Sommerville’s biggest concern with this idea is the instability of software projects due to the ‘creative’ or ‘design’ nature of the software engineering discipline. Even though he recognises the influence of process improvement on product quality, he regards individual skills and differences as an obstacle to process control. Sommerville argues that the effectiveness of process control depends on the type and size of the product. Schach also acknowledges that the Software Engineering discipline has its own unique properties. All the prominent authors, however, agree that fewer defects in the final product could be achieved through process quality management and improvement.

In the next section, an overview of two software quality improvement frameworks (CMM and PSP) that are built on the TQM philosophy is provided.

2.1.4 Software quality improvement frameworks

The TQM philosophy (as discussed in Section 2.1.2) is substantiated by numerous organisational quality improvement frameworks (Kan, 2003). Two such frameworks are the Capability Maturity Model (CMM) and the Personal Software Process (PSP). These frameworks were developed by Watts Humphrey who based his work on the TQM philosophies of Crosby, Deming and Juan (Humphrey, 2000; Paulk et al., 1993). In the first sub-section of this discussion, an overview of the CMM framework is provided. This is followed by a detailed discussion of the PSP framework in order to provide conceptual understanding regarding specific principles and practices that are referenced in the remainder of this report.

2.1.4.1 CMM framework

The CMM is specifically aimed at addressing software process improvement, software process assessment and software capability evaluations within both large and small organisations (Paulk et al., 1993). This framework describes five maturity levels that an organisation have to move through in order to improve both their software process capability and their software product quality. Each level (except the initial level) contains key process areas that indicate the goals that must be achieved to reach a specific maturity level (Paulk et al., 1993). The five CMM maturity levels, together with the key process areas and characteristics of each level, are briefly summarised in Table 2-1.

While the CMM is focused specifically on improving software quality within an organisation, Humphrey expanded his work to develop a CMM-based framework, called the Personal Software Process (PSP), aimed at individual software developers. Many of the key process areas of CMM are also covered by the PSP (Saiedian & Carr, 1997) as indicated in Table 2-1. In the next section, the PSP framework is discussed in more detail.

Table 2-1: CMM framework illustrating key process areas addressed by CMM and PSP

Maturity Level	**Characteristics**	**Key process areas (CMM)**	**Key process areas (PSP)**
1. Initial (Unpredictable process capability)	Ad hoc, chaotic Lack of sound management practices Few processes defined Success depends on individual efforts	None	None
2. Repeatable (Disciplined process capability)	Basic management processes established Process is stable to repeat	Software configuration management Software quality assurance Software subcontract management Software project tracking and oversight Software project planning Requirements management	Software project tracking Software project planning
3. Defined (Standard or consistent process capability)	Software process is documented Projects use an approved software process	Peer reviews Intergroup coordination Software product engineering Integrated software management Training program Organisation process definition Organisation process focus	Peer reviews (indirectly via personal reviews) Software product engineering Integrated software management Organisation process definition Organisation process focus
4. Managed (Predictable process capability)	Measures of process quality collected Process/product quantitatively controlled	Quantitative process management Software quality management	Quantitative process management Software quality management
5. Optimised (Continuous process improvement capability)	Continuous process improvement Quantitative feedback	Defect prevention Technology change management Process change management	Defect prevention Technology change management Process change management

(Source: Adapted from Paulk et al., 1993 and Saiedian & Carr, 1997)

2.1.4.2 The PSP framework

Given the importance of the PSP in the context of this study, this section defines the PSP framework, focusing firstly on the PSP principles, and secondly, on the features that are included in each of the first three PSP levels.

The PSP framework was specifically designed as a self-improvement process to guide individual software developers in increasing their planning accuracy and improving the quality of their software programs (Humphrey, 2005). PSP is based on the following planning and quality principles (Humphrey, 2000, p. 5):

- Every developer is unique; to be most effective, developers must plan their work and base these plans on their own, personal data.
- To consistently improve their performance, developers must personally use well-defined and measured processes.
- To produce quality products, developers must feel personally responsible for the quality of their products. Superior products are not produced by mistake; developers must strive to do quality work.
- It costs less to find and fix defects earlier in a process than later.
- It is more efficient to prevent defects than to find and fix them.
- The right way is always the fastest and cheapest way to do a job.

These PSP principles are based on the TQM philosophies of Crosby, Deming and Juran (Paulk et al., 1993).

Similar to CMM, the PSP framework is divided into levels or process versions, each with its own features and objectives (Humphrey, 1994) (see Table 2-2). The first PSP level, PSP0, is aimed at introducing software developers to a defined measured process (Objective 0). The features included in this level guide developers in measuring the time they spend in the pre-defined PSP process phases, as well as keeping track of the defects they inject and remove in each phase. In an expansion of PSP0, PSP0.1 introduces coding standards, size measurement and the process improvement proposal. PSP1 focuses on planning and tracking (Objective 1) by incorporating size estimation and test reporting. This level is enhanced in PSP1.1 with the introduction of task and schedule planning. PSP2 focuses on quality management (Objective 2) by introducing design reviews and code reviews to help developers find and remove defects earlier in the software development life cycle. As a follow-up on PSP2, PSP2.1 introduces design templates with the intent to quantify the completeness of design artefacts. PSP3 focuses on scaling up to larger programs (Objective 3) by adding a cyclic development process.

Table 2-2: The PSP framework

PSP levels	Level Features	Objective
PSP0	Defined process Basic process measures: • Time recording • Defect tracking Defect type standard	0. Defining and Using Processes
PSP0.1	Coding standard Size measurement Process improvement proposal	
PSP1	Size estimation Test reporting	1. Planning and Tracking
PSP1.1	Task planning Schedule planning	
PSP2	Code review Design review	2. Quality Management
PSP2.1	Design templates	
PSP3	Cyclic development	3. Scaling Up

(Source: Adapted from Humphrey, 2005, p. 8)

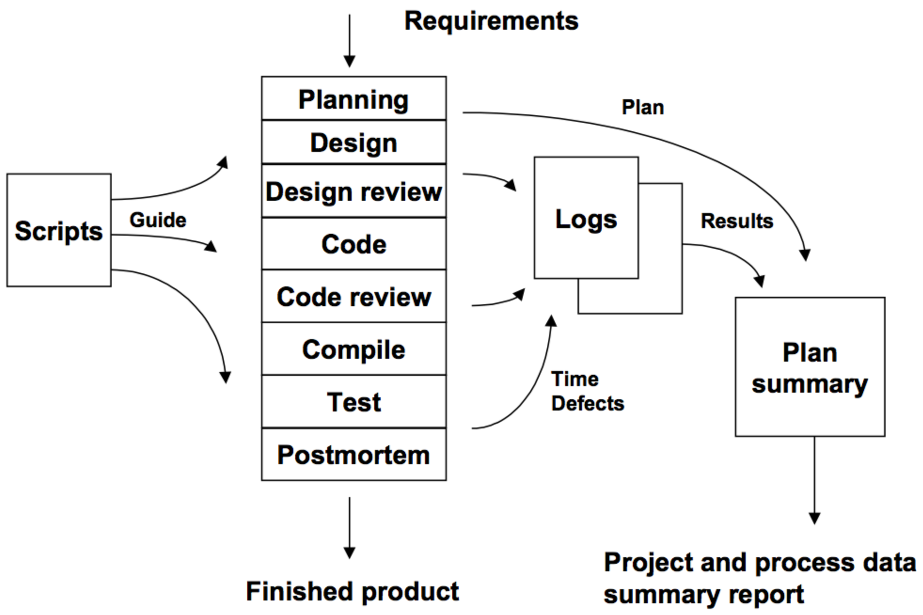
In the following sub-sections, the features of each of the different levels (PSP0, PSP1 and PSP2) of the PSP framework are discussed. The specific features of PSP3 are excluded from this discussion as it falls outside the scope of this study.

PSP0 (Defining and using processes)

Humphrey (2005) states that “a defined process specifies precisely how to do something” (p. 6). Defined processes serve as guidance and motivation to consistently follow “disciplined software engineering practices” (Humphrey, 2005, p. 6). He proposes that defined processes should include measures to enable developers to understand their performance, manage their work, and plan and manage the quality of their products.

The PSP0 level proposes a defined process that is divided into eight phases: planning, design, design review, code, code review, compile, test and post-mortem (see Figure 2-4) (Humphrey, 2000). Specific process scripts have been developed to indicate the exact actions that should be performed in each of these phases. The following in-process data (time, size and defect) are collected throughout the whole development process (Humphrey, 2000, p. 15):

- Time spent in each development phase;
- Size of the artefacts (design documents, actual lines of code) created;
- Defects (including descriptions) injected and removed in each development phase; and
- Time spent on fixing the documented defects.



(Source: Humphrey, 2000, p. 7)

Figure 2-4: PSP process flow

A plan summary form is used to record planning (estimated) time, size and defect data. Actual time and defect data are recorded in “logs” while developers do their tasks according to the scripts. During the post-mortem phase, when the all task(s) are completed, the plan summary form is updated with the actual time and defect data from the logs, as well as the actual measured size(s) from the finished product.

PSP1 (Planning and tracking)

The main principle behind the PSP planning process is that “estimates are made by comparing the planned job with previous jobs” (Humphrey, 2005, p. 69). The prerequisite for planning is that a developer needs to have personal historical data on the previous products that he/she created. The predictability of the estimations is determined by the amount of historical data as well as the accuracy of the data (Humphrey, 2005).

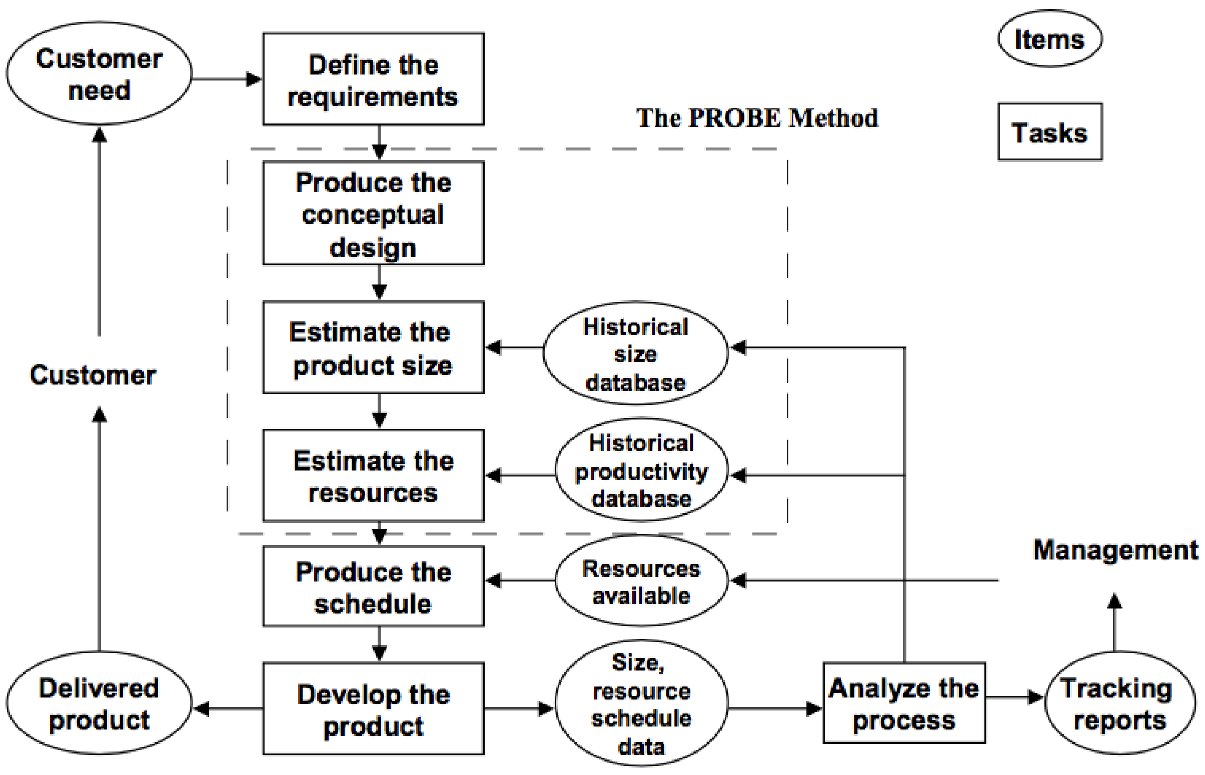
In PSP1, planning is done from a conceptual design that is based on clearly defined requirements. The size of the product is estimated from the conceptual design. The estimated size determines the total time required to do the task (which is based on actual historical productivity data). The total time is divided into the development phases based on the historical time-in-phase data. This planning process is called proxy-based estimation (PROBE) (Humphrey, 2000; 2005) and is illustrated in Figure 2-5. The completed planning process will result in estimations for the size of the program, the total development time and the time in each phase. The total development time is used to schedule tasks for each day to perform the total job. An earned-value method is used to track task completion.

PSP2 (Quality management)

Humphrey (2005) states that software developers tend to spend most of their time finding and fixing defects and as a result tend to neglect the attributes that matter to end-users. He therefore argues that even though defects are not top priority, defect management is essential to product quality.

Quality management in PSP2 is based on three principles (Humphrey, 2000, pp. 23–24):

1. *The individual developer who creates the product has to be responsible for the quality thereof*. The PSP provides developers with quality measures and guidelines to improve the quality of their products.



(Source: Humphrey, 2000, p. 11)

Figure 2-5: PSP project planning process

2. *Find and fix defects as early as possible in the life cycle*. The PSP provides developers with review practices (design review and code review) and guidelines to find and fix defects before the program is compiled and tested.
3. *The most effective way to manage defects is to prevent their initial introduction*.

In dealing with the third quality management principle, PSP2 prescribes three ways to prevent defects (Humphrey, 2000, p. 24):

- By recording defects and reviewing what caused them, the developer becomes more sensitive towards making these defects during development. This argument comes from the assumption that people tend to make the same mistakes over and over again.
- By using effective design techniques to create reviewable designs, developers will make fewer design mistakes.
- By producing better designs, developers will make fewer coding mistakes.

The quality principles of PSP2 are implemented through the use of the following quality improvement methods: design reviews, code reviews, design templates and quality measurements.

Design reviews and code reviews

Software review methods (such as inspections, walkthroughs and personal reviews) are widely used for early defect removal (Fagan, 1976; Humphrey, 2000; Pressman, 2005; Schach, 2011; Sommerville, 2004) as testing alone is seen as ineffective and time-consuming (Humphrey, 2005; Schach, 2011). Humphrey (2005) argues that the reason why reviews are more efficient than testing is that in reviews you “find defects directly” and in testing you “only get the symptoms” (p. 195). The primary objective of reviews is to find defects earlier in the development process so that they do not get intensified during testing (Pressman, 2005). Humphrey (2000; 2005) further argues that defects are caused by individual software developers and that these individuals need to manage their personal behaviour through measurement and tracking, and apply methods for early defect removal and defect prevention.

PSP2 proposes the use of personal design reviews and code reviews to find and fix defects before testing (Humphrey, 2005). The underlying principle of personal reviews is that people tend to make the same mistakes repeatedly (Humphrey, 2000). Therefore, developers should analyse and use their collected defect data to create checklists in order to specifically address their personal defect habits. Humphrey (2005) claims that the use of personal checklists during reviews would increase the efficiency of reviews.

PSP2 also recommends the following review principles (Humphrey, 2005, p. 169):

- Personally review all of your own work before you move to the next development phase.
- Fix all defects before giving the product to someone else.
- Use a personal checklist and follow a structured review process.
- Follow sound review practices.

- Measure the review time, the size of products reviewed, and the number and types of defects you found and missed.
- Use data to improve your personal review process.
- Design and implement your products so that it can be easily reviewed.
- Review your data to identify ways to prevent defects.

Humphrey (2005) further states that "doing thorough design and code reviews will do more to improve the quality and productivity of your work than anything else you can do" (p. 163).

Design templates
Designs form an integral part of the quality of a software system (ICSE, 2011). Humphrey (2005) argues that developers do not create thorough designs because they believe that it will make the development process longer. He further argues that if the developer only writes small programs (which is typical in many undergraduate programming courses) the advantages of a thorough design would not be as evident. Hence developers never learn how to do thorough or complete designs because they tend not to use it while writing small programs. They are therefore unlikely to have the necessary design skills and experience to write bigger programs.

PSP2 guides a software developer in creating 'complete' designs through the use of four design templates (Humphrey, 2005) that each corresponds with four design categories [as defined by de Champeaux, Lea and Faure (1993) in their two-dimensional design specification structure]. Humphrey (2005, p. 247) further proposes specific unified modelling language (UML) diagrams that can be used to model certain system parts and system behaviour in each of the four design categories:

- The *external static* category models object attributes and the relationships between classes with class diagrams.
- The *external dynamic* category models the interaction of users with a system. During the requirements workflow of software development, these interactions are modelled through use case diagrams and descriptions (Schach, 2011).

During the analysis phase, these interactions are modelled with activity and sequence diagrams (Schach, 2011).

- The *internal static* category is modelled through pseudocode or flowcharts. Humphrey (2005) argues that even though the internal static category has no UML equivalent, most UML tools provide a way to enter and store the pseudocode with a design.
- The *internal dynamic* behaviour of objects is modelled through state charts.

Table 2-3 summarises the relation between these four design categories and the PSP design templates.

Table 2-3: PSP design specification structure

PSP design template	**Design category**	**Definition**	**UML diagrams**
Functional Specification Template (FST)	External-static	Define attributes of system parts and the relationships between system parts.	Class diagram
Operational Scenario Template (OST)	External-dynamic	Define interaction between system parts.	Use case diagram Use case description Activity diagram Sequence diagram
Logic Specification Template (LST)	Internal-static	Define the detail logical structure within a system part.	No UML equivalent, but pseudocode is normally used.
State Specification Template (SST)	Internal-dynamic	Define the internal state behaviour within a system part.	State diagram

Quality measures

In PSP, individual developers gather data to enable them to increase planning accuracy and improve the quality of software programs. "With time, size and defect data, there are many ways to measure, evaluate, and manage the quality of a program" (Humphrey, 2000, p. 17). The PSP defines several quality measures to enable a developer to analyse and improve personal development processes: defect density, review rate, various development time ratios, defect rations, yield, defects per hour, defect removal leverage (DRL) and appraisal to failure ration (A/FR). Each of these measures are calculated by using the in-process data (time, size, defect) (Humphrey, 2000). Table 2-4 provides a brief description of each of these quality measures together with recommendations and good practice guidelines.

Table 2-4: PSP quality measures

Quality measure	**Description**	**Recommendations & good practices**
Defect density	The number of defects found in a program relative to its size - usually expressed in terms of defects per 1000 lines of code.	The lower the defect density the better the quality of the program. Five or fewer defects/KLOC is good. The number of defects found in testing is a good indication of defects that remains in the program.
Review rate	The time spend on design and code reviews normally expressed as lines of code (LOC) per hour.	An effective review rate is between 150 and 200 LOC per hour.
Development time ratios	The ratio of time spent in any two phases. The following three ratios are important quality measures: • Design time to coding time. • Design review time to design time. • Code review time to coding time.	Design-to-coding ratio is the most important quality ratio measure. Design time should at least be the same as coding time. Design review time should at least be half of design time. Code review time should at least be half of coding time.
Defect ratios	The ratio of defects found in one phase compared to another phase. The main defect ratios are: • Code review defects to Compile defects indicates the quality of the code. • Design review defects to Test defects indicates design quality.	Twice as many defects must be found in code review than in compilation. Twice as many defects must be found in design review than in test.
Yield	The percentage of defects found in a specific phase. Process yield refers to the percentage of defects found before the first compile or test.	The suggested guideline for high quality programs is a process yield of at least 70%.
Defects per hour	The number of defects that is injected or removed per hour.	Use defect removal rate to guide personal planning.
Defect removal leverage (DRL)	The measurement of the relative effectiveness of two defect removal phases. Typical ratios that is used is: • Design review to test. • Code review to test.	These ratios indicate if the reviews are more effective than testing.
Appraisal to failure ratio (A/FR)	The A/FR indicates the quality of the engineering process using Juran's cost-of-quality parameters. The A stands for appraisal cost and is indicated as the time spent in design reviews and code reviews. The F stands for failure cost and is indicated by the time spent in compile and testing.	The A/FR is an indication of effort to find and fix defects early in the development life cycle.

(Source: Adapted from Humphrey, 2000, pp.18–22)

Summary

Overall, PSP is a continuous improvement process designed to guide individual software developers in increasing their planning accuracy and improving the quality of their software programs. The PSP framework is divided into four main levels: PSP0, PSP1, PSP2 and PSP3 (see Table 2-2). PSP0 guides a developer in using a defined process and measurement data to improve his/her performance. PSP1 is based on the principle that every developer is unique and must therefore use his/her personal data to plan his/her work. This is accomplished through the use of the PROBE technique to improve the predictability of the development process. The PSP2 quality management strategy is based on three principles: (1) the individual developer who creates the product needs to be responsible for the quality thereof; (2) find and fix defects as early as possible in the life cycle; and (3) the most effective way to manage defects is to prevent their initial introduction. PSP2 addresses these principles through the use of design reviews, code reviews, design templates and quality measurements. PSP3 focuses on scaling up to larger programs by including a cyclic development process.

The PSP principles are widely used in industry to improve the quality of work done by software developers. Before taking a closer look at what happened when higher education students attempted to use the PSP principles (as discussed in Section 2.1.4.2) as part of their software development processes, an explanation of the general software quality problems in higher education is needed.

2.2 Software Quality Problems in Higher Education

Over the past 40 years, numerous studies have been conducted to evaluate the programming performance of undergraduate CS students and identify possible reasons for the low quality of their programs. In many of these studies, students had difficulties completing even relatively small and basic programs. The remainder of the discussion in this section provides an overview of some of the key studies that have been conducted over the past four decades to identify both general programming problems and design problems experienced by novice programmers.

2.2.1 General programming problems

In 2001, the “McCracken Working Group (MWG)” (McCracken et al., 2001) conducted a multi-national, multi-institutional study in the United States of America (USA) and other countries during which they assessed the programming competency of Computer Science students who completed their first or second programming courses. In this study, students were required to write three small programs - the simplest of which was to build a Reverse Polish Notation (RPN) Calculator. Even with this simple exercise, the students performed much worse than expected. Most students blamed their inability to complete the programming exercise in the given time as the major reason for failure. The researchers argued that the students’ poor performance was due to their inability to abstract the problem from requirements (lacking problem solving skills) and “bad” programming habits (fixing syntax without understanding the task). As a result, the MWG proposed that future research should analyse narrative data gathered from students to gain better insight into the students’ development processes and problem-solving behaviour.

The ITiCSE 2004 “Leeds” working group (Lister et al., 2004) conducted a follow-up study on the McCracken group’s research to investigate alternative reasons for poor programming performance. They found that the main reasons for students’ poor performance are the “lack of problem-solving skills”, “a fragile grasp of basic programming principles”, and the inability to perform “routine programming tasks, such as tracing through code” (Lister et al*.,* 2004, p. 119). Lister et al. (2004) argue that because students are taught by example, without the ability to read and interpret code they would not be able to write code.

In 2013, a replicated study of the original MWG study was done to determine if the conclusions of the original study still stand after more than a decade of teaching intervention in Computer Science (Utting et al., 2013). During this study, a “skeleton” class layout and pre-written test cases were given to the students as part of the programming exercise (“the clock problem”). This means that the students did not have to create the design from scratch and they had a method of verification for correctness. The study found that students perform better in programming tasks when they have enough scaffolding code as a starting point and when they do not need to do the “whole design”. The authors also indicated that test-driven-development (TDD), where test cases are given by the instructor, improved the programming performance of students. They further noted that modern students are more connected to resources through the Internet, which might influence the strategies that they will use to solve programming problems. Given the important role that complete designs play in defect prevention (Humphrey, 2005), it is no surprise that these students were able to produce higher quality programs than students from the original study.

As part of a study discussing a 'goals and plans' strategy to solve programming problems, Soloway et al. (1982) introduced the 'Rainfall Problem':

> *Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.*

This short length problem consists of four major tasks: get input, sum the input, calculate the average and output the average. The results of Soloway et al.'s (1982) initial study indicate that only nine out of 31 participants solved this problem correctly. The rainfall problem and variations thereof have since been used in numerous empirical studies (Ebrahimi, 1994). Although these studies considered different aspects of programming, they all recognised various difficulties experienced by students in solving this apparently simple problem. Ebrahimi (1994) identifies the "misunderstanding of plan composition and semantic misinterpretation of language constructs" (p. 457) as the two major causes of students' errors in solving the rainfall problem. Simon (2013), in using an adapted version of the rainfall problem in an introductory programming course examination paper, reports that the 149 students received an average mark of 23% for this problem. He concludes that despite improvements in education over the last 30 years, students still make the same kind of errors when doing the rainfall problem. In another 'rainfall' study, Lakanen et al. (2015) used an adapted version of the rainfall problem to analyse implementation approaches and program errors made by novice programmers. In contrast to some of the other 'rainfall studies' (Soloway et al. 1982; Ebrahimi, 1994; Simon, 2013)), students in Lakanen et al.'s (2015) study performed very well and obtained an average score of 68.8% in a written exam paper. The most common defect made by students in this study was not guarding against division by zero when calculating the average.

In a study that looked more closely at the defects made by students, Ahmadzadeh et al. (2005) collected compiler generated error messages on a large scale. They report that there is a strong correlation between the time spent in debugging and the marks obtained by the students. Ahmadzadeh et al. (2005) further argue that students "who are skilled at debugging are faster and more effective at finding the errors and able to achieve higher marks". Weak debuggers revealed a lack of understanding of the intent of the program or code that they had to debug. Most weak programmers could isolate the defect but were unable to correct it. Good programmers who struggle with debugging lack an understanding of the "actual program implementation". They also found that most good debuggers are good programmers, but less than half of the good programmers are good debuggers. Of particular interest in this study is Ahmadzadeh et al.*'s* (2005) use of defect measurements to evaluate student performance.

2.2.2 Design problems

While general programming problems (as discussed in Section 2.4.1) can have a significant impact on the quality of software, designs are also critical in ensuring the quality of a software system. It is therefore alarming that, even after having completed their undergraduate studies, most CS students are still unable to design software and generally lack even basic design knowledge (Chen et al., 2005; Eckerdal et al., 2006a; Eckerdal et al., 2006b; Hu, 2016; Loftus et al., 2001). The discussion in this section presents a chronological overview of five studies that investigated CS students' design ability, as well as their understanding of software design and software design criteria. All five studies used the 'super alarm clock' task originally used in Chen et al.'s (2005) study. For this task, students were instructed to design a 'super alarm clock' (based on the provided requirements) that would help university students manage their own sleep patterns. Each of the five studies, however, proposed different categories to evaluate the quality of students' designs.

In their multi-national, multi-institutional study, Chen et al. (2005) created five categories for design evaluation: standard graphical, ad-hoc graphical, code or pseudocode, textual and mixed. They analysed the complexity of the students' designs according to the number of parts, use of grouping structures and interaction among parts. Their results indicate that the design representation style progresses from textual to standard graphical as the developer's experience level increases, and that most students did not indicate the structural groupings and interaction between parts. They found that recognition of requirements ambiguity also increases with experience and resulted in higher success when used. In addition to the design task, students also performed a 'design criteria prioritisation' task in which they had to rank the importance of software design criteria for different design scenarios. In this regard, the students identified the clarity of designs and the management of the design process as the most critical elements of the design process.

In another attempt to examine students' design ability, Eckerdal et al. (2006a, p. 404; 2006b, p. 200) proposed six semantic categories to evaluate the various degrees to which a stated requirement was met:

- Nothing ("little or no intelligible content");
- Restatement ("merely restate requirements from the task description");
- Skumtomte ("add a small amount to restating the task");
- First Step ("include some significant work beyond the description");
- Partial Design ("provide an understandable description of each part and an overview of the system that illustrates the relationships between the parts"); and
- Complete ("show a well-developed solution").

Overall, approximately 20% of the participants created no designs while more than 60% of them made no significant progress towards design. Only 9% created a partial or complete design. Their results (Eckerdal et al., 2006a) indicate a positive correlation between courses completed and the completeness of design. They did, however, find no correlation between design completeness and the students' performance in CS courses.

In revisiting Eckerdal et al.'s (2006a) original question (*Can graduating students design software systems*?), Loftus et al. (2011) conducted an extended study that explored students' abilities to design software in groups and to evaluate each other's designs. The first part of the experiment included a 'super alarm clock' design task that was completed by seven groups consisting of ten members each. The designs were rated by both the researchers and the students according to the criteria listed in Table 2-5. The design criteria were developed based on the information that a system architect would be looking for in a complete design document. The artefacts list possible elements that can be used to evaluate the design, while the weight assigned to each design criteria adds up to 10. These artefacts also correspond closely with the UML elements included in Humphrey's PSP design templates (see Table 2-3). During the design evaluation, a nominal mark of 0, 0.25, 0.5, 0.75 or 1 was awarded for each design area. This nominal mark was then multiplied by the indicated weight to determine the 'real mark' for the design area. Lotus et al. then mapped these 'real marks' to Eckerdal et al.'s (2006a) original design completeness categories (see Table 2-6). This mapping was done to create an objective instrument that could be used to evaluate different kinds of design. Lotus et al. (2011) also emphasised that the weightings (as indicated in Table 2-5) can be adjusted based on the size and nature of the system to be developed. Their 'simpler' categorisation can therefore be used to assess "designs developed under different software engineering methodologies" (Lotus et al., 2011, p. 110).

Table 2-5: Marking criteria for group designs with weightings

Design criteria	**Possible artefacts**	**Weight (Total = 10)**
Problem analysis	Use case diagrams Use case descriptions	2
Behavioural architecture	Sequence, communication or activity diagrams	1.5
Structural architecture	Component Diagrams	1
Detailed behaviour	Pseudo-code or Flow charts	1
Detailed structural	Detailed class diagrams	0.5
Database design	ER diagrams	1.5
GUI design	Mock-ups or Storyboards	0.5
Security	Discussion on security considerations	0
Linkage of Artefacts	Linkage by placing lines between diagrams	2

(Source: Adapted from Lotus et al., 2011, p. 109)

The results of Lotus et al.'s (2011) study indicate that students' strongest design ability is to analyse the problem by means of use-case diagrams. Detailed class diagrams and relationships between classes were not used at all. Their experiment also revealed that structural design abilities were stronger than behavioural design abilities. Although only one group was able to produce a “complete design”, students were generally able to recognise better designs.

Table 2-6: Mapping of real marks to original completeness categories

Rating mark (RM)	**Completeness Category**
$RM = 0$	Nothing
$0 < RM < 2$	Restatement
$2 \leq RM < 4$	Skumtomte
$4 \leq RM < 6$	First Step
$6 \leq RM < 8$	Partial Design
$8 \leq RM \leq 10$	Complete Design

(Source: Adapted from Lotus et al., 2011, p. 108)

In a further expansion of Eckerdal et al.’s original study, Thomas et al. (2014) used a phenomenographic approach to gain insight into 35 students’ understanding of the “produce a design” phenomenon. In comparison to the original Eckerdal et al. (2006a) study, a larger portion of Thomas et al.’s (2014) participants delivered designs that fall in the upper three categories (first step, partial, and complete) of Eckerdal et al.’s design completeness classification model. Their phenomenographic data (based on analysis of the students’ designs, as well as one design created by an expert) resulted in six hierarchical categories of design understanding (see Table 2-7).

Table 2-7: Hierarchical outcome space: Students' understandings of "produce a design"

Category Code	**Description**
0	Informal design. Normally text-based; can include some pictures, but no formal artefacts.
1	Uses some formal structures for analysis like use case diagrams but does not indicate system structure or behaviour.
2	Focuses on formal design techniques such as class diagrams and structural relationships (static).
3	Formal design techniques that express (dynamic) behaviour such as sequence diagrams or flowcharts.
4	Uses multiple artefacts and includes relationships between components indicating a clear link between static and dynamic.
5	The notations are relaxed and only the essential artefacts are included (expert).

(Source: Thomas et al., 2014, p. 95)

Based on their evaluation of the students' designs, Thomas et al. (2014, p. 97) also identified five features that are critical to the understanding of the 'produce a design' phenomenon: language of computing, structure of components, behaviour of components, link structure and behaviour, and appropriate artefacts as needed. They urge educators to create learning conditions that will help students to recognise these critical features.

In a duplication of Eckerdal et al.'s (2006a) experiment, Hu (2016) used the original design completeness categories to assess his students' designs. The results of his study indicate that half of the students created a "partial design", a third created a "complete design", and the rest (16.6%) created a "first step design". In addition to completing the design task, Hu (2016) also asked students to implement their designs and to describe their typical design process (as part of a post-activity reflection exercise). Based on the students' reflections, Hu (2016) identified four design process approaches followed by novice programmers. Closer inspection of these approaches, however, suggests that it would be more accurate to describe these as *development practices* since each of the approaches include both design and implementation practices. The development practices identified by Hu (2016, p. 201) can therefore be described as follows:

- *Noun-extraction:* Students define classes, interfaces and data structures from the problem description. They fix problems while they are coding to let the defined data structures work.
- *Task-oriented:* Students start with tasks that are not completely new to them and then start coding tasks that "seem(s) less likely to be abandoned later". Code are modified frequently, but designs are not improved.
- *Console-dialog-oriented:* Students create a "driver" program that prompts specific tasks as input. They then create classes and methods to support these tasks.
- *Code-first:* No design artefacts are created, only code. Students start with simple tasks first and then code more difficult ones. Testing and changes occur throughout the process.

In summary, Hu's (2016) study reveals that students were unable to effectively apply design principles to indicate the comprehension of design trade-offs. Students were unable to perform proper requirements analysis, especially with unfamiliar problems, which in turn lead to difficulties in starting the design process. In cases where the students' design diagrams were of a low quality, the designs eventually became useless artefacts. Students rarely updated their designs when code changed during implementation. Hu (2016) suggests that the focus of a design course should be on "process knowledge" (p. 203) and that design activities should be used throughout all programming courses to improve the overall quality and usefulness of designs.

The five studies described above all confirm that it is possible to evaluate the completeness of students' designs according to some sort of hierarchical structure. The UML elements included in Lotus et al.'s (2011) design evaluation framework correspond closely with the PSP design templates. In most of these studies, very few students were able to create complete designs. Novice students generally found it difficult to correctly identify requirements (Chen et al. 2005; Hu, 2016) and consequently delivered designs that were either not much more than a textual description of the original task (Chen et al., 2005; Eckerdal et al., 2006a) or could at most be described as the first-step of partial designs (Hu, 2016; Thomas et al., 2014). Lotus et al. (2011) concurred that their students' strongest design ability was to analyse the problem by means of use-case diagrams. While the first four studies only considered the actual designs created by the students, Hu (2016) was able to gain much deeper insight into his students' design processes through the inclusion of an implementation and narrative component to his study. He suggests that design courses should have a stronger focus on "process knowledge" (p. 203).

2.3 The Use of PSP Principles by Higher Education Students

The discussion in Section 2.2 highlighted some of the major programming and design problems that directly influence the quality of software developed by students. Hilburn and Towhidnejad (2000) argue that not enough attention is given to the quality of software developed by programming students. They suggest that instructors should adopt "a software development process that emphasise quality techniques, methods, and analysis" (Hilburn & Towhidnejad, 2000, p. 171). Such a process should be taught to and used by programming students throughout all their courses. Humphrey (1999) reasons that, despite all the literature that guides software developers on “good” practices and effective methods, the only generally accepted short-term priority for a software developer remains “coding and testing”. Humphrey (1999) claims that one of the biggest challenges in software development is to persuade software developers to use effective methods. Software developers tend to stick to a personal process that they have developed from the first small program they have written, and it is difficult to convince them to adopt better practices (Humphrey, 1999). Hilburn and Towhidnejad (2000) propose the PSP as a good candidate to contribute towards the implementation of a curriculum that focuses on software quality.

