ATTRIBUTES CONTRIBUTING TO STUDENTS' USE OF QUALITY SOFTWARE DEVELOPMENT PRACTICES

Guillaume Nel
Lecturer, Department of Information Technology, Central University of Technology (Free State) and PhD student, Department of Computer Science and Informatics, University of the Free State, South Africa

Liezel Nel
Adjunct Professor, Department of Computer Science and Informatics, University of the Free State, South Africa

Johannes Cronje
Dean, Faculty of Informatics and Design, Cape Peninsula University of Technology, South Africa

ABSTRACT
In 2001 the "McCracken group", through a multi-institutional study, concluded that many students finishing their introductory programming courses could not program due to a lack of problem solving skills. In 2004 Lister established that students have a fragile grasp of skills to read and interpret code. Humphrey suggests that educators must shift their focus from the programs that the students create to the data of the processes the students use. This paper addresses the problem of poor performing students through an investigation of their quality appraisal techniques (QATs) and development processes. Firstly, a survey was conducted to determine the current software development practices used by a group of undergraduate Computer Science students. Numeric data collected revealed that the current practices used by the majority of students would not be sufficient to produce quality programs. Secondly, a case study was conducted to gain a deeper understanding of the various factors that are likely to influence students' intention to use QATs. Analysis of numeric data collected through a survey revealed that students' intentions to use QATs are driven by ease of use, compatibility, usefulness, result demonstrability, subjective norm and career consequences. Thirdly, an experiment was conducted to determine students' perceptions on the use of process measurement data to improve their current software development practices. Analysis of numeric and narrative data revealed that performance measurement data could provide students with useful information to adopt proper development practices.

KEYWORDS
problem solving, software development process, quality appraisal techniques, personal software process, undergraduate education

INTRODUCTION
Despite all the efforts of Computer Science educators to train students to develop software programs of the highest standard, the programming performance of undergraduate students is often worse than expected. This can be attributed to the lack of problem solving skills (McCraken et al., 2001), as well as poor code reading and interpretation skills (Lister et al., 2004). Humphrey (1994; 1999) created the Personal Software Process (PSP) that guides software developers in the use of process measurement and quality appraisal techniques (QATs) (in the form of personal design reviews and code reviews) to improve the quality of their programs. He suggests that educators must shift their focus from the programs that the students create to the data of the processes the students use (Humphrey, 1999). Various researchers reported on their experiences with the incorporation of PSP in educational environments (Börsteler et al., 2002; Jenkins & Ademoye, 2012; Towhidnejad & Salimi, 1996; Williams, 1997).

The aim of this paper is threefold:

1. To discover which QATs and software development practices are used by undergraduate Computer Science students at a selected South African University of Technology.
2. To identify factors that influence students' intent to use QATs.
3. To investigate the role of process measurement data as a contributor to the use of quality software development practices.

LITERATURE REVIEW
In 2001 the "McCracken group" (McCraken et al., 2001) conducted a multi-national, multi-institutional study in the United States of America (US) and other countries during which they assessed the programming competency of Computer Science students who completed their first or second programming courses. They found that the majority of the students' programming performance was much worse than expected. Students indicated "the lack of time to complete the exercise" (McCraken et al., 2001, p. 133) as the major reason for poor performance. The research group also found that students struggle to abstract the problem from the exercise description (McCraken et al., 2001) and therefore lack the ability to do problem solving. The group argues that students might have inappropriate (bad) programming habits because they treat program code as text and simply try to fix syntax instead of focusing on the task that the code must accomplish. They suggest that future research should analyse narrative data gathered from students to gain better insight into the students' development processes and problem-solving behaviour.

Lister et al., (2004) conducted a follow-up study on the McCracken group's research to investigate alternative reasons for poor programming performance. Their findings indicate that many students have "a fragile grasp of both basic programming principles and the ability to systematically carry out routine programming tasks, such as tracing (or 'desk checking') through code" (Lister et al., 2004, p. 119). According to Perkins, Hancock, Hobbs, Martin and Simmons (1989), students' code reading and interpretation skills can be linked to their ability to review and debug code.

Software quality can be defined as software that conforms to the user requirements (Crosby, 1979). Software review methods are widely used in the industry to improve the quality of software programs (Fagan, 1976; Schach, 2011), as testing alone is seen as a very ineffective and time-consuming debugging strategy (Schach, 2011). According to Humphrey (2005), effective defect management is essential in order to manage cost and schedule during software development and also contributes to software quality. Humphrey states that testing alone is not the most effective way to remove defects. He proposes the inclusion of additional quality appraisal techniques such as inspections, walkthroughs and personal reviews. An inspection is a kind of structured team peer review process that was introduced by Mike Fagan (1976). Walkthroughs are less formal, with fewer steps than inspections (Schach, 2011). Fagan (1976) concludes that a developer's productivity increases when he uses inspections because less time is spent on unit testing. Schach (2011) indicates the advantages in time, cost and essentially project success when defects are discovered early in the development life cycle.