Humphrey (2000) created a personal software process (PSP) course in which a software developer gradually learns to adopt his/her software practices according to personal measurements (also see Table 2-2). There are various versions of the PSP course that are suited for either beginning university students, advanced undergraduate students, graduates or industry developers (Humphrey, 2000). Analyses of thousands of PSP students’ measurement data indicate that personal reviews improve program quality and that students spend less time in the testing phase if they use QATs (design reviews and code reviews) (Humphrey, 2005). The course data also indicates an improvement on predictions based on historical data. Humphrey (1999) states that PSP trained students in an educational environment will only use these methods if the educator grade them on the use thereof, and that most students eventually will fall back on a process of coding and testing. He suggests that Computer Science educators must shift their focus from the programs that the students create to the data of the processes that the students use.

Although there are numerous studies that report on the use of PSP principles for process improvement in industry, the number of PSP studies in the higher education domain is limited (Kuhrmann et al. 2016). An extensive search on some of the major academic databases (Academic Search Ultimate, EBSCOhost and Scopus), as well as Google Scholar revealed very few educational PSP studies during the past five years (2015 - 2019). Due to implementation and adoption challenges identified in earlier studies, educational researchers seem to have either terminated their PSP integration efforts (Pando & Ojeda, 2019) or shifted their instruction (and research) efforts to focus more specifically on strategies and tools that can be used to overcome the identified challenges (Denwattana et al., 2019; Gomez-Alvarez et al., 2016; Pando & Ojeda, 2019; Raza et al., 2017; Raza et al., 2019; Rong et al., 2018). Only educational studies that specifically reported on efforts to use basic PSP principles for the improvement of students' development process quality, were selected for further discussion in this section.

In the selected PSP educational studies, researchers have either incorporated selected PSP principles as part of their traditional teaching approaches or conducted a PSP course with their students. For these PSP courses, some instructors chose to use a scaled-down or adapted version of the full PSP course. Based on their own and their students' experiences, these researchers identified various attributes and/or challenges that influenced the success of their students' attempts to improve the quality of their development practices solely through the application of PSP principles.

In one of the first PSP-related studies conducted in an educational environment, Towhidnejad and Salimi (1996) experimented with an integrated learning approach of PSP concepts as part of two first-year CS semester courses. During the first semester, the focus was on collecting time-in-phase process data and making size estimations. They indicate that the students' improvement on making accurate size estimation was not as good as time estimations. They argue that the reason for this was that students were unfamiliar with the complexity level of new programming constructs (like loops) and therefore, initially found it difficult to estimate the size of programs. Initially, the students also resisted to keep track of time because they thought they knew how to do time management, and also found it difficult to relate the concept of time management to programming. During the second semester, the students used code reviews as a defect removal strategy and kept record of defect data. Students found it easier to adopt code reviews as part of their quality improvement process since they perceived it as more closely related to programming. Code reviews also helped the students to decrease the number of syntax defects before the first compile. Overall, the instructors' biggest challenges were (1) to motivate students to follow PSP practices, and (2) to get students to collect accurate and reliable data.

In an attempt to train better software developers, the University of Utah incorporated PSP concepts in all their undergraduate CS courses (Williams, 1997). The specific PSP concepts addressed were time management and defect removal. Students recorded time-in-phase and size data that were used for estimations. For quality, they used code reviews and recorded defects removed in specific development phases. Data were compiled in summary reports that the students themselves had to analyse in an effort to improve their planning accuracy and program quality. Williams (1997) indicate that, according to students, time management was not relevant to software development and obstructed their focus on the learning of programming. Similarly to the findings of Towhidnejad and Salimi (1996), Williams also reports that students found code reviews more relevant to programming. For this reason, the instructors decided to first do the quality management principles and then perform time management and estimation practices. Williams (1997) further reports that although students demonstrated accurate theoretical knowledge of PSP principles, they struggled with the application thereof. Overall, there was no improvement on productivity with the use of code reviews and no significant improvement on planning through the use of the PROBE estimation technique. Williams (1997) suggests that discussions of group statistical feedback data might influence students' intention to capture more accurate individual process measurement data.

Similarly, Carrington et al. (2001) included selected elements of the PSP in a large second-year programming course (with 360 students) at the University of Queensland. The objective of this intervention was to "help students develop good software development habits early, and to encourage them to see software development as a systematic discipline rather than a trial and error activity" (p. 81). Students were trained during four lectures that focused on defect recording, time and size measurement, code reviews and checklists. Students also had to complete a final assignment in which they had to analyse and reflect on their own, personal data. Carrington et al. (2001) report that the additional work caused a cognitive overload for some students, especially for those who struggled with basic programming skills. On the other hand, some of the more skilled programmers "felt that the PSP practices interfered with their well-established habits" and made the assignments "less entertaining and more tedious" (p. 85). Carrington et al. (2001) emphasise that when students write a program by incrementally coding and testing one line of code at a time, they are unable to log data in the correct prescribed PSP phases. Although the final assignment reports indicated that most students did not record defect and time data correctly, the authors, however, believe that the whole process of using PSP principles, encouraged students to start questioning their current development practices.

Since Grove (1998) regarded the complete PSP course as possibly being too difficult for beginning programmers, he instead created and used a scaled down version called the Software Development Process Log (SDPL). Grove (1998) indicates that students initially struggled with the distinction between development phases and therefore struggled to collect reliable data. The use of SDPL, however, had a positive impact on students' attitudes toward software process improvement (Grove, 1998). In this regard, the students recognised the value of using a proper programming methodology, good designs and reviews. They further realised the importance of process measurement data as a motivator for continuous improvement.

As part of a comparative experiment conducted at Carnegie Mellon, Hou and Tomayko (1998) applied key components of PSP with 65 first year CS students. They compared the results of these students to those of a group of similar size who did not receive such training. The process data logged by the PSP-group revealed interesting insights regarding their development practices. Even though these students captured time in the design phase, there was no evidence that they created any design documentation. After the introduction of code reviews, there was a significant decline in the number of compile defects recorded. Students in the non-PSP group took significantly longer to produce their final projects and on average obtained lower marks. Hou and Tomayko (1998) believe that the better performance of the PSP-group can be attributed to less time spent in testing/debugging. Based on additional narrative feedback, students from the PSP-group could be divided into two distinct groups: those who indicated that PSP helped them to better plan their personal schedule and those who regarded the extra work of capturing process data as a burden.

Similarly to the views of Hou and Tomayko's (1998) students, Bullers (2004) also states his disappointment in the low value that students place on the PSP discipline. In describing a study that integrated PSP practices into an introductory database course, Bullers (2004) mentions that his students generally struggled to appreciate the value of gathering data on their personal development practices. Some students even objected that the PSP principles were not related to the database course syllabus. From the instructor's side, the biggest concern was the time spent on monitoring, analysing and grading the PSP data gathered by students during the course.

Börstler et al. (2002) report on their experiences in teaching some PSP variations at different universities. At Montana Tech of the University of Montana, students initially showed resistance to PSP, but the general reaction at the end of the course was that they felt "more aware of their programming practices and shortcomings" (p. 45). Although some master's students at Drexel University also initially showed resistance towards PSP, several of them reported incorporating at least some PSP parts in their work environments. At Umeå University, the use of PSP was optional in a second-year C++ course with only six of 78 students opting to use it throughout the course. The students' main reason for abandoning PSP was that it "impose[d] an excessively strict process on them" (p. 44) and that they did not believe that the extra effort would be worthwhile. An evaluation of the Purdue University students' attitude towards PSP revealed that they regarded PSP activities as "extra work" (p. 45), and did not show appreciation for the potential benefits of this disciplined process. Students strongly recommended that PSP topics should rather be placed in more advanced programming courses when students are already familiar with language specific syntax and the development environment.

Although Runeson (2001) agrees that it would be harder for first-years to learn PSP practices together with the development of programming skills, he argues that first-year students find it easier to adopt PSP practices than graduates do. Based on his experiences in teaching PSP courses to students of various levels at Lund University in Sweden from 1996 to 1999, Runeson (2001) found that first-year students are more focused on programming issues than process issues and can therefore gain most from PSP. Since first-year students have less established development habits, it is easier to convince them to use PSP practices as part of their natural development processes. This also supports Humphrey's (1999) statement that students are likely to stick to the development process they used on their very first program.

In an attempt to test the process improvement claims of PSP, Prechelt and Unger (2001) conducted an experiment to compare the performance of a group of CS master's students consisting of both PSP-trained programmers (P-group) and non-PSP trained programmers (N-group). They report that 18 of the 24 P-group participants did not use PSP techniques at all. Prechelt and Unger (2001) claim that the low level of PSP usage might be explained by the "different temperaments of the programmers", the small size of the PSP tasks, as well as the absence of "a working environment which actively encourages PSP usage" (p. 471). They call for further investigations into the technical, social and organisational attributes (beyond the level of training and infrastructure provided) that might influence the use of PSP methods. Through further experimentation, Prechelt (2001) proved that it is possible to learn and apply defect logging and defect analysis (DLDA) without spending time on doing a full PSP course. DLDA can be applied on any software phase with the intention to remove defects earlier in the development life-cycle. Prechelt (2001) argues that early defect removal or defect prevention is an attribute of a "mature process" that can be applied on "immature" software processes (p. 57).

In focusing on early defect removal, Jenkins and Ademoye (2012) conducted a pilot and follow-up experiment in which students performed personal code reviews using the standard PSP code review checklist (Humphrey, 2005). Since the pilot study indicated that students found it difficult to use the provided checklist, the follow-up study used a different approach as tutorials were used to first train the students to use the checklists. As a result, the students in the follow-up study found the checklist reviews easier to use. According to the students, the major problem they experienced with the code reviews was the time it took to complete the review. In both experiments the qualitative feedback from the students indicate that they believe the process of using code reviews improved the quality of their programs although there are no concrete evidence to support this statement.

In another study that focused on the use of checklists to conduct personal reviews, Rong et al. (2012) designed an experiment to ascertain whether checklist based reviews work well for inexperienced first-year students. They let one group use checklists while the other group conducted ad-hoc reviews without checklists. In order to determine the impact that using checklists during reviews have on beginners, Rong et al. (2012) used the review rate, review efficiency and sensitivity of a checklist as measures to compare the results of the two groups. The checklist group used the PSP code review checklist as described by Humphrey (2005) - the same checklist used by Jenkins and Ademonye (2012) in their study. Rong et al. (2012) conclude that checklists are helpful to guide beginner programmers during code reviews with the resulting review rates close to the suggested 200 lines of code (LOC)/hour benchmark prescribed by Humphrey. However, they found no concrete evidence that code reviews with checklists will improve the efficiency of the review. For future research, the authors suggest finding methods to improve the effectiveness of checklists and identifying other attributes that might influence the use of code reviews.

As part of a more recent study, Rong et al. (2016) argue that it is not acceptable to wait for novice programmers to become more experienced before quality software can be produced. They therefore set out to determine if software engineering students were able to produce high quality or even defect-free code. Rong et al. (2016) state that the main challenge of using PSP to improve software quality is that the developer must possess the ability to adjust his/her process according to lessons learnt from past experience. They argue that this is impossible for students with limited experience and therefore propose a process called PSP+ that includes additional steps for “Seeding” and “Guided Sharing” (p. 365). The “Seeding” step is presented in the form of a lecture that focuses on best practices in software engineering and defect free programming (DFP). The “Guided Sharing” step includes an instructor-led open discussion on interpretation of process data and sharing of “lessons learnt” after the programming assignments. Based on the results of a comparative study that involved one PSP group and one PSP+ group, Rong et al. (2016) conclude that PSP+ showed potential in helping students produce defect-free programs “without sacrificing productivity” (p. 371). Narrative data collected from both groups suggest that detailed design and code reviews are the most useful practices for achieving DFP. While the PSP group valued code reviews as more important, the PSP+ group (who were initially sensitised to best practices) indicated that detailed designs was the most useful practice for achieving DFP.

Summary

Based on the findings of the various educational PSP studies discussed in this section, both positive and negative effects of the use of PSP principles have been noted. On the positive side, the use of a defined and measurement process made students more aware of the shortcomings in their current development practices (Börstler et al., 2002; Carrington et al., 2001; Grove, 1998) and had a positive impact on the students' attitudes toward software process improvement (Grove, 1998). After being introduced to code reviews, students were able to remove defects earlier in the development life cycle (Grove, 1998; Hou & Tomayko, 1998; Jenkins & Ademoye, 2012; Prechelt, 2001; Rong et al., 2016; Towhidnejad & Salimi, 1996) and consequently spent less time in testing/debugging. Early defect removal or defect prevention is an attribute of a "mature process" that can be applied on "immature" software processes (Prechelt, 2001, p. 57). Detailed designs were highlighted as a major contributor to defect prevention (Grove, 1998; Rong et al., 2016).

On the negative side, the following problems were noted with regard to the use of PSP principles:

- Students *struggled to capture accurate and reliable data* (Carrington et al., 2001; Grove, 1998; Prechelt, 2001; Towhidnejad & Salimi; 1996). In some studies this was attributed to students' *inability to distinguish between the development phases* (Carrington et al., 2001; Grove, 1998). Carrington et al. (2001) emphasised that students struggle to distinguish between the various phases when they code and test one line of code at a time. Prechelt (2001), however, attributed the inaccurate data to a *lack of self-discipline* on the students' side and labelled it as a "personality issue" (p. 61). In cases where students were *struggling with basic programming skills*, the additional tasks of capturing data caused a cognitive overload (Carrington et al., 2001). In this regard, Börstler et al. (2002) recommend that PSP topics should only be introduced in more advanced programming courses, while Grove (1998) instead created and used a scaled down version of PSP.
- Many students *abandoned the use of PSP practices*. In a number of studies, students objected to process measurement because they either found it *unrelated to software development* (Bullers, 2004; Towhidnejad & Salimi, 1996;

Williams, 1997) or regarded it as *extra effort* (Börstler et al., 2002; Carrington et al., 2001; Hou & Tomayko, 1998). In some studies, students objected to the use of the PSP defined process because this process was *not compatible with their current development practices* (Carrington et al., 2001) or they regarded it as *too strict* and therefore could not see the potential benefits of using this disciplined process (Börstler et al., 2002).

- Students *struggled with the application of PSP principles* even though they demonstrated accurate theoretical knowledge of these principles (Williams, 1997).
- Students with *limited software development experience* are *unable to adjust their processes according to lessons learnt from past experience* (Rong et al., 2016).

Humphrey (1999) claims that one of the biggest challenges in software development is to convince software developers to adopt better practices as they tend to stick to a personal process that they have developed from the first small program they have written. In the next section, various technology adoption models are considered.

2.4 Technology Adoption Models

There are numerous theoretical models that can be used to examine individual intentions to adopt information technology tools. The key models are summarised in Table 2-8.

Although software development methodologies, and more specifically QATs, cannot necessarily be regarded as technological tools, a study conducted by Riemenschneider et al. (2002) provides empirical evidence that established models of individual intentions for tool adoption can be used to provide insights into methodology adoption by software developers in a large organisation. For their study, Riemenschneider et al. (2002) selected the following existing technology acceptance models: Technology Acceptance Model (TAM); TAM2; Perceived Characteristics of Innovating (PCI); Theory of Planned Behaviour (TPB); and Model of Personal Computer Utilisation (MPCU). After evaluation of these five models, Riemenschneider et al. (2002, p. 1139) identified the following 12 constructs (which include both common and unique constructs from the selected models) as being appropriate in the context of software development methodology adoption: behavioural intention, usefulness, ease of use, subjective norm, voluntariness, compatibility, result demonstrability, image, visibility, perceived behavioural control (internal), perceived behavioural control (external), and career consequences.

Table 2-8: Theoretical models of individual adoption

Models	Core Constructs	Source
TRA	Attitude toward behaviour; Subjective norm.	(Fishbein & Ajzen, 1975)
TPB	Attitude toward behaviour; Subjective norms; Perceived behavioural control.	(Ajzen, 1985; 1991)
TAM	Perceived usefulness; Perceived ease of use.	(Davis, 1989)
TAM 2	Perceived usefulness; Perceived ease of use; Subjective norms.	(Venkatesh & Davis, 2000)
UTAUT	Performance expectancy; Effort expectancy; Social influence; Facilitating conditions.	(Venkatesh et al., 2012)
DOI	Innovation attributes; Innovators characteristics.	(Rogers, 1983)
SCT	Outcome expectations (Performance); Outcome expectations (Personal); Self-efficacy; Affect; Anxiety;	(Compeau & Higgins, 1995)
MM	Extrinsic motivation; Intrinsic motivation.	(Davis et al., 1992)
MPCU	Job-fit; Complexity; Long-term consequences; Affect towards use; Social factors; Facilitating conditions.	(Thompson et al., 1991)
IS Success	System quality; Information quality; Service quality; System use; User satisfaction; Net benefits.	(DeLone & McLean, 2003)
PCI	Relative advantage; Ease of use; Image; Visibility; Compatibility; Result demonstrability; Voluntariness; Trialability.	(Moore & Benbasat, 1991)

In using the newly developed model to evaluate the intentions of a group of developers in a large organisation to use a newly introduced software methodology, Riemenschneider et al. (2002) identified usefulness, compatibility, subjective norm and voluntariness as significant determinants of methodology adoption. They further argue that the introduction of a new methodology will not be successful if the software developers do not regard it as useful. A new methodology will be regarded as useful when it enables developers to be more productive and increase their performance level. If the developers regard the methodology as useful, the likelihood of adoption is higher if this new process is compatible with their current processes. They also warn that developers who believe in the usefulness and compatibility of a software process innovation might avoid using the innovation because of the negative views of peers and supervisors (subjective norm) who oppose the use thereof. The significance of subjective norm as a determinant was attributed to the importance of teamwork in software development. Riemenschneider et al. (2002) suggest that new methodologies should be introduced step-by-step. The same principle is followed in the PSP course as developers are gradually introduced to the PSP framework objectives through seven levels (Humphrey, 1994).

In describing a conceptual framework for examining the acceptance of agile methodologies, Chang and Tong (2009) identified perceived usefulness, perceived ease of use, perceived compatibility, result demonstrability and perceived maturity as possible determinants of methodology acceptance. Chang and Thong (2009) argue that the insignificance of result demonstrability in Riemenschneider et al.'s study may be attributed to the long development cycles of real-world methodologies – preventing software developers "from observing the results in a short period of time" (p. 811). They (Chan & Thong, 2009) acknowledge that the adoption of innovations often require a radical change in the developers' existing work practices. If the innovation is not compatible with the developers' current practices, they are unlikely to adopt it - as also noted in Riemenschneider et al.'s study (2002).

Various other studies have also identified perceived usefulness as a significant factor in predicting professional developers' intention to use software process innovations such as programming languages (Agarwal & Prasad, 2000) and CASE tools (Iivari, 1996). Overall, these studies also suggest that an innovation is only likely to be accepted if it is perceived as being useful in increasing job performance (Chan & Thong, 2009).

2.5 Summary

In this chapter, four key areas applicable to this study were considered. Firstly, theoretical background regarding the software quality management principles and practices that are of particular relevance to this study were presented. As part of this background discussion, the meaning of quality in the context of software development was defined; TQM and the underlying philosophies that contributed to the forming of the TQM movement were discussed; the suitability of TQM principles for the software engineering discipline were explored; and an overview of two software quality improvement frameworks (CMM and PSP) that are built on the TQM philosophy were provided. A detailed discussion of specific principles and practices of the PSP framework relevant to this study were also provided. Secondly, given the higher education context in which this study is situated, general programming and design problems that could influence the quality of software developed by students were considered. Thirdly, various attributes and/or challenges that influenced the success of higher education students' attempts to improve the quality of their development practices through the use of PSP principles were presented. Finally, various theoretical models that could be used to examine individual intentions to adopt technology tools were described.

Over the past 40 years, numerous studies have been conducted (Lister et al. 2004; McCracken et al., 2001; Soloway et al., 1982; Utting et al., 2013) to evaluate the programming performance of undergraduate Computer Science (CS) students (often referred to as 'novice programmers') and identify possible reasons for the low quality of programs developed by these novices. Even though these landmark studies placed a huge emphasis on the relation between problem solving skills and the quality of the final product, the quality of the processes followed by the novices were not measured. Software quality improvement efforts need to consider both product quality and process quality (Kan et al., 1994). Defects can have a major impact on both of these quality categories (Humphrey, 2005). The underlying philosophies of TQM have been used successfully to improve the quality of various manufacturing industries (Crosby, 1979; Deming, 1986; Feigenbaum, 1983; Ishikawa, 1985; Juran, 1999). These philosophies are focused on defect prevention to lower the cost of rework. Quality improvement mechanisms should be built-in throughout the whole development process to reduce the time spent in testing. All process improvement efforts must be measured in order to be improved. The key to process improvement is the understanding of measurement data. The individuals who create the product must be committed to take responsibility for the quality thereof and have the ability to adjust their performance based on measurements. Kan et al. (1994) argue that these TQM philosophies can also be used to improve the quality in the software development industry. The CMM and PSP software process improvement frameworks are based on the TQM philosophies. The CMM is specifically aimed at addressing software process improvement, software process assessment and software capability evaluations within both large and small organisations (Paulk et al., 1993). PSP is a continuous improvement process designed to guide individual software developers in increasing their planning accuracy and improving the quality of their software programs (Humphrey, 2000; Humphrey, 2005). The PSP process improvement strategies are based on the following principles: the developer must use a defined process and measurement data to improve his/her performance; and every developer is unique and must therefore use personal data to plan his/her work. The PSP quality management strategy is based on the following principles: personal responsibility for individual quality, early defect removal, and defect prevention. These principles are addressed through the use of the following strategies: design reviews, code reviews, design templates and quality measures. These strategies can therefore be collectively referred to as quality appraisal techniques (QATs). A number of researchers have reported on attempts to incorporate PSP principles and strategies in order to improve the quality of novice programmers' software development practices (Börstler et al., 2002; Bullers, 2004; Carrington et al., 2001; Grove, 1998; Hou & Tomayko, 1998; Jenkins & Ademoye, 2012; Prechelt, 2001; Prechelt & Unger, 2001; Rong et al., 2012; Rong et al., 2016; Runeson, 2001; Towhidnejad & Salimi, 1996; Williams, 1997). One of the major PSP quality strategies is focused on the creation of complete designs (Humphrey, 2005). Several attempts have been made to develop models to evaluate the quality of novice programmers' software designs (Chen et al., 2005; Eckerdal et al., 2006a; 2006b; Hu, 2016; Loftus et al., 2001). Of these researchers, only Hu (2016) was able to gain deeper insight into his students' design processes by including a product implementation and reflective (narrative) component to his study. Even though he suggests that design courses should have a stronger focus on process knowledge, no mention is made of process measurements. The PSP framework includes measurements that can reveal deeper insights regarding the quality of design, as well as the quality of the entire software development process. However, none of the educational researchers mentioned above recognised the potential value of the PSP framework as an evaluation model to assess the quality of individual development processes.

This gap was addressed in the first research activity of this study (Phase 1) where the PSP framework was used as an evaluation framework to address the following research question:

> *RQ1.1: What is the quality of the typical software development processes followed by novice programmers?*

CS researchers have mentioned both successes and problems regarding their students' use of the defined PSP processes and QATs (i.e. complete designs, design reviews, code reviews and measurements). Through the use of PSP principles, novice programmers became more aware of their actual development processes and had a more positive attitude towards software process improvement (Börstler et al., 2002; Carrington et al., 2001; Grove, 1998). After the introduction of code reviews, novice programmers were able to remove defects earlier in the development life cycle (Grove, 1998; Hu & Tomayko, 1998; Jenkins & Ademoye, 2012; Prechelt, 2001; Rong et al., 2016; Towhidnejad & Salimi, 1996). Detailed designs were highlighted as a major contributor to defect prevention (Grove, 1998; Rong et al., 2016). Despite these positive outcomes, a number of problems were also noted. In some studies, the novice programmers struggled to capture accurate and reliable data (Carrington et al., 2001; Grove, 1998; Prechelt, 2001; Towhidnejad & Salimi; 1996). There were also cases where the novices completely abandoned the use of PSP practices. These novices regarded the use of defined measured processes as (1) unrelated to software development (Bullers, 2004; Towhidnejad & Salimi, 1996; Williams, 1997); (2) extra effort (Börstler et al., 2002; Carrington et al., 2001; Hou & Tomayko, 1998); (3) incompatible with their current development practices (Carrington et al., 2001); or (4) too strict (Börstler et al., 2002). The above mentioned successes and problems can serve as a starting point in identifying attributes that could influence novice programmers' use of PSP principles. Williams (1997) specifically mentions that the novice programmers in his study struggled with the application of PSP principles even though they demonstrated accurate theoretical knowledge of these principles. This could suggest a difference between how novices perceive their development processes and what they actually do. A number of authors have proposed the use of narrative data to gain better insight into students' development processes (Hu 2016; McCracken et al., 2001). Humphrey (1999) suggests that educators must shift their focus from the programs that the students create to the data of the processes the students use. Rong et al. (2012) also suggest that there could be other attributes that might influence students' use of quality processes. In an attempt to form a better understanding of novice programmers' actual development processes (including their use of QATs) when compared to their perceived processes as well as the attributes that could potentially influence novice programmers' use of QATs, the second research activity of this study (Phase 2) was directed by the following two research questions:

> *RQ2.1: How do novice programmers' perceived software development processes (including their use of QATs) differ from their actual processes?*
>
> *RQ2.2: What are the attributes that could potentially influence novice programmers' use of QATs?*

Humphrey (1999) claims that one of the biggest challenges in software development is to convince software developers to adopt better practices as they tend to stick to a personal process that they have developed from the first small program they have written. Runeson (2001) also found it easier to convince first-year students (who do not yet have well established development habits) to use PSP practices as part of their natural development process. Students in Towhidnejad and Salimi's (1996) study found it easier to adopt code reviews as part of their quality improvement process since they regarded it as more closely related to programming. Various researchers have reported on the challenges they experienced in motivating their students to adopt PSP practices (Börstler et al., 2002; Bullers, 2004; Carrington et al., 2001; Hou & Tomayako, 1998; Towhidnejad & Salimi, 1996; Williams, 1997). Prechelt and Unger (2001) claim that in their study, the low level of PSP usage could be explained by the "different temperaments of the programmers", the small size of the PSP tasks, as well as the absence of "a working environment which actively encourages PSP usage" (p. 471). They call for further investigations into the technical, social and organisational attributes (beyond the level of training and infrastructure provided) that might influence the use of PSP methods. Rong (2012) calls for further investigations into attributes that might influence the use of QATs such as code reviews. A number of authors have shown that adoption models that were originally developed to examine individual intentions to adopt information technology tools can also be used for software process and software development methodology adoption (Agarwal & Prasad, 2000; Chang & Tong, 2009; Iivari, 1996; Riemenschneider et al., 2002). In the third research activity of this study (Phase 3), an adapted version of Riemenschneider et al.'s (2002) methodology adoption model was used to examine the intent of novice programmers to adopt QATs as part of their natural development process. Phase 3 was directed by the following research question:

RQ3.1: What are the factors that could influence novice programmers' intent to adopt QATs?

In the next chapter, the methodology and results of the Phase 1 research activity are discussed.

Chapter 3: Evaluating the quality of novice programmers' typical software development processes (Phase 1)

As explained in the introductory chapter of this thesis report (see Section 1.2), this research study followed a mixed-methods approach based on Plowright's (2011) FraIM framework. This study was divided into three phases in order to distinguish between the three main sources of data (cases). Each of these phases also followed a different methodology. It was therefore deemed more appropriate to create a separate discussion for the methodology and results of each research activity. Consequently, the methodology and results of the Phase 1 research activity are discussed in this chapter (Chapter 3), while the details of each of the other two phases are covered in Chapter 4 (Phase 2) and Chapter 5 (Phase 3) respectively.

The aim of the Phase 1 research activity was to use the PSP framework (see Table 2-2) as a basis for evaluating the quality of novice programmers' typical software development processes. In addressing this aim, Phase 1 was guided by the following research question:

RQ1.1: What is the quality of the typical software development processes followed by novice programmers'?

Using Plowright's basic FraIM structure as a roadmap (see Figure 1-1), this chapter describes both the methodology and the results of the Phase 1 research activity. Firstly, the selection of a survey approach as the main data source management strategy for this phase of the study and sampling decisions that were made are outlined. Secondly, the data collection process and analysis procedures are explained. Thirdly, a discussion of the evidence and claims that transpired from this part of the empirical investigation is provided. Finally, a summary of the main findings from Phase 1 is presented. To inform the emergent nature of this research study, further investigations are also suggested.

3.1 Cases

For this part of the investigation, a survey approach (Plowright, 2011) was followed to assess the quality of novice programmers' typical software development processes using the PSP framework as an evaluation model.

3.1.1 Data source management strategy

From a methodological perspective, a survey approach was regarded the most suitable data source management strategy for Phase 1. The main information source for this investigation included a large number of potential cases (approximately 500), which included all the undergraduate CS students who were registered in the Department of Information Technology at the selected UoT at the time. The potential cases were already grouped by year level as well as the various courses they were enrolled for. The researcher, however, still had some control in selecting cases (participants) based on these groupings. A survey approach allowed the researcher to draw participants from "naturally occurring groups" (Plowright, 2011) while making provision for some interruption of the participants' natural setting. By involving the participants in a data collection activity during one of their normal scheduled classes, the degree of naturalness in which the data were collected ("ecological validity") would be lower than that of a case study approach. The intention of this survey approach was, however, not to capture accurate portrayals of the participants' behaviour in their natural setting, but rather to gather reflections on actions and behaviours that they have followed in the past.

3.1.2 Sampling decisions

The research population for this survey included all first-, second- and third-year CS students at the selected UoT, giving a total population of approximately 500 students. These students are enrolled in one of two academic streams, namely software development or web development. The intent of the survey was to uncover general information (regarding software development processes) with a limited amount of detail but enough responses (from novice programmers) to provide an opportunity for generalisation within the study population. Therefore, a purposive sampling strategy (Babbie, 2010) was used to specifically select students enrolled for the software development stream. The sampling strategy can also be regarded as convenient

(Patton, 2015) since all the students were studying at the same department where the researcher was employed. This minimised any potential logistical issues as all participants were located geographically at the same university/department and were easily accessible to the researcher. The resulting sample comprised all the students enrolled for the three compulsory software development courses (one for each of the three year levels) at the selected institution. Since the web development students could also enrol for these courses (as electives), these students were not excluded from the original population.

3.2 Data Collection Methods

For this survey, data was collected by means of ‘asking questions’ in a paper-based self-completion questionnaire ~~(~~Plowright, 2011) (see Appendix A). The purpose of the questionnaire was to evaluate the software development processes typically used by undergraduate CS students at the selected UoT. The questionnaire integrated both structured (close-ended questions) and less structured (open-ended questions) approaches to asking questions (Babbie, 2010). The questionnaire was divided into two sections: demographic information and software development processes. Four different types of questions [Yes/No, multiple choice (either single or multiple answer), rating and open-ended] were used to elicit responses from the participants.

In the demographic section, participants were only asked to enter the subject code of their software development module. This information was used to verify the academic year-level of each participant.

In the software development processes section, nine questions were included to gather information regarding the participants’ perceptions of the software development processes they typically follow while working on programming assignments (see Table **3-1**). Each of these questions were specifically structured to relate to one of the levels of Humphrey’s (2005) PSP framework (PSP0, PSP1 or PSP2).

Table 3-1: Origin of questionnaire questions

Category	Related questions	Question type	PSP level
Software development process and basic measurement	Q1: Percentage time spent in development phases	Open-ended (totalled to 100%)	PSP0
	Q3: Software Life Cycle model	Open-ended	PSP0
	Q8: Record defects	Yes / No checklist	PSP0
	Q9: Record actual time in phases	Yes / No checklist	PSP0
Estimation and planning	Q10: Use of estimation technique	Yes / No checklist	PSP1
	Q11: Estimation Technique	Open-ended	PSP1
Quality management and design	Q2: Use of defect removal strategies	MCQ (multiple answer)	PSP2
	Q4: Most efficient defect removal strategy	MCQ (single answer)	PSP2
	Q5: Design modelling technique	MCQ (multiple answer)	PSP2.1
Performance	Q6: Average mark for programming assignments	Rating (10-point scale)	n/a
	Q7: Reason for failure	MCQ (single answer)	n/a

An open-ended question (Q3) was included to determine which software life cycle participants typically used. Humphrey claims that most developers do not know how to use operational processes (2005:7) and prescribes a process structure consisting of eight major phases (see Section 2.3.1). Six of these phases (planning, design, coding, compile, test and post-mortem) form part of the PSP0 baseline process (Humphrey, 2005). Based on Humphrey's (2005) suggestion that in .NET programming environments compile defect time should be recorded under the testing phase, the compile and test phases were combined into one phase called testing. The post-mortem phase was also removed because students were generally unfamiliar with this concept. Q1 (time spent in development phases) therefore only included planning, design, coding and testing as the major software development phases. The basic measurements of PSP0 include time-in-phase data and defect data (Humphrey, 2005). Participants were therefore also asked to indicate, by means of Yes/No questions, if they keep record of the most common mistakes they make (Q8) and the actual time they spend in the different development phases (Q9).

PSP1 is concerned with software estimation and planning. The PSP framework proposes a planning framework that uses a PROBE technique (Humphrey, 2005) to estimate the size and duration of a project. The questionnaire firstly included a Yes/No question to establish if the participant uses estimation techniques to determine the size of and time spent on his/her assignments (Q10). This was followed by an open-ended question (Q11) in which the participant was probed to describe the technique(s) that he/she uses for time and/or size estimation.

PSP2 is concerned with quality management and design. Quality management includes the defect removal strategies (design review, code review and testing) that is used by a programmer. The questionnaire therefore included a question (Q2) to determine which defect removal strategies (design review, code review and testing) the participant uses, as well as a question (Q4) where the participant could indicate which one of these strategies he/she regards as the most effective. In the context of PSP2, design includes the different techniques that programmers use to model their designs. The PSP framework proposes a design specification structure with four categories in which specific UML diagrams (see Table 2-3) can be used to model the designs (Humphrey, 2005). The design question (Q5) therefore included the following six response options: “I never do any designs”, “Class diagrams”, “State charts”, “Flowcharts and/or Pseudocode”, “Sequence and/or Activity diagrams” and “Use cases”. Two non-PSP related questions (Q6 and Q7) were added to the questionnaire to elicit the participants’ perceptions regarding their overall programming performance.

The questionnaire was distributed during normal lectures and participants were given 20 minutes to complete it. A total of 251 participants completed the survey. This sample included 74 (29.48%) first-year, 113 (45.02%) second-year and 64 (25.50%) third-year students.

3.3 Data Analysis

All the data collected through the questionnaire was captured in a Microsoft Excel spreadsheet. Narrative data collected from the open-ended questions was imported into NVivo for Mac, Version 11. An open coding process (Babbie, 2010) was then employed to group related responses. Based on the categories that emerged from the data, related responses to each question were then grouped and converted to numeric data. The next step in the data analysis process was to import all the numeric data in IBM SPSS Statistics for Windows, Version 25.0 for further analysis. After careful data inspections (exploration via box-plots and identification of unusual cases), the data of five participants (three first-years, one second-year and one third-year) were removed from the data set that was used for further analysis and reporting. The final data set therefore included data for 246 participants - 71 (28.86%) first-years, 112 (45.53%) second-years and 63 (25.61%) third-years.

3.4 Evidence and Claims

The discussion in this section is grouped according to the four main question categories set out in Table 3-1: current process and measurement, estimation and planning, quality management and design, and performance.

3.4.1 Software development process and basic measurement

The discussion in this section focuses on the perceived time spent in development phases and the operational process model(s) that students typically follow. Students first had to indicate how much of their development time they typically spent in each of the four main software development phases (planning, design, coding and testing). The students indicated that on average they typically spent most of their development time in coding (49.84%) (see Table 3-2 and Figure 3-1). They seem to spend 24.81% of their time in testing/debugging, which is almost half of their coding time. The perceived time spent in planning (13.32%) is almost equal to the design time (12.03%). The time for planning and design combined was 26.17%. Overall the perception exists that they spent approximately a quarter of their time before coding, half their time during coding and a quarter of their time debugging after coding. Students of all year levels reported almost similar perceived time values, which can be regarded as an indication that a first-year student and a third-year student typically use quite similar development processes.

Table 3-2: Perceived time spent in development phases

Development Phase	**1st Year (N = 71)**	**2nd Year (N = 112)**	**3rd Year (N = 63)**	**All students (N = 246)**
Planning	11.8%	13.2%	15.3%	13.32%
Design	10.5%	12.6%	12.9%	12.03%
Coding	53.1%	47.8%	49.8%	49.84%
Testing/Debugging	24.6%	26.5%	22.1%	24.81%

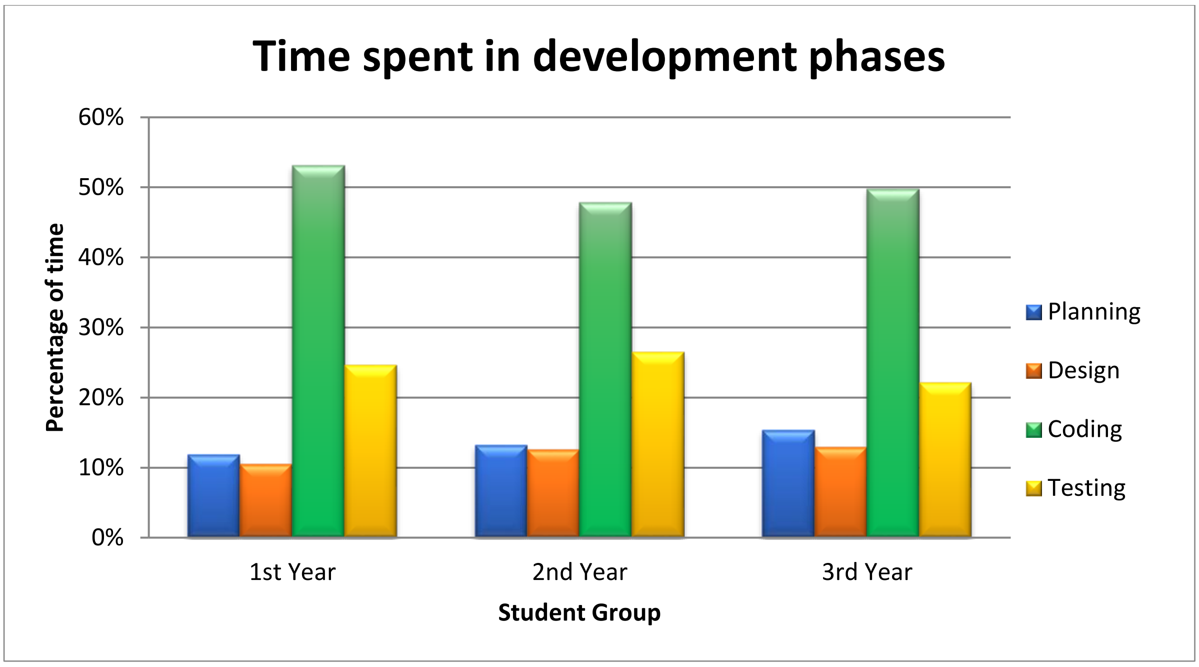


Figure 3-1: Perceived time spent in development phases

When students were asked to indicate their preferred software development life cycle model, the majority of second-year (68.8%) and third-year (63.5%) students indicated that they use “Code-and-Fix” (see Table 3-3 and Figure 3-2). None of the first year students could name their preferred software life cycle. This is not surprising since operational process models are only covered in the second-year curriculum. The senior students’ reliance on code-and-fix strategies serves as an indication that they lack a thorough design phase in their development process and explains why they spent the least amount of time in the design phase (see Figure 3-1).

Table 3-3: Preferred software life cycle

Life cycle	**1st Year (N = 71)**	**2nd Year (N = 112)**	**3rd Year (N = 63)**	**All students (N = 246)**
Don't know	100%	4.5%	4.8%	32.1%
Code-and-fix	-	68.8%	63.5%	47.6%
Open source	-	7.1%	3.2%	4.1%
Waterfall	-	15.2%	14.3%	10.6%
Prototype	-	3.6%	4.8%	2.8%
Agile	-	0.9%	9.5%	2.8%

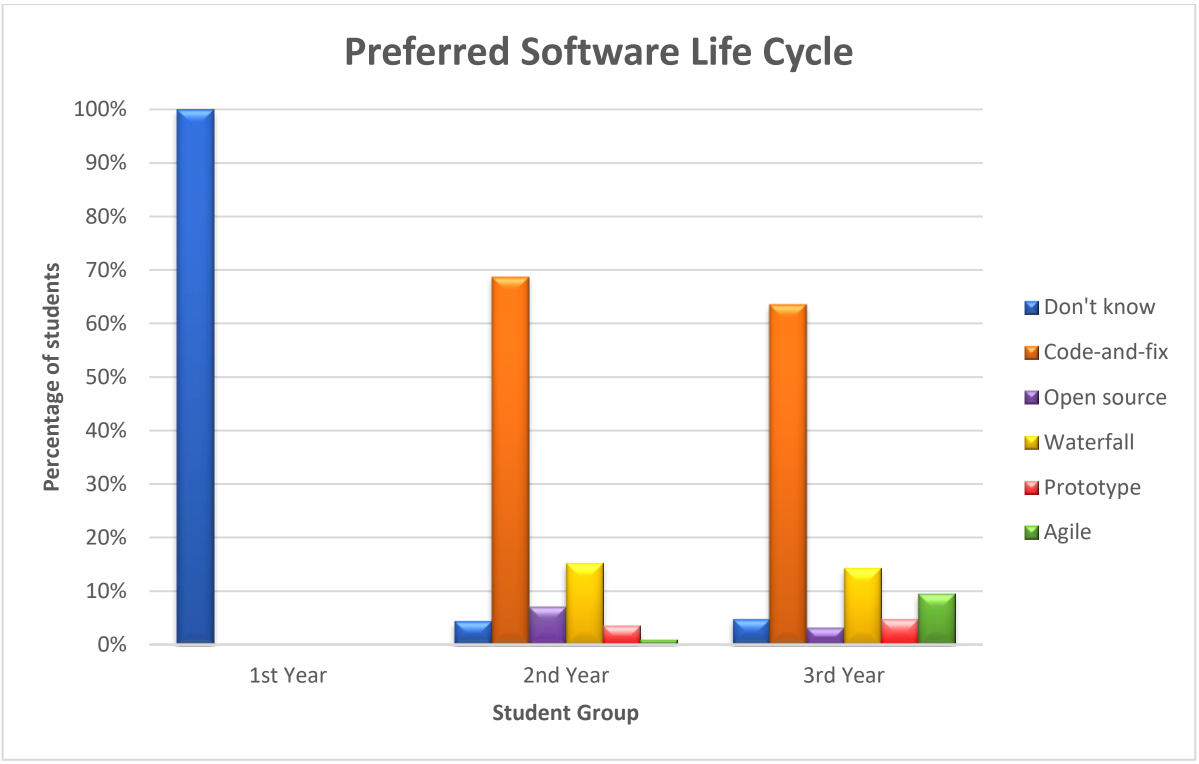


Figure 3-2: Preferred software life cycle

PSP0 prescribes two process measures: “time spent per phase” and “defects found per phase” (Humphrey, 2005, p. 20). This measurement data can be used for planning and managing projects and serve as a performance indicator when making process changes (Humphrey, 2005). Defect measurement data should include the following: the time to fix the defect, the phase in which a defect was injected, the phase in which the defect was removed, and a description of the defect (Humphrey, 2005). The description of the defects should be clear enough so that the programmer can use it during reviews and for defect prevention purposes (Humphrey, 2005). Since only 30.1% of the students indicated that they keep record of the defects they make (see Table 3-4), it can be concluded that the majority of the students have no specific strategy to manage their defects and/or prevent future occurrences of the same defects. The students’ year level also does not have a significant impact on their use of such a strategy.

It is also important to note that the reported percentages of time spent in the various development phases (Table 3-2) should mostly be regarded as individual perceptions since only 16.3% of the students indicated that they keep actual time records (see

Table 3-5). Very few first-year students (11.3%) and even fewer second-year students (3.6%) indicated that they keep record of the actual time they spend in the various development phases. This is in contrast to the much larger portion of the third-year students (44.4%) who indicated that they keep time records.

Table 3-4: Keep record of common defects

Response	1st Year (N = 71)	2nd Year (N = 112)	3rd Year (N = 63)	All students (N = 246)
Yes	31.0%	25.9%	36.5%	30.1%
No	69.0%	74.1%	63.5%	69.9%

Table 3-5: Keep record of actual time spent in software development phases

Response	1st Year (N = 71)	2nd Year (N = 112)	3rd Year (N = 63)	All students (N = 246)
Yes	11.3%	3.6%	44.4%	16.3%
No	88.7%	96.4%	55.6%	83.7%

3.4.2 Estimation and planning

The discussion in this section focuses on the estimation and planning techniques typically used by students. The majority of students (87.8%) indicated that they do not use any time estimation techniques. Of the 30 students who use such estimation techniques, 14 (46.7%) could not name the specific technique they used (see Table 3-6). Seven students (23.3%) indicated that their estimations are based on conceptual designs. Four students (13.3%) indicated that they used the PROBE technique prescribed by the PSP framework (Humphrey, 2005). The first step in the PROBE technique is, however, to create a conceptual design that is used to make size estimations. One student mentioned that he used “functionality and complexity”, which might indicate some resemblance to the “technical complexity factor” (TCF) of function point computation (Schach, 2011, p. 273). Two students (6.7%) indicated that they used “guessing”, but did not mention any evidence of what these “guesses” are based on. Another two students (6.7%) indicated that they used “scheduling”, which is one of the activities that is conducted after estimations are made (Humphrey, 2005).

Table 3-6: Estimation techniques used

Estimation technique	No. of students
Do not know	14
Conceptual design	7
PROBE	4
Guessing	2
Scheduling	2
Functionality and complexity	1

3.4.3 Quality management and design

The main aim of the PSP quality management strategy is to find defects as early as possible in the development life cycle by reviewing artefacts after production (Humphrey, 2005) (also see Section 2.3.4). As part of this strategy, developers' first focus should be on producing a complete design and document it according to the four PSP design templates (Humphrey, 2005). Then the developer can use personal design reviews and code reviews to find defects before testing (Humphrey, 2005). The discussion in the following sub-sections elaborates on evidence collected regarding the design modelling techniques and defect removal strategies typically used by students.

3.4.3.1 Design modelling techniques

The UML modelling techniques used by the students were grouped according to the relevant design template quadrants of the PSP2 framework (Humphrey, 2005) (see Table 3-7 and Figure 3-3). The participants were also provided with a "never do any designs" option and an "other" option where they could specify any additional model(s) used. No responses were recorded for the "additional models" option. Overall, 46.3% of the students indicated that they never do any designs.

The *internal static* category of designs is modelled through pseudocode or flowcharts during the design phase of development. These designs are typically used to model the logical flow of statement execution for program segments. From the data presented in Table 3-7 and the graphical representation in Figure 3-3, it becomes evident that first-year and second-year students who do designs mostly used flowcharts or pseudocode, while third-year students mostly used class diagrams. For all year levels the use of static modelling structures were considerably higher than dynamic modelling structures, which were rarely used.

Table 3-7: Design modelling techniques used

Design quadrant	UML techniques	1st Year	2nd Year	3rd Year	All students
-	No designs used	56.3%	41.1%	44.4%	46.3%
External static	Class diagrams	7.0%	23.2%	34.9%	21.5%
Internal static	Flowcharts Pseudocode	39.4%	49.1%	25.4%	40.2%
External dynamic	Use cases Interaction diagrams	1.4%	2.7%	4.8%	2.8%
Internal dynamic	State diagrams	0.0%	0.0%	0.0%	0.0%

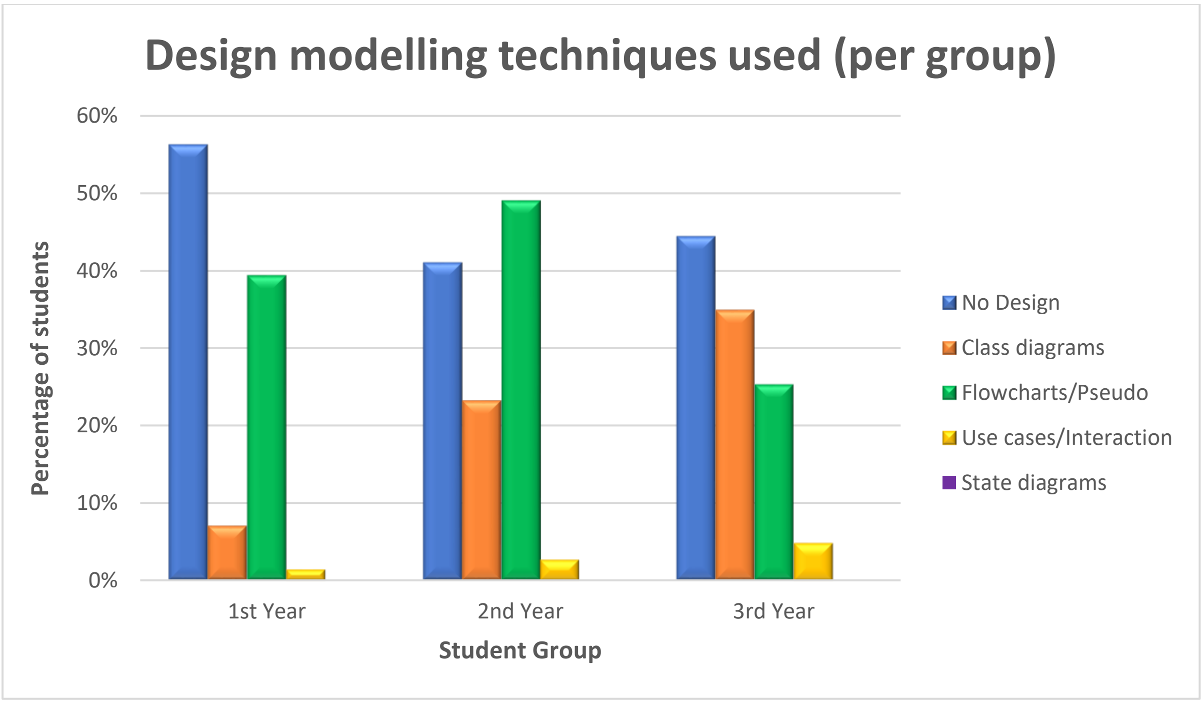


Figure 3-3: Design models used (grouped by year level)

Designs from the *external static* category were seemingly used by 21.5% of the students. These designs, which are modelled with class diagrams, are created during the design phase of development to show the interaction between objects and the relationships between classes. The indicated use of class diagrams showed an increase from first-year (7.0%) to third-year level (34.9%). The first-years only start with object-oriented programming towards the end of their second semester, which explains their limited use of class diagrams. Further analysis of the data revealed a significant correlation at the 0.01 level [Pearson correlation: .251; sig (2-tailed): .000] between the students' year level and their use of external static design models (see Table 3-8). No such correlations were observed for any of the other design quadrants.

Table 3-8: Correlations (year level & design quadrants)

		Year	External static	Internal dynamic	Internal static	External dynamic
Year	Pearson Correlation	1	.251**	.b	-0.099	0.074
	Sig. (2-tailed)		0.000		0.123	0.248
	Sum of Squares and Cross-products	133.740	18.724	0.000	-8.780	2.228
	Covariance	0.546	0.076	0.000	-0.036	0.009
	N	246	246	246	246	246
External static	Pearson Correlation	.251**	1	.b	0.034	0.089
	Sig. (2-tailed)	0.000			0.599	0.165
	Sum of Squares and Cross-products	18.724	41.581	0.000	1.671	1.492
	Covariance	0.076	0.170	0.000	0.007	0.006
	N	246	246	246	246	246
Internal dynamic	Pearson Correlation	.b	.b	.b	.b	.b
	Sig. (2-tailed)					
	Sum of Squares and Cross-products	0.000	0.000	0.000	0.000	0.000
	Covariance	0.000	0.000	0.000	0.000	0.000
	N	246	246	246	246	246
Internal static	Pearson Correlation	-0.099	0.034	.b	1	0.009
	Sig. (2-tailed)	0.123	0.599			0.887
	Sum of Squares and Cross-products	-8.780	1.671	0.000	59.159	0.183
	Covariance	-0.036	0.007	0.000	0.241	0.001
	N	246	246	246	246	246
External dynamic	Pearson Correlation	0.074	0.089	.b	0.009	1
	Sig. (2-tailed)	0.248	0.165		0.887	
	Sum of Squares and Cross-products	2.228	1.492	0.000	0.183	6.801
	Covariance	0.009	0.006	0.000	0.001	0.028
	N	246	246	246	246	246

**. Correlation is significant at the 0.01 level (2-tailed).

b. Cannot be computed because at least one of the variables is constant.

The *external dynamic* category of designs models the interaction of users with a system by means of use cases and interaction diagrams. Since the creation of these diagrams only form part of the second-year curriculum, it might explain the slight increment in the perceived use of these modelling techniques from year one (1.4%) to year three (4.8%). The low usage of use cases and interaction diagrams in the first-year (1.4%) and second-year (2.7%) can be attributed to the nature of the exercises and assignments given on these levels. Most of the exercise and assignment specifications given during the first two years are in an analysed form. There is therefore no need for the students to gather or analyse requirements on these levels. During the third year of study, students have to work on much bigger projects. It is therefore expected that use cases and interaction diagrams will form part of their requirements and analysis process. The data, however, shows that only 4.8% of the third-year students indicated that they made use of use cases and interaction diagrams, which is a possible area of concern. The overall low usage of these diagrams (used by only 2.8% of the students) can be linked to the types of exercises or assignments that the students have to complete as these rarely force them to do "real" or complete requirements and analysis.

The *internal dynamic* behaviour of objects is modelled through state charts. The lack of usage of internal dynamic modelling (not used by any of the students, as indicated in Table 3-7) can be regarded as a direct result of the limited use of external dynamic modelling. Internal dynamic modelling should take place during the analysis phase when user interaction with the system is modelled in the form of object states that change through user interaction (Schach, 2011).

The only design modelling style that seemed to remain fairly constant through the year levels were the use of "no-designs". This is not surprising given the students' heavy reliance on "code-and-fix" as their primary development life cycle (see Figure 3-2).

A design score was also calculated for each student in an effort to obtain a clearer picture of the overall quality of the students' designs. A student received one (1) mark for each of the design template quadrants in which he/she indicated use of the specific modelling technique. These marks (to a maximum of four) were then totalled and converted to a percentage to give the total design score for each student. The average design score for the entire group of participants was 16.16% (see Table 3-9). The drop in average design score from the second year (18.75%) to the third year (16.27%) reveals another area of concern as one would expect a higher number of senior students to produce better quality designs.

Table 3-9: Average design scores per year level

	1st year	**2nd year**	**3rd year**	**All students**
Average design score	11.97%	18.75%	16.27%	16.16%

3.4.3.2 Defect removal strategies

The discussion in this section examines the students' indicated use of three defect removal strategies: design reviews, code reviews and debugging/testing. Even though Humphrey (2005) claims that the "reviewability" (p. 185) of a design is not that important if you review your own designs, he still states that it is impossible to conduct an efficient design review on a poorly documented and incomplete design. As presented in Table 3-10, the students' indicated use of both design reviews and code reviews increased slightly from the first year to the third year of study. Additional data analysis (see Table 3-11) revealed a significant correlation between students' design scores and their indicated use of design reviews as a defect removal strategy [Pearson correlation: .238; Sig. (2-tailed): .000]. The observed increase in average design scores from the first year to the third year (see Table 3-9) could therefore explain the slight increase in the indicated use of design reviews from 7.0% in the first year to 14.3% in the third year (see Table 3-10). The indicated use of code reviews ranged from 43.7% in the first year to 55.6% in the third year (an increase of 11.9% between the year levels). Overall, the students least used design reviews (10.2%), followed by code reviews (50.4%). The most popular technique for removing defects was debugging, with an overall indicated usage of 93.1% (see Table 3-10).

Table 3-10: Indicated use of defect removal strategies

Strategy	**1st year (N = 71)**	**2nd year (N = 112)**	**3rd year (N = 63)**	**All students (N = 246)**
Design review	7.0%	9.8%	14.3%	10.2%
Code review	43.7%	51.8%	55.6%	50.4%
Testing/Debugging	88.7%	96.4%	92.1%	93.1%

Table 3-11: Correlations (design score and indicated use of design reviews)

		Design review	Design score (%)
Design review	Pearson correlation	1	.238**
	Sig. (2-tailed)		0.000
	N	246	246
Design score (%)	Pearson correlation	.238**	1
	Sig. (2-tailed)	0.000	
	N	246	246

**. Correlation is significant at the 0.01 level (2-tailed).

Students also had to indicate which one of the three defect removal strategies they regarded as the most effective. Given the majority perception (76.8% of students) that testing/debugging was the most efficient defect removal strategy (see Table 3-12 and Figure 3-4), it is not surprising that an overwhelming majority of students (93.1%) reported that they used debugging for fixing defects (see Table 3-10). This perception might change if more of them keep record of the actual time that they spend in the different development phases as well as the time that it takes to find and fix defects late in development life cycles. Since 46.3% of the students indicated that they did not do any designs (see Table 3-7), it is not surprising that only 2.4% of the students regarded design reviews as the most effective technique for removing defects (see Table 3-12). The data also indicated a slight increase in the perceived effectiveness of code reviews from first-year to third-year level. The slight increase in the use of code reviews is proportional to the increase in students' perception regarding the effectiveness of code reviews as a defect removal strategy (a 13.6% increase from first-year to third-year – see Table 3-12).

Table 3-12: Most efficient defect removal strategy

Strategy	**1st Year (N = 71)**	**2nd Year (N = 112)**	**3rd Year (N = 63)**	**All students (N = 246)**
Design review	0.0%	3.6%	3.2%	2.4%
Code review	19.7%	14.3%	33.3%	20.7%
Testing/Debugging	80.3%	82.1%	63.5%	76.8%

3.4.4 Performance

In order to obtain a reflective response from the students on the causes of failure, they were asked to indicate the average mark that they normally obtained for programming assignments. As indicated in Figure 3-5, the reported average marks form a normal distribution curve around the mean value of 59.7%.

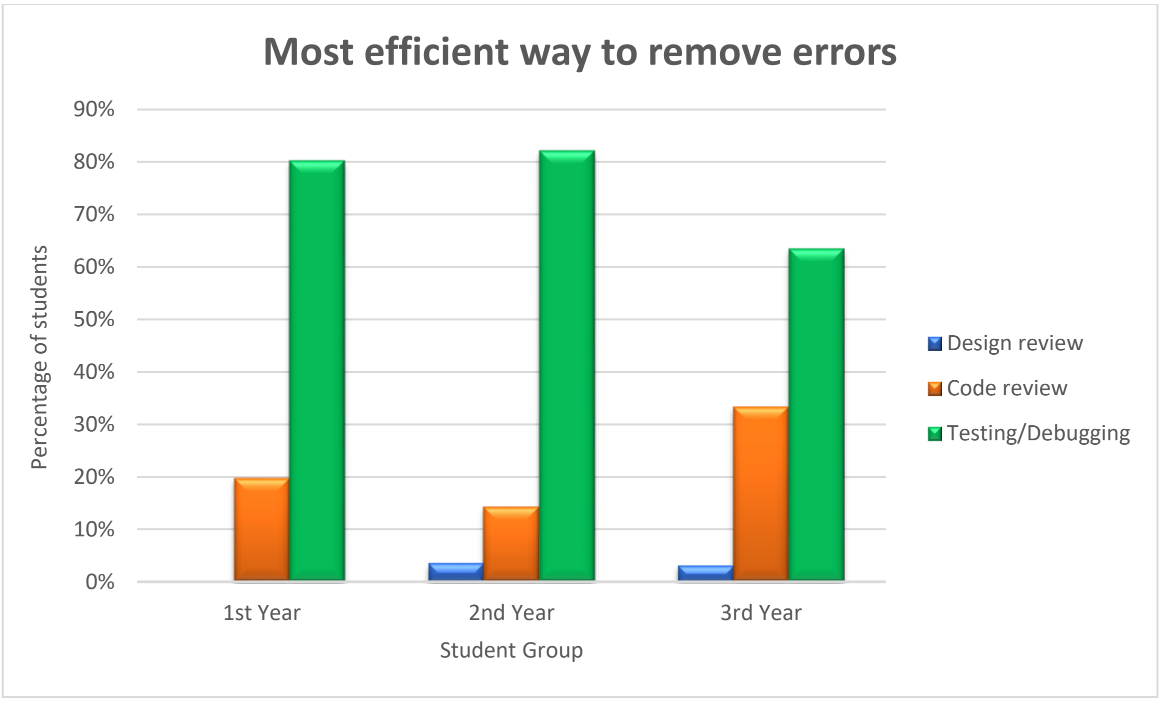


Figure 3-4: Most efficient defect removal strategy

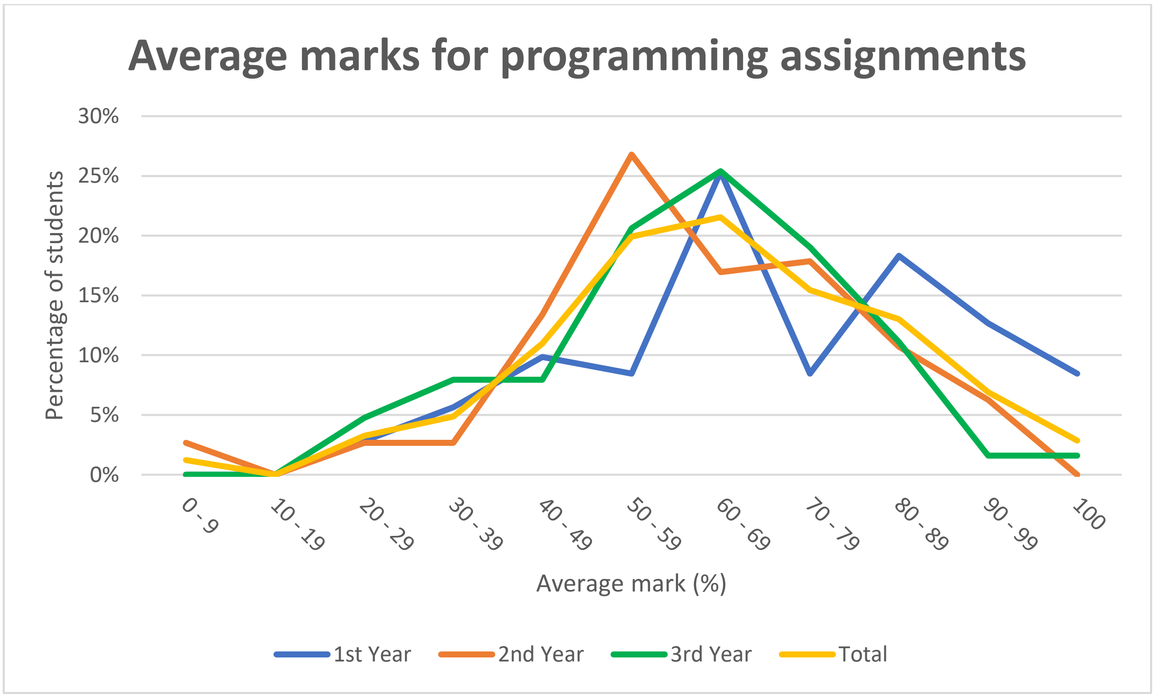


Figure 3-5: Average marks for programming assignments

Students then had to select (from three provided options) the main reason why they did not score full marks in all their programming assignments. The data analysis revealed distinct differences between the responses from students in the different year levels (see Table 3-13 and Figure 3-6). The majority of first-year students (50.7%) believed that their lack of programming skills was the major cause of poor results. This corresponds with Runeson's (2001) findings regarding first-year students' tendency to focus on programming issues rather than process issues. Second-year (47.3%) and third-year (61.9%) students mostly placed the blame on their inability to identify defects. Towards the third year of study, fewer students (15.9%) regarded their 'lack of skill' as the major reason for failure. Although the students in all year levels regarded 'time' as a stumbling block to their success, it was not seen as the major contributor (with values ranging between 16.9% and 27.7%). This is in contrast with students from McCracken et al.'s (2001) study who indicated 'time' as the major reason for their failure to successfully complete programming exercises.

Table 3-13: Main reason for not scoring 100% for assignments

Main reason	**1st Year (N = 71)**	**2nd Year (N = 112)**	**3rd Year (N = 63)**	**All students (N = 246)**
Not enough time	16.9%	27.7%	20.6%	22.8%
Unable to identify defects	26.8%	47.3%	61.9%	45.1%
Lack of skills	50.7%	25.0%	15.9%	30.1%

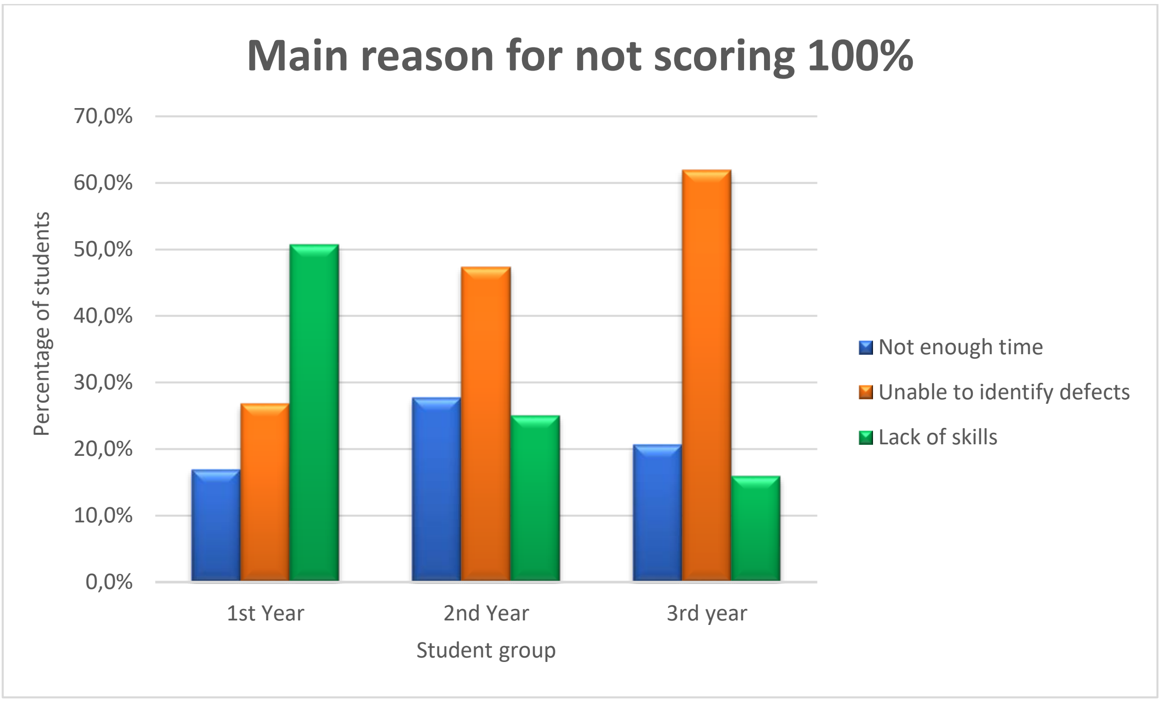


Figure 3-6: Main reason for not scoring 100% for assignments

The students indicated ‘testing/debugging’ as their primary technique for fixing defects (see Table 3-10). Since debugging treats the consequence of a defect, these students are likely to find it a lot more difficult and time-consuming to find defects (Humphrey, 2005). This could also explain why students regarded the ‘identification of defects’ as a major contributor to poor results (see Figure 3-6). This effect is likely to increase towards the third-year when assignments are more comprehensive – making it even more difficult to identify defects (Humphrey 2005).

Without a process that accommodates designs (see Table 3-7), students would spend little time on design reviews (see Table 3-10) and would consequently not be able to identify defects early in the development life cycle. These students therefore have to rely on code reviews and testing/debugging as their primary techniques for finding and fixing defects (as indicated in Table 3-10). Given the students’ heavy reliance on code-and-fix strategies (see Figure 3-2), it is most likely that the thinking process of ‘how to solve a problem’ would occur during the coding phase and not during the design phase. This could explain why students perceived that they spend most of their time in the coding phase (see Figure 3-1).

Without measurement data, a student is unlikely to be aware of his/her actual development process and will subsequently continue to use exactly the same methods and techniques. Software developers tend to stick to a personal process that they have developed from the first small program they have written, and it is difficult to convince them to adopt better practices (Humphrey, 1999).

3.5 Summary

In this chapter, the methodology and results of Phase 1 of this research study were reported. Firstly, the selection of a survey approach as the main data source management strategy, as well as the sampling decisions made were outlined. Secondly, the data collection and analysis strategies were explained. Thirdly, a discussion of the evidence and claims that transpired from this part of the empirical investigation was provided. The main findings can be summarised as follows:

The aim of the Phase 1 research activity was to evaluate the quality of novice programmers' software development processes through the use of the PSP framework as an evaluation model (see RQ1.1). Overall, the evidence revealed that most students on all three levels of study relied on a code-and-fix development process, which resulted in them not spending nearly enough time on designs. Almost half of the students indicated that they never did any designs, while the average design quality score of those who did create designs was calculated as 16.16%. The students' lack of design completeness (indicated by the low design scores) could also explain the limited time spent on designs. Since most of the students (69.9%) indicated that they did not keep track of their most common defects, they are unlikely to realise the effectiveness of complete and accurate designs. The majority of the students (76.8%) regarded testing/debugging as the most effective technique to remove defects, which corresponded with their preferred development life cycle of code-and-fix. As a result of the limited use of designs and the lack of completeness thereof, only 2.4% of the students perceived design reviews as the most effective way to remove defects. Few students (20.7%) indicated that they perceived code reviews as the most effective, but almost half of them (50.4%) indicated that they used it. Even though a large portion of the students (45.1%) believed that their inability to identify defects was the major contributor to failure, most students did not make use of any effective defect removal strategies (QATs). The overall lack of process measurement and process awareness might explain why the students did not adopt more effective development processes and QATs.

It should be noted, however, that these findings are based solely on the students' perceptions regarding the software development processes they typically followed. There are no indications as to how closely related these perceived processes are to their actual software development processes. Actual process measurement data could also provide a better indication of the quality of students' development processes. In addition, a number of authors (Hu 2016; McCracken et al., 2001) have proposed the use of narrative data to gain better insight into students' development processes. Rong et al. (2012) also suggested that there could be other attributes that might influence students' use of quality processes.