Humphrey (2005) regards inspections and walkthroughs as team quality techniques. He proposes that individual software developers should review their work before peer inspection, hence the term "personal reviews". He indicates 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 is "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 develop from the first small program they have written, and it is difficult to convince them to adopt better practices. Humphrey (2005) created a PSP course in which a software developer gradually learns to adopt his/her software practices according to personal measurements. The aim of the course is to improve program quality through personal reviews and to enable a software developer to make more accurate estimations based on personal historical performance data (collected by the individual). Analyses of thousands of PSP students' measurement data indicate that personal reviews improve program quality and that students spent less time in the testing phase if they use quality appraisal techniques (design reviews and code reviews). 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 grades 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 the students use. A number of researchers reported on their experiences with the incorporation of personal software process techniques in educational environments.

Jenkins and Ademoye (2012) conducted a pilot and follow-up experiment in which students used personal code reviews to improve the quality of their individual programs. Although there is no concrete evidence to support this statement, the narrative feedback from the students in both experiments indicate that they believe the process of using code reviews improved the quality of their programs.

Towhidnejad and Salimi (1996) incorporated a simplified version of PSP as part of two first-year Computer Science courses. They report that PSP helped students to improve their time management and time estimations, as well as to decrease the number of syntax defects. Students only accepted PSP as an integral part of their development practices in their second semester of PSP usage. The educators' biggest challenges were (1) to motivate students to follow the PSP defined process 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 Computer Science courses. Williams (1997) reports that although students demonstrated accurate theoretical knowledge of PSP principles, they struggled with the application thereof. He remarks that discussion of group statistical feedback data might influence students' intention to capture more accurate individual process measurements. His biggest challenge was to motivate students to use PSP as part of their natural program development practices.

Börsteler et al., (2002) report on their experiences of teaching some PSP variations at different universities. At Montana Tech, University of Montana, students showed initial 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 showed initial resistance to PSP several of them reported incorporating at least some PSP parts in their work environments. An evaluation of Purdue University students' attitude towards PSP reveals 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 later programming courses when students are already familiar with language-specific syntax and development environment. 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 was worthwhile.

In an attempt to test the process improvement claims of PSP, Prechelt and Unger (2000) conducted an experiment to compare the performance of 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 (2000) 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.

METHODOLOGY

This research study followed a mixed methods approach based on the Framework of Integrated Methodologies (FraIM) as suggested by Plowright (2011). The context of this study was the Information Technology department at a selected South African University of Technology. The study was divided into three cases in order to distinguish between the three main sources of data (Plowright, 2011).

CASE 1 METHODOLOGY

In Case 1 a survey (Plowright, 2011) was conducted to gather information regarding undergraduate Computer Science students' perceptions of the quality appraisal techniques and software development processes they normally use when developing programs. The research population for this case included all first, second and third year Computer Science students at the selected institution. Data was collected by means of "asking questions" in a paper-based self-completion survey containing closed questions (Plowright, 2011). The survey was distributed and completed during normal lectures. A total of 251 students (the sample) completed the survey. This sample included 74 first-year, 113 second-year and 64 third-year students. The numerical data collected through the survey was analysed in MS Excel and the results grouped according to the year level of the respondents.

CASE 2 METHODOLOGY

In Case 2 a case study (Plowright, 2011) was conducted to gain a deeper understanding of the various factors that are likely to influence students' intention to use QATs. The research population for this case was restricted to fourth-year Computer Science students from the selected institution, who were registered for the Software Engineering module (55 students). These students were selected because they were already familiar with the various techniques that can 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).

There are numerous theoretical models that can be used to examine individual intentions to adopt information technology tools. Although software development methodologies and more specifically QATs cannot necessarily be regarded as technological tools, a study conducted by Riemenschneider, Hardgrave and Davis (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) (Davis, 1989);
- TAM2 (Venkatesh & Davis, 2000);
- Perceived Characteristics of Innovating (PCI) (Moore & Benbasat, 1991);
- Theory of Planned Behaviour (TPB) (Ajzen, 1985); and
- Model of Personal Computer Utilisation (MPCU) (Thompson, Higgins & Howell, 1991).

After evaluation of these five models Riemenschneider et al., (2002, p.1139) identified 12 constructs (which include both common and unique constructs from the selected models) as appropriate in the context of methodology adoption. The selected constructs are defined as follows in the context of Case 2:

- **Behavioural intention (BI)** – the extent of the student's intention to use QATs.
- **Usefulness (U)** – the extent to which the student thinks that using QATs will enhance his/her programming performance.
- **Ease of use (EOU)** – the extent to which the student perceives that using QATs will be free of effort.
- **Subjective norm (SN)** – the extent to which the student thinks that others, who are important to him/her, think he/she should use QATs.
- **Voluntariness (VOL)** – the extent to which the student 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 student develops systems.
- **Result demonstrability (RD)** – the extent to which the results or benefits of using QATs are apparent to the student.
- **Image (IMG)** – the extent to which the use of QATs is perceived to enhance the student's image/status in his/her social system.
- **Visibility (VIS)** – the extent to which the use of QATs can be observed in the student's learning environment.
- **Perceived behavioural control – internal (PBC-I)** – the student's perceptions of internal constraints on using QATs.

- **Perceived behavioural control – external (PBC-E)** – the student's perceptions of external constraints on using QATs.
- **Career consequences (CC)** – the extent to which the adoption of QATs will influence the student's chance to secure employment after completing his/her degree.

The survey constructed for Case 2 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 Case 2. Each item was based on a 4-point Likert scale (1 = strongly disagree and 4 = strongly agree). The numerical data collected through the survey was analysed using SPSS software.

CASE 3 METHODOLOGY

In Case 3 an experiment (Plowright, 2011) was conducted to gain a deeper understanding of students' development processes through the collection of actual process data. The population for this case included all third year Computer Science students at the selected institution. These students were selected since they already had intermediate programming skills and experience in software defect removal strategies. From this population six students were randomly selected to participate in the practical experiment. Data collection included observations, asking questions (post-activity survey and interviews) as well as artefact analysis (Process Dashboard© data and program code) (Plowright, 2011).

The Case 3 experiment consisted of four steps as summarised in Table 1. The instructor first conducted a tutorial activity to teach students how to log and interpret performance-measurement data using the Process Dashboard© software. During this tutorial students were required to do an exercise in which they had to log time, size and defect measurements in different phases of the software development life cycle. The various defect types and examples of defects categorised into types were also discussed. After the tutorial the students completed an individual programming exercise during which they had to capture performance data using the Process Dashboard© software. For this programming exercise the students had to implement the code to simulate the "Quick Pick" option of the South African National Lottery (LOTTO©) draw.[1]

TABLE 1: EXPERIMENT DESIGN

Activity	Duration	Rationale
1. Instructor presents performance measurement tutorial.	1 hour	Teach students to do process measures and interpret process data.
2. Students do programming exercise. while Instructor makes observations.	3 hours	Capture process measures while doing programming exercise (student). Record student behaviour and questions asked (instructor).
3. Students complete post-activity survey.	15-20 min	Explore students' perceptions of process measuring.
4. Instructor conducts interviews with students	10 min (per student)	Gain deeper insights into students' development processes.

The students received an extensive background document on how "LOTTO" draws work. In the "Quick Pick" option a user of the system first had to select the number of player lotto draw records that should be generated. The requested number of records then had to be generated randomly, sorted and written to a text file. Each draw record had to contain the draw date, draw number (starting from 1) and seven unique numbers ranging from 1 to 49 (the six lotto numbers in ascending order followed by a bonus number). Students could use any resources, including the Internet, to complete this activity. While the students worked on the individual programming exercise the instructor moved around the students and recorded his observations as well as all questions from the students. After this exercise the students had to complete a post-activity survey that consisted of mostly open-ended questions. The purpose of this survey was to explore the students' perceptions on the capturing and interpreting of process-measurement data. In the final activity of Case 3 the instructor conducted interviews with all six students. During these interviews open-ended questions were used to gather narrative data regarding the students' development processes.