Chapter 4 describes a follow-up investigation (Phase 2) that looked more closely at the differences between the perceived and actual development processes followed by a group of novice programmers while using the PSP framework. This included a closer examination of attributes that might influence these programmers' use of QATs.

Chapter 4: Understanding novice programmers' actual development processes and use of QATs (Phase 2)

In Phase 1 of this research study, the PSP framework was used to evaluate the quality of novice programmers' software development processes. Since the data collected as part of Phase 1 was solely based on the students' perceptions of their typical development processes, a number of additional questions were raised regarding the actual processes followed by these students (as outlined in the Conclusion section of Chapter 3). The aim of the Phase 2 research activity was twofold. Firstly, to form a better understanding of the differences between novice programmers' perceived and actual development processes (including their use of QATs) through the use of actual process measurement data (as prescribed by the PSP framework), supplemented by narrative data. Secondly, to identify attributes that could potentially influence novice programmers' use of QATs. This is also in line with Humphrey's (1999) suggestion that educators must shift their focus from the programs that the students create to the data of the processes that the students use. Consequently, Phase 2 of this research study was directed by the following two research questions:

- *RQ2.1: How do novice programmers' perceived software development processes (including their use of QATs) differ from their actual processes?*
- *RQ2.2: What are the attributes that could potentially influence novice programmers' use of QATs?*

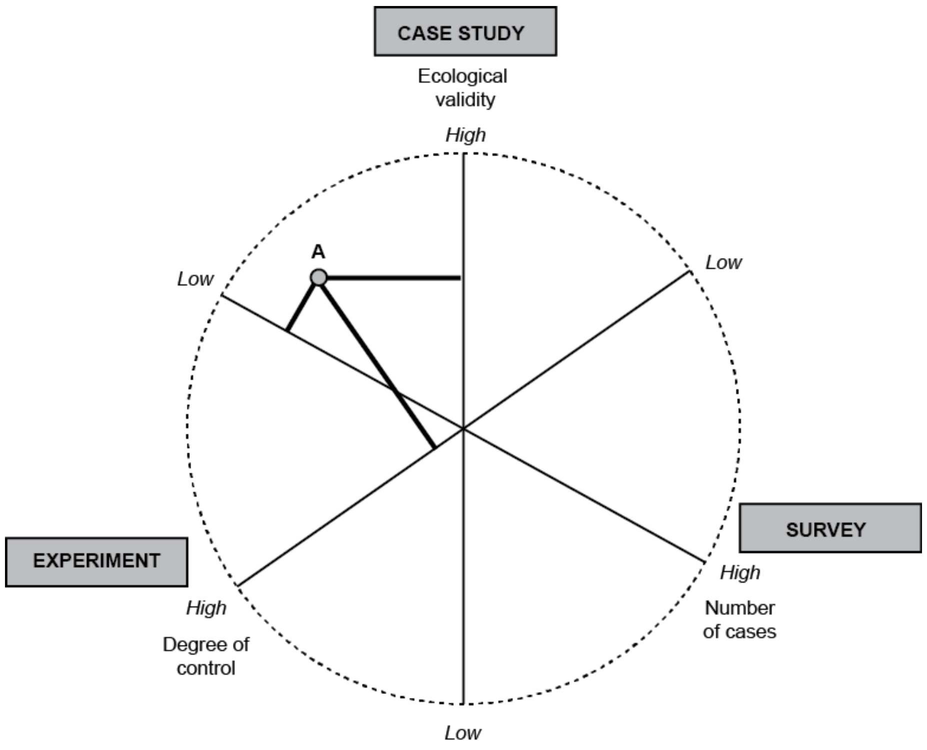
Following an adapted version of Plowright's basic FraIM structure (see Figure 1-1), this chapter describes both the methodology and the results of Phase 2. Firstly, the selection of an experimental case study approach as the main data source management strategy, as well as the sampling decisions made in this phase of the study are outlined. Secondly, the data collection and analysis strategies are explained. Thirdly, evidence of the perceived and actual development process followed by each of the participants are provided. In the course of these overviews, the participants' actual process measurement data and narrative data are also considered. Fourthly, further discussions of the data are offered as guided by the Phase 2 research questions.

4.1 Cases

For this part of the investigation, an integrated experimental case study approach (Plowright, 2011) was followed to gain a deeper understanding of novice programmers' actual development processes and their use of QATs through the collection of both actual process measurement data and narrative data. The discussion in the following sub-sections clarifies the selection of this particular data source management strategy and explains the sampling decisions.

4.1.1 Data source management strategy

For Phase 2, I wanted to collect detailed, in-depth information from a small number of participants, therefore favouring either a case study or an experiment as the preferred data source management strategy. The research would, however, not be conducted in (1) a conventional/natural setting where (2) I had little control over the selection of participants – failing to satisfy two of Plowright's (2011) main criteria for selecting a case study strategy. Although students (novice programmers) regularly do development tasks/exercises in the institutional computer laboratories (the chosen research location), I would manipulate the development conditions by requiring students to follow a defined development process unfamiliar to them – therefore creating an unnatural situation. For Phase 2, I would also not be completely bounded by natural occurring groups within the undergraduate CS student population when selecting research participants. The lack of naturalness in the research location and situation ("ecological validity"), and the relatively high degree of control over participant selection therefore favoured an experimental strategy. The data source management strategy could, however, not be described as a true experiment since this was not a comparative study with treatment and control groups. I therefore decided to settle for an integrated data source management approach that can be described as an experimental case study (see Point A in Figure 4-1). Accordingly, Phase 2 involved a low number of cases, with a medium level of ecological validity where I had a relatively higher level of control over the selection of cases (participants).



(Source: Adapted from Plowright, 2011)

Figure 4-1: Experimental case study as data source management strategy

4.1.2 Sampling decisions

The population for this phase included all third- and fourth-year CS students enrolled for the software development stream at the selected UoT. These students already had intermediate programming skills and experience in the use of software defect removal strategies. Since I wished to select only a small subset of the population for this phase of the research study, I employed purposive sampling (Babbie, 2010). I therefore identified a total of 15 top performing students from my third- and fourth-year courses who I believed possessed the necessary skills to complete the various activities that would form part of the Phase 2 study. A participant information sheet (see Appendix B) was distributed to all the identified students as an invitation to participate in the research activity. The sampling strategy can therefore also be regarded as convenient (Patton, 2015) since I had easy access to the participants. Participants also had to be available during a pre-determined time slot – minimising any potential logistical issues. The resultant sample comprised six male students in their third year of study in the software development stream.

4.2 Methodology and Data Collection Methods

The methodology that was followed for the Phase 2 experimental case study comprised the six steps as summarised in Table 4-1. Data was collected during five of these steps. The various data collection strategies included making observations, asking questions (pre-activity questionnaire, post-activity questionnaire and focus group discussion) and analysing artefacts (Process Dashboard© data and program code) (Plowright, 2011). Each of the six steps and the corresponding data collection strategies (where applicable) are described in more detail in the following sub-sections.

Table 4-1: Summary of methodology and data collection strategies

Activity	Duration	Rationale
1. Participants complete pre-questionnaire	5 – 10 min	Gather information regarding participants' perceived software development processes.
2. Instructor presents performance measurement tutorial.	1 hour	Teach participants to capture process measures and interpret process data.
3. Participants do programming exercise. 4. Instructor makes observations.	3 hours	Capture process measures while doing programming exercise (Participants). Record participant behaviour and questions asked (Instructor).
5. Participants complete post-questionnaire	15 – 20 min	Explore participants' perceptions of process measurement and evaluate their process improvement proposals.
6. Instructor conducts a focus group discussion with participants	20 min	Gain deeper insights into participants' development processes.

4.2.1 Pre-activity questionnaire

The participants first completed a pre-activity questionnaire (see Appendix C) that was based on the questionnaire used in Phase 1 (see Appendix A). Although the majority of the questions were kept unchanged, some changes were deemed necessary to better align the questions with the aim of Phase 2. The question that asked participants to indicate which software life cycle they normally used (compare with Question 1 in Appendix A) was changed from an open-ended question to a multiple-choice (multiple-option) question. The five life cycle options were based on the main life cycles that were identified from the Phase 1 data. For the 'time spent in phases' question (compare with Question 2 in Appendix A), design review and code review phases were added because some Phase 1 participants indicated that they used design reviews and code reviews as defect removal strategies. I also wanted to investigate the perceived and actual time spent during these phases and the effectiveness of these removal strategies.

Two new questions were also added. Participants who indicated that they made use of defect removal strategies (in Question 3) were asked to specify which of the indicated checklists they used (see Question 4), as well as the origin of those checklists (see Question 5). Two questions (asking participants for their student number, initials and surname) were added in Section 2 to allow for matching of data collected in the other Phase 2 steps.

4.2.2 Performance measurement tutorial

After the participants have completed the pre-activity questionnaire, I (as the instructor) conducted a tutorial activity to teach the participants how to log and interpret performance measurement data using the Process Dashboard© software[3]. This software application is part of an open-source initiative to support developers in using PSP or Team Software Process (TSP). The creators of this software believe that Process Dashboard© can remove one of the barriers (sufficient tool support) to adopt PSP or TSP. Process Dashboard© offers the basic functionalities listed in Table 4-2 and is often adopted by PSP/TSP instructors in the offering of the Carnegie Mellon Software Engineering Institute's official PSP and TSP courses. The software creators claim that the major strengths of Process Dashboard© are in its ease of use, flexibility, platform independence and price. From an ease-of-use perspective, the tool optimises the ease of collecting common metrics (time and defects) and provides easy access to process scripts to guide developers through hierarchically arranged tasks. The tool coexists easily with other development environments because of its small screen footprint.

As part of the process measurement tutorial, participants worked on a programming exercise (see Appendix D) so that they could practice the capturing of process measurements with Process Dashboard©. I started the tutorial with a discussion of the

[3] https://www.processdash.com

generic PSP2.1 process script, which gave an overview of the different development phases and the required steps of each phase. Specific attention was also paid to defect types and examples of each. The participants then completed the tutorial exercise while following the PSP2.1 process script and capturing measurements. I adapted the tool so that the participants only logged actual data, and not any planning or estimation data.

Table 4-2: Process Dashboard© functionalities

Functionality	**Description**
Data collection	Time, defects, size for plan and actual data
Planning	Integrated scripts, templates, forms and summaries, PROBE
Tracking	Earned value
Data analysis	Charts and reports aid the analysis of historical data trends
Data export	Export data to excel

As part of the tutorial, I also discussed the analysis and interpretation of time and defect data at the end of the exercise. One hour was set aside for completion of the entire tutorial. This included the instructor-led explanations and discussions. No data (for research purposes) was collected during this step.

4.2.3 Programming exercise

After the tutorial, the participants completed an individual programming exercise (see Appendix E) during which they had to capture performance data using the Process Dashboard© software. The purpose of the exercise was to:

1. Use Process Dashboard© to capture process measurement data while doing a programming assignment following the PSP2.1 process script.
2. Capture the following process data through Process Dashboard©:
   - Time spent in development phases.
   - Defects injected and removed in specific phases.
   - Time spent removing defects.
   - Size of the product.
3. Interpret data collected through Process Dashboard©.

For the programming exercise, the participants had to implement the code to simulate the “Quick Pick Option” of the South African National Lottery (LOTTO$^{©}$) draw. The participants received an extensive background document on how “LOTTO” draws work (see Appendix F). In this “Quick Pick Option” program, a user first has to select the number of player lotto draw records that should be generated. The requested number of records are then generated randomly, sorted and written to a text file. The participants could use any resources, including the Internet, to complete this activity. While they worked on the individual programming exercise, I moved between the participants and recorded any relevant observations as well as all questions asked by the participants. Participants were given three hours to complete the programming exercise.

4.2.4 Post-activity questionnaire

After this exercise, the participants had to complete a post-activity questionnaire (see Appendix G) that consisted of mostly open-ended questions. The purpose of this questionnaire was to explore the participants’ views regarding the capturing and interpreting of process measurement data and their beliefs on how this data could be used to improve their personal development process. The first part of the questionnaire (Questions 1 to 3) was used to determine the difficulties that the participants encountered when capturing time and defect data with Process Dashboard$^{©}$. Question 4 required the participants to reflect on the process they followed when doing reviews and to propose improvements that could reduce testing time (based on their recorded time-in-phase data). Question 5 focused on defect data and asked the participants to propose techniques for removing defects earlier in the development life cycle. Question 6 was an open-ended question where the participants had to reflect on the value of process measurement data. Question 7 was another open-ended question asking participants for proposals for personal process improvement.

4.2.5 Focus group discussion

For the final step of Phase 2, I initially planned to conduct a focus group discussion with all six participants. In order to provide me with enough time to review the questionnaire responses and artefacts created by each participant during the preceding Phase 2 activities, these interviews were scheduled for the next day. Only three of the six participants, however, showed up for the scheduled discussion. The other three participants declined participation in the discussion (even after they were given an option to reschedule). During this discussion, open-ended questions were used to gather narrative data regarding the participants' development processes. These questions focused on the actual process that each of the three participants followed to solve the given programming problem.

4.3 Data Analysis

The numerical data collected through the two questionnaires was analysed in Microsoft Excel, while narrative data from the questionnaires and the focus group discussion was analysed in NVivo for Mac, Version 11. The built-in Process Dashboard$^{©}$ functionality was used to export a summary of each participant's captured process measurement data to Microsoft Excel spreadsheets for further analysis and comparison. For the purpose of this analysis (and the discussions to follow), data collected through the pre-questionnaire were labelled as "perceived" since it painted a picture of the software development processes, as well as the basic measurement, quality management and design strategies that the participants ***thought*** they typically used in their programming assignments. In contrast, the data collected through artefact analysis (process measurement data and program code) were labelled as "actual" since it was collected or created during or in direct response to an actual programming activity. Since Phase 2 involved six participants only, I was able to do an in-depth analysis of each participant's individual data. During the analysis process, I used the various guidelines as set out in Humphrey's (2000) PSP quality measures (see Table 2.2) to evaluate the collected data.

The first step in the analysis was to analyse the pre-questionnaire data. Firstly, the perceived time-in-phase data was analysed to determine if the participant typically spent enough time on the relevant strategies (design, design review and code review) to allow for the early removal of defects (before testing). The following time-in-phase recommendations are made as part of the PSP quality measures:

- Design time should at least be the same as coding time.
- Design review time should at least be half of design time.

- Code review time should at least be half of coding time.

As part of the pre-questionnaire data analysis, the following data comparisons and/or evaluations were also made for each participant:

- The perceived testing time was compared to the perceived time spent on techniques to remove defects earlier.
- The perceived development process was compared to the perceived design modelling techniques and design time.
- The perceived choice of defect removal strategies was evaluated based on the perceived time indicated for performing the defect removal strategies and compared to the perception of the effectiveness of the strategies.
- The perceived use of checklist(s) was compared to the perceived use of defect removal strategies.
- The perceived source of the checklist(s) was compared to the participant's perceived capturing of time and defect data.

Secondly, the following comparisons between actual and perceived data were made:

- The actual design documentation was compared to the perceived design modelling techniques and the actual time spent in design.
- The actual time-in-phase data was compared to the perceived time-in-phase data and the perceived development process.
- The actual time spent on design review and code review was compared to the perceived defect removal strategies to deduce the actual strategies used to remove defects.
- The actual defect data was evaluated by comparing the phases in which defects were removed to the perceived defect removal strategy.
- The actual effectiveness of the defect removal strategies was compared to the perceived effectives indicated in the pre-questionnaire.

Lastly, the quality of actual defect descriptions (the clarity with which the cause of a defect was described) was evaluated to deduce if it would be usable for future defect prevention.

4.4 Evidence

Since the Phase 2 participants were all considered to be top-performers, my initial expectations were that the collected data would not contain a lot of variance. Surprisingly, there were six very different experiences that revealed interesting details regarding each participant's individual development process. The observed differences therefore warrant an individual discussion of each participant's data. The discussion in this section hence focuses on presenting the individual data collected from each of the six participants during Steps 1, 3, 4, 5 and 6 of the Phase 2 experimental case study. In each sub-section, the pre-questionnaire data, program and actual PSP data, and post-questionnaire data of a participant are discussed.

For each participant, the pre-questionnaire data discussion focuses on the participant's perceived development processes. The program code and PSP data sub-section describes the time taken to complete the programming exercise, the size of the final program, the functional requirements achieved, and the conceptual parts used by the participant to solve the problem. The discussion in the post-questionnaire sub-section describes the participant's reflection on his process measurement experience as well as the personal process changes he proposed after analysing his process measurement data.

The section concludes with a short description of my overall observations, the evidence gathered from the participants' combined captured process measurement data, and the data collected during the focus group discussion.

4.4.1 Participant 1

A comparison of Participant 1's perceived and actual development process data is summarised in Table 4-3. Details are discussed in the sub-sections to follow.

Table 4-3: Recorded data for Participant 1

	Perceived	**Actual**
Development process	Code-and-fix	Code-and-fix
Design modelling	Use cases	Code comments
Development time in phase %		
Planning	25%	9.68%
Design	10%	3.23%
Design review	4%	0.81%
Coding	55%	32.30%
Code review	0%	6.45%
Testing/Debugging	6%	47.60%
Defect removal strategies	Debugging	Design review Code review Debugging
Use of checklists	None	None

4.4.1.1 Pre-questionnaire

Participant 1 indicated that he typically spent most of his development time in the coding phase (55%), followed by planning (25%) and design (10%). Since he spent only 4% of his time (in design reviews) finding defects before testing, the low time indication for testing (6%) is questionable. Although he selected code-and-fix as his development process, this does not correspond with the relatively high percentage of perceived time indicated for planning and design (total of 35%). The participant, however, indicated that he utilised “use cases” for design modelling, which could explain the perceived time spent in planning and design.

His primary defect removal strategy, “debugging”, also does not match the corresponding perceived time-in-phase data of 6% for testing. The participant furthermore believed that “debugging” was the most effective way to remove defects. No use of review checklists was indicated, which corresponds with the indicated time spent on reviews. According to the participant, he did no actual recording of defects or the time it took to find and fix them. He believed that poor performance in programming assignments can be attributed to “insufficient programming skills”.

4.4.1.2 Program and actual PSP data

According to the actual PSP data, the participant's total development time to complete the assignment was 124 minutes. The total size of the submitted program was 64 lines of code (see Appendix H). All the major functional requirements were achieved with the program code. The program functionality was accomplished through three methods. A driver method was used to get input from the user and served as the main entry point in the program from which other methods were invoked. Two additional methods were created: "GenerateNumbers" for generating the required lotto rows in a two-dimensional array structure, and "StoreNumbersToDataFile" to write the lotto numbers from the array to a text file. The generated numbers were, however, not checked for duplicates and the output was not sorted.

No design documentation was created, which corresponds with the low percentage of actual time spent in design (3.23%) and with his preferred code-and-fix development process (as indicated in the pre-questionnaire) (see Table 4-3). The high actual testing time (47.6%) can be attributed to his development style. This value is also much higher than the perceived value of 6% indicated in the pre-questionnaire.

The participant recorded a total of nine defects that were all injected during the coding phase (see Table 4-4). One defect was removed during code review and the other eight during testing. The defect removal efficiency of the code review was 7.5 defects per hour and 8.14 defects per hour for the testing. This indicates that the actual review technique and debugging strategy used by the participant were equally effective and contributed to his believe that debugging is the most effective technique for removing defects. The defect descriptions were clear but, in some cases, described the consequence of the defect instead of the cause of the defect (e.g. "Not giving enough feedback").

4.4.1.3 Post-questionnaire

Participant 1 reported that time recording, identifying defect types and describing defects were "*easy*", but he had some trouble categorising and describing "*run-time*" defects. He also indicated that recording time in the correct phase was "*very easy*", even though he had trouble differentiating between the coding and the testing phase.

Table 4-4: Defect data for Participant 1

Defect type	**Phase injected**	**Phase removed**	**Time to fix (minutes)**	**Defect description**
Assignment	Code	Test	1.4	Specifying number of rows in a two-dimensional array.
Assignment	Code	Test	3.2	Object null pointer exception.
Assignment	Code	Test	4.5	Invalid rank specifier when assigning two-dimensional array.
Interface	Code	Code review	2.2	Not writing rows on separate lines.
Function	Code	Test	3.4	Not using "GetLongLength" method to get number of rows and columns.
Interface	Code	Test	2.1	Not giving enough feedback.
Interface	Code	Test	0.9	Moving cursor to the next line after feedback.
Interface	Code	Test	6.1	Invalid parameter passed to Sort method.
Function	Code	Test	2.1	Index out of bound exception when looping through numbers.

His first step in doing the assignment was to identify and understand the main functional requirements of the problem. He then documented the major requirements as code comments (also see Appendix H) instead of creating an actual design document. He believed that these design comments helped him to identify defects in his code and perceived these comments as a code review.

Based on his time data, he believed that if he spent more time on designing and reviewing the design, he would be able to find and fix defects earlier and that this would help to reduce the time spent in testing.

After he analysed his defect data, he concluded that spending more time on designs and reviews could lower the number of defects that he would find in testing. He also believed that the measurement data could help him to measure his own "*productivity*" in terms of the time spent in the different development phases. As for process improvement, he proposed to learn how to do effective designs, spend more time on reviewing and "*move away*" from code-and-fix.

4.4.2 Participant 2

A comparison of Participant 2's perceived and actual development process data is summarised in Table 4-5. Details are discussed in the sub-sections to follow.

Table 4-5: Recorded data for Participant 2

	Perceived	**Actual**
Development process	Code-and-fix Iterative	Code-and-fix
Design modelling	Flow charts Pseudo code	None
Development time in phase %		
Planning	5%	7.80%
Design	10%	2.84%
Design review	0%	0.00%
Coding	60%	70.20%
Code review	10%	0.00%
Testing/Debugging	15%	19.10%
Defect removal strategies	Debugging	Debugging
Use of checklists	None	None

4.4.2.1 Pre-questionnaire

Participant 2 indicated that he typically spent 60% of his development time in the coding phase, which is relatively high in comparison to his perceived time spent in planning and design (15%). He indicated his perceived time for code review at 10% and for testing at 15%, which provide a fairly realistic total of 25% to find and fix defects. He selected code-and-fix and iterative as his perceived development processes, which correspond with the low perceived design time of 15%. The participant, however, indicated that he utilised "flowcharts and pseudo code" for design modelling, which would require him to spend at least as much time in design as in coding. He also indicated that he typically spent 0% of his time on design reviews, which is likely to lead to design defects that will only be discovered late in the development life-cycle. Since his perceived design time is relatively low, it might also indicate that the designs are not on a reviewable quality level, which could explain the 0% of perceived time spent on design reviews.

He selected “debugging” as his only defect removal strategy, which could explain why he typically spent more time on debugging (15%) than on reviews (10%). Even though the participant believed that “debugging” was the most effective way to remove defects, he still indicated that he typically spent 10% of his time on code reviews without the use of any checklists.

According to the participant, he did not usually do any actual recording of defects or the time it took to find and fix them. He believed that poor performance in programming assignments can be attributed to “insufficient time”.

4.4.2.2 Program and actual PSP data

The participant’s total development time to complete the assignment was 124 minutes. The total size of the submitted program was 76 lines of code (see Appendix I). All functional requirements were 100% achieved with the program code. The program functionality was accomplished through four methods. A driver method was used to get input from the user and to generate the lotto row numbers. Three additional methods were implemented and called from the driver program: “CheckDuplicates” to search for duplicates before storing a generated number in a one-dimensional array structure; “Sort” to sort the generated lotto row; and “LottoNumbers” to write the lotto numbers from the array to a text file.

No design documentation was created, which corresponds with the low percentage of actual time spent in design (2.84%) as well as his perceived use of code-and-fix and iterative development processes (as indicated in the pre-questionnaire). The participant spent 0% of his actual development time on code reviews and design reviews, which indicates that no reviews were done. This data does not correspond with his original perception that he typically spent 10% of his time in code reviews (see Table 4-5). The actual coding time (70.2%) was higher than the anticipated 60% indicated in the pre-questionnaire. The actual testing time of 19.1% was close to the perceived value of 15% indicated in the pre-questionnaire.

The participant recorded a total of four defects, all injected during the coding phase and all removed during testing (see Table 4-6). The defect removal efficiency of the testing was 8.89 defects per hour. The defect descriptions clearly explained the cause of the defects and could therefore be useful in the construction of personalised checklists.

Table 4-6: Defect data for Participant 2

Defect type	**Phase injected**	**Phase removed**	**Time to fix (minutes)**	**Defect description**
Function	Code	Test	4.2	Logical error in loop condition
Assignment	Code	Test	5.1	Accessing an empty array
Function	Code	Test	4.4	Infinite loop
Interface	Code	Test	11.0	Did not use file append

4.4.2.3 Post-questionnaire

The participant struggled to "*record time*" and mentioned that he specifically found it "*very difficult to record time in the correct phase*". He also mentioned that he was unable to "*log*" data in the correct phase because he struggled to comprehend "*what must be done*" in the different phases. He also reported that "*describing a defect*" was difficult even though he found it easy to identify the type of defect. He could not describe some of the defects and therefore did not record them, even though he fixed them. The participant acknowledged that he did not do any code or design reviews.

Based on a review of his time data, he believed that "*doing each phase step-by-step*" and finding defects earlier would reduce his testing time. After analysing his defect data, he concluded that "*logging all defects*" and "*doing design reviews and code reviews*" could also enable him to remove defects earlier. He believed that the measurement data could help him to "*rate*" himself on his ability to detect defects and could also serve as "*encouragement*" to improve. As for personal process improvement, he proposed spending more time on designs, design reviews and code reviews. He also acknowledged that he did not have a clear understanding of "*what must be done*" during the various software development phases.

4.4.3 Participant 3

A comparison of Participant 3's perceived and actual development process data is summarised in Table 4-7. Details are discussed in the sub-sections to follow.

Table 4-7: Recorded data for Participant 3

	Perceived	**Actual**
Development process	Code-and-fix	Code-and-fix
Design modelling	Flow charts	None
Development time in phase %		
Planning	20%	17.00%
Design	10%	0.00%
Design review	5%	0.00%
Coding	40%	33.30%
Code review	15%	0.00%
Testing/Debugging	10%	49.67%
Defect removal strategies	Code review Debugging	Debugging
Use of checklists	Code review	None

4.4.3.1 Pre-questionnaire

Participant 3 indicated that his typical development time was distributed as follows: 20% for planning, 10% for design, 5% for design review, 40% for coding, 15% for code review and 10% for testing (see Table 4-7). The much higher perceived planning time in comparison to the perceived design time raised questions regarding what exactly he did for planning and design respectively. Both perceived review times (design review and code review) correspond well with the perceived time spent on design and coding. The low value for perceived testing time (10%) could be an indication that the participant typically detected most defects during the design review and code review phases.

Although he selected code-and-fix as his development process, this does not correspond with the high percentage of perceived time indicated for planning and design (30%). The participant, however, indicated that he utilised "flowcharts" for design modelling, which could explain the time spent in design.

His perceived defect removal strategies include debugging and code review, but not design review for which he allocated some time (5%). He indicated that he typically used code review checklists compiled from both existing checklists and his own defect data. He also indicated that he kept record of his defects, which corresponds with the way in which he compiled his checklists. The participant also believed that code reviews was the most effective way to remove defects, which corresponds with the amount of perceived time that he allocated for code reviews. He believed that poor performance in programming assignments can be attributed to “the inability to identify all defects”.

4.4.3.2 Program and actual PSP data

The participant’s total development time to complete the assignment was 147 minutes. The total size of the submitted program was 63 lines of code (see Appendix J). The project consisted of two driver methods ("Main" and "Main2") – each contained in a separate file. Only the code in the "Main" method executed automatically at runtime. Not all functional requirements were completely achieved with the program code. Participant 3 did not implement the parts to get input from the user and write the numbers to a text file. The "Main" method implementation generated six non-duplicate numbers and stored it in an array list. The code also sorted the numbers and displayed it on the screen.

No design documentation was created, which corresponds with the zero time spent in design and with his preferred code-and-fix development process model (as indicated in the pre-questionnaire). His actual testing time of 41.5% was much higher than the anticipated 10% as indicated in the pre-questionnaire (see Table 4-7). He spent zero time in design and design review, which indicates that no designs were done and that design defects likely slipped through to testing. This could explain the high amount of time he actually spent in testing. The participant also logged time in compile, which indicates that some defects were fixed during a compile phase. In a .NET programming environment such as Microsoft Visual Studio, compile defects should actually have been recorded under testing, as suggested by Humphrey (2005). Test defects and compile defects are defects picked up late in the life cycle, which essentially totalled his actual compile and test defect removal time to 49.67%. This was much higher than the anticipated 10% indicated in the pre-questionnaire.

The participant recorded a total of five defects that were all injected during the coding phase (see Table 4-8). Four of the defects were removed during testing and one defect during the compile phase. The defect removal efficiency of the testing was 3.93 defects per hour. His defect descriptions were vague and focused on the consequences of the defects and not on the causes. Descriptions like these are unlikely to benefit the participant in improving his defect management.

Table 4-8: Defect data for Participant 3

Defect type	Phase injected	Phase removed	Time to fix (minutes)	Defect description
Syntax	Code	Test	3.8	Syntax error
Interface	Code	Test	6.2	Output format wrong
Checking	Code	Compile	9.3	Output format wrong
Checking	Code	Test	4.4	Output format wrong
Checking	Code	Test	25.2	Output format wrong

4.4.3.3 Post-questionnaire

The participant reported that "*recoding time*" was "*easy*", but to "*record time in the correct phase*" was "*difficult*". He struggled to differentiate between coding and testing, which is a common effect of using a code-and-fix process. The participant also mentioned that he found it difficult to "*identify the type of defect*" and to "*describe a defect*". This acknowledgement, together with his very vague defect descriptions (see Table 4-8), made it highly unlikely that he typically recorded defects to create his own checklists for future use (as indicated in his pre-questionnaire). The participant also struggled to record the time to find and fix the recorded defects. He acknowledged that he did not do any code review or design review.

Based on his actual time data, he believed that better planning for more "*efficient code*" and reviewing his design would reduce his testing time.

After he analysed his defect data, he concluded that code reviews and "*compiling more often*" would enable him to remove defects earlier. He also mentioned that the measurement data could help him to "*manage time spent on defects*" and to do better planning. As for process improvement, he proposed doing better planning before implementing the code.

4.4.4 Participant 4

A comparison of Participant 4's perceived and actual development process data is summarised in Table 4-9. Details are discussed in the sub-sections to follow.

Table 4-9: Recorded data for Participant 4

	Perceived	**Actual**
Development process	Code-and-fix Prototyping	Code-and-fix
Design modelling	Use cases Flow charts	None
Development time in phase %		
Planning	25%	14.5%
Design	15%	0.0%
Design review	5%	0.0%
Coding	40%	46.5%
Code review	0%	0.0%
Testing/Debugging	15%	39.0%
Defect removal strategies	Debugging	Debugging
Use of checklists	None	None

4.4.4.1 Pre-questionnaire

For Participant 4, the indicated perceived time spent on design (15%) was questionable because his perceived development processes are code-and-fix and prototyping (see Table 4-9). The low amount of perceived time spent on design review (5%) could be an indication that the design might not be of a reviewable quality. The indicated use of use cases and flowcharts for design modelling could explain the 15% of perceived time spent in design. His indication of 0% perceived time for code review could be an indication that he typically solved most of his syntax defects during testing. Most of his defects would therefore be resolved late in the life cycle which could in turn, lead to much higher testing time than the perceived 15%.

He indicated “debugging” as his only defect removal strategy, which raises questions regarding the perceived time portion allocated for design review. The participant indicated that he kept track of common defects as well as the time to find and fix them. He also indicated that he typically made use of checklists based on his previous errors and existing checklists. What was questionable, however, was that he did not use these checklists for design reviews and code reviews. The participant believed that “debugging” was the most effective way to remove defects, which corresponded with his choice of defect removal strategies. He believed that poor performance in programming assignments could be attributed to “insufficient time”.

4.4.4.2 Program and actual PSP data

Participant 4’s total development time to complete the assignment was 114 minutes. The total size of the submitted program was 32 lines of code (see Appendix K). Not all functional requirements were completely achieved with the program code. The program functionality was accomplished through one driver method. No code was attempted to get input from the user for the generation of multiple rows of lotto numbers. Instead, an outer loop structure was used to generate ten rows of lotto numbers. For each row, six non-duplicate numbers were created and stored in a one-dimensional array. Neither sorting nor output to a text file was attempted with the code. The participant did, however, attempt to display the content of the array, but the code had some syntax defects that could not be resolved.

No design documentation was created, which corresponds with the no actual time spent in design as well as his preferred code-and-fix and prototyping development strategies (as indicated in the pre-questionnaire) (see Table 4-9). His actual testing time (39%) was much higher than anticipated in the pre-questionnaire (15%), but corresponds with no actual design review and code review times. The participant’s actual testing time was most probably much higher than perceived because defects were discovered late in the development life cycle and he did not employ techniques for early defect removal.

The participant recorded four defects injected during coding that were all removed during testing (see Table 4-10). The defect removal efficiency of the testing was 4.29

defects per hour. His defect descriptions were vague and also indicated that he did not know what caused the defects.

Table 4-10: Defect data for Participant 4

Defect type	Phase injected	Phase removed	Time to fix (minutes)	Defect description
Assignment	Code	Test	3.4	Unknown
Syntax	Code	Test	3.0	Unknown
Function	Code	Test	17.0	Could not create a list
Function	Code	Test	28.9	Unknown

4.4.4.3 Post-questionnaire

The participant reported that "*recoding time*", "*recording time in the correct phase*" and "*identifying defect types*" were easy. However, he found it difficult to describe defects and considered it as "*something to get use to*". He struggled to differentiate between coding and testing, which is a common effect of using a code-and-fix process. The participant acknowledged that he should have logged more time under testing instead of coding. He also acknowledged that he did not do any code review or design review, which corresponds with the actual time spent in these phases.

Based on his time data, he believed that if he "*writes all code and then test*" instead of small iterative cycles of code-and-test, he would be able to reduce his testing time. After analysing his defect data, he proposed that he should not spend too much time on one error, which contributed to his argument of larger code-and-test cycles. He believed that the measurement data could help him to do better time management and "*improv[e] on ways to detect defects*". He also mentioned that the data made him realise that he had problems with software development. As for process improvement, he surprisingly proposed "*no changes*" to his current process.

4.4.5 Participant 5

A comparison of Participant 5's perceived and actual development process data is summarised in Table 4-11. Details are discussed in the sub-sections to follow.

Table 4-11: Recorded data for Participant 5

	Perceived	**Actual**
Development process	Code-and-Fix	Code-and-fix
Design modelling	Data flow diagrams Flow charts	None
Development time in phase %		
Planning	10%	10.00%
Design	15%	1.88%
Design review	15%	0.00%
Coding	40%	84.40%
Code review	10%	0.00%
Testing/Debugging	10%	3.75%
Defect removal strategies	Design review Code review Debugging	Debugging
Use of checklists	Code review	None

4.4.5.1 Pre-questionnaire

Participant 5 indicated that his typical development time was distributed as follows: 10% for planning, 15% for design, 15% for design review, 40% for coding, 10% for code review and 10% for testing (see Table 4-11). The perceived time spent on design was questionable since he indicated his typical development process as code-and-fix. The perceived time allocated for code review might be a bit low compared to the amount of perceived time spent in coding. The low amount of perceived time indicated for testing could be an indication that he typically found and fixed most defects in design review and code review. Although he selected code-and-fix as his development process, the participant indicated that he utilised flowcharts and data flow diagrams for design modelling, which could explain the perceived time spent in design.