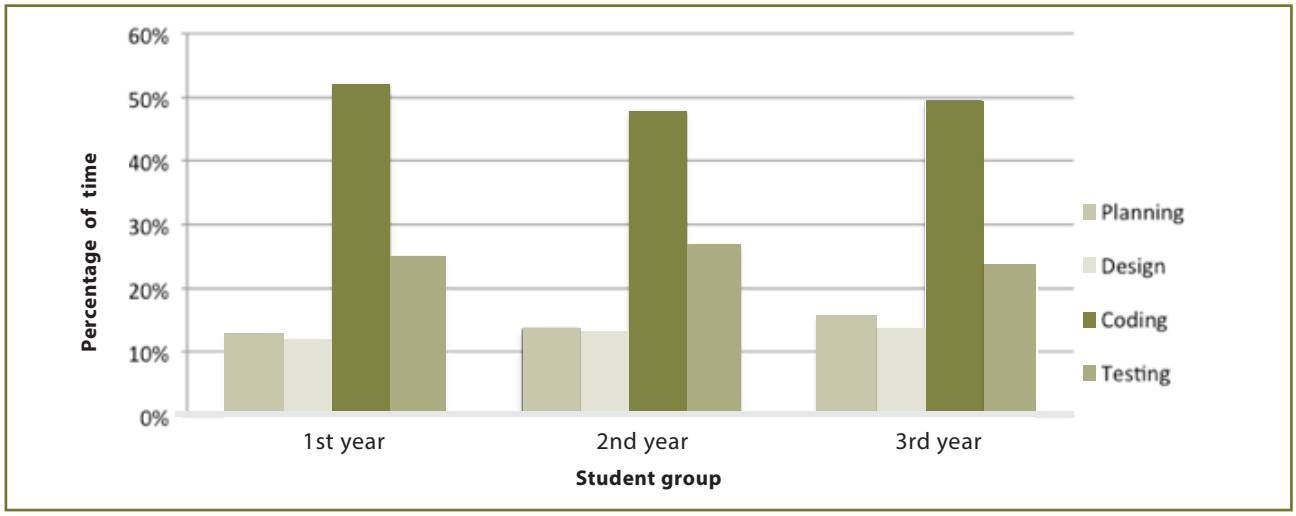
DISCUSSION OF RESULTS

CASE 1: PRE-SURVEY

Students first had to indicate how much of their development time is spent in each of the provided phases (see Figure 1). On average, students spent 25% of their development time on planning and design. They also indicated that most of their development time is spent on coding and contributes to 50% of the total development time. They spent 25% of their time on testing and debugging, which is roughly half the time that they spent on coding. Students of all year levels indicated almost similar results, which is an indication that a first year student and a third year student make use of similar development practices. It should be noted that the reported times are mostly estimates (individual perceptions) since only 16% of the students indicated that they record the actual time that they spend in the different development phases. The majority (88%) of students indicated that they do not use any time estimation techniques.

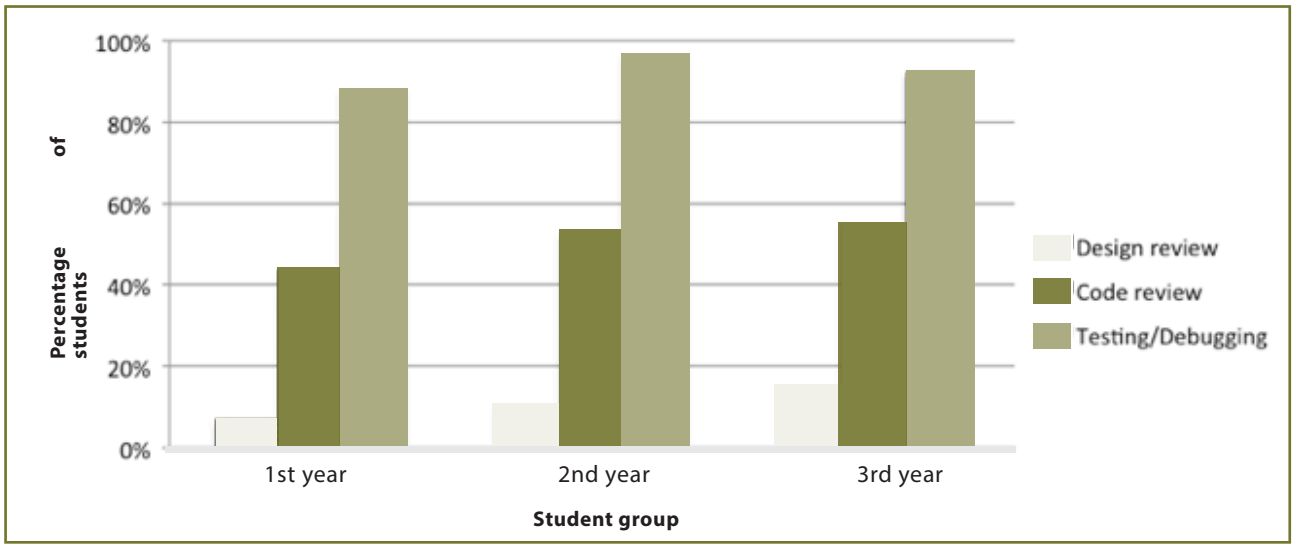
1 https://www.nationallottery.co.za

FIGURE 1: ESTIMATED TIME SPENT IN DEVELOPMENT PHASES



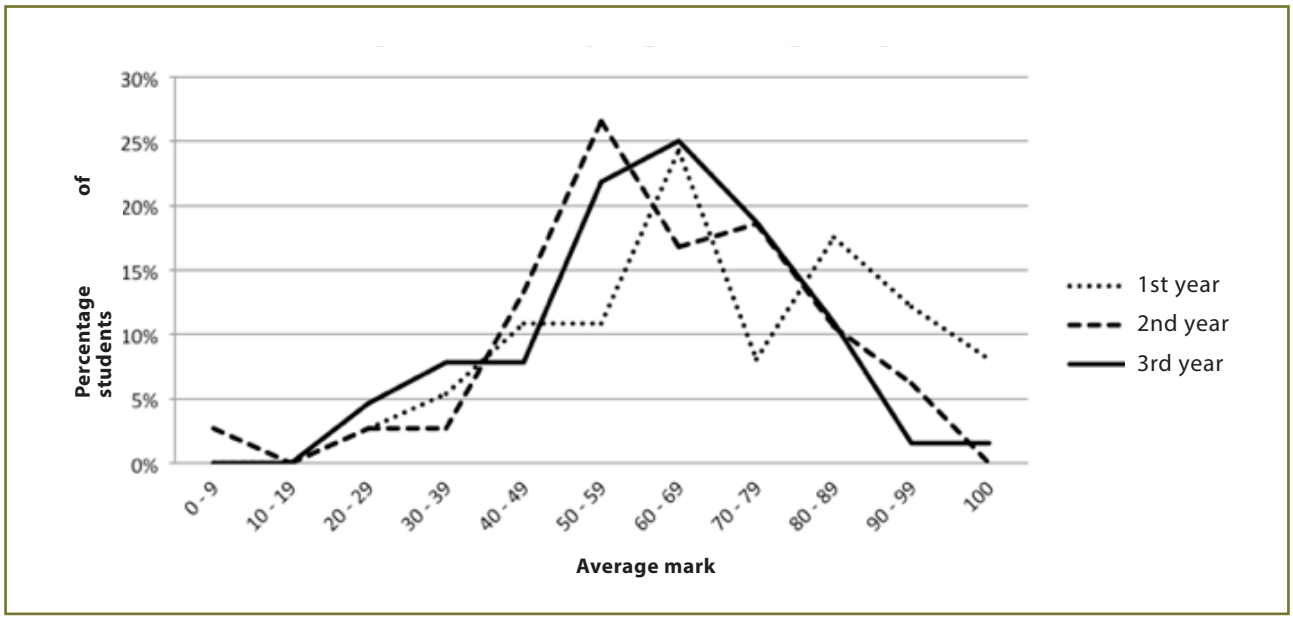
The next section of the survey focused on defect removal strategies. Students reported that they primarily use debugging for fixing defects as opposed to design and code reviews (see Figure 2). The use of design and code reviews increment slightly (10%) from first- to third-year students. Only 30% of the students indicated that they keep record of the defects they make.

FIGURE 2: USE OF DEFECT REMOVAL STRATEGIES



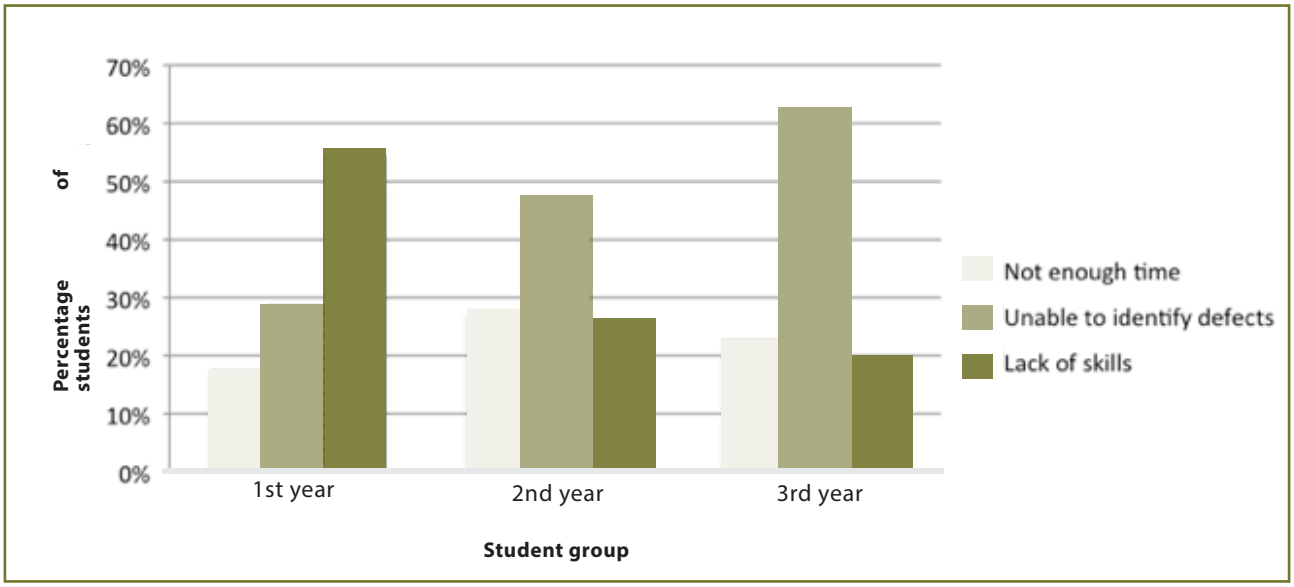
Students were also asked to give an indication of the average mark they obtain for their programming assignments. As indicated in Figure 3 the reported average marks form a normal distribution curve around 59.5%.