His defect removal strategies included design review, code review and debugging, which correspond with the time allocated for these phases. He indicated that he typically made use of code review checklists compiled from his own defect data. He also kept record of his defects, which corresponds with the way in which he compiled his code review checklists. The participant also indicated that he kept track of the amount of time that he typically spent on fixing defects. There was no indication of the use of checklists for design reviews, which raised questions regarding his perceived time spent on and the technique used for design reviews. The participant also believed that code reviews was the most effective way to remove defects and that poor performance in programming assignments could be attributed to the inability to identify all defects.

4.4.5.2 Program and actual PSP data

Participant 5's total development time to complete the assignment was 140 minutes. The total size of the submitted program was 28 lines of code (see Appendix L). The code contained some defects that could not be resolved. The participant implemented one method called "LottoTest" that was invoked from a "Page_Load" method. The code indicated no attempt to get input from the user. A "Dictionary" data structure was created for storing the lotto numbers and a "Random" object for generating each number. An iteration structure that repeated six times was created, but the code to generate and store each number contained defects. No attempt was made to check for duplicate numbers. The code also contained a string array that stored the values from the dictionary structure as strings using LINQ. A second loop was created that iterated through a string array to display the values of the array. Despite the defects present in the code, it was apparent that the participant's intention was to create code that generated six numbers and displayed these six numbers. This participant most probably did not have the technical knowledge to use a dictionary data structure to accomplish the desired functionalities. It was not clear why the participant decided to use a data structure that was beyond his technical capability.

No design documentation was created, which corresponds with the low percentage of actual time spent in design (1.88%) and with his preferred code-and-fix development process (as indicated in the pre-questionnaire) (see Table 4-11). Only the actual planning time data corresponds with the perceived time-in-phase data indicated in the pre-questionnaire. His actual design time (1.88%) indicated that almost no designs were done and therefore no design review could occur. There was also no actual time spent in code reviews. Given that no actual time was spent in design reviews and code reviews, the low amount of actual time in testing was questionable. However, there was a substantially large proportion of time logged in coding (84.4%), which led to the assumption that the participant either did not capture time data correctly, or that he ‘got stuck’ in the coding phase and could not produce a workable program.

The participant recorded one defect that was injected during the design phase and he attempted to remove it in the testing phase (see Table 4-12). The defect removal efficiency of his testing was 10 defects per hour. Since the participant only managed to produce 28 lines of code during the actual time spent in coding, this could be a further indication of his low technical skill level as he was unable to produce a fully functional program. The single recorded defect description was confirmation that the participant did not know what caused the defect and therefore could not resolve it.

Table 4-12: Defect data for Participant 5

Defect type	**Phase injected**	**Phase removed**	**Time to fix (minutes)**	**Defect description**
Interface	Design	Test	2.4	Unknown - still fixing

4.4.5.3 Post-questionnaire

The participant reported that recording time, recording time in the correct phase, identifying defect types and describing defects were just “*new*”, and not difficult. He emphasised that his focus was not on taking process measurements, but rather on coding. He also stated that recording defect data was not difficult, but “*something to get used to*”. He struggled to find and fix defects during his code review and acknowledged that he did not log that time as code review but as coding because he forgot to log time in a different phase. Based on his time data, he believed “*keeping track of code and design time*” would reduce his testing time.

After he analysed his defect data, he concluded that he “*did bad*” and would “*try to do much better*” next time. He also indicated that he removed most of his defects during the coding phase (which does not correspond with his defect data). According to the participant, measurement data could help him to keep track of the time in different phases. He also felt that knowledge of this data could enable him to divide time better and to “*attend*” to all phases of the development life-cycle. As for process improvement, he proposed collecting more accurate process data, especially regarding defects.

4.4.6 Participant 6

A comparison of Participant 6's perceived and actual development process data is summarised in Table 4-13. Details are discussed in the sub-sections to follow.

Table 4-13: Recorded data for Participant 6

	Perceived	**Actual**
Development process	Code-and-fix Prototyping	Code-and-fix
Design modelling	Data flow diagrams Class diagrams Flow charts	None
Development time in phase %		
Planning	20%	23.00%
Design	20%	0.00%
Design review	5%	0.00%
Coding	20%	7.08%
Code review	25%	0.00%
Testing/Debugging	10%	69.90%
Defect removal strategies	Design review Code review Debugging	Debugging
Use of checklists	Code review	None

4.4.6.1 Pre-questionnaire

For Participant 6, the perceived time spent on design (20%) was questionable because he indicated that his typical development process was a combination of code-and-fix and prototyping (see Table 4-13). The perceived time spent on design review (5%) also indicated that the design might not be of a reviewable quality. The indicated use of flowcharts, data flow diagrams and class diagrams for design modelling could explain the large amount of perceived time spent in planning and design. The perceived time allocation for code review (25%) compared to testing (10%) could serve as an indication that the participant typically removed most of his defects during code review. The low amount of perceived time indicated for testing (10%) could be an indication that he followed an effective early defect removal strategy. The greater portion of perceived time in code reviews could be an indication that he preferred code reviews over design reviews or testing.

His perceived defect removal strategies included design review, code review and debugging, which correspond with the perceived time allocated for these phases. Participant 6 indicated that he typically made use of code review checklists compiled from his own defect data and from existing checklists. This does not correspond with the indication that he typically did not keep track of any defects or time spent on finding and fixing these defects. The participant believed that debugging is the most effective way to remove defects even though there was no indication that he typically kept track of the time it took to remove defects. He believed that poor performance in programming assignments could be attributed to insufficient programming skills.

4.4.6.2 Program and actual PSP data

The participant's total development time to complete the assignment was 113 minutes. The total size of the submitted program was 12 lines of code (see Appendix M).

No design documentation was created, which corresponds with the zero percentage of actual time spent in design and with his preferred code-and-fix and prototyping development strategies (as indicated in the pre-questionnaire). Only the actual planning time data corresponds with the intended time-in-phase data of the pre-questionnaire (see Table 4-13). The actual high planning time (23%) could serve as an indication that prototyping was used. The actual design time indicated that no designs were done and therefore no design review could occur. The low amount of actual time spent in coding (7.08%) combined with the small program size indicated that a small portion of code was created (presumably the “prototype”) and then tested.

The participant recorded one defect injected during the coding phase that he attempted to remove during testing (see Table 4-14). The defect removal efficiency of the testing was 0.76 defects per hour. The large amount of actual testing time (69.90%) and the single defect that was logged (but not resolved) could be an indication that the participant ‘got stuck’ on one defect, rendering him unable to continue. The defect description served as confirmation that the participant did not know what caused the defect and therefore could not resolve it.

Table 4-14: Defect data for Participant 6

Defect type	Phase injected	Phase removed	Time to fix (minutes)	Defect description
Function	Code	Test	63.9	Unknown - output is totally wrong

The program code did not reveal any answers regarding his development process as it only consisted of the 12 lines of code that were generated by the compiler for a default project. No code was added to reflect an attempt to solve the problem. During his interview, Participant 6 claimed that he "*explored*" with some code, but could not get a single line to compile. He therefore deleted everything before submitting his program.

4.4.6.3 Post-questionnaire

The participant reported that recording time was "very difficult", but to record it in the correct phase was "difficult". He also reported that identifying defect types was "very difficult", while it was "difficult" to describe defects. His reason for struggling with time recording and defect recording was that it was difficult for him "*to use something new*". He acknowledged that he did not do any reviews.

Based on his time data, the participant believed that improving his technical skills would reduce his testing time. He stated that his biggest problem was syntax and logical defects. He believed that doing design reviews and code reviews could help to reduce his number of test defects. According to the participant, measurement data could also help him to realise in which phase he should spend more time. As for process improvement, he proposed improving his basic programming skills.

4.4.7 Instructor observations

I made the following main observations while the participants were completing the programming exercise:

- Participants searched the Internet in an attempt to find solutions for the exercise.
- No designs were created to solve the exercise problem.

- Some participants forgot to start and stop the Process Dashboard© timer when switching phases.
- Some defects were not logged.
- Participants struggled to distinguish between the “coding” and the “testing” phase.
- Participants struggled to describe their logged defects.

Given the participants’ inability to distinguish between the coding and the testing phase, they did not log their re-work coding in the correct phase. Most of them logged that time under coding, which explained why re-work (testing) time was lower than coding time. More precise measurements would therefore have resulted in much higher testing times.

4.4.8 Overall Process Dashboard© performance analysis

The six participants took 135 minutes on average to create the program. This time frame included all phases of development: planning, design, design review, coding, code review and testing. I decided to end the programming exercise after 150 minutes as enough useful experimental data was accumulated. At that time, the participants also indicated that they would not be able to identify and fix all remaining defects even without a time limit.

The participants on average spent their actual development time as follows:

- 17% on planning,
- 1% on design,
- 0% on design reviews,
- 45% on coding,
- 1% on code reviews, and
- 36% on testing or debugging.

If averages are compared, the actual time that these participants spent on design was much lower than the perceived times reported in the pre-questionnaire. Most of them also spent much more time in testing than expected. The actual testing time would, however, be much higher if they had to continue to produce fully functional programs. The participants on average produced 45 lines of code, which resulted in a productivity of 20 lines of code per hour. They recorded an average of five defects with 90% of these defects injected during coding. The limited actual time spent on designs explains why most defects were injected during coding. Ninety-five percent of the defects were removed in the testing phase – an indicator that debugging was used as the primary technique for defect removal. Given the average time spent on reviews (1%), it is not surprising that so few defects (2%) were discovered during reviews. Since the participants (on average) only spent 1% of their actual time in the design phase, the complete absence of design reviews is understandable. This resulted in defects being discovered late in the development life cycle (testing), which made it more difficult to identify them.

4.4.9 Focus group discussion

A focus group discussion was conducted with Participants 4, 5 and 6 to gain a deeper understanding of the development processes they followed to create the program. The only artefacts that these participants created (in addition to the captured Process Dashboard© measurement data) were their actual programming code. Since they did not create any designs, I decided to rather redirect the discussion to focus on their problem solving processes.

All three participants indicated that their first step in solving the problem was to do an Internet search for possible solutions. They all found code that they thought could possibly solve the problem. They copied the code and then tried to change it to solve the problem. They also indicated that this was the method they typically followed when completing their programming assignments. Feiner and Krajnc (2009) made a similar observation in their experiment where most students indicated that their first step in solving a programming problem was to search "the Internet" or "use Google". In a questionnaire conducted at the end of their experiment, the students revealed their general acceptance of "Copy & Paste" programming (Feiner & Krajnc, 2009, p. 84) as part of their software development process.

In reflecting on their problem-solving process, the three participants from this study indicated that they should rather have started by first solving the problem logically (using flowcharts or pseudo code) and only then have resorted to searching for code snippets to accomplish specific tasks. They also indicated that they did not find it easy to write pseudo code to solve problems and therefore preferred to search for code solutions where the logical thinking has already been done. Generally, they found it "*hard to start*" solving a problem.

4.5 Discussion of Evidence

In the previous section, the Phase 2 evidence was evaluated according to the quality guidelines of the PSP framework. A number of issues were also raised regarding the differences between the perceived and actual development processes that the participants followed. In order to form a better understanding of these novice programmers' development processes and their use of QATs, a thorough exploration of the evidence is needed. The discussion in this section is structured around the two main themes of Phase 2: differences between perceived and actual development processes (including the use of QATs), and attributes influencing the use of QATs.

4.5.1 Differences between perceived and actual software development processes

Evidence has been provided of distinct differences between the participants' perceived and actual software development processes. Major differences were noted with regard to their design practices as well as the time spent in the testing/debugging and coding phases.

Table 4-15 summarises and compares the perceived and actual design practices followed by the participants during the Phase 2 experimental case study.

Table 4-15: Comparison of participants' perceived and actual design practices

Participant	% Time spent on designs		Design modelling techniques	
	Perceived	Actual	Perceived	Actual
1	10%	3.23%	Use cases	Code comments
2	10%	2.84%	Flowcharts Pseudo code	None
3	10%	0	Flowcharts	None
4	15%	0	Use cases Flowcharts	None
5	15%	1.88%	DFD Flowcharts	None
6	20%	0	DFD Flowcharts Class diagrams	None

All the participants spent much less time in design than originally perceived. Despite the substantial amount of perceived design time (ranging from 10% to 20%), most of the participants ended up spending almost no time on designs. Similarly to the students in Hou and Tomayko's (1998) study, some of the participants in this study created no design documentation even though they captured time in the design phase. The participants' small amount of actual design time also corresponded with the design modelling techniques that they actually used (none) and the total lack of formal design documentation. A similar lack of designs was also noted in other design studies (Eckerdal et al., 2006a; Lotus et al., 2011). In Eckerdal et al.'s (2006a) study, 80% of the students created either no designs or made no significant progress towards design. The only exception in this study was Participant 1, who actually used some form of design modelling. This participant's design can be classified in the "informal design" category of Thomas et al. (2014) as it was only text based.

Table 4-16 summarises and compares the participants' perceived and actual time spent in testing/debugging and coding during the experiment. Comparisons of the participants' perceived and actual time-in-phase data are complicated by the fact that students generally struggle to distinguish between the development phases (Grove, 1998) and consequently find it difficult to capture accurate and reliable process data (Towhidnejad & Salimi, 1996). With the exception of Participant 5, most participants in this study completely underestimated the time that they normally spent resolving defects during testing. Although the actual coding time for Participant 2 and Participant 4 was higher than perceived, they both indicated that they should have captured even more time in testing because a lot of their coding occurred as a result of re-work to resolve defects (which should have been logged as testing time). Carrington et al. (2001) emphasise that when students write a program by incrementally compiling and testing one line of code at a time, they are unable to log data in the correct prescribed PSP phases. The higher than expected testing times could therefore also be an indication that participants ended up doing more "fixing than coding" instead of the code-and-fix strategy that most of them believed that they typically used.

Table 4-16: Comparison of participants' perceived and actual testing/debugging and coding time

Participant	% Time spent on testing		% Time spent on coding	
	Perceived	Actual	Perceived	Actual
1	6%	47.60%	55%	32.30%
2	15%	19.10%	60%	70.20%
3	10%	49.67%	40%	33.30%
4	15%	39.00%	40%	46.50%
5	10%	3.75%	40%	84.40%
6	10%	69.90%	20%	7.08%

Early defect removal occurs when defects are removed before the testing/debugging phase (Humphrey, 2005). This is accomplished through the use of design reviews and code reviews (as examples of QATs). Table 4-17 summarises and compares the participants' perceived and actual defect removal strategies, as well as their perceived and actual time spent on these strategies.

With the exception of Participant 1, all the participants ended up using debugging as their only defect removal strategy during the programming exercise. According to Humphrey (1999), one of the biggest challenges in software development is to persuade software developers to use effective methods. Although Participant 1 indicated that he typically only used debugging, he did attempt to use design reviews and code reviews during the programming exercise. The low amount of time that he ended up spending on design review and code review was, however, not indicative of someone who really depended on these strategies to remove defects.

Table 4-17: Comparison of participants’ perceived and actual defect removal strategies

Participant	Defect removal strategy		% Time spent on design review		% Time spent on code review		% Time spent on testing/debugging	
	Perceived	Actual	Perceived	Actual	Perceived	Actual	Perceived	Actual
1	Debugging	Design review Code review Debugging	4%	0.81%	0%	6.45%	6%	47.60%
2	Debugging	Debugging	0%	0%	10%	0%	15%	19.10%
3	Code review Debugging	Debugging	5%	0%	15%	0%	10%	49.67%
4	Debugging	Debugging	5%	0%	0%	0%	15%	39.00%
5	Design review Code review Debugging	Debugging	15%	0%	10%	0%	10%	3.75%
6	Design review Code review Debugging	Debugging	5%	0%	25%	0%	10%	69.90%

Participant 2 and Participant 4 indicated debugging as their only perceived defect removal strategy, which correspond with their use thereof during the exercise. Participants 3, 5, and 6 indicated that they typically used other strategies (design review and code review) as well, but ended up using only debugging during the exercise. Of interest is the fairly huge amount of perceived time these three participants believed they were using for design reviews and code reviews. This might directly influence the perceived time that they thought they spent on testing/debugging because they believed that they would pick up defects earlier in the life cycle (with design reviews and code reviews). The overall low perceived design review times might be an indication of the participants’ inability to create reviewable designs (Humphrey, 2000). However, it is still questionable that some participants indicated that they used design reviews and even assigned a substantial amount of time to it while they ended up not doing any designs at all. Since even small programming exercises requires some code to be written, there will always be at least some code to review. Students in Towhidnejad and Salimi's (1996) study found it easier to adopt code reviews as part of their quality improvement process because they regarded it as more closely related to programming. However, software developers tend to stick to a personal process that they have developed from the first small program they have written, and it is difficult to convince them to adopt better practices (Humphrey, 1999).

4.5.2 Attributes influencing the use of QATs

The discussion in this section takes a reflective look at all the collected evidence using PSP0 and PSP2 as lenses in an attempt to identify specific problems experienced by the participants in following these guidelines/practices. Since the adoption of QATs would likely require a change in behaviour from the participants, the self-theory of intelligence is included as a third, supporting lens for this discussion. Ultimately, these identified problems are related to attributes that could potentially influence novice programmers' use of QATs.

4.5.2.1 PSP0: Software development process and basic measurements

In using PSP0 as the first lens for this reflection, four potentially influencing attributes were identified.

Understanding of development phases

One of the PSP quality improvement practices states that: "To do high-quality work, you must measure and manage the quality of your development process" (Humphrey, 2005, p. 157). Given the negative impact that defects have on the quality of the development process, it is not surprising that "most software professionals agree that it is a good idea to remove defects early, and they are even willing to try doing it" (Humphrey, 2005, p. 142). In this study, the participants tried to do the same, but they all opted for a code-and-fix development process which is not the best strategy to follow for early defect removal. They therefore spent most of their actual development time in the planning, coding and testing phases. The code-and-fix model does not make provision for any phases that can be linked to quality appraisal practices such as design, design review and code review (Schach, 2011). Some participants indicated that they did not know exactly what to do in these development phases (design, design review, code review), but acknowledged that the process measurement data made them aware that something needs to be done in these phases. Participant 4 indicated that he mostly “confused coding and testing”, while Participant 2 attributed his struggles to the fact that he “could not differentiate on what must be done on each phase”. As for process improvement, Participant 2 proposed “try[ing] to understand what must be done in each phase” and “spend[ing] more time on design, design review and code review”. In this regard, Participant 1 proposed that for process improvement he should “learn how to do design effectively” and “spend more time reviewing so I don’t do more of code and fix”. Participant 5 indicated that the recording of time data in specified phases forced him to “attend all sections as far as possible”. This is an indication that Participant 5 did not know exactly what to do in each of these phases.

Technical programming skills

One of the code review principles of PSP is that one must, first of all, produce a reviewable product (Humphrey, 2005). Only Participant 2 created a fully functional program for the programming exercise while Participant 1 successfully implemented most of the major functionalities. Participant 6 indicated that his biggest problem was his lack of technical programming skills and that this was the main reason why he produced only a small amount of workable code. As for process improvement, he suggested “*recapping on OPG1 stuff [the basics of coding] because my problem was mainly syntax and logical errors*”. Participant 5 also produced very few lines of workable code and used data structures beyond his technical capability. In his post-questionnaire, however, he did not mention any shortcomings in his technical ability to produce code. Participant 4 produced slightly more workable code than Participants 5 and 6, but in his post-questionnaire stated that process measurement data “*is good for making one see his/her problems in software development”*.

Accuracy of measurement data

Time measurement data is typically used to “analyse your process, to understand strengths and weaknesses, and to improve” (Humphrey, 2005, p. 15). Therefore, the accuracy of time data will directly influence process improvement decisions. All the participants indicated that they had some difficulty capturing accurate time data in the correct phase:

Participant 1: "*Forgot to switch the timer between phases when busy testing and had to go back to coding*".

Participant 2: "*I could not log data in the correct phase and sometimes forgot to log data*".

Participant 3: "*I had a problem recording time while I was in testing*".

Participant 4: "*I record[ed] time in the wrong phase, and it was a bit confusing when it came to the part where I had to test because I was working more on the coding*".

"*I confused coding and testing for most of my recording*".

Participants 5 and 6 did not indicate any specific problems with capturing accurate time data, but acknowledged that capturing process measurement data is "new" to them. The most common problem that the participants experienced was to distinguish between "re-work" (as result of fixing a defect) and normal work. The re-work time was supposed to be logged under the phase in which the defect was discovered. If this is not done correctly, it will be impossible to accurately compute the efficiency of the defect removal strategy. This confusion could be attributed to the nature of the code-and-fix process model whereby most coding occurs because of the fixing of defects. Towhidnejad and Salimi (1996) also reported that only half of their students collected accurate and reliable data.

Ability to find and fix defects

From the participants' defect descriptions, it was evident that some of them could not resolve all the defects regardless of their defect removal strategy. Since these participants mostly relied on testing to resolve defects, it could point to a lack of debugging skills. Humphrey (2005) explained that in reviews you "find defects directly", and in testing you "only get the symptoms". Only Participants 1 and 2 managed to create 100% working programs. Their defect descriptions were much better than the rest of the participants as they described the cause of the defects and not the consequence thereof. Participant 3 had vague generic descriptions (e.g. "*Syntax Error*" and "*Output Format Wrong*") that at best indicated the kind of defect that occurred, but not the cause of the defect. Participant 4 had one vague description

("*Could not create a list*") that described the consequence of the defect and not the cause thereof. The remainder of his identified defects were described as "*Unknown*" and it was therefore not clear if he managed to resolve these defects. Participants 5 and 6 each had one defect description. In both cases the defect description reflected unresolved defects:

Participant 5: "*Unknown - Still fixing*"

Participant 6: "*Unknown - output is totally wrong*"

In PSP, defect descriptions are used to create personalised checklist items (Humphrey, 2005). These descriptions therefore need to be clear and precise. Most participants in this study indicated some difficulty logging all defects and also struggled to describe the defects:

Participant 1: "*Difficult to come up with the right description for run-time errors*".

Participant 2: "*I fixed some defects without recording them*".

"*I could not understand which defects to record*".

Participant 3: "*I struggled identifying type of defects and to describe them*".

Participant 5: "*During my code review, most of the time I was fixing errors and did not remember to log the defects in Process Dashboard*".

Based on the defect descriptions of each participant, the only descriptions that could be useful for defect management were created by Participants 1 and 2. Poor descriptions of defects are likely to lead to difficulties when this data must to be used to create personal checklist items. When not all defects are logged, it can lead to misinterpretation of the severity of defects and the causes of the lost time in the phase during which the defect was removed.

4.5.2.2 PSP2: Quality Management and Design

In using PSP2 as the second lens for this reflection, three potentially influencing attributes were identified: design skills, design review and code review skills, and value of process measurement data.

Design skills

The PSP quality-management strategy recommends that developers' first focus should be on producing "a thorough and complete design and then document[ing] the design with the four PSP design templates" (Humphrey, 2005, p. 155). Even though Humphrey claimed that the "reviewability" of a design is not that important if you review your own designs, he also stated that "without a well-documented and complete design, it is impossible to do a competent design review" (Humphrey, 2005, p. 185). All the participants in this study indicated some formal design modelling techniques that they typically used, but none of them attempted (during the Phase 2 activity) to use any of these techniques to create formal design documentation. Some participants also indicated that they did not know how to create effective designs. In explaining his strategies for process improvement, Participant 1 said that he would have to "*learn how to design effectively*" and "*design the requirements, review to identify defects so that I have less design defects when I code*".

Design review and code review skills

Participants 1 and 2 indicated that they typically did not do reviews and therefore did not use any checklists. Participant 4 also indicated that he did not do reviews but created checklists based on his previous defects and from existing checklists. Participant 3 indicated that he typically used checklists for code reviews compiled from his previous defects and from existing checklists, but he ended up not doing any reviews. Participants 5 and 6 also indicated that they typically made use of checklists for code reviews, compiled from their own defects, but ended up not doing any reviews.

However, only Participant 1 ended up doing some form of reviews. He described his technique as follows:

> "*In my design review I actually ensured that my requirements were well design[ed], even though I just passed through it*".
>
> "*For my code review I actually commented what was required by the requirements*".
>
> "[I] spend little time on reviewing by just scanning through [the] code and initial design".

Value of process measurement data

The main purpose of requesting the participants to gather process measurement data on their own development processes was to provide them with an opportunity to reflect on this data and to propose process improvements or changes based on the collected data. After the participants analysed their own process measurement data, they were therefore probed to propose process changes on how to reduce their testing time and methods to remove defects earlier in the life cycle. Specific attention was given to proposals that would influence early defect removal and reduce the time spent in testing. In reviewing these proposals, I was specifically looking for indications that a participant's reflection on his time and defect data showed some signs that could be interpreted as encouragement to use QATs (design reviews and code reviews). The participants displayed varying interpretations of their process measurement data and the potential value of the data. Table 4-18 summarises the participants' responses in this regard.

Participant 1 revealed good insight into his time-in-phase data and realised that by spending more time on design, design review and code review he could reduce his testing time. Through interpretation of his defect data, he identified a problem with his current review technique ("*I spent little time on reviewing by just scanning through the code and initial design*"), but failed to see any use for the defect descriptions. His primary focus of interpretation was on time rather than on defects: "*Process measurement data helps you to measure yourself on the different development phases, like what it is your doing most when working on a project"*. He also was the only participant who acknowledged a lack in his design skill and who realised that by addressing this shortcoming he would be able to change his development process, as illustrated by the following comment: "*Learn how to design effectively and being able to spend more time reviewing so I don't do more of code and fix"*.

After reviewing his process measurement data, Participant 2 proposed that he should "*try to identify and fix defects as early as possible*" to reduce testing time. He acknowledged that performing "*design reviews and code review*" could help him to reduce testing defects. He saw some value in defect data, but did not indicate how defect descriptions could help him to prevent future defects: "*Every error that you fix you must log whether big or small. Process measurement data helps you to rate* *yourself on detecting errors. It encourages you to improve as you write code*". Participant 2 revealed good insight into potential process improvement strategies: "*Spend more time on design, design review and code review*". In addition, he also acknowledged his lack of skill to perform each of these quality appraisal activities: "*Try to understand what must be done in each phase*".

Table 4-18: Summary of participants' proposed process changes

	How to reduce test time	**How to remove defects earlier**	**Changes to development process**
1	• Spend more time reviewing. • Design the requirements and then review to identify defects of the design. • Spend more time on code review after completing each segment of code. • Find and fix defects earlier by doing reviews.	• Do more thorough reviews.	• Learn to do design effectively. • Spend more time on reviewing. • Do not do code-and-fix.
2	• Do each phase step-by-step. • Try to identify and fix defects as early as possible.	• Do design reviews and code reviews.	• Spend more time on design, design review and code review. • Try to understand what must be done on each phase.
3	• Plan well. • Code efficient. • Review design for accuracy. • Fix problems as quickly as possible during testing.	• Review code. • Compile more often. • Do better planning to minimise defects.	• Fast coding. • Plan well before implementing anything.
4	• Code in full before testing. • Correct all defects at once.	• Do not spend too much time on one defect.	• No changes.
5	• Keep track of time spent in phases. • Do designs.	• Do better because this was bad.	• Become familiar with the data capturing tool (Process Dashboard©). • Remember to log all defects.
6	• Recap the basics of coding. • Avoid syntax and logical errors.	• Do code review and design review.	• Know the basics (programming principles and language syntax).

After reviewing his time-in-phase data, Participant 3 made a couple of questionable statements regarding strategies that he could follow to reduce testing time. "*Review my design*" could be an indication that he acknowledges the role that design reviews can play in reducing testing time. However, he did not give any indication that he should create a reviewable design to enable the review. Through the statements "*code as efficient as I can*" and "*fix errors as quick as possible*", he voiced his believe that doing tasks quicker will allow him to reduce his testing time. The statement "*I would plan well*" is very vague and does not point to any specific task-oriented activity/strategy to reduce testing time. After interpreting his defect data, Participant 3 identified three strategies for the earlier removal of defects. Although he realised that code reviews can lead to early defect removal, his proposed strategy to "*compile my program more often*" is still likely to result in defects being discovered late – essentially making the code-and-fix life cycle even worse. "*If I plan well, I would avoid having many defects*" is a vague statement and does not indicate a specific process related activity to reduce test defects. Even though Participant 3 indicated that process measurement data "*helps to manage time spent on error fixing*", he did not clearly indicate how this would be accomplished. He also believed that process measurement data "*gives ideas on how to conduct your product in terms of planning, coding, reviewing and testing*". This statement could be an indication that he has never really thought of development phases before and that the process measurement data would guide him to perform his development in more phases than just code-and-fix. It is also interesting to note that he left out the "design" phase in this statement. This might be an indication that he was confused with the difference between planning and design. As for improving his own process, he proposed "*fast coding*", which does not point to any concrete process improvement activity. He did, however, state "*planning well before implementing anything*", which might be an indication that he referred to a "design" phase that must be completed before coding.

In the interpretation of his process measurement data, Participant 4 focused on changes within his current process instead of the adoption of new techniques that would result in a process change. After reviewing his time-in-phase data, he proposed a technique where he would "*cod[e] in full before testing - to correct all the errors I have at once*" as a way of improving his current highly incremental code-and-fix development style. Although he did not come up with any process improvement ideas to remove defects earlier than in the testing phase, he could see (from his defect data) that he should "*try not to spend too much time on one defect*". He regarded process measurement data as a good way "*to do time management*". This is a good observation given that historical time data can be used for future project estimations. Although he indicated that measurement data can be used "*for recording of ways to improv[e] defects*", this was not clearly demonstrated in his proposal for earlier defect removal. Even though he stated that process measurement data "*is good for making one see his/her problems in software development"* and was unable to create a fully functional program, he still recommended "*no changes [for] now*" to his current software development process.

Participant 5 focused on the task of capturing process measurement data during the interpretation of his time-in-phase data and indicated no specific way in which this data could help him to improve his own development process. He made the following two remarks in this regard: "*Just have to keep track of amount of time during coding and other phases like design because that's what kept the process of testing much higher*" and "*One can adjust by practicing to keep track of time*". He also did not show any meaningful insight in his interpretation of defect data, as is evident from the following excerpt: "*I would try to do much better than this because this was fairly bad enough*". His interpretation of the value of process measurement data did, however, give some indication that he was beginning to think about doing development in different phases and that the data served as a tool for time management: "*Helps me to keep track of time spend on each section so that you can divide your time accordingly to attend all sections as far as possible*". In his development process improvement proposal, he indicated that he wanted to improve the *way* in which he captured measurement data and not *what* he would change about his development process: "*The changes that I would make would be to familiarise myself with Process Dashboard and remember the defect log for every section*". The participant's interpretations revealed that he had no idea of the actual purpose of process measurement data and saw the recording of this data as a separate activity that was not directly influencing his development practice. This is also evident in his description for doing reviews: "*During my code review, most of the time I was fixing errors and only doing the coding and I couldn't keep the thought of Process Dashboard to switch between defect log at all"*.

Since Participant 6 indicated technical programming skills as his major problem, his measurement data interpretations were mostly clouded by acknowledgements of his technical skill limitations instead of focusing on process improvements: "*Recapping on the basics of coding because my problem mainly was syntax and logical errors*". However, he did indicate that "*doing coding and design review will help*" to remove defects earlier in the life cycle, but failed to mention anything regarding his lack of designs. He also indicated that the process measurement data made him aware of his lack of knowledge in terms of "*where [in which phase] one should spend more time*", but did not mention anything regarding how he should spend the time. Despite some process-oriented interpretations of his defect data and the value of process measurement data, he still believed that the only way to improve his development process was by improving his technical skill. This is evident from the following statement: "*By knowing the basic programming principles, because I spent much time in coding and testing because of syntax and logical errors*". He therefore believed that that logical errors are the effect of poor knowledge of basic programming principles and not a lack of problem solving and design skills.

The PSP quality guidelines claim that quality can only be improved if it is measured and that the quality measurements should indicate "the effectiveness of the process for removing the defects" (Humphrey, 2005, p. 143). Participants 1, 2, 4 and 6 believed that debugging is the most effective defect removal strategy, which corresponded with their usage thereof. Participants 3 and 5 believed that code reviews are the most effective method for removing defects but ended up not doing any code reviews at all. Only Participant 1 made use of reviews and resolved one defect during code review at an efficiency rate that is just lower than his debugging efficiency rate. Those participants who did not do any reviews, consequently had no measurements to indicate the effectiveness of their use of QATs. Without the existence of these efficiency metrics there will be no motivation to adopt QATs as defect removal strategies.

Overall, the participants who struggled to produce a working program (Participants 3, 4, 5 & 6) displayed a less meaningful interpretation of their process measurement data. Their improvement proposals focused more on changing the activities in their current software process than on changing the process itself. Even though their current software development processes did not result in good performance, they still believed that they followed the optimum process and just needed to perform better at what they were already doing. On the contrary, the better performing participants (Participants 1 & 2) proposed process-oriented changes such as spending more time on creating effective designs and learning how to do more effective reviews. Although their current practices resulted in success, they were still motivated to find ways to further improve the quality of their current process, beyond their current capability.

4.5.2.3 Self-theory of intelligence

Through the participants' post-mortems (as recorded in the post-questionnaire and focus group discussion), they have provided some personal insights regarding their own intelligence and abilities. According to Dweck's (2000) Self-theory of Intelligence, a student's implicit assessment of his/her own intelligence and abilities could ultimately influence his/her individual motivations and behaviours. Given the development processes followed by the participants in the Phase 2 exercise, the actual adoption of QATs would require a fairly drastic change in normal "development" behaviour from them. It is also likely that not all participants will be equality motivated to adopt such a new behaviour. In using self-theory of intelligence as the third lens for this reflection, two behavioural attributes were identified.

Motivation

As part of her self-theory of intelligence, Dweck (2000) described the reactions of students in situations of failure as a "helpless" pattern. Some of these "helpless" responses were visible in the post-questionnaire responses of the participants. Participants 4, 5 and 6 blamed their own inability to get used to the new environment (Process Dashboard$^{©}$), the new practices and the unfamiliar problem as their reason(s) for failure (to capture accurate time and defect data) as is evident from the following responses:

Participant 4: "*My mind is slowly getting used to it [the process of capturing data], so as a result I confused times*".

Participant 5: "*Remembering to always check time was a problem because I was used to just coding*".
"*Just getting used to adjust to this new system*".

Participant 6: "*The big problem is being new to a program. If I get used to it, it would not be a problem*".

"*I forgot the steps to follow since the problem is new to me*".

Signs of "helplessness" were also observed during the focus group discussion. Participants 4, 5 and 6 acknowledged a deterioration in their problem-solving strategies and described how they made use of maladaptive practices such as "cargo culting" (O'Dell, 2017, p. 78) to produce code. Some of these participants also indicated that they "gave up". According to Participant 4, more time would not have helped him: "*Even if I had more time, I would not be able to solve this problem*". Participant 5 realised that he had "*more failure than success*" and that he, in future, "*would try to do much better than this because this was fairly bad enough*". Participant 6 condemned his ability by blaming his lack of basic programming skills: "*Recapping on OPG1 stuff [the basic of coding] because my main problem was syntax and logical errors*". The other three participants (1, 2 & 3) never showed any signs of questioning or blaming their own abilities. Instead, they started devising self-improvement strategies - thereby portraying behaviours that can be more closely linked to what Dweck (2000) referred to as the "mastery-oriented" pattern.