FIGURE 3: AVERAGE MARKS FOR PROGRAMMING ASSIGNMENTS



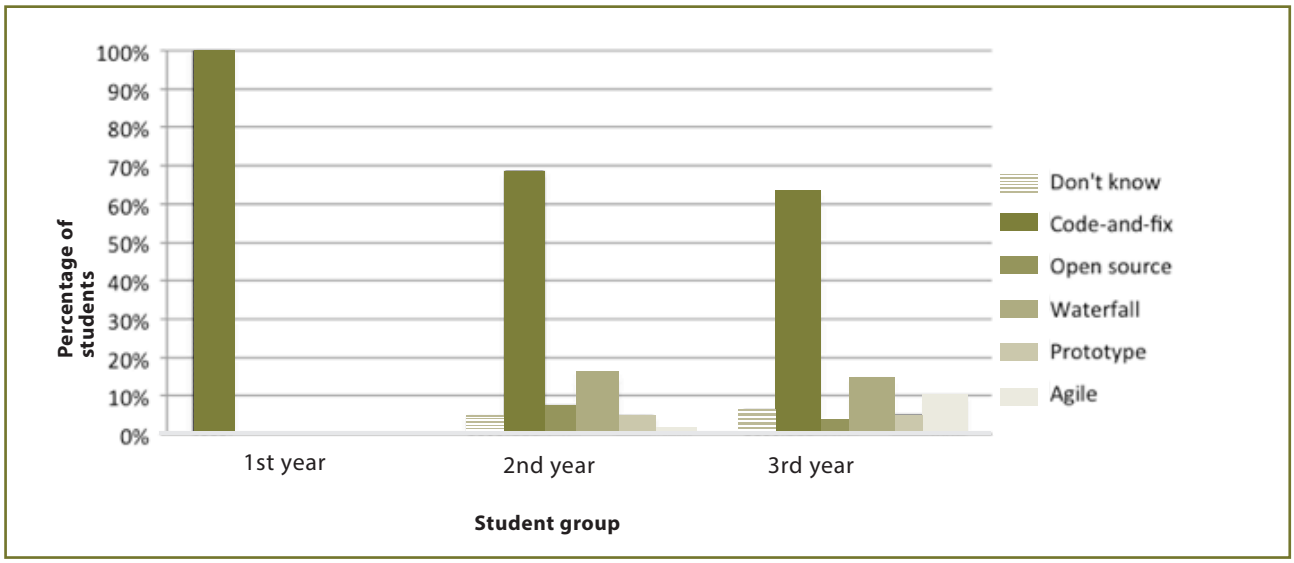
Students then had to select (from three provided options) the main reason why they do not score full marks in all their programming assignments. The data analysis revealed distinct differences between the responses from students in the different year levels (see Figure 4). The majority of first-year students (54%) believe that their lack of programming skills is the major cause of poor results. Second-year (47%) and third-year (62%) students mostly put the blame on their inability to identify defects. Towards the third year fewer students (16%) regard their "lack of skill" as the major reason for failure. Although the students in all year levels regard "time" as a stumbling block to their success it is not seen as the major contributor (with values ranging between 17% and 27%).

FIGURE 4: MAIN REASON FOR NOT SCORING 100% FOR ASSIGNMENTS



When students were asked to indicate their preferred software development life-cycle model the majority of second-year (68%) and third-year (63%) students selected "code-and-fix" (see Figure 5). It is not surprising that all the first year students selected the "don't know" option, since the first Software Engineering course is part of 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.

FIGURE 5: PREFERRED SOFTWARE LIFE CYCLE



Without a process that accommodates designs, students would spend little time on design reviews and consequently would not be able to identify defects early in the development life cycle. The students therefore have to rely on code reviews and debugging as their primary technique for finding and fixing defects. When using code-and-fix strategies the "thinking" process of "how to solve a problem" would occur during the coding phase – not during the design phase – which explains why students spent most of their time in the coding phase. Since the students indicated "debugging" as their primary technique for fixing defects (see Figure 2) it is no surprise that they struggle to identify defects. They treat the consequence of a defect, which makes it a lot more difficult and takes more time to find the actual defect. This also explains why students see the "identification of defects" as a major contributor to poor results (see Figure 4). This effect will increase towards the third year when assignments are more comprehensive – therefore making it more difficult to identify defects. The student, however, will not realise this because he/she is using exactly the same process that worked for him/her from the first year. This explains why there is almost no difference in the time spent in phases from first to third year (see Figure 1).

CASE 2: CASE STUDY

Initial analysis of the Case 2 survey data revealed that the voluntariness and perceived behavioural control – internal constructs displayed low construct reliability (Cronbach's alpha < 0.64). Only the 10 remaining constructs were therefore retained for further analysis (see Table 2).[2]

TABLE 2: 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 does not require a lot of mental effort. • I find QATs easy to use. • QATs are not cumbersome to use. • Using QATs does 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 is compatible with all aspects of my programming tasks. • Using QATs fits 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 is a status symbol amongst software developers.	0.745
Visibility (VIS) Mean = 2.4521 SD = 0.68888	• QATs are very visible at the Department.[3] • 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 is available to me. • Formal guidance is available to me in using QATs. • A specific group is available for assistance with QATs 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 = 0.6273	• 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

3 Although the true name of the academic department and institution concerned was used on the actual instrument it will not be disclosed here in order to protect the anonymity of the selected institution.

The next step was to identify the constructs that can be regarded as significant determinants of students' intentions (BI) to use QATs. Each construct was tested individually using least-squares regression analysis. Table 3 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.

TABLE 3: 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

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 some 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.

The results of this research study show that the perceived compatibility of software process innovations (such as QATs) with a developer's pre-existing software development process have a highly significant influence on intention to use. Chan and Thong (2009, p. 811) emphasise 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. In a study comparing the PSP experiences of first-year and graduate students, Runeson (2001) concludes 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.

There are numerous examples of prior studies that have found perceived usefulness as 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 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 useful.

In support of prior studies, the Case 2 results also show that subjective norm significantly affects intention. Riemenschneider et al., (2002) 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 who oppose the use thereof. Chang and Thong (2009) conclude 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 Case 2 are also required to complete a number of group projects. 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 (Börsteler et al., 2002). These students also mentioned that they "encouraged each other to follow PSP practices" (p. 45).

The difference in the two studies regarding the effect of ease of use on adoption could possibly be attributed to the difference in contexts – working environment vs. 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 conclude that the diverse views on the resulting 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 Case 2 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 use 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 Börsteler 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.

CASE 3: PROGRAMMING EXPERIMENT

The discussion in this section considers the data that was collected during Steps 2, 3 and 4 of the Case 3 programming experiment.

INSTRUCTOR OBSERVATIONS

The instructor made the following main observations while the students were completing the exercise:

- Students searched the Internet to find solutions for the exercise.
- No designs were created to solve the exercise problem.
- Some students forgot to start and stop the Process Dashboard© timer when switching phases.
- Some defects were not logged.
- Students struggled to distinguish between the "coding" and the "testing" phase.
- Students struggled to describe their logged defects.

The students did not log the re-work coding in the correct phase. Most of them logged that time under coding, which explains why re-work or testing time was lower than coding time (also see section on Process Dashboard© performance data.). More precise measurements would result in much higher testing times. Towhidnejad and Salimi (1996) also reported that only half of their students collected accurate and reliable data.

PROGRAM CODE

Not one of the students in the group produced a fully-functional program (according to the given specifications) during the allotted time frame. Two of the students (Student A and Student B) created programs that accomplished almost all of the given requirements. Both of their programs generated the specified number of player lotto draw records and wrote these records to the output file. They were, however, unable to calculate the draw date and draw number (for each record generated), which also had to be part of the output file. Students C, D and E had executable programs. Student C's program could only generate a fixed number of lotto draw records (10) without duplicates, while the output was written to the screen instead of the required text file. These draw records also did not include the draw date and draw number. Student D's program calculated incorrect draw numbers and wrote these numbers to the screen. Student E's program only contained a user input screen with no code to solve the lotto draw problem. The program created by Student F could not be executed. Inspection of the intended code logic revealed that his program was unable to generate any random numbers and did not contain any code to generate output.

PROCESS DASHBOARD© PERFORMANCE DATA

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

On average the students spent their 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.

The actual time that these students captured while working shows a good correlation with the times reported in the pre-survey (see Figure 1). The actual testing or debugging time, however, would be much higher if these students had to continue to produce fully-functional programs. The students on average produced 45 lines of code, which resulted in a productivity of 20 lines of code per hour. Each student recorded an average of five defects, with 90% of these defects injected during coding. The limited time spent on designs also indicates that most defects would be 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 that only 1% of the time was spent in reviews, this would yield few defects (2%) to be found during reviews. No design reviews were conducted because of the lack of designs and only 1% of the time spent on the design phase. This resulted in defects being discovered late in the development life cycle (testing), which makes it more difficult to identify them.

POST-ACTIVITY SURVEY

Students indicated that capturing time measurement data in the correct phases was easy, but identifying and describing defects was difficult. For process improvement some students indicated that they would spend more time on creating effective designs, and need to learn the skill to do effective reviews to pick up defects earlier in the life cycle. Participants in Prechelt and Unger's (2000) study made almost identical remarks. In McAlpin and Liu's (1995) study the programmers' software quality increased because they were motivated to spend more time on designs and reviews.

Most students were surprised by how much time they spent on testing and indicated that debugging might not be the most effective way to find and fix defects. In one of Humphrey's (1994) earlier studies about process feedback he remarks that programmers are typically surprised at how much data they can gather from small exercises and how quickly they can start using the measurement data to improve their personal software practices.

INTERVIEWS

An interview was conducted with each student in order to gain a deeper understanding of the development processes each one followed to create the program. The only artefacts that the students created (in addition to the captured Process Dashboard© measurement data) were the actual code (see section on Program code). The students did not create designs and therefore these interviews focused on what each student did during the problem-solving process.

The students all 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. As part of Feiner and Krajnc's (2009) experiment they asked their students what their first step would be in solving a given programming assignment. They report that most of their students indicated that they would search "the Internet" or "use Google" first. A survey conducted at the end of their experiment also revealed their students' general acceptance of "Copy & Paste" programming as part of their software development process (Feiner & Krajnc, 2009, p. 84). All the students in our experiment indicated that this is the method they usually follow when completing their programming assignments.

In retrospect, all the students indicated that they should rather have started by first solving the problem logically (using flowcharts or pseudo code) and then searched for code snippets to accomplish specific tasks. They also indicated that they do not find it easy to write pseudo code to solve problems and therefore prefer to search for code solutions where the logical thinking has already been done. Generally, they find it "hard to start" solving a problem.

CONCLUSION

In this paper various attributes contributing to the poor quality of student programs have been mentioned. The findings of Case 1 revealed that most students rely on a process of "code-and-fix", as predicted by Humphrey (1999). "Code-and-fix" remains the predominate process of choice from first- to third-year level, which indicates no process improvement through these years of study. Students also regard "testing" as the most effective strategy to remove defects. The case study conducted in Case 2 revealed that students' usage of QATs is driven by ease of practice, compatibility, convenience, 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). More in-depth research is needed to identify additional factors that could possibly affect students' desire to use QATs. After a selected group of third-year students participated in a practical experiment (Case 3), they – through the use of process measurement data – realised (1) how much time they were actually spending on testing or rework, and (2) that testing is not the most effective method to find and solve defects. Process-measurement data could therefore be regarded as a potential contributor to the usefulness construct of students' willingness to adopt QATs.