Behaviour

Linking to another attribute of the self-theory of intelligence, some participants revealed a "performance goal" orientation in which they "want to look smart (to themselves or others) and avoid looking dumb" (Dweck, 2000). In their perceived time-in-phase process data (as captured in the pre-questionnaire), Participants 5 and 6 indicated a process that included enough time in the design, design review and code review phases to remove defects early in the life cycle. Their perceived defect removal strategies also indicated that they used QATs such as design reviews and code reviews. Only Participants 5 and 6 indicated that they used all the listed defect removal strategies (design review, code review and debugging). Participant 3 indicated that he used code reviews and debugging. However, during the programming exercise, Participants 3, 5 and 6 ended up not using any of their perceived QATs and did not spend any time at all on design, design reviews or code reviews. Participants 3, 4, 5 and 6 indicated that they used checklists based on previous defects for conducting reviews. Not one of these participants described their defects clearly so that it could be used for future defect prevention as checklist items. Participants 5 and 6 were the only participants who could not produce workable programs during the programming exercise. As for their process improvement proposals, not one of Participants 4, 5 and 6 revealed any "learning-oriented goals" (Dweck, 2000) that could ultimately contribute to the use of QATs. Despite only being able to produce a partially working program, Participant 4 even went as far as stating that he would not make any changes to his current software development process. Although Participant 3 made some unsubstantiated statements for process improvement, there was some indication of awareness of the use of QATs to reduce testing time and to find defects earlier in the life cycle - as is evident from the following quotes:

> "*I would review my design for accuracy*".
>
> "*I should review my code*".
>
> "*Planning well before implementing anything*".

On the contrary, both Participant 1 and Participant 2 did not try to "look smart" when they completed the pre-questionnaire. Both participants indicated that they only used debugging as defect removal strategy and that they did not use any checklists. There was also no indication of the creation of checklists based on previously collected personal data. During the programming exercise, only Participant 1 used code reviews. Participants 1 and 2 also captured the most defects in their error logs and the high quality of their defect descriptions made theirs the only descriptions that could be usable for future defect prevention. In the post-questionnaire, both of these participants also indicated some learning-oriented goals that could ultimately contribute to their use of QATs.

Participant 1:	"*Learn how to do designs effectively*".
	"*Spend more time on reviewing and fixing while reviewing*" instead of "*just scanning through code and the initial design*".
Participant 2:	"*Spend more time on design, design reviews and code reviews*".
	"*Try to understand what must be done in each phase*".

These goals "reflected a desire to learn new skills, master new tasks, or understand new things" as specified by Dweck (2000) as being attributes of beholders of an incremental theory of intelligence. It is therefore interesting to note that only Participants 1 and 2 managed to produce completely functional programs.

Table 4-19 provides a summary of the identified skills and behavioural characteristics of each participant, mapped to the identified attributes.

Table 4-19: Mapping of participants skills and behaviours to identified attributes

Attributes	**Participant**					
	1	**2**	**3**	**4**	**5**	**6**
PSP0						
Understanding of development phases	X	X	X	X	X	X
Technical programming skills	✓	✓	✓	✓	X	X
Accuracy of measurement data	X	X	X	X	X	X
Ability to find and fix defects	✓	✓	✓	✓	X	X
PSP2						
Design skills	X	X	X	X	X	X
Design review and code review skills	X	X	X	X	X	X
Value of process measurement data	✓	✓	X	X	X	X
Self-theory of intelligence						
Motivation orientation	Mastery	Mastery	Mastery	Helpless	Helpless	Helpless
Achievement goal orientation	Learning	Learning	Performance	Performance	Performance	Performance

4.6 Summary

In this chapter, the methodology and the results of Phase 2 of this research study were reported. The aim of the Phase 2 research activity was twofold: (1) to form a better understanding of the differences between novice programmers' perceived and actual development processes and their use of QATs through the use of actual process measurement data (as prescribed by the PSP framework) supplemented by narrative data; and (2) to identify attributes that could potentially influence novice programmers' use of QATs. Firstly, the selection of an experimental case study approach as the main data source management strategy, as well as the sampling decisions made were outlined. Secondly, the data collection and analysis strategies were explained. In the main research activity, participants used the PSP0 and PSP2 framework scripts to guide them in capturing process measurement data while solving a programming problem. Thirdly, a detailed discussion of each participants' perceived and actual process data were presented. Fourthly, in order to form a better understanding of these novice programmers' development processes and use of QATs, a more thorough exploration of the evidence (as guided by the two main themes of Phase 2) were provided. The main findings can be summarised as follows:

Software development processes and use of QATs

Overall, the participants mostly used a code-and-fix style of development, as was also indicated in their perceived processes. There was a total lack of formal design documentation despite the indication of some formal design techniques that they typically used. Most participants spent a lot more time in testing than they perceived. Most participants used debugging as their only defect removal strategy. Although almost half of the participants indicated that they typically used design reviews and code reviews as well, they ended up using only debugging.

Attributes influencing the use of QATs

In using PSP0, PSP2 and the self-theory of intelligence as lenses, nine attributes that could potentially influence a novice programmer's use of QATs were identified (see Table 4-19). The attributes listed under PSP0 and PSP2 can be regarded as technical attributes, while the self-theory of intelligence attributes are motivational and behavioural in nature. These attributes can be regarded as critical skills and personal behaviours that could potentially influence the successful use of QATs.

To ensure effective use of QATs, the programmer first needs to make the decision to change his/her current software development process. However, it remains unclear which factors could influence a novice programmer's intent to adopt QATs as part of his/her personal software development process. This issue is addressed in Phase 3 of this research study, as described in the next chapter.

Chapter 5: Factors influencing novice programmers' intent to adopt QATs (Phase 3)

The overall aim of this research study was to explore the attributes that could potentially influence novice programmers' effective use of QATs in an educational context. In this regard, the Phase 2 research activity identified nine attributes that could potentially influence a novice programmer's use of QATs. However, it was noted that in order to ensure effective use of these QATs, the programmer first needs to make the decision to change his/her personal software development process. The aim of Phase 3 of this study was therefore to identify factors that could influence novice programmers' intent to adopt QATs. Phase 3 was therefore directed by the following research question:

RQ3.1: What are the factors that could influence novice programmers' intent to adopt QATs?

As with the discussion of the previous two research phases, the Phase 3 discussion also uses Plowright's basic FraIM structure (see Figure 1-1) as a roadmap to describe the methodology and results of this research activity. Firstly, the chosen data source management strategy and sampling decisions are explained. Secondly, an explanation is provided about how data was collected in Phase 3. Thirdly, the data analysis strategies are described and evidence that transpired from this part of the empirical investigation is provided.

5.1 Cases

For this part of the investigation, a case study approach (Plowright, 2011) was followed to identify various factors that are likely to influence novice programmers' intention to adopt QATs. The discussion in the following sub-sections clarifies the selection of the chosen data source management strategy and explains the sampling decisions.

5.1.1 Data source management strategy

For Phase 3, I wanted to collect data from students who already had sufficient programming knowledge as well as some experience with process measurement and reviews (code and design). Since the participants were to be sourced from the selected institution, the only group of students who matched the selection criteria were the fourth year Computer Science students who were registered for the Software Engineering module. The major focus of this module was object-oriented analysis and design, but students were also exposed to process measurement practices (using Process Dashboard©) and the use of reviews (design reviews and code reviews) as defect removal strategies. Since these students were already in their fourth year of studying programming, it could be assumed that they also had sufficient knowledge of basic programming. However, they had no industry programming experience and could therefore still be regarded as novice programmers. My choice of research participants was therefore limited by the available potential cases. Given the relatively small number of students registered for this module (55 students), I involved as many of these students as possible in the research activity, thereby satisfying one of Plowright's (2011) main criteria for the use of a case study as data source management strategy.

5.1.2 Sampling decisions

As explained in Section 5.1.1, the research population for this case was restricted to the fourth-year Software Engineering students from the selected institution (55 students). These students were selected because they were already familiar with the various techniques that could be used to improve the quality of their programs. Data was collected by means of 'asking questions' in a paper-based self-completion survey (Plowright, 2011). The survey was distributed and completed at the end of a scheduled lecture. Forty-seven students (the sample) completed the survey (85% response rate).

5.2 Data Collection Methods

There are numerous theoretical models that can be used to examine individual intentions to adopt methodologies. In this regard, Riemenschneider et al. (2002) identified 12 constructs that are appropriate in the context of software development methodology adoption. These constructs were defined as follows in the context of Phase 3:

- *Behavioural Intention (BI)* – the extent of the novice programmer's intention to use QATs.
- *Usefulness (U)* – the extent to which the novice programmer thinks that using QATs will enhance his/her programming performance.

- *Ease of use (EOU)* – the extent to which the novice programmer perceives that using QATs will be free of effort.
- *Subjective norm (SN)* – the extent to which the novice programmer believes that others, who are important to him/her, think he/she should use QATs.
- *Voluntariness (VOL)* – the extent to which the novice programmer perceives the adoption of QATs as non-mandatory.
- *Compatibility (C)* – the extent to which QATs are perceived as being consistent/compatible (incorporable) with the current manner in which the novice programmer develops systems.
- *Result Demonstrability (RD)* – the extent to which the results or benefits of using QATs are apparent to the novice programmer.
- *Image (IMG)* – the extent to which the use of QATs are perceived to enhance the novice programmer's image/status in his/her social system.
- *Visibility (VIS)* – the extent to which the use of QATs can be observed in the novice programmer's learning environment.
- *Perceived behavioural control* – internal (PBC-I) – the novice programmer's perceptions of internal constraints on using QATs.
- *Perceived behavioural control* – external (PBC-E) – the novice programmer's perceptions of external constraints on using QATs.
- *Career consequences (CC)* – the extent to which the adoption of QATs will influence the novice programmer's chance to secure employment after completing his/her degree.

The survey constructed for Phase 3 (see Appendix N) was based on the validated measurement scales from Riemenschneider et al.'s (2002) research study, with rewording of a number of items to make it relevant in terms of the context of the current study. Each item was based on a 4-point Likert scale (1 = strongly disagree and 4 = strongly agree).

5.3 Data Analysis and Evidence

The numerical data collected through the survey was analysed using IBM SPSS Statistics for Windows, Version 25.0. During initial analysis, Cronbach's alpha was used to assess the reliability of measurement items for each of the 12 constructs. For 10 of the constructs, the values of Cronbach’s alpha were between 0.640 and 0.841. These values were regarded as acceptable given the limited number of test items (Berger & Hänze, 2015). However, the values for *voluntariness* (Cronbach’s alpha = 0.418) and *perceived behavioural control - internal* (Cronbach’s alpha = 0.346) displayed low construct reliability. Therefore, only the 10 constructs with Cronbach’s alpha values of 0.640 or higher were retained for further analysis (see Table 5-1).

The next step was to identify the constructs that could be regarded as significant determinants of novice programmers’ intentions (BI) to use QATs. Each construct was tested individually using least-squares regression analysis. Table 5-2 shows the results of each construct test, indicating the names of the constructs as well as the beta coefficients, significance levels and R^2 values.

Ease of use and compatibility showed the highest significance, followed by usefulness and result demonstrability ($p < 0.01$). Subjective norm and career consequences were also significant ($p < 0.05$), while PBC-E, visibility and image were not significant. A comparison between these significant determinants and those identified in Riemenschneider et al.’s (2002) study reveal a number of interesting commonalities as well as several notable differences. When compared to the six significant determinants identified in the present study, Riemenschneider et al.’s study only identified compatibility, usefulness and subjective norm as significant determinants of methodology use intentions.

5.4 Summary

In this chapter, the analysis and results of Phase 3 of this research study were reported. The aim of the Phase 3 research activity was to identify factors that could influence novice programmers’ intent to adopt QATs. Firstly, the selection of a case study approach, the data source management strategy, as well as the sampling decisions made, were outlined. Secondly, the data collection strategies were explained. Thirdly, the data analysis strategies were described and evidence that transpired from this part of the empirical investigation was provided.

Table 5-1: TAM constructs retained

Construct	Scale items	alpha

Behavioural intention (BI)
Mean = 3.6809
SD = 0.45951
	• I intend to use QATs in future programming tasks.
	• Given the opportunity, I would use QATs.
		0.640

Usefulness (U)
Mean = 3.4433
SD = 0.37795
	• Using QATs improves my programming performance.
	• Using QATs increases my productivity.
	• Using QATs enhances the quality of my programs.
	• Using QATs makes it easier to do my programming tasks.
	• The advantages of using QATs outweigh the disadvantages.
	• QATs are useful in programming tasks.
		0.681

Ease of use (EOU)
Mean = 2.8156
SD = 0.45872
	• Learning QATs was easy for me.
	• I think QATs are clear and understandable.
	• Using QATs do not require a lot of mental effort.
	• I find QATs easy to use.
	• QATs are not cumbersome to use.
	• Using QATs do not take too much of my time.
		0.663

Subjective norm (SN)
Mean = 3.0071
SD = 0.73717
	• People who influence my behaviour think I should use QATs.
	• People who are important to me think I should use QATs.
	• My fellow students think I should use QATs.
		0.760

Compatibility (C)
Mean = 2.9504
SD = 0.56027
	• QATs are compatible with the way I develop systems.
	• Using QATs are compatible with all aspects of my programming tasks.
	• Using QATs fit well with the way I work.
		0.783

Image (IMG)
Mean = 2.9929
SD = 0.67204
	• Software developers who use QATs have more prestige than those who do not.
	• Software developers who use QATs have a high profile.
	• Using QATs are a status symbol amongst software developers.
		0.745

Visibility (VIS)
Mean = 2.4521
SD = 0.68888
	• QATs are very visible at the Department[4].
	• It is easy for me to observe others using QATs.
	• I have had plenty of opportunity to see QATs being used.
	• I can see when other students use QATs.
		0.748

Personal behavioural control – external (PBC-E)
Mean = 2.8553
SD = 0.57891
	• Specialised instruction and education concerning QATs are available to me.
	• Formal guidance is available to me in using QATs.
	• A specific group is available for assistance with QAT difficulties.
	• For making the transition to QATs, I felt I had a solid network of support (e.g. knowledgeable fellow students, student assistants, lecturers, etc.)
	• The Department provides most of the necessary help and resources to enable students to use QATs.
		0.724

Career Consequences (CC)
Mean = 3.2270
SD = 0.59123
	• Knowledge of QATs puts me on the cutting edge in my field.
	• Knowledge of QATs increases my chance of getting a job.
	• Knowledge of QATs can increase my flexibility of changing jobs.
	• Knowledge of QATs can increase the opportunity for more meaningful work.
	• Knowledge of QATs can increase the opportunity for preferred jobs.
	• Knowledge of QATs can increase the opportunity to gain job security.
		0.841

Result Demonstrability (RD)
Mean = 3.1383
SD = 0.62730
	• I would have no difficulty telling others about the results of using QATs.
	• I believe I could communicate to others the consequences of using QATs.
	• The results of using QATs are apparent to me.
	• I would have no difficulty explaining why QATs may or may not be beneficial.
		0.825

[4] Although the real name of the academic department and institution concerned were used on the actual instrument, it will not be disclosed here in order to protect the anonymity of the selected institution.

Table 5-2: Regression analysis of constructs

Construct	β	Standard error of β	t	Sig.	R^2
Ease of use	0.412	0.136	3.023	0.004**	0.169
Compatibility	0.341	0.111	3.065	0.004**	0.173
Usefulness	0.467	0.167	2.788	0.008**	0.147
Result demonstrability	0.280	0.101	2.779	0.008**	0.146
Subjective norm	0.224	0.870	2.587	0.013*	0.129
Career consequences	0.274	0.108	2.526	0.015*	0.124
Personal behavioural control - external	0.216	0.114	1.897	0.064	0.074
Visibility	0.156	0.097	1.614	0.113	0.055
Image	0.139	0.100	1.396	0.170	0.041

Notes: * $p<0.05$, ** $p < 0.01$

The case study conducted in Phase 3 revealed that students' intentions to adopt QATs are driven by six factors: ease of use, compatibility, usefulness, result demonstrability, subjective norm and career consequences. These usage intentions differ from those identified in studies that involved professional programmers (Agarwal & Prasad, 2000; Chan & Thong, 2009; Iivari, 1996; Riemenschneider et al., 2002). By combining these six factors with the nine attributes that could potentially influence a novice programmer's use of QATs (as identified in Phase 2), we end up with a list of 15 attributes that could potentially influence novice programmers' effective use of QATs.

This concludes the discussions of the methodology and results of each of the three research activities that were conducted as part of this study. Chapter 6 concludes the work by synthesising the empirical findings and outlining implications for research and practice. This final chapter also describes the limitations of this study and makes recommendations for future research.

Chapter 6: Discussions, Recommendations and Conclusions

The mixed-methods study presented in this thesis was conducted as a continuation of more than four decades of research that focused on evaluating the programming performance of novice programmers in educational environments and identifying possible reasons for the low quality of programs developed by them. Building on the principles of Total Quality Management (TQM) and related software quality improvement frameworks (such as CMM and PSP) that have been successfully implemented in the software industry, the study aimed to provide greater insights regarding attributes that could potentially influence novice programmers' effective use of QATs in an educational context. To this end, this study set out to answer the following main research question:

> *What are the attributes that could potentially influence novice programmers' effective use of quality appraisal techniques?*

In order to answer this main research question, three research activities were conducted (Phase 1, Phase 2 and Phase 3) to answer the following four subsidiary research questions:

RQ1.1:	*What is the quality of the typical software development processes followed by novice programmers?*
RQ2.1:	*How do novice programmers' perceived software development processes (including the use of QATs) differ from their actual processes?*
RQ2.2:	*What are the attributes that could potentially influence novice programmers' use of QATs?*
RQ3.1:	*What are the factors that could influence novice programmers' intent to adopt QATs?*

This chapter commences with a discussion of the empirical findings and conclusions from each of the three research activities to provide answers to the main and subsidiary research questions (as defined for each of the phases). Following this, the contributions and implications of this study are outlined. Next, limitations present in the current study are acknowledged and recommendations for future research are proposed. Lastly, a final conclusion to this study is presented.

6.1 Empirical Findings and Conclusions

This section outlines the empirical findings and conclusions from each of the three main research activities that formed part of this research study. In the first three sub-sections, the research questions as addressed by each of the phases are discussed. The final sub-section focuses on answering the main research question.

6.1.1 Phase 1 - Evaluating the quality of novice programmers' perceived software development processes

As a starting point to this research study, the aim of the Phase 1 research activity was to use the PSP framework as a basis for evaluating the quality of novice programmers' typical software development processes. Phase 1 was guided by the following research question:

RQ1.1: What is the quality of the typical software development processes followed by novice programmers?

In Phase 1, a survey approach was followed to gather information regarding novice programmers' perceptions of the software development process they typically use when developing programs. The participants comprised a large group of undergraduate CS students from a selected UoT. Data collected by means of 'asking questions' in a paper-based self-completion questionnaire were analysed and compared to the quality guidelines as set out in the PSP framework.

Overall, the evidence revealed that most students relied on a process of code-and-fix, as Humphrey (1999) predicted. Code-and-fix was clearly the process of choice from first- to third-year level, which indicated no process improvement through these years of study. Students also viewed testing as the most effective strategy to remove defects. Within the different levels of Humphrey's (2005) PSP quality framework, a number of interesting observations were made regarding the development processes followed by these novice programmers.

Based on Humphrey's (2005) PSP0 guideline that a developer should spend at least the same amount of time in the design phase than in the coding phase, it became clear that the students did not spend nearly enough time in the design phase. This seemed to be a direct consequence of the students' predominant reliance on development life cycles that did not include a design phase. It is possible that the students will continue with the same low-quality development process throughout their undergraduate studies because they do not really know how they spend their development time and are not aware of the common defects they make. The majority of the students (83.7%) did not keep track of the actual time spent in the different development phases, while 69.9% lacked a defect management strategy to prevent similar defects in future. This overall lack of process measurement and process awareness might explain why the students do not adopt more effective development processes and QATs.

In the process of developing quality software, PSP1 strongly emphasises the use of software estimation and planning techniques (Humphrey, 2005). The majority of students (87.8%), however, indicated that they did not use any time estimation techniques. In contrast to results of the MWG study (McCracken et al., 2001) where students listed time as the major reason for their failure, only 22.8% of the Phase 1 students regarded lack of time as the major contributor to their failure to create fully functional programs.

Within the PSP quality framework, PSP2 proposes a design specification structure with four design categories to describe the different modelling techniques that programmers use to model their designs (Humphrey, 2005). A design quality score was calculated for each student based on his/her use of the specified modelling techniques. Almost half of the students (46.3%) indicated that they never did any designs, while the average design quality score of those who did create designs was calculated as 16.16%. This overall low design score is an indication that the designs produced by the students are not "complete, accurate, and precise enough to ensure its quality implementation" (Humphrey, 2005, p. 220). Humphrey (2005) stated that “the lack of a precise design is the source of many implementation errors” and “an under-specified design can be expensive and error prone” (p. 221). The students’ lack of design completeness (indicated by the low design scores) could also explain the limited time spent in the design phase. Defect measurements could also be useful to indicate design defects and limitations that a developer has in design skills. Since the majority of the students (69.9%) indicated that they did not keep track of their most common defects, they are unlikely to realise the usefulness of complete and accurate designs.

As part of the quality management process in PSP2, developers are encouraged to employ effective QATs (such as design reviews and code reviews). Most students (76.8%) regarded testing/debugging as the most effective technique to remove defects, which corresponded with their preferred development life cycle of code-and-fix. As a result of the limited use of designs and the lack of completeness thereof, only 2.4% of the students perceived design reviews as the most effective way to remove defects. Few students (20.7%) indicated that they perceived code reviews as the most effective, but almost half of them (50.4%) indicated that they used it.

Since the questionnaire data only revealed insights regarding the students’ perceptions of the software development process they typically use when developing programs, there was no indication whether these were the processes they actually used. Further investigations were therefore needed to explore any possible differences between the perceived and actual development processes of novice programmers. To improve the quality of students’ programs, Humphrey (1999) suggested that CS educators should focus on their students’ actual process data of the programs they create. As a follow-up on Phase 1, Phase 2 of this research study therefore looked into the actual development processes followed by a group of novice programmers while using the PSP framework.

6.1.2 Phase 2 - Understanding novice programmers’ actual development processes and use of QATs

Humphrey (1999) claimed that one of the biggest challenges in software development is to convince software developers to adopt better practices as they tend to stick to a personal process that they have developed from the first small program they have written. Runeson (2001) also found it easier to convince first-year students (who do not yet have well established development habits) to use PSP practices as part of their natural development process. As part of the Phase 2 research activity, an integrated experimental case study approach was followed in an attempt to (1) form a better understanding of the differences between novice programmers' perceived and actual development processes (including the use of QATs) through the use of actual process measurement data (as prescribed by the PSP framework), supplemented by narrative data; and (2) identify attributes that could potentially influence novice programmers' use of QATs. Phase 2 was therefore guided by two research questions - each of which are addressed in the following sub-sections.

RQ2.1: How do novice programmers' perceived software development processes (including their use of QATs) differ from their actual processes?

Overall, the participants mostly used a code-and-fix style of development as indicated in their perceived processes. Similar to the findings of Hou and Tomakyo (1998), these students also created no actual design documentation despite their indications of the numerous design modelling techniques (use cases, flowcharts, pseudo code, data flow diagrams, class diagrams) they typically used. Other design studies (Eckerdal et al., 2006a; Lotus et al., 2011) also reported a total lack of formal design documentation with their students. However, most participants indicated that they typically used flowcharts - a design modelling technique that would have been perfectly suitable for modelling the logic for this specific exercise. This could suggest that although they have the necessary theoretical knowledge, they did not know how to implement it correctly. Williams (1997) made a similar observation regarding students' theoretical knowledge of PSP principles. The overall lack of formal design documentation for the exercise could also be attributed to the relatively small size of the exercise. Schach (2011) suggested that a code-and-fix model without any design could be suitable for short programming exercises (less than 200 lines of code).

Most participants in this study completely underestimated the time they typically spent resolving defects during testing. Since most participants used debugging as their primary (and only) defect removal strategy, the higher than expected testing times were not surprising. Although half of the participants indicated that they typically used design reviews and code reviews as well, they ended up using only debugging. What is still of concern, however, is that some participants indicated that they used design reviews and even assigned a substantial amount of time to it while they ended up not doing any designs at all. Students in Towhidnejad and Salimi's (1996) study found it easier to adopt code reviews as part of their quality improvement process since they regarded it as more closely related to programming.

RQ2.2: What are the attributes that could potentially influence novice programmers' use of quality appraisal techniques?

Following the evidence that resulted from the Phase 2 research activity, nine attributes that could potentially influence novice programmers' use of QATs as prescribed by the PSP framework were identified.

Understanding of development phases

Grove (1998) indicated that beginner programmers struggle with the distinction between development phases. Some Phase 2 participants indicated that they did not know exactly what to do in the design, design review and code review development phases, but acknowledged that the defined process made them aware that something needed to be done in these phases. Carrington et al. (2001) emphasised that students struggle to distinguish between the various phases when they code and test one line of code at a time.

Technical programming skills

One of the code review principles of PSP is that one must firstly produce a reviewable product (Humphrey, 2005). Some of the Phase 2 participants who struggled with the programming exercise, identified their lack of basic programming skills as the major cause of their inability to produce working code. Carrington et al. (2001) argued that the additional work required to implement the PSP can cause a cognitive overload for some students, especially for those who struggle with basic programming skills. For this reason, Runeson (2001) agreed that it would be harder for first-years to learn PSP practices together with the development of programming skills, but he stated that these students are the ones that can benefit the most from following such practices.

Accuracy of measurement data

Towhidnejad and Salimi (1996) reported that only half of their students collected accurate and reliable data. All the Phase 2 participants indicated that they had difficulties to capture data in the correct phases. The major problem was that they could not differentiate between coding and testing. This confusion could have been caused by the coding-rework that they had to do in order to fix a defect. Because of their heavy reliance on a code-and-fix style of programming, it created a situation where most of the coding was done in response to fixing a defect. The confusion led to inaccurate data that should have been logged under testing and not under coding. This could also have influenced the time that was logged to find and fix a defect. Carrington et al. (2001) emphasised that when students write a program by incrementally compiling and testing one line of code at a time, they are unable to log data in the correct prescribed PSP phases. Grove (1998) argued that his students initially struggled with the distinction between development phases and therefore had problems to collect reliable data. Williams (1997) suggested that discussions of group statistical feedback data might influence students' intention to capture more accurate individual process measurement data.

Ability to find and fix defects

Humphrey (2005) stated that the process of debugging is “the time required to get from symptoms to defects” (p. 195). He defined debugging as “the process of finding the defective code that caused the program to behave improperly” (p. 195). From the participants' defect descriptions, it was evident that some of them could not resolve all the defects regardless of their defect removal strategy. Some of the Phase 2 participants also struggled to describe their defects clearly. In most cases, the poor description of the defects could be linked to their inability to determine the cause of the defect. Poor descriptions of defects are likely to lead to difficulties when defect data are used to create personal checklist items. None of the participants in this study realised that the purpose of defect data was to act as a means of defect prevention. Some participants also mentioned that they did not log all defects. Prechelt (2001) indicated that more than half of his PSP participants could not “keep the self-discipline required for defect logging" (p. 61) and regarded it as a "personality issue". Not logging all defects can lead to misinterpretations regarding the severity of defects, as well as the exact phase (planning, design, coding or testing) during which the defect occurred. According to the participants' data, most defects were injected during the coding phase. Consequently, their defect measurement data did not provide enough detailed information to prevent similar defects in future. Carrington et al. (2001) also noted that that most of their students did not record defect data accurately. However, they failed to mention specific details regarding the quality of their students' defect descriptions and the consequences thereof.

Design skills

The PSP quality management strategy recommends that developers' first focus should be on producing "a thorough and complete design and then document the design with the four PSP design templates" (Humphrey, 2005, p. 155). Various studies reported that even after having completed their undergraduate studies, most CS students generally lack basic design knowledge (Chen et al., 2005; Eckerdal et al., 2006a, 2006b; Hu, 2016; Loftus et al., 2001). None of the Phase 2 participants created formal design documents. Even the better performing participants indicated that they lacked the skills necessary to create effective designs. The lack of a well-documented and complete design directly influenced the use of design reviews as a defect removal strategy. Rong et al.'s (2016) students indicated that detailed design is one of the most useful practices for achieving defect free programming (DFP).

Design review and code review skills

None of the Phase 2 participants used properly defined strategies for performing design reviews and code reviews. Even though Humphrey claimed that the "reviewability" of a design is not that important if you review your own designs, he still stated that "without a well-documented and complete design, it is impossible to do a competent design review" (Humphrey, 2005, p. 185). The lack of complete designs could therefore have been the reason for the absence of design reviews. Most participants produced code that could be reviewed, but none of them used code reviews either. Even though some of the Phase 2 participants indicated that defects could be removed earlier, and that testing time could be reduced through the use of design reviews and code reviews, none of them attempted to use it. The only logical conclusion that can be drawn is that the participants lacked proper strategies for performing design reviews and code reviews. The total lack of reviews also caused a lack of defect removal efficiency measurements for their reviews. The absence of an efficiency indicator for the value of reviews could cause a barrier to adopting reviews as defect removal strategies. As for Jenkins and Ademoye's (2012) students, the additional time that they took to complete code reviews could negatively influence the value they placed on reviews and consequently the adoption thereof. On the contrary, Towhidnejad and Salimi (1996), as well as Williams (1997), reported that students found code reviews more relevant to programming, and therefore, easier to adopt than time management. Rong et al. (2012) suggested future research in methods that could improve the effectiveness of reviews.

Value of process measurement data

Various studies reported on the low value that students place on PSP principles and the consequential abandonment of PSP practices. In several studies, students objected to process measurement because they either found it unrelated to software development (Bullers, 2004; Towhidnejad & Salimi, 1996; Williams, 1997) or regarded it as extra effort (Börstler et al., 2002; Carrington et al., 2001; Hou & Tomayko, 1998). In some studies, students objected to the use of the PSP defined process because this process was not compatible with their current development practices (Carrington et al., 2001) or they regarded it as too strict and therefore could not see the potential benefits of using this disciplined process (Börstler et al., 2002).

In contrast, the Phase 2 participants placed a similar value on PSP principles as Grove's (1998) students. They recognised the value of using proper programming methodologies, good designs and reviews. They further realised the importance of process measurement data as a motivator for continuous improvement. Even though not all Phase 2 participants illustrated the same level of interpretation of their process data, most of them agreed that it provided valuable insights regarding their development processes. This is similar to Börstler et al.'s (2002) study where students showed initial resistance to PSP, but the general reaction at the end of the course was that they were "more aware of their programming practices and shortcomings" (p. 45). Those Phase 2 participants who struggled to produce a working program displayed a less meaningful interpretation of their process measurement data. A possible explanation for this could be, as reported by Carrington et al. (2001), that the additional work caused a cognitive overload for some students, especially for those who struggled with basic programming skills. One of the Phase 2 participants revealed some disconnected behaviour towards PSP similar to the claim by Williams (1997) that students saw time management as irrelevant to software development and obstructed their focus from programming. Some of the process improvement proposals made by the participants in Phase 2 did not transpire from their own, personal data. They instead made proposals that could be theoretically regarded as good improvements. In Williams' (1997) study, the students also demonstrated accurate theoretical knowledge of PSP principles, but struggled with the application thereof.

Individual motivations and behaviours

According to Dweck's (2000) Self-theory of Intelligence, a students' implicit assessment of his/her own intelligence and abilities could ultimately influence his/her individual motivations and behaviours. During the post-mortem reflections, some participants revealed a "helpless" orientation, and consequently blamed their own inability to get used to the new environment (Process Dashboard©), the new practices and the unfamiliar problem as their reason(s) for failure. They also acknowledged a deterioration in their problem-solving strategies and described how they used maladaptive practices like "cargo culting" (O'Dell, 2017, p. 78) to produce code.

Some participants revealed a "performance goal" orientation in which they "want[ed] to look smart (to themselves or others) and avoid looking dumb" (Dweck, 2000). Participants who did not try to "look smart" indicated some learning-oriented goals that could ultimately contribute to process improvement. These goals "reflected a desire to learn new skills, master new tasks, or understand new things", as indicated by Dweck (2000) as being attributes of beholders of an incremental theory of intelligence. These participants were the only ones who managed to produce completely functional programs.

6.1.3 Phase 3 - Factors influencing novice programmers' intent to adopt QATs

Various researchers have reported on the challenges they experienced in motivating their students to adopt PSP practices (Börstler et al., 2002; Bullers, 2004; Carrington et al., 2001; Hou & Tomayako, 1998; Towhidnejad & Salimi, 1996; Williams, 1997). There have also been numerous calls for further investigations into the factors (other than training) that might influence the adoption of PSP methods (Prechelt & Unger, 2001; Rong et al., 2012). In Phase 3, a case study approach was followed to identify factors that could influence novice programmers' intent to adopt QATs. Phase 3 was therefore directed by the following research question:

RQ3.1: What are the factors that could influence novice programmers' intent to adopt QATs?

As a starting point in identifying these factors, Riemenschneider et al.'s (2002) 12 constructs for methodology adoption were used: behavioural intention; usefulness; ease of use; subjective norm; voluntariness; compatibility; result demonstrability; image; visibility; perceived behavioural control (internal); perceived behavioural control (external); and career consequences.

Results of the Phase 3 case study revealed that students' intentions to use QATs are driven by ease of use, compatibility, usefulness, result demonstrability, subjective norm and career consequences. These usage intentions differ from those identified in studies that involved professional programmers (Agarwal & Prasad, 2000; Chan & Thong, 2009; Iivari, 1996; Riemenschneider et al., 2002).

The results of Phase 3 also showed that the perceived compatibility of software process innovations (such as QATs) with a novice programmer's pre-existing software development process had a highly significant influence on intention to use. Chan and Thong (2009) emphasised that the adoption of innovations often require a radical change in the developers' existing work practices. If the innovation is not compatible with the developers' current practices, they are unlikely to perceive it as beneficial. Carrington et al.*'s* (2001) students objected to the use of the PSP defined process because it was not compatible with their current development practices. In several other studies (Bullers, 2004; Towhidnejad & Salimi, 1996; Williams, 1997), students objected to process measurement because they found it unrelated to software development. In a study comparing the PSP experiences of first-year and graduate students, Runeson (2001) argued that it is easier to convince first-year students to use PSP as part of their development process since they have not yet formed established development habits. This argument supports Humphrey’s statement (1999) that students are likely to stick to the development process they used on their very first program. Both Towhidnejad and Salimi (1996) and Williams (1997) reported that students found code reviews more relevant to programming and therefore easier to adopt than time management.

There are numerous examples of prior studies that have found perceived usefulness a significant factor in predicting professional developers’ intention to use software process innovations such as software development methodologies (Chan & Thong, 2009; Riemenschneider et al., 2002), programming languages (Agarwal & Prasad, 2000) and CASE tools (Iivari, 1996). Overall, these studies suggest that an innovation is only likely to be accepted if it is perceived as being useful in increasing job performance (Chan & Thong, 2009). Similar to software developers in an industry environment, student developers are also influenced by a reward structure. They want to be productive and attain high marks for their assignments. If they do not see QATs as beneficial to their productivity, they are unlikely to regard it as being useful. Börstler et al.’s (2002) students abandoned the use of PSP principles because they could not see the potential benefits of such a defined process. From a productivity viewpoint, Jenkins and Ademoye’s (2012) reported that even though the students in their study believed that reviews improved the quality of their programs, they indicated that the biggest problem with code reviews was the extra time that it took. In this regard, Rong et al. (2012) could not find any concrete evidence that the efficiency of reviews were improved by the use of checklists. Even though the use of checklists serve as a useful guide for novice programmers, Rong et al. (2012) still stressed that researchers need to find methods to improve the effectiveness of reviews. Hou and Tomayko (1998) reported a significant decline in their students’ compile defects after the introduction of code reviews. They also reported that non-PSP trained students took significantly longer to produce their final projects and on average obtained lower marks.