The students' feedback indicated that they also lack design and problem-solving skills. This provides further verification for the findings of McCracken et al.'s study (2001). This lack of design and problem-solving skills could potentially be the main driver behind the students' preference towards a "code-and-fix" development process. However, data collected during Case 3 revealed that the students' development process could rather be described as "copy-paste-and-fix", since very little code was produced from scratch. An in-depth investigation is necessary to see if "copy-paste-and-fix" is the prevailing development process for most undergraduate computer programming students. If this is found to be the case, educators could focus on equipping students with proper "Copy and Paste" skills [as suggested by Feiner and Krajnc's (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., 1989), it could also enable them to effectively reuse code snippets copied from the Internet and other sources. It is, however, recommended that educators enforce effective design techniques from the first programs that students write in an effort to ensure that they will not fall back on an unstructured "code-and-fix" or "copy-paste-and-fix" life cycle. Case 3 has shown that the effect of process-measurement data should be regarded as a valuable contributor to any process improvement changes that educators want to enforce on students. The ultimate ideal is that students be able to adapt their processes according to their personal data.

ACKNOWLEDGMENTS

This paper is based on research conducted under the supervision of Profs J. C. Cronje and L. Nel, in partial fulfilment of the requirements for the Doctoral Degree in Computer Information Systems in the Faculty of Natural and Agricultural Sciences at the University of the Free State, and is published with the necessary approval.

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.

Ajzen, I. (1985). From intentions to action: A theory of planned behavior. In J. Kuhl and J. Beckmann (Eds), *Action control: From cognition to behavior* (pp.11-39). New York, NY, Springer Verlag.

Börsteler, 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.

Chan, F. K. Y. & Thong, J. Y. L. (2009). Acceptance of agile methodologies: A critical review and conceptual framework. *Decision Support Systems, 46*, 803-814.

Crosby, P. B. (1979). *Quality is free*. New York, NY, McGraw-Hill.

Davis, F. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly, 13*(3), 318-339.

Fagan, M.E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 15*(3), 182-211.

Feiner, J. & Krajnc, E. (2009). Copy & paste education: Solving programming problems with web code snippets. *Proceedings of the Interactive Computer Aided Learning (ICL 2009) conference* (pp. 81-88).

Humphrey, W. S. (1994). Process feedback and learning. *Proceedings of the 9th International Software Process Workshop* (pp. 104-106).

Humphrey, W.S. (1999). Why don't they practice what we preach? The personal software process (PSP). *Annals of Software Engineering, 6*(1-4), 201-222.

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

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.

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). A multi-national study of reading and tracing skills in novice programmers. In *Working Group reports from ITiCSE on innovation and technology in computer science education* (ITiCSE-WGR '04) (pp. 119-150). New York, NY, ACM.

Iivari, J. (1996). Why are CASE tools not used? *Communications of the ACM, 39*(10), 94-103.

McAlpin, J. & Liu, J. B. (1995). Experiencing disciplined software engineering at the personal level. *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing* (pp. 124-127).

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). New York, NY, ACM.

Moore, G. & Benbasat, I. (1991). Development of an instrument to measure the perceptions of adopting an Information Technology innovation. *Information Systems Research, 2*(3), 192-222.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989). Conditions of learning in novice programmers. *Journal of Educational Computing Research, 2*(1), 37-55.

Plowright, D. (2011). *Using mixed methods: Frameworks for an integrated methodology*. London, UK, SAGE Publications (Kindle edition).

Prechelt, L. & Unger, B. (2000). An experiment measuring the effects of personal software process (PSP) training. *IEEE Transactions on Software Engineering, 27*(5), 465-472.

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.

Runeson, P. (2001). Experiences from teaching PSP for freshmen. *Proceedings of the 14th Conference on Software Engineering Education and Training, IEEE* (pp. 98-107).

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

Thompson, R., Higgins, C. & Howell, J. (1991). Personal computing: Toward a conceptual model of utilization. *MIS Quarterly, 15*(1), 125-143.

Towhidnejad, M. & Salimi, A. (1996). Incorporating a disciplined software development process in to introductory computer science programming courses: Initial results. *Proceedings of the 26th Annual Frontiers in Education Conference (FIE '96), Vol. 2.* (pp. 497-500).

Venkatesh, V. & Davis, F. (2000). A theoretical extension of the technology acceptance model: Four longitudinal field studies. *Management Science, 46*(2), 186-204.

Williams, L.A. (1997). Adjusting the instruction of the personal software process to improve student participation. *Proceedings of the 27th Annual Frontiers in Education Conference (FIE '97), Vol. 1. Teaching and Learning in an Era of Change* (pp. 154-156).