In support of prior studies, the Phase 3 results also showed that subjective norm significantly affects intention. Riemenschneider et al. (2002) warned that developers who believe in the usefulness and compatibility of a software process innovation might avoid using the innovation because of the negative views of peers and supervisors who oppose the use thereof. Chang and Thong (2009) concluded that the significance of subjective norm as a determinant can be attributed to the importance of teamwork in software development. The student software developers in the context of Phase 3 were also required to complete a number of group projects as part of their other modules. Even in cases where they are working on individual projects, the students often form study groups to help one another. This creates a social learning environment where students could be subjected to peer influences. Students at the University of Utah who followed PSP practices during pair programming activities reported a higher level of enjoyment and higher confidence levels in their own work (Börstler et al., 2002). These students also mentioned that they "encouraged each other to follow PSP practices" (p. 45).

The differences noted between studies conducted in industry contexts and educational contexts regarding the effect of ease of use on adoption, could possibly be attributed to the difference between work and education environment. While professional developers are already using the innovation, students are still in the process of learning how to use it. The professionals may have already moved beyond early concerns regarding the effort required to use the innovation (Chan & Thong 2009, p. 811). Chang and Thong (2009) also argued that the diverse views on the result demonstrability of methodology use in Riemenschneider et al.'s study may be attributed to the long development cycles of real-world methodologies – preventing software developers "from observing the results in a short period of time" (p. 811). While the students in Phase 3 have not necessarily used QATs in their own development projects, they might believe that they have adequate theoretical knowledge regarding the benefits of using QATs. In an attempt to improve students' utilisation of PSP, Williams (1997) found that even if students have theoretical knowledge of the process, they might still struggle to apply it.

The significance of career consequences as a determinant of students' intention to adopt QATs could also be attributed to the educational context. Since students are preparing to enter the job market, they are likely to regard their familiarity with industry-used techniques as something that will influence their chances of securing employment after the completion of their degree. In Börstler et al.'s (2002) study, students who have used PSP in their first-year course reported that their knowledge of software engineering principles helped them to obtain summer internships.

6.1.4 Attributes that could potentially influence novice programmers' effective use of QATs

This exploratory research study was directed by one main research question:

What are the attributes that could potentially influence novice programmers' effective use of quality appraisal techniques?

By using the PSP quality improvement framework as an evaluation tool, the results of the Phase 1 research activity revealed that the novice programmers typically used a code-and-fix development strategy, their designs were almost non-existent, they mostly relied on testing to remove defects and they did not make use of measurements to gain insight into their development processes. The quality of the processes used by these novices did not change from first- to third-year level - an indication that they have already formed established development practices that will be difficult to change. The majority of the more experienced novices (second- and third-year students) did, however, recognise that the low quality of their programs were caused by their 'inability to identify defects'. In contrast, the less experienced novices believed that 'lack of technical skills' was the major contributor to low quality products.

In Phase 2, the low-quality of novice programmers' development practices was confirmed through a comparison of perceived and actual development processes. Surprisingly, for some of these novices' their perceived usage of QATs was totally different from their actual use thereof. Even in their proposal improvements these individuals revealed a strong theoretical knowledge of techniques that should be used for early defect removal and defect prevention. The noted differences between the perceived and actual use of QATs could possibly be attributed to these individuals' performance goal orientation. Further analysis of the actual process data and narrative data revealed that the following difficulties could have hampered their effective use of QATs:

- Lack of understanding of what exactly needs to be done in the prescribed development phases;
- Lack of technical programming skills;
- Inaccurate measurement data;
- Inability to identify defects;
- Lack of design review and code review skills;
- Lack of design skills;
- Inability to interpret measurement data;
- Helpless motivational orientation during problem solving; and
- Lack of learning-oriented achievement goals.

Based on these identified difficulties, nine attributes that could influence novice programmers’ use of QATs were identified. These attributes can be linked to individual technical skills and abilities as well as individual motivations and behaviours. However, in order to ensure effective use of QATs, the developer first needs to make the decision to adopt a new development process that includes QATs. The six factors (in Phase 3) that have been shown to influence a novice programmer’s intent to adopt such a process (ease of use, compatibility, usefulness, result demonstrability, subjective norm and career consequences) can therefore also be regarded as attributes impacting the effective use of QATs. By combining the results of Phase 2 and Phase 3, this research study therefore identified 15 attributes that could potentially influence novice programmers’ effective use of QATs as summarised in Table 6-1. In this summary, the original intention to adopt attributes (factors) are grouped with the two attributes that can be linked to the self-theory of intelligence (achievement goal orientation and motivational orientation) as behavioural and motivational attributes that could potentially influence a novice programmer’s intent to adopt QATs. The remaining seven attributes can be linked to individual skills and abilities.

The PSP quality management principles (defect prevention and early defect removal) rely heavily on the creation of complete designs and the use of proper review strategies. Without a complete design it would be impossible to perform design reviews. In Phase 2, the novice programmers showed limited design abilities and a total lack of review skills. They were also unsure about what exactly needed to be done in the prescribed development phases (other than coding and testing). Numerous studies have reported that even after completion of their undergraduate studies, most CS students are still unable to design software and generally lack even basic design knowledge. It is therefore not an easy task to get novice programmers to create complete designs that would be suitable for defect prevention and early defect removal. Given that the Phase 3 participants indicated 'ease of use' and 'result demonstrability' as major factors for adoption, it is likely that they would only adopt a new strategy if it is quick and easy to use and the effect thereof is noticeable immediately. It is therefore unlikely that novice programmers will recognise the 'usefulness' of complete designs.

Table 6-1: Attributes that could influence effective use of QATs

Category	**Attributes**
Skills and abilities	• Understanding of development phases • Accuracy of measurement data • Value of process measurement data • Technical programming skills • Ability to find and fix defects • Design skills • Design review and code review skills
Behaviour and motivation	• Career consequences • Compatibility • Ease of use • Result demonstrability • Subjective norm • Usefulness • Achievement goal orientation • Motivational orientation

The use of designs and reviews would also require the ability and willingness from these novices to make a drastic change to their current well-established code-and-fix practices. What make designs and reviews even less likely to be used is that the novices in this study indicated 'compatibility' as a major factor for adoption. Thong (2009) emphasised that the adoption of innovations often requires a radical change in the developers' existing work practices. If the innovation is not compatible with the developers' current practices, they are unlikely to perceive it as beneficial.

It was enlightening that most of the novices in this study realised that the low quality of their programs was caused by their inability to find and fix defects. Instead of focusing on efforts to get novice programmers to adopt quality processes (see Section 2.3), it might therefore be easier to direct their focus towards the costs of their failures (defects) through the use of process measurement data. Based on TQM and PSP principles, the foundation of quality improvement lies in process measurement data and the interpretation thereof. While the additional task of capturing data can cause a cognitive overload for novice programmers (Carrington et al., 2001), the amount of data can also be overwhelming (Kan et al., 1994). Kan et al. (1994) therefore suggest that the number of metrics need to be selected carefully and that the information extracted from the data should be focused, accurate and useful.

Based on these findings and arguments, a process improvement strategy that focuses on a measured defect management approach might be more effective to encourage the use of QATs. In following such an approach, students only need to measure the defects they make and the time they spend on fixing these defects (thereby reducing the number of metrics). These focused defect measurements could help to make students more aware of the cost of rework - especially when defects are picked up late in the development process (with testing). Students would therefore not be forced into following defined processes that are not compatible with their current processes. Instead, minimal but useful measurement data might convince them to consider the adoption of QATs.

6.2 Contributions and Implications of the Study

The Software Engineering discipline recommends various best practices for achieving quality in software projects. These practices are typically taught to CS students as part of their undergraduate studies. The PSP framework was specifically created as a self-improvement process to guide software developers in following good development practices. A number of researchers have reported on attempts to incorporate PSP principles as part of their Computer Science curriculums. Several attempts have also been made to develop models for the evaluation of the quality of students' software designs. Previous studies have, however, failed to recognise the potential value of the PSP framework as an evaluation model to assess the quality of individual development practices. The unique contribution of Phase 1 of this study was therefore to use the PSP framework as a model for evaluating the quality of novice programmers' software development processes. Humphrey (1999) suggested that educators should shift their focus from the programs that the students create to the data of the processes that they followed. The model created in Phase 1 could be used by educators to "quickly" assess the quality of their students' development processes without the effort of gathering actual process measurement data. This model could also be used to identify "gaps" in the education of novice programmers. However, it should be noted that there could be differences between the perceived and actual development processes of novice programmers as illustrated in Phase 2 of this study.

Various researchers have reported on their own experiences with the incorporation of PSP principles in educational environments in an attempt to improve the quality of their student's development processes (Börstler et al., 2002; Jenkins & Ademoye, 2012; Towhidnejad & Salimi, 1996; Williams, 1997). While all of these attempts have had some level of success, a number of problems regarding the students' use of the PSP principles were mentioned. The unique contribution of Phase 2 of this study was the identification of a list of critical success factors for novice programmers' use of QATs (based on data gathered from actual process measurement data and narrative feedback). The focus of Phase 2 was to specifically investigate attributes that could influence the use of QATs (design, design review and code review) as primarily utilised in the context of PSP. This list of attributes can be used as cautionary guidelines when educators attempt to use PSP principles or QATs to improve the quality of novice programmer's development practices. The identified technical attributes can also be used as a skills development guide for novice programmers to ensure the success of quality improvement efforts.

Various researchers have reported on the challenges they experienced in motivating their students to adopt PSP practices (Börstler et al., 2002; Bullers, 2004; Carrington et al., 2001; Hou and Tomayako, 1998; Towhidnejad & Salimi, 1996; Williams, 1997). There have also been numerous calls for further investigations into the factors (other than training) that might influence the adoption of PSP methods (Prechelt & Unger, 2001; Rong et al., 2012). There are numerous theoretical models (e.g. TAM, TAM2, PCI, TPB and MPCU) that are typically used to examine individual intentions to adopt technology tools - not processes. Riemenschneider et al. (2002) showed that these models could be used to provide insights into methodology adoption by software developers in a large organisation. Through the Phase 3 research activity, Riemenschneider et al.'s (2002) adoption model was used to specifically examine novice programmers' intent to adopt QATs as part of their natural development processes in an educational context. No other reported studies could be found that specifically used technology adoption models to examine individual intentions to adopt PSP practices.

The results of the Phase 2 research activity revealed a possible link between self-theory of intelligence and an individual's ability and/or intent to change his/her current software development practices. Quality improvement, as suggested by Deming's (2000b) PDSA cycle, requires continuous improvement of a product or process through statistical variation control. Since continuous improvement through the interpretation of process measurement data is central to the PSP framework, self-theory of intelligence could have a direct impact on the effective utilisation of PSP's QATs. Another unique contribution of this study is the suggested relation between self-theories of intelligence and the use of QATs in the specific context of the PSP. Due to the exploratory nature of this research study, the 'Why' of this suggested relation fell outside the scope of the stated research aim.

6.3 Limitations and Recommendations for Future Research

This study has offered an exploratory perspective on attributes that could potentially influence novice programmers' effective use of QATs. As a direct consequence of this methodology, the study encountered a number of limitations that need to be acknowledged. In most cases, these limitations could be addressed by future research.

The research activities conducted as part of this study focused on specific groups of students (as novice programmers) within a specific context (a selected South African

UoT). Since this was an exploratory study, the aim was merely to explore the research topic and not to provide final and conclusive answers to all the research questions. Consequently, no claims are made to the generalisability of the findings and conclusions. The findings and conclusions of this study could, however, be used as a departure point for the creation of new hypotheses to inform further research and debate in similar contexts.

Although all the Phase 2 participants recorded time in the planning and design phases, no formal design documentation were created. They could therefore not present any evidence of what they actually did in these phases. In this regard it would be interesting to investigate the actual planning and design processes followed by novice programmers.

As indicated in Phase 2, the students' lack of designs could potentially be the main driver behind their preference towards a code-and-fix development process. However, during the Phase 2 focus group discussion, the students indicated that their first step in solving programming problems was to "search the Internet" for solutions. Although the students were allowed to use the Internet during the programming activity, no data was collected on what they searched for, what they found and how the 'discovered' code were incorporated as part of their own program code. An in-depth investigation could also shed more light on the extent of which this 'copy-paste-and-fix' style of programming is used by undergraduate computer programming students. If this is found to be the dominant style among students, instructors could focus on equipping students with proper "Copy and Paste" skills [as suggested by Feiner and Krajnc (2009)]. While additional attempts to improve students' code reading and interpretation skills could advance their ability to review and debug their own code (Perkins et al., 1986), it could also enable them to effectively reuse code snippets copied from the Internet and other sources. However, it is recommended that instructors enforce effective design techniques from the first programs that students write in an effort to ensure that the students will not fall back on an unstructured code-and-fix or copy-past-and-fix life cycle. It would also be interesting to experiment with design activities that could motivate students to perform designs and realise the value of good designs. This could also include experiments with technologies and tools that could help students to form better design practices.

One particular aspect that had a significant impact on the students' programming performance was their inability to find and fix defects (debugging). The only defect data that were captured were the defect descriptions, the phase in which the defect was injected and resolved, and the time it took to fix the defect. No specific data was collected to provide insight regarding the actual defect removal processes that were used. Further investigations in this regard could shed more light on why students are unable to resolve specific defects. In an attempt to encourage earlier removal of defects, in would be interesting to have students experiment with different review strategies in order to improve the effectiveness of their reviews and possibly encourage the usage of such strategies.

As part of the attribute identification process in Phase 2, it was suggested that a student's self-theory of intelligence could impact his/her individual motivations and behaviours. Future research could be conducted to further investigate the possible influence of self-theory on the development practices of novice programmers.

6.4 Conclusion

In spite of all the efforts of CS instructors to get their students to improve the quality of their software programs through the use of PSP principles and strategies, the results are often worse than expected. This study was an attempt to direct the focus of CS instructors towards attributes that could influence the effective use of QATs (design review, code review, designs templates and quality measures) by novice programmers. The identified attributes were grouped into two categories: skills and abilities needed for the effective use of QATs; and behavioural and motivational attributes influencing the intention to adopt QATs (also see Table 6-1). It was, however, argued that the mastery of these skills and abilities could be influenced by the adoption attributes. Based on the overall findings and supporting arguments, a process improvement strategy that focuses on a measured defect management approach might be more effective to encourage the use of QATs by novice programmers.

Although I am still a long way from achieving my ultimate goal in life - convincing students to use QATs to improve the quality of their software programs - this study provided me with better insights regarding attributes that could potentially influence the software quality improvement attempts of my students.

> “Quality work is not done by accident; it is done only by skilled and motivated people.”
>
> \- *Watts S. Humphrey*

List of References

Agarwal, R., & Prasad, J. (2000). A field study of the adoption of software process innovations by information systems professionals. IEEE Transactions on Engineering Management, 47(3), 295-308. https://doi.org/10.1109/17.865899

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. In J. Cunha & W. Fleischman (Chairs), Proceedings of the 10th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE’05) (pp. 84-44). Association for Computing Machinery. https://doi.org/10.1145/1067445.1067472

Ajzen, I. (1985). From intentions to action: A theory of planned behavior. In J. Kuhl and J. Beckmann (Eds.), Action Control (pp.11–39). SSSP Springer Series in Social Psychology. Springer. https://doi.org/10.1007/978-3-642-69746-3_2

Ajzen, I. (1991). The theory of planned behavior. Organizational Behavior and Human Decision Processes, 50(2), 179–211. https://doi.org/10.1016/0749-5978(91)90020-T

Babbie, E. (2010). The practice of social research (12th ed.). Cengage Learning.

Berger, R., & Hänze, M. (2015). Impact of expert teaching quality on novice academic performance in the jigsaw cooperative learning method. International Journal of Science Education, 37(2), 294–320. https://doi.org/10.1080/09500693.2014.985757

Börstler, J., Carrington, D., Hislop, G.W., Lisack, S., Olson, K., & Williams, L. (2002). Teaching PSP: challenges and lessons learned. IEEE Software, 19(5), 42–48. https://doi.org/10.1109/MS.2002.1032853

Bullers, W.I. (2004, January). Personal software process in the database course. In R. Lister & A. Young (Eds.), Proceedings of the 6th Australian Conference on Computing Education (ACE’04) (Vol. 30, pp. 25–31). Australian Computer Society, Inc.

Carrington, D., McEniery, B., & Johnston, D. (2001). PSP^{SM} in the large class. In Proceedings of the 14th Conference on Software Engineering Education and

Training, In search of a software engineering profession (Cat. No. PR01059) (pp. 81–88). IEEE. https://doi.org/10.1109/CSEE.2001.913824

Chan, F.K.Y., & Thong, J.Y.L. (2009). Acceptance of agile methodologies: A critical review and conceptual framework. Decision Support Systems, 46(4), 803–814. https://doi.org/10.1016/j.dss.2008.11.009

Chen, T., Tew, A.E., Fincher, S., Cooper, S., Stoker, C., Simon, B., Bouvier, D., Powers, K., Blaha, K., Sanders, D., Johnson, H., Robins, A., Eckerdal, A., Ratcliffe, M., McCartney, R., Tenenberg, J., Moström, J.E., Schwartzman, L., Vandegrift, T., ... Monge, A. (2005). Students designing software: a multi-national, multi-institutional study. Informatics in Education, 4(1), 143–162.

Compeau, D.R., & Higgins, C.A. (1995). Computer self-efficacy: development of a measure and initial tests. MIS Quarterly, 19(2), 189–211. https://doi.org/10.2307/249688

Cresswell, J.W. (2014). Research design: Qualitative, quantitative and mixed methods approaches (4th ed.). Sage.

Crosby, P.B. (1979). Quality is free. McGraw-Hill.

Crosby, P.B. (1984). Quality without Tears. McGraw-Hill.

Davis, F. (1989). Perceived usefulness, perceived ease of use, and user acceptance of Information Technology. MIS Quarterly, 13(3), 318–339. https://doi.org/10.2307/249008

Davis, F.D, Bagozzi, R.P., & Warshow, P.R. (1992). Extrinsic and intrinsic motivation to use computers in the workplace. Journal of Applied Social Psychology, 22(14), 1111–1132. https://doi.org/10.1111/j.1559-1816.1992.tb00945.x

de Champeaux, D., Lea, D., & Faure, P. (1993). Object-oriented system development. Addison-Wesley.

DeLone, W.H., & McLean, E.R. (2003). The DeLone and McLean model of information systems success: A ten-year update. Journal of Management Information Systems, 19(4), 9–30. https://doi.org/10.1080/07421222.2003.11045748

Deming, W.E. (1986). Out of the crisis: quality, productivity, and competitive position. MIT Press.

Deming, W.E. (2000a). Out of the crisis [Kindle edition]. MIT Press.

Deming, W.E. (2000b). The new economics for industry, government, education (2nd ed.). MIT Press.

Denwattana, N., Saengsai, A., & Charoenchaimonkon, E. (2019). AppDOSI: An application for analyzing and monitoring the personal software process. In Proceedings of the 4th International Conference on Information Technology (InCIT) (pp. 280–283). IEEE. https://doi.org/10.1109/INCIT.2019.8911993

Dweck. C. (2000). Self-Theories: Their role in motivation, personality, and development [Kindle edition]. Taylor & Francis.

Early, J.F., & Coletti, O.J. (1999). The quality planning process. In J.M. Juran & A.B. Godfrey (Eds.), Juran's Quality Handbook (5th ed., pp. 3.1–3.50). McGraw-Hill.

Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. International Journal of Human-Computer Studies, 41(4), 457–480. https://doi.org/10.1006/ijhc.1994.1069

Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., & Zander, C. (2006a, March). Can graduating students design software systems?. ACM SIGCSE Bulletin, 38(1), 403–407). https://doi.org/10.1145/1124706.1121468

Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., & Zander, C. (2006b). Categorizing student software designs: Methods, results, and implications. Computer Science Education, 16(3), 197–209. https://doi.org/10.1080/08993400600912376

Elshennawy, A.K., Maytubby, V.J., & Aly, N.A. (1991). Concepts and attributes of total quality management. Total Quality Management, 2(1), 75–98. https://doi.org/10.1080/09544129100000008

Fagan, M.E. (1976). Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3), 182–211. https://doi.org/10.1147/sj.153.0182

Feigenbaum, A. (1983). Total Quality Control (3rd ed.). McGraw-Hill.

Feiner, J., & Kranjc, E. (2009, September 23–25). Copy & paste education: solving programming problems with web code snippets. In Proceedings of the Interactive Computer Aided Learning conference (ICL 2009) (pp. 81–88).

Fishbein, M., & Ajzen, I. (1975). Belief, attitude, intention and behavior: an introduction to theory and research. Addison-Wesley.

Gitlow, H.S., & Gitlow, S.J. (1987). The Deming guide to quality and competitive position. Prentice-Hall.

Gómez-Alvarez, M.C., Gasca-Hurtado, G.P., Manrique-Losada, B., & Arias, D.M. (2016, June). Method of pedagogic instruments design for software engineering. In Proceedings of the 11th Iberian Conference on Information Systems and Technologies (CISTI) (pp. 1–6). IEEE. https://doi.org/10.1109/CISTI.2016.7521377

Grove, R.F. (1998, August). Using the personal software process to motivate programming practices. In Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education (ITiCSE'98) (pp. 98–101). Association for Computing Machinery. https://doi.org/10.1145/282991.283046

Hilburn, T.B., & Townhidnejad, M. (2000, March). Software quality: a curriculum postscript?. In Proceedings of the 31st SIGCSE technical symposium on Computer Science Education (SIGCSE'00) (pp. 67–171). Association for Computing Machinery. https://doi.org/10.1145/330908.331848

Hou, L., & Tomayko, J. (1998, March). Applying the personal software process in CS1: An experiment. In Proceedings of the 29th SIGCSE technical symposium on Computer Science Education (SIGCSE'98) (pp. 322–325). Association for Computing Machinery. https://doi.org/10.1145/273133.274322

Hu, C. (2016, February) Can students design software? The answer is more complex than you think. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16) (pp. 199–204).

Association for Computing Machinery. https://doi.org/10.1145/2839509.2844563

Humphrey, W.S. (1988, March) Characterizing the software process: a maturity framework. IEEE Software, 5(2), 73–79. https://doi.org/10.1109/52.2014

Humphrey, W.S. (1994). Process feedback and learning. In Proceedings of the 9th International Software Process Workshop (pp. 104–106). IEEE. https://doi.org/10.1109/ISPW.1994.512776

Humphrey, W.S. (1995, September). Making process improvement personal. IEEE Software, 12(5), 82–83. https://doi.org/10.1109/52.406762

Humphrey, W.S. (1999). Why don't they practice what we preach? The personal software process (PSP). Annals of Software Engineering, 6, 201–222. https://doi.org/10.1023/A:1018997029222

Humphrey, W.S. (2000). The personal software process. Carnegie Mellon University.

Humphrey, W.S. (2005). PSP: A self-improvement process for software engineers. Pearson Education Inc.

ICSE. (2011, May). Welcoming message of ICSE 2011. The 33rd International Conference on Software Engineering. Honolulu, Hawaii. Retrieved December 22, 2018, from http://2011.icseconferences.org/content/welcome

Iivari, J. (1996, October). Why are CASE tools not used? Communications of the ACM, 39(10), 94–103. https://doi.org/10.1145/236156.236183

International Organization for Standardization (ISO). (2015). ISO 9001: International standards for quality management. International Organization for Standardization.

Ishikawa, K. (1989). Introduction to Quality Control. JUSE Press.

Ishikawa, K., & Lu, D. (1985). What is total quality control? Prentice-Hall.

Jancikova, A., & Brycht, K. (2009). TQM and organizational culture as significant factors in ensuring competitive advantage: a theoretical perspective. Economics & Sociology, 2(1), 80–95. https://doi.org/10.14254/2071-789X.2009/2-1/8

Jenkins, G.L., & Ademoye, O. (2012). Can individual code reviews improve solo programming on an introductory course? Innovations in Teaching and Learning in Information and Computer Sciences (ITALICS), 11(1), 71–79. https://doi.org/10.11120/ital.2012.11010071

Juran, J.M. (1992). Juran on quality by design: the new steps for planning quality in goods and services. The Free Press.

Juran, J.M. (1999). How to think about quality. In J.M. Juran & A.B. Godfrey (Eds.), Juran’s Quality Handbook (5th ed., pp. 2.1–2.18). McGraw-Hill.

Juran, J.M., & Godfrey, A.B. (1999). The quality control process. In J.M. Juran & A.B. Godfrey (Eds.), Juran’s Quality Handbook (5th ed., pp. 4.1–4.29). McGraw-Hill.

Kan, S.H. (2003). Metrics and models in software quality engineering (2nd ed.). Addison-Wesley Longman Publishing Co., Inc.

Kan, S.H., Basili, V.R., & Shapiro, L.N. (1994). Software quality: An overview from the perspective of quality management. IBM Systems Journal, 33(1), 4–19. https://doi.org/10.1147/sj.331.0004

Krüger, V. (2001). Main schools of TQM: “the big five”. The TQM magazine, 13(3), 146–155. https://doi.org/10.1108/09544780110366042

Kuhrmann, M., Diebold, P., & Münch, J. (2016). Software process improvement: a systematic mapping study on the state of the art. PeerJ Computer Science, 2, e62. https://doi.org/10.7717/peerj-cs.62

Lakanen, A., Lappalainen, V., & Isomöttönen, V. (2015, November). Revisiting rainfall to explore exam questions and performance on CS1. In Proceedings of the 15th Koli Calling Conference on Computing Education Research (pp. 40–49). ACM. https://doi.org/10.1145/2828959.2828970

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K. Seppälä, O., Simon, B., & Thomas, L. (2004, December). A multi-national study of reading and tracing skills in novice programmers. ACM SIGCE Bulletin, 36(4), 119–150). https://doi.org/10.1145/1041624.1041673

Loftus, C., Thomas, L., & Zander, C. (2011, March). Can graduating students design: revisited. In Proceedings of the 42nd ACM technical symposium on Computer Science Education (SIGCSE'11) (pp. 105–110). Association for Computing Machinery. https://doi.org/10.1145/1953163.1953199

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In Working group reports from ITiCSE on Innovation and technology in computer science education (ITiCSE-WGR'01) (pp. 125–180). Association for Computing Machinery. https://doi.org/10.1145/572133.572137

McMillan, J.H., & Schumacher, S. (2006). Research in education: evidence-based inquiry (6th ed.). Pearson Education Inc.

Moen, R. (2009, September). Foundation and history of the PDSA cycle. In Proceedings of the Asian network for quality conference. Tokyo. Retrieved October 27, 2018, from https://deming.org/uploads/paper/PDSA_History_Ron_Moen.pdf

Moore, G.C., & Benbasat, I. (1991). Development of an instrument to measure the perceptions of adopting an information technology innovation. Information Systems Research, 2(3), 192–222. https://doi.org/10.1287/isre.2.3.192

Naur, P., & Randell, B. (Eds.). (1969). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968. NATO, Scientific Affairs Division.

Neyestani, B (2017, February) Principles and contributions of total quality management (TQM) gurus on business quality improvement. SSRN. https://doi.org/10.2139/ssrn.2950981

O'Dell, D.H. (2017, February). The debugging mindset. Queue, 15(1), 71–90. https://doi.org/10.1145/3055301.3068754

Pailthorpe, B.C. (2017). Emergent design. The International Encyclopaedia of Communication Research Methods, 1–2. https://doi.org/10.1002/9781118901731.iecrm0081

Pando, B., & Ojeda, T. (2019). PSP-CI: A tool for collecting developer's data with continuous integration. In Á. Rocha, C. Ferrás & M. Paredes (Eds.), Information Technology and Systems, ICITS 2019, Advances in Intelligent Systems and Computing (Vol 918, pp. 103–112). Springer. https://doi.org/10.1007/978-3-030-11890-7_11

Patton, M.Q. (2015). Qualitative research & evaluation methods: integrating theory and practice (4th ed.). Sage Publications.

Paulk, M.C., Weber, C.V., Garcia, S.M., Chrissis, M.B., & Bush, M. (1993). Key practices of the capability maturity model, Version 1.1 (CMU/SEI-93-TR-025). Software Engineering Institute, Carnegie Mellon University. https://resources.sei.cmu.edu/asset_files/TechnicalReport/1993_005_001_16214.pdf

Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

Plowright, D. (2011). Using Mixed Methods: Frameworks for an Integrated Methodology [Kindle edition]. SAGE Publications.

Prechelt, L (2001). Accelerating learning from experience: avoiding defects faster. IEEE Software, 18(6), 56–61. https://doi.org/10.1109/52.965803

Prechelt, L., & Unger, B. (2001). An experiment measuring the effects of personal software process (PSP) training. IEEE Transactions on Software Engineering, 27(5), 465–472. https://doi.org/10.1109/32.922716

Pressman, R.S. (2005). Software engineering: a practitioner's approach (6th ed.). McGraw-Hill.

Quality (n.d.). In Cambridge English Dictionary online. Retrieved October 14, 2019, from https://dictionary.cambridge.org/dictionary/english/quality

Raza, M., Faria, J.P., & Salazar, R. (2019). Assisting software engineering students in analyzing their performance in software development. Software Quality Journal, 27(3), 1209–1237. https://doi.org/10.1007/s11219-018-9433-7

Raza, M., Faria, J.P., Amaro, L., & Henriques, P.C. (2017, July). WebProcessPAIR: recommendation system for software process improvement. In Proceedings of the 2017 International Conference on Software and System Process (ICSSP 2017) (pp. 139–140). Association for Computing Machinery. https://doi.org/10.1145/3084100.3084365

Riemenschneider, C.K., Hardgrave, B.C., & Davis, F.D. (2002). Explaining software developer acceptance of methodologies: a comparison of five theoretical models. IEEE Transactions on Software Engineering, 28(12), 1135–1145. https://doi.org/10.1109/TSE.2002.1158287

Rogers, E.M. (1983) Diffusion of innovations (3rd ed.). The Free Press.

Rong, G., Li, J., Xie, M., & Zheng, T. (2012, April). The effect of checklist in code review for inexperienced students: an empirical study. In Proceedings of the 25th Conference on Software Engineering Education and Training (pp. 120–124). IEEE. https://doi.org/10.1109/CSEET.2012.22

Rong, G., Zhang, H., Liu, B., Shan, Q., & Shao, D. (2018, February). A replicated experiment for evaluating the effectiveness of pairing practice in PSP education. Journal of Systems and Software, 136, 139–152. https://doi.org/10.1016/j.jss.2017.08.011

Rong, G., Zhang, H., Qi, S., & Shao, D. (2016, May). Can software engineering students program defect-free? An educational approach. In Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C) (pp. 364–373). IEEE. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7883322&isnumber=7883261

Runeson, P. (2001, February). Experiences from teaching PSP for freshmen. In Proceedings of the 14th Conference on Software Engineering Education and Training, In search of a software engineering profession (pp. 98–107). IEEE. https://doi.org/10.1109/CSEE.2001.913826

Saiedian, H., & Carr, N. (1997, July). Characterizing a software process maturity model for small organizations. ACM SIGICE Bulletin, 23(1), 2–11. https://doi.org/10.1145/1031167.1031168

Saunders, M., Lewis, P., & Thornhill, A. (2016). Research methods for business students (7th ed.). Pearson Education Limited.

Schach, S.R. (2011). Object-oriented and classical software engineering (8th ed.). McGraw-Hill.

Simon. (2013, March). Soloway's rainfall problem has become harder. In Proceedings of 2013 Learning and Teaching in Computing and Engineering (pp. 130–135). IEEE. https://doi.org/10.1109/LaTiCE.2013.44

Soloway, E., Ehrlich, K., Bonar, J.G., & Greenspan, J. (1982). What do novices know about programming? In A. Badre & B. Shneiderman (Eds.), Directions in Human–Computer Interactions (Vol. 6, pp. 27–54). Ablex Publishing.

Sommerville, I. (2004). Software Engineering (7th ed.). Pearson Education Limited.

Stevens, T. (1994, July 4). Dr. Armand Feigenbaum on the cost of quality and the hidden factory. Industry Week. https://www.industryweek.com/quality/dr-armand-feigenbaum-cost-quality-and-hidden-factory

Thomas, L., Eckerdal, A., McCartney, R., Moström, J.E., Sanders, K., & Zander, C. (2014, July). Graduating students' designs: through a phenomenographic lens. In Proceedings of the 10th annual conference on International Computing Education Research (ICER'14) (pp. 91–98). Association for Computing Machinery. https://doi.org/10.1145/2632320.2632353

Thompson, R.L., Higgins, C.A., & Howell, J.M. (1991, March). Personal computing: toward a conceptual model of utilization. MIS Quarterly, 15(1), 125–143. https://doi.org/10.2307/249443

Towhidnejad, M., & Salimi, A. (1996). Incorporating a disciplined software development process into introductory computer science programming courses: initial results. In Technology-Based Re-Engineering Engineering Education Proceedings of Frontiers in Education 26th Annual Conference (FIE'96) (Vol. 2, pp. 497–500). IEEE. https://doi.org/10.1109/FIE.1996.572893

Utting, I., Tew, A.E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersone, M., Kolikant, Y.B., Sorva, J., & Wilusz, T. (2013, June). A fresh look at novice programmers' performance and their teachers' expectations. In Proceedings of the ITiCSE working group reports conference on Innovation and Technology in Computer Science Education – working group reports

(ITiCSE – WGR'13) (pp. 15–32). ACM. https://doi.org/10.1145/2543882.2543884

Venkatesh, V., & Davis, F.D. (2000). A theoretical extension of the technology acceptance model: four longitudinal field studies. Management Science, 46(2), 186–204. https://doi.org/10.1287/mnsc.46.2.186.11926

Venkatesh, V., Thong, J.Y., & Xu, X. (2012, March). Consumer acceptance and use of information technology: extending the unified theory of acceptance and use of technology. MIS Quarterly, 36(1), 157–178. https://doi.org/10.2307/41410412

Williams, L.A. (1997, November). Adjusting the instruction of the personal software process to improve student participation. In Proceedings of the Frontiers in Education 27th Annual Conference, Teaching and Learning in an Era of Change (FIE'97) (Vol. 1, pp. 154–156). IEEE. https://doi.org/10.1109/FIE.1997.644830

Appendix A : Student Questionnaire (Phase 1)

Software Development Process Questionnaire (Phase 1)

Dear Student

Thank you for giving your attention to this questionnaire. The approximate time needed to complete this questionnaire is 15 - 20 minutes. The purpose of these questions is mainly to find out which software development processes you make use of for the programming assignments of the subject that is specified at the top of this page. By completing this questionnaire you give the researcher consent to use your information for research purposes only. Responses will be confidential and your privacy will be protected to the maximum extent allowable by law. Participation is voluntary. Completing or failing to complete this questionnaire has absolutely no bearing on your marks for this subject.

Please mark your selected answers with an **X**.

Section 1 - Demographic Information

Which Software Development course are you currently registered for? ______________

Section 2 – Current Software Development Process

1. Make use of percentages to indicate how much time you normally spend in each of the following software development phases. Please make sure that the percentages add up to 100%.

Planning:	________________
Design:	________________
Coding:	________________
Testing/Debugging:	________________
	100%

Note: If you normally do not spend any time in one or more of these phases, you can write 0% next to that phase.

2. Which of the following defect removal strategies do you normally use? (Check all that apply.)

☐ Design review ☐ Code review ☐ Testing/Debugging

3. Which Software Life Cycle do you normally use?

____

4. Which **one** of these do you think is the most efficient way to remove errors from a program? (Only select one.)

☐ Design review ☐ Code review ☐ Testing/Debugging

5. Which of the following do you normally use to model your designs? (Check all that apply)

☐ I never do any designs **OR**

☐ Class diagrams
☐ State diagrams
☐ Flowcharts and/or Pseudo code
☐ Sequence and/or Activity diagrams
☐ Use cases

Other (please specify): ____________________

6. What is the average mark that you normally get for your programming assignments in this subject?

☐ 0-9 ☐ 10-19 ☐ 20-29 ☐ 30-39 ☐ 40-49 ☐ 50-59

☐ 60-69 ☐ 70-79 ☐ 80-89 ☐ 90-99 ☐ 100

7. If you don't get 100% for all your programming assignments, please indicate the **main reason** why you think you don't get 100%? (Only select one.)

☐ Insufficient time ☐ Insufficient programming skills

☐ Unable to identify or locate all defects

8. Do you keep record of the most common mistakes you make in your programs?

☐ Yes ☐ No

9. Do you keep record of the actual time you spend on the different software development phases?

☐ Yes ☐ No

10. Have you ever used an estimation technique to determine how big a program is going to be and how much time it will take to complete the program?

☐ Yes ☐ No

11. If you answered *Yes* to **Question 10**, please specify which estimation technique(s) you use:

____

Thank you for kindly participating and completing this questionnaire.

Appendix B : Participant Information Sheet (Phase 2)

Attributes contributing to the effective use of quality appraisal techniques by undergraduate Computer Science students

Dear Student

Thank you for your willingness to consider participating in the “Quality appraisal techniques” research project. This study is being conducted in partial fulfilment of a Ph.D. degree for Guillaume Nel under the supervision of Prof Johannes Cronje.

Participation in this research project requires signed consent from participants. Before you sign the consent form, we want to provide you with the necessary background regarding this project so that you can make an informed decision as to whether you will be participating or not.

The Research Ethics Committee of the Faculty of Natural and Agricultural Science, University of the Free State has approved this research study [UFS-HSD2015/0115]. This information sheet and the attached consent form are only part of the process of informed consent. If you want more details about something mentioned here, or information not included here, you should feel free to ask. Please take the time to read this participant information sheet carefully and to understand any accompanying information.

What is the purpose of this study?

The purpose of this study is to explore the factors contributing to quality of computer programmes developed by undergraduate Computer Science students.

Why have I been invited to participate?

You have been invited to participate since you are an undergraduate Computer Science student at the South African Higher Education Institution that have been selected for this study. Please note that participation is voluntary and that there will be no consequences if you decided not to participate. Since this study does not form part of any academic module that you are registered for, there will also be no academic implications if you decide not to participate.

What will I be asked to do?

You will be asked to participate in a 5-hour experiment session that will be conducted in one of the institutional computer laboratories. During this session you will be asked to do the following:

- Attend a 1-hour tutorial session during which various performance measurement aspects will be discussed.
- Complete a pre-activity questionnaire. The purpose of this questionnaire will be to explore your current software development practices.
- Complete an individual programming exercise during which you have to capture performance data using the Process Dashboard© software.
- Complete a post-activity questionnaire. The purpose of this questionnaire will be to explore your perceptions on the capturing and interpreting of process measurement data.
- Be interviewed by the researcher. During this interview you will be asked to give narrative feedback regarding the development processes you used during the individual programming exercise. With your permission, the researcher will audiotape the interview solely for the purposes of accurately transcribing the conversions. You will have an opportunity to review and correct the transcript.

Are there any possible benefits and or risks from participation in this study?

There will be no direct benefits to participants. However, your participation might give you valuable insight into the processes that you currently use to solve programming problems. There are no known risks to participating in this study.

What if I change my mind during or after the study?

Should you decide at any time during the experiment that you no longer wish to participate, you may withdraw your consent without providing an explanation. Any anonymous data collected up to the point of withdrawal will be retained for use in the study.

What happens to the information I provide?

Tapes and transcription data will be stored in a locked cabinet and only the researcher will have the key. Any local electronic data will be stored on secured computers where only the researcher and supervisors can gain access to the data. All physical records of identifying information will be destroyed one year after publication of the study results. Electronic records with identifying information will be destroyed one year after the publication of the study results.

How will the results of the study be published?

Written findings will be published online or in print journals; and both written and video reporting may be presented at local, provincial, national or international academic conferences for the purpose of furthering an understanding of quality appraisal techniques used by undergraduate Computer Science students. Your identity will remain anonymous in all written presentations of data via pseudonyms and the reporting of aggregated results.

What if I have questions about this study?

Please feel free to contact the researcher or supervisors if you require further information about the study. If you have concerns or complaints about the conduct of this study, please contact the Research Ethics Coordinator of the Department of Computer Science & Informatics, University of the Free State.

Contact details

Researcher: guilnel@cut.ac.za

Supervisors: Johannes.cronje@gmail.com

Departmental Research Ethics Coordinator: pieterb@ufs.ac.za

How do I give my consent to participate?

Complete the attached consent form if you understand and agree to take part in this study. Please submit the completed consent form to your lecturer. You may keep this information sheet for your own records.

Appendix C : Pre-Activity Questionnaire (Phase 2)

Software Development Process Pre-Questionnaire

The purpose of this questionnaire is to determine which software development practices you currently make use of for you programming assignments. The approximate time needed to complete this questionnaire is 5 tot 10 minutes.

Please mark your selected answers with an **X**.

Section 1 – Current Software Development Process

1. Which Software Life Cycle do you normally use? (Check all that apply.)

☐ Code-and-fix

☐ Waterfall

☐ Iterative

☐ Prototyping

☐ Agile

2. Make use of percentages to indicate how much time you normally spend in each of the following software development phases. Please make sure that the percentages add up to 100%.

Planning:	________________
Design:	________________
Design review:	________________
Coding:	________________
Code review:	________________
Testing/Debugging:	________________
	Total = 100%

Note: If you normally do not spend any time in one or more of these phases, you can write 0% next to that phase.

3. Which of the following defect removal strategies do you normally use? (Check all that apply.)

☐ Design review

☐ Code review

☐ Testing/Debugging

4. If you selected either “Design review” or “Code review” in Question 3, which of the following checklists do you normally use? (Check all that apply.)

☐ Design review checklists

☐ Code review checklists

☐ None

5. If you make use of checklists, how do you compile your checklist items? (Check all that apply.)

☐ Compile checklist based on previous errors that I have made.

☐ Use an existing checklist (compiled by someone else).

6. Do you keep record of the most common mistakes you make in your programs?

☐ Yes ☐ No

7. Do you keep record of the actual time you spend on the different software development phases?

☐ Yes ☐ No

8. Which **one** of the following do you think is the most efficient way to remove errors from a program? (Only select one.)

☐ Design reviews ☐ Code reviews ☐ Testing/Debugging

9. Which of the following do you normally use to model your designs? (Check all that apply)

☐ I never do any designs **OR**

☐ Class diagrams

☐ Flowcharts and/or Pseudo code

☐ Use cases and/or Interaction diagrams

☐ State charts/diagrams

Other (please specify): _____________

10. What is the average mark that you normally get for your programming assignments in this subject?

☐ 0-9	☐ 10-19	☐ 20-29	☐ 30-39	☐ 40-49	☐ 50-59
☐ 60-69	☐ 70-79	☐ 80-89	☐ 90-99	☐ 100	

11. If you don't get 100% for all your programming assignments, please indicate the **main reason** why you think you don't get 100%? (Only select one.)

☐ Insufficient time ☐ Insufficient programming skills

☐ Unable to identify or locate all defects

Section 2 - Demographic Information

Student number: ______________________________

Initials & surname: ______________________________

Gender: ☐ Male ☐ Female

Age: ☐ 18 ☐ 19 ☐ 20 ☐ 21 ☐ 22 ☐ 23+

Thank you for kindly participating and completing this questionnaire.

Appendix D : Performance Measurement Tutorial (Phase 2)

Performance Measurement Class Tutorial

Overview

This class tutorial describes a simulation exercise on how to capture process measurement data.

Purpose

The purpose of this tutorial is:

1. To get an overview of how to use the Process Dashboard© software to capture process measurement data.
2. To be able to capture the following process data through Process Dashboard©:
   - **a.** Time spend in development phases.
   - **b.** Defects injected and removed in specific phases.
   - **c.** Time spend to remove defects.
   - **d.** Size of the product.
3. To be able to interpret data collected through Process Dashboard©.

Table of Contents

Exercise Requirements

Write a program to calculate the **mean** and **standard deviation** of a set of *n* real numbers.

Your program can read the *n* real numbers from the keyboard, a file, or some other source.

Use a **list** to store the *n* numbers for the calculations. If necessary, a variable or static array(s) or other data structure(s) may be used to hold the data.

Thoroughly test the program. You need to conduct at least two tests using the data values provided in Table 1. Expected results for each of the provided test data sets are indicated in Table 2.

Table 1: Test data

Test Data Set 1	**Test Data Set 2**
160	15.0
591	69.9
114	6.5
229	22.4
230	28.4
270	65.9
128	19.4
1657	198.7
624	38.8
1503	138.2

Table 2: Expected results

Test	**Expected Value**	
	Mean	*Std. Dev*
Test Data Set 1	550.6	572.03
Test Data Set 2	60.32	62.26

Mean and Standard Deviation

Overview

The mean is the average of a set of data. The average is the most common measure of location for a set of numbers. The average locates the centre of the data.

Standard deviation is a measure of the spread or dispersion of a set of data. The more widely the values are spread out, the larger the standard deviation. For example, say we have two separate lists of exam results from a class of 30 students; one ranges from 31% to 98%, the other from 82% to 93%. The standard deviation would be larger for the results of the first exam.

Calculating mean and standard deviation

The formula for calculating the mean is

$$x_{avg} = \frac{\sum_{i=1}^{n} x_i}{n}$$

The formula for standard deviation, σ (sigma), is

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} \left(x_i - x_{avg}\right)^2}{n-1}}$$

where

- Σ is the symbol for summation
- *i* is an index to the *n* numbers
- *x* is the data in the set
- *n* is the number of items in the set

Assignment instructions

Use the *d*efect type standards listed in Table *3* to determine in which category a defect should be logged.

Table 3: Defect Type Standards

#	Category	Description	Sub-description
10	Documentation	Comments	
		Manuals	
20	Syntax	Spelling	
		Punctuation	
		Typos	
		Instruction formats	
		Begin-End Brackets	
		Spelling	
30	Packaging	Library	
		Version Control	
40	Assignment	Naming	Declaration
			Duplicate names
		Scope	
		Initialization	Variables and objects
		Range	Variable limits
			Array range
50	Interface	Internal	Incorrect module interaction
			Incorrect module-external data structure
			Incorrect input parameters
		Input/Output	File
			Display
			Printer
			Communication
		User Interface	Formats
			Content
			Large response time
			Lack of naturalness
			Inconsistency

#	Category	Description	Sub-description
			Redundancy
			Complexity
			Lack of flexibility
			Non-responsiveness
			Unpredictable flows
			Visual stimulation
60	Checking	Error Messages	
		Inadequate checks	
70	Data	Structure	
		Content	
80	Function	Loops	Incorrect Initial value
			Incorrect terminal value
			Incorrect control value processing
		Computation	Incorrect Equation
			Wrong manipulation
		Algorithmic	Incorrect/missing processing
			Unnecessary processing
			Duplicate Logic
			Unachievable path
			Illogical conditions or Impossible Cases
90	System	Timing	
		Memory	
100	Environment	Design, Compile, Test support systems	

Before starting the assignment, review the process script in Table 4 to ensure that you understand the "big picture" before you begin. Also, ensure that you have all of the required inputs before you begin.

Table 4: Process Script

Purpose:	To guide the development of small programs
Entry Criteria:	- Requirements statement - Defect type standard (see Table 3) - Process Dashboard

Step	Activities	Description
1	Planning	- Read through the requirements statement. - Record time in the Time Recording log.
2	Design (optional)	- Review the requirements and produce a design to meet them. - Record in the Defect Recording log any requirements defects found. - Record time in the Time Recording log.
3	Design Review (optional)	- Review the design. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
4	Code	- Implement the design. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
5	Code Review (optional)	- Review the code. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
6	Compile (optional)	- Compile the program until error-free. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
7	Test	- Test until all tests run without error. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
8	Post-mortem	- Determine the total program size. - Enter this data in the Project Plan Summary form.

Exit Criteria	- A thoroughly tested program. - Completed Time and Defect Recording logs.

Submission Instructions

Submit your assignment package to the instructor.

The assignment package should consist of:

- Process Dashboard© backup file.
- Program source code.

Important Notes

- Remember, you must complete this program before the end of today's tutorial session.
- Keep your program simple.
- You must submit a 100% working program.
- If you are not sure about something, ask your instructor for clarification.

Appendix E : Programming Assignment (Phase 2)

Programming Assignment 1

Overview

Program 1 should simulate the "**Quick Pick Option**" during which a Terminal generates a certain number of lotto game rows for a player.

Purpose

The purpose of this assignment is:

2. To use the Process Dashboard© software to capture process measurement data while doing a programming assignment.
3. To be able to capture the following process data through Process Dashboard©:
    a. Time spend in development phases.
    b. Defects injected and removed in specific phases.
    c. Time spend to remove defects.
    d. Size of the product.
4. To be able to interpret data collected through Process Dashboard©.

Table of Contents

Prerequisites

You will need the following resources in order to complete this assignment:

- Performance Measurement Class tutorial.
- Assignment Domain Background Information Document.

Program 1 Requirements

Write a program that generate any number of randomly selected lottery row numbers and store it in a text file. Your program needs to get the number of rows that should be generated as input through the keyboard from the user. You may use any data structure(s) to store the numbers in memory.

You must make use of a text file in which the numbers are stored permanently. The lottery row numbers should be stored in separate lines. Each lottery row contains six unique numbers ranging from 1 to 49 and needs to be separated by spaces. The numbers in each row needs to be in ascending order before storing it in the text file.

Program 1 Example Data

If the user decides to generate **10** lottery rows, the following data needs to be created in the text file:

```
9 12 23 24 32 42
14 17 20 28 40 45
5 15 17 28 34 49
2 13 23 32 38 42
2 5 18 29 35 48
8 15 33 37 43 44
5 9 13 21 43 47
3 26 29 31 36 40
3 4 12 31 40 49
18 20 23 26 31 36
```

If the user decides to generate **7** lottery rows on another execution of the program, the following data needs to be created in the text file:

```
16 20 23 28 43 46
2 6 7 8 11 18
1 2 23 26 28 45
9 19 24 27 37 39
9 17 22 35 46 48
15 17 32 33 35 49
12 31 33 34 35 42
```

Take note that your program should generate different randomly selected numbers. The above examples are only to illustrate the layout of the text file that is created.

Assignment Instructions

Before starting the assignment, review the process script in Table 1 to ensure that you understand the "big picture" before you begin. Also, ensure that you have all of the required inputs before you begin.

Submission Instructions

Submit your assignment package, consisting of the following, to the instructor.

- Process Dashboard© backup file.
- Program source code.
- Any other documentation (designs) created during the development.

Important Notes

- You only have three hours to complete this program.
- Keep your program simple.
- You must submit a 100% working program.
- If you are not sure about something, ask your instructor for clarification.

Table 1: Process Script

Purpose:	To guide the development of small programs
Entry Criteria:	- Requirements statement - Defect type standard - Process Dashboard

Step	Activities	Description
1	Planning	- Read through the requirements statement. - Record time in the Time Recording log.
2	Design (optional)	- Review the requirements and produce a design to meet them. - Record in the Defect Recording log any requirements defects found. - Record time in the Time Recording log.
3	Design Review (optional)	- Review the design. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
4	Code	- Implement the design. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
5	Code Review (optional)	- Review the code. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
6	Compile (optional)	- Compile the program until error-free. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
7	Test	- Test until all tests run without error. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
8	Post-mortem	- Determine the total program size. - Enter this data in the Project Plan Summary form. - Analyse your data. - Complete your post-activity questionnaire

Exit Criteria	- A thoroughly tested program. - Completed Time and Defect Recording logs.

Appendix F : Programming Assignment Background (Phase 2)

Assignment Domain Background Information

Table of Contents

Overview

All the programming assignments that you are going to do during this experiment will be based on one problem domain – “The National Lottery” also referred to as “LOTTO”. All background information, activities and aspects involved in the “LOTTO” domain will be covered in this document. This document will therefore be an integral part of all programming assignments.

What is LOTTO?

LOTTO is a game of chance where the player should select six (6) numbers from a field of 49 numbers (hence the term LOTTO 6/49). If these six (6) numbers match the six drawn numbers of the LOTTO draw, the player wins the first category prize.

How to play LOTTO?

1. Each LOTTO Game Row costs R3.50 inclusive of VAT.

2. On any one LOTTO Entry Coupon a Player may play a minimum of one Game Row.

3. A Player must manually mark in black or blue pen or pencil six (6) numbers from the numbers 1 to 49 inclusive on each of one or more Game Rows.

4. The completed LOTTO Entry Coupon with the appropriate amount due shall then be submitted to the Retailer who will process the completed LOTTO Entry Coupon through the Terminal. A Ticket recording each selection marked on the coupon will be issued to the Player.

5. The Retailer shall process LOTTO Entry Coupons through the Terminal, and Tickets will be issued only through the Terminal.

6. If the Terminal rejects the LOTTO Entry Coupon, the entry is not valid.

Quick Pick Option

A Ticket that contains numbers randomly generated by the Terminal may also be purchased. This is called a QuickPick Ticket. The Player must request such a Ticket from the Retailer and decide the number of Game Rows the Terminal needs to generate. The number of rows generated determines the price of the ticket. Each row will cost R3.50.

Example of QuickPick Ticket with 7 rows:

```
16 20 23 28 43 46
2  6  7  8  11 18
1  2  23 26 28 45
9  19 24 27 37 39
9  17 22 35 46 48
15 17 32 33 35 49
12 31 33 34 35 42
```

LOTTO Draws

Players can purchase LOTTO tickets any day. A public draw, that determines the winning numbers for a game, is conducted twice per week every Wednesday and Saturday.

In LOTTO draws seven (7) numbers are drawn at random from a Drawing Machine containing forty-nine (49) balls numbered 1 to 49.

The first six (6) numbers drawn are the Main Numbers and the seventh number drawn is the Bonus Number.

The Draw is broadcasted live on TV. The LOTTO Game sales are closed approximately 30 minutes before the Draw on Wednesday and Saturday evenings. During this time all Game Rows are collected from the Terminals to determine the Total Game row sales. According to these figures ,the total sales and total price money can be given during the live shows.

With every draw an updated script, which includes the number of times each number has been drawn is given to the LOTTO presenters. The updated script is written on the assumption that each and every number will be drawn during that specific draw, so that the presenter can read the relevant number and times that it has been drawn with ease. Refer to Table 1 for examples of previous LOTTO draws.

Table 1: LOTTO draw examples

Draw Date	**Draw Number**	**Main Numbers**	**Bonus Number**
2017/04/26	1	14 19 25 30 44 45	19
2017/04/29	2	15 20 26 29 40 46	28
2017/05/03	3	5 15 18 24 43 44	17

LOTTO Prizes

Winning Categories and Distribution of Price Money

The winning amount distributed to the players (called the price pool) is 45% of the total sales, which is determined from each game row that is sold for R 3.50. Refer to Table 2 for the winning categories of the LOTTO and the distribution of the prize money.

Table 1: LOTTO prize money distribution

Winning Categories	Matching Numbers	% Distribution of winnings
Division 1	6 Correct Numbers	18.25%
Division 2	5 Correct Numbers + Bonus Number	4.00%
Division 3	5 Correct Numbers	9.00%
Division 4	4 Correct Numbers + Bonus Number	5.00%
Division 5	4 Correct Numbers	16.75%
Division 6	3 Correct Numbers + Bonus Number	11.00%
Division 7	3 Correct Numbers	36.00%

Moments after each draw, the number of row matches in each category is determined and the exact pay-out amount for each of these matches. All that a player needs to do is match his/her selected numbers with the drawn numbers and claim his/her payouts at the nearest Ticket Centre. The Terminal at the Ticket Centre can also do this kind of matching.

Example of Price Divisions for Draw 2 with 10 million game rows:

Draw Date:	2017/04/30
Draw Number:	2
Main Numbers:	15 20 26 29 40 46
Bonus Number:	28
Total Sales:	R 35 000 000.00
Total Price Pool:	R 15 750 000.00

The results (per division) are indicated in Table 2.

Table 2: Results for Draw 2

Division Results			
Winning Categories	**Matching Numbers**	**Number of row winners**	**Price money per winning row**
Division 1	6 Correct Numbers	2	R 1 437 187.50
Division 2	5 Correct Numbers + Bonus Number	2	R 315 000.00
Division 3	5 Correct Numbers	177	R 8 008.47
Division 4	4 Correct Numbers + Bonus Number	435	R 1 810.34
Division 5	4 Correct Numbers	9255	R 285.05
Division 6	3 Correct Numbers + Bonus Number	12362	R 140.15
Division 7	3 Correct Numbers	164386	R 34.49

What are the odds of winning a LOTTO prize?

There are 13,983,816 different combinations that one can play when choosing six out of 49 numbers. When calculating the permutations, many people make the assumption that the numbers selected must match the drawn numbers in the exact order drawn. If this was the case, the odds of matching six numbers would be astronomically more than 13,9 million. The odds of getting 5 numbers plus bonus are 1 in 2.3 million - nearly 6 times easier than getting 6 numbers. To get 5 numbers is 1 in 55 thousand - 40 times easier than getting 5 + bonus, etc. Refer to Table 3 for the odds of winning a prize.

Table 3: LOTTO winning odds

Winning Categories	Matching Numbers	Odds
Division 1	6 Correct Numbers	1 in 14 million
Division 2	5 Correct Numbers + Bonus Number	1 in 2.3 million
Division 3	5 Correct Numbers	1 in 55 thousand
Division 4	4 Correct Numbers + Bonus Number	1 in 22 thousand
Division 5	4 Correct Numbers	1 in 1 thousand
Division 6	3 Correct Numbers + Bonus Number	1 in 800
Division 7	3 Correct Numbers	1 in 61

Appendix G : Post-Activity Questionnaire (Phase 2)

Post-Activity Questionnaire

The purpose of this questionnaire is to explore your perceptions on the use of measurements during your development process for Programming Assignment 1.

1. On a scale of 1 - 4, indicate the ease with which you were able to perform the following actions in Process Dashboard. (Mark your selected answers with an **X**.)

	Very difficult	**Difficult**	**Easy**	**Very easy**
a. Record time.	1	2	3	4
b. Record time in the correct phase.	1	2	3	4
c. Identify the type of defect.	1	2	3	4
d. Describe a defect.	1	2	3	4

2. Which challenges/problems did you experience with the **time measurement**?

3. Which challenges/problems did you experience with the **recording of defects**?

4. Review your “Time in Phase” data in Process Dashboard.

4.1 If you logged any time during the “Design Review” and/or “Code Review” phases, write down a complete description of what you did during these phases.

4.2 What percentage of your time did you spend in the “Test” phase? ______________

4.3 What actions would you take in the next assignment to reduce your “Test” time?

____
____
____
____
____
____

5. Review your “Defects Removed” data in Process Dashboard.

5.1 In which phase did you remove the most defects? ____________________

5.2 What would you do differently in the next assignment to ensure that you remove defects earlier in the life cycle?

____
____
____
____
____
____

6. What value do you see in process measurement data?

____
____
____
____

7. Which changes would you make to your development process to improve your programming performance?

____
____
____
____
____
____

Thank you for kindly participating and completing this questionnaire!

Appendix H : Program Code (Participant 1)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace Quick_Pick_Option
{
    class Program
    {


        static void Main(string[] args)
        {

            int numberOfRows;

            //Get number of rows
            Console.Write("Please Enter Number of Rows: ");
            numberOfRows = Convert.ToInt32(Console.ReadLine().Trim());

            //Generate numbers
            int[,] quickPickNumbers = GenerateNumbers(numberOfRows);

            int rc = StoreNumbersToDataFile(quickPickNumbers);

            if(rc !=-1)
                Console.WriteLine("Numbers successfully saved");

            else
                Console.WriteLine("Numbers not successfully saved");

            Console.ReadKey();

        }//End Main

        static int[,] GenerateNumbers(int numberOfRows)
        {

            int number;

            //Numbers will be in rows and columns
            int[,] quickPickNumbers = new int[numberOfRows,6];

            //Create random number object
            Random random = new Random();

            //Generating
            for (int row = 0; row < quickPickNumbers.GetLongLength(0); ++row)
            {

                for (int col = 0; col < quickPickNumbers.GetLongLength(1); ++col)
                {

                    //Generate number
                    number = random.Next(49);

                    //Check if available
                    //Put a number in a certain column
                    quickPickNumbers[row, col] = number;

                }

            }

            return quickPickNumbers;

        }

        static int StoreNumbersToDataFile(int[,] quickPickNumbers)
        {

            int rc = 0;

            try
            {

                //Creating file
                FileStream file = new FileStream("Numbers.txt", FileMode.Create);
                StreamWriter strFile = new StreamWriter(file);

                long len = quickPickNumbers.GetLongLength(1);

                //Writing to file
                for (int row = 0; row < quickPickNumbers.GetLongLength(0); ++row)
                {

                    for (int col = 0; col < quickPickNumbers.GetLongLength(1); ++col)
                    {

                        strFile.Write(quickPickNumbers[row, col] + " ");

                    }

                    //Moving file pointer to the next line
                    strFile.WriteLine();

                }

                strFile.Close();
                file.Close();

            }
            catch (Exception)
            {

                rc = -1;

            }

            return rc;

        }//End StoreNumbersToDataFile

    }
}
```

Appendix I : Program Code (Participant 2)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace Lotto
{
    class Program
    {
        static void Main(string[] args)
        {
            int rows;
            int number;
            Random newNumber = new Random();

            Console.WriteLine("Please enter number of rows to generate");
            rows = Convert.ToInt32(Console.ReadLine());
            int[] quickpick = new int[6];
            for (int i = 0; i < rows; i++)
            {
                for (int x = 0; x < 6; ++x)
                {
                    number = newNumber.Next(1, 49);
                    if (!(CheckDuplicates(quickpick, number)))
                    {
                        quickpick[x] = number;
                    }
                    else
                    {
                        --x;
                    }

                }
                LottoNumbers(Sort(quickpick),rows);
            }

            Console.ReadKey();
        }
        public static int[] Sort(int[] array)
        {
            int temp;
            int[] SortedArray = new int[6];
            for (int x = 0; x < array.Length -1; ++x)
            {

                if (array[x] < array[x + 1])
                {
                    temp = array[x];
                    array[x] = array[x + 1];
                    array[x + 1] = temp;
                }
            }
            return array;
        }
        public static bool CheckDuplicates(int[] array, int randomNumber)
        {
            bool flag = false;
            for (int x = 0; x < 6; ++x)
            {
                if (array[x] == randomNumber)
                {
                    flag = true;
                }
            }
            return flag;
        }
        public static void LottoNumbers(int[] quickpick,int rows)
        {
            FileStream lottoNumbers = new FileStream("LottoNumbers.txt",
FileMode.Append, FileAccess.Write);
            StreamWriter writer = new StreamWriter(lottoNumbers);

                writer.WriteLine();
                for (int i = 0; i < quickpick.Length; i++)
                {
                    writer.Write(quickpick[i] + " ");
                }


            writer.Close();
            lottoNumbers.Close();

        }
    }
}
```

Appendix J : Program Code (Participant 3)

Note: Participant 3 submitted a C# project folder that contained two .cs files, each containing a single Main() method. The code from both files were used to determine the total lines of code generated (63 lines).

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace ConsoleApplication1
{
    class Program
    {

       static void Main(string[] args)
        {
            ArrayList numbers = new ArrayList();

            Random RandomClass = new Random();
            int randomNumber;

            for (int i = 0; i < 6; i++)
            {
                do
                {
                    randomNumber = RandomClass.Next(1, 49);
                }
                while (numbers.Contains(randomNumber));

                numbers.Add(randomNumber);
                Console.Write(" " + randomNumber + " ");
            }


            numbers.Sort();
            numbers.Reverse();
        }
    }
}
```

Program_R2.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RandomNumbers
{
    class Program
    {
        static void Main2(string[] args)
        {
            int smallest = 1;
            int biggest = 49;
            int Totalnumbers = 10;

            List<int> possible = new List<int>();
            for (int i = smallest; i <= biggest; i++)
            {
                possible.Add(i);
            }

            List<int> result = new List<int>();
            Random rand = new Random();
            for (int i = 0; i < Totalnumbers; i++)
            {
                int random = rand.Next(1, possible.Count) - 1;
                result.Add(possible[random]);
                possible.RemoveAt(random);
            }
            Console.WriteLine("Random Numbers between 1 to 49 : ");
            foreach (int i in result)
            {
                Console.WriteLine(i + " ");
            }
            Console.Read();
        }
    }

}
```

Appendix K : Program Code (Participant 4)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        private static Random _rnd;

        static void Main(string[] args)
        {
            _rnd = new Random();
            string inputString;
             int lotto;

            Int32[] lottoNumbers = new Int32[6];


            for (int i = 0; i < 10; i++)
            {
                for (Int32 idx = 0; idx < 6; idx++)
                {


                    do
                    {

                        lotto = _rnd.Next(1, 50);
                    }

                    while (lottoNumbers.Contains(lotto));


                    lottoNumbers[idx] = lotto;


                }

            }
            Console.WriteLine("{0,6}", lottoNumbers[]);
            inputString = Console.ReadLine();
        }


    }
}
```

Appendix L : Program Code (Participant 5)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{

    protected void Page_Load(object sender, EventArgs e)
    {


        LottoTest();


    }




    private void LottoTest()
    {
        Dictionary<int, int> numbers = new Dictionary<int, int>();
        Random generator = new Random();
        while (numbers.Count < 6)
        {
            numbers[generator.Next(1, 49)] = 1;
        }



        string[] lotto = numbers.Keys.OrderBy(n => n).Select(s => s.ToString()).ToArray();


          foreach (String _str in lotto)
            {
                Response.Write(_str);
                Response.Write("  ");




            }

    }


}
```

Appendix M : Program Code (Participant 6)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{

    protected void Page_Load(object sender, EventArgs e)
    {

    }

}
```

Appendix N : Phase 3 survey

Software Engineering IV: Perceptions regarding Quality Appraisal Techniques

Throughout your years of study, you have been introduced to various techniques that can be used to improve the quality of your programs. The aim of this survey is to find out more about your perceptions regarding Quality Appraisal Techniques (QATs). For the purpose of this survey, QATs are defined as **Designs**, **Design Reviews** and **Code Reviews**.

Participation in this research study is voluntary and any data gathered will be regarded as confidential. Completion and submission of this survey is regarded as your consent to participate.

The survey will take approximately 20 - 25 minutes of your time and your participation is highly appreciated.

Instructions:

When completing this survey, think about each statement by itself and indicate your level of agreement from your perspective as a software development student. Give the answer that truly applies to you, and not what you would like to be true, or what you think others want to hear.

Select ONE level of agreement for each statement. Mark your selection with an **X**.

	Strongly Disagree	**Slightly Disagree**	**Slightly Agree**	**Strongly Agree**
Usefulness				
1. Using QATs improves my programming performance.	❐	❐	❐	❐
2. Using QATs increases my productivity.	❐	❐	❐	❐
3. Using QATs enhances the quality of my programs.	❐	❐	❐	❐
4. Using QATs makes it easier to do my programming tasks.	❐	❐	❐	❐
5. The advantages of using QATs outweigh the disadvantages.	❐	❐	❐	❐
6. QATs are useful in programming tasks.	❐	❐	❐	❐
Ease of use				
7. Learning QATs was easy for me.	❐	❐	❐	❐
8. I think QATs are clear and understandable.	❐	❐	❐	❐
9. Using QATs do not require a lot of mental effort.	❐	❐	❐	❐
10. I find QATs easy to use.	❐	❐	❐	❐
11. QATs are not cumbersome to use.	❐	❐	❐	❐
12. Using QATs do not take too much of my time.	❐	❐	❐	❐
Behavioural Intention				
13. I intend to use QATs in future programming tasks.	❐	❐	❐	❐
14. Given the opportunity, I would use QATs.	❐	❐	❐	❐

	Strongly Disagree	Slightly Disagree	Slightly Agree	Strongly Agree
Social Factors				
15. People who influence my behaviour think I should use QATs.	❐	❐	❐	❐
16. People who are important to me think I should use QATs.	❐	❐	❐	❐
17. My fellow students think I should use QATs.	❐	❐	❐	❐
Voluntariness				
18. Although it might be helpful, using QATs are certainly not compulsory in programming tasks.	❐	❐	❐	❐
19. My lecturers do not require me to use QATs.	❐	❐	❐	❐
20. My use of QATs are voluntary.	❐	❐	❐	❐
Compatibility				
21. QATs are compatible with the way I develop systems.	❐	❐	❐	❐
22. Using QATs are compatible with all aspects of my programming tasks.	❐	❐	❐	❐
23. Using QATs fit well with the way I work.	❐	❐	❐	❐
Image				
24. Software developers who use QATs have more prestige than those who do not.	❐	❐	❐	❐
25. Software developers who use QATs have a high profile.	❐	❐	❐	❐
26. Using QATs are a status symbol amongst software developers.	❐	❐	❐	❐
Visibility				
27. QATs are very visible at the Department.	❐	❐	❐	❐
28. It is easy for me to observe others using QATs.	❐	❐	❐	❐
29. I have had plenty of opportunity to see QATs being used.	❐	❐	❐	❐
30. I can see when other students use QATs.	❐	❐	❐	❐
Perceived Behavioural Control – Internal				
31. I feel that there is no gap between my existing skills and knowledge and those required to use QATs.	❐	❐	❐	❐
32. I have the knowledge necessary to use QATs.	❐	❐	❐	❐
Perceived Behavioural Control – External				
33. Specialised instruction and education concerning QATs are available to me.	❐	❐	❐	❐

	Strongly Disagree	Slightly Disagree	Slightly Agree	Strongly Agree
34. Formal guidance is available to me in using QATs.	❐	❐	❐	❐
35. A specific group is available for assistance with QAT difficulties.	❐	❐	❐	❐
36. For making the transition to QATs, I felt I had a solid network of support (e.g. knowledgeable fellow students, student assistants, lecturers, etc.)	❐	❐	❐	❐
37. The Department provides most of the necessary help and resources to enable students to use QATs.	❐	❐	❐	❐
Career Consequences				
38. Knowledge of QATs puts me on the cutting edge in my field.	❐	❐	❐	❐
39. Knowledge of QATs increases my chance of getting a job.	❐	❐	❐	❐
40. Knowledge of QATs can increase my flexibility of changing jobs.	❐	❐	❐	❐
41. Knowledge of QATs can increase the opportunity for more meaningful work.	❐	❐	❐	❐
42. Knowledge of QATs can increase the opportunity for preferred jobs.	❐	❐	❐	❐
43. Knowledge of QATs can increase the opportunity to gain job security.	❐	❐	❐	❐
Result Demonstrability				
44. I would have no difficulty telling others about the results of using QATs.	❐	❐	❐	❐
45. I believe I could communicate to others the consequences of using QATs.	❐	❐	❐	❐
46. The results of using QATs are apparent to me.	❐	❐	❐	❐
47. I would have no difficulty explaining why QATs may or may not be beneficial.	❐	❐	❐	❐

THANK YOU FOR TAKING THE TIME TO COMPLETE THIS SURVEY