3rd International Conference on Emerging Technology and Interdisciplinary Sciences (ICETIS 2024)

An exploration of novice programmers' actual software development processes and use of quality appraisal techniques

Guillaume Nel*

[1]*Department of Information Technology, Central University of Technology, Free State, South Africa, guilnel@cut.ac.za,* https://orcid.org/0000-0003-1750-0960

[2]*Department of Computer Science & Informatics, University of the Free State, South Africa.*

*Corresponding author

Abstract: Achieving high-quality software projects is a central goal in Software Engineering, with best practices typically taught to Computer Science (CS) undergraduates. The Personal Software Process (PSP) framework guides developers in good development practices, yet incorporating PSP principles into curricula has encountered challenges, particularly in students' effective use of quality appraisal techniques (QATs) such as design, design review, and code review. This study investigated attributes influencing novice programmers' use of QATs within the PSP context. An experimental case study was conducted involving six third-year CS students. Data were collected through actual process measurements and narrative feedback to compare their perceived and actual development processes. The analysis revealed significant discrepancies between students' perceived and actual use of QATs. Critical success factors affecting the adoption of QATs were identified, including an understanding of development phases, technical programming skills, accuracy in measurement data, ability to identify defects, design and review skills, and motivational orientation. These findings contribute a list of attributes serving as cautionary guidelines for educators aiming to enhance software development practices among novice programmers. Addressing these factors can improve the integration of PSP principles and the effective use of QATs in educational settings.

Keywords: Novice programmers, Personal software process, Process measurement data, Quality appraisal techniques, Software development processes, Software quality improvement

Introduction

Software review methods are widely used in 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). 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 the students create to the data of the students' processes (Humphrey, 1999). Various researchers reported their experiences incorporating PSP in educational environments (Börstler et al., 2002; Contreras-Vas et al., 2021; Kusakabe et al., 2020; Quinn, 2023; Towhidnejad & Salimi, 1996). Despite its potential to bridge the gap between industry and educational training, PSP is not widely used in academic settings (Pando Soto & Rodríguez Rafael, 2020)—with very limited examples from the past ten years.

The aim of this paper is twofold. Firstly, to form a better understanding of the differences between novice programmers' perceived and actual development processes (including using 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 shift their focus from the programs that the students create to the data of the processes they use.

Literature review

According to Humphrey (2005), effective defect management is essential to managing costs and schedules 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. 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, a software developer's only generally accepted short-term priority is "coding and testing".

Both positive and negative effects of using PSP principles in educational environments 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; 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 to "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 about the use of PSP principles:

- Students struggled to capture accurate and reliable data (Carrington et al., 2001; Contreras-Vas et al., 2021; Grove, 1998; Kusakabe et al., 2020; Prechelt, 2001; Towhidnejad & Salimi, 1996). Some studies attributed this 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). When students struggled 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 only introducing PSP topics 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) or regarded it as extra effort (Börstler et al., 2002; Carrington et al., 2001; Contreras-Vas et al., 2021; Hou & Tomayko, 1998; Kusakabe et al., 2020). 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; Pando Soto & Rodríguez Rafael, 2020).
- Students struggled with the application of PSP principles even though they demonstrated accurate theoretical knowledge of these principles (Contreras-Vas et al., 2021; Kusakabe et al., 2020).
- Students with limited software development experience are unable to adjust their processes according to lessons learnt from past experience (Kusakabe et al., 2020; 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. Actual process measurement data could also better indicate the quality of students' development processes. In addition, several authors (Hu 2016; McCracken et al., 2001) have proposed using 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.

Research method

For this 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 explains the sampling decisions, methodology, and data collection methods

Sampling decisions

The population for this study included all third-year CS students enrolled for the software development stream at a selected university in South Africa. 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 research study phase, 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 study. A participant information sheet was distributed to all the identified students as an invitation to participate in the research activity. Therefore, the sampling strategy can 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.

Methodology and data collection methods

The methodology followed for the experimental case study comprised the six steps as summarised in Table 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 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.	3 hours	Capture process measures while doing programming exercise (Participants).
4.	Instructor makes observations.		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.

Pre-activity questionnaire

The pre-activity questionnaire (paper-based self-completion format) integrated both structured (close-ended questions) and less structured (open-ended questions) approaches to asking questions (Babbie, 2010). 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 2). Each of these questions was explicitly structured to relate to one of the levels of Humphrey's (2005) PSP framework (PSP0, PSP1, or PSP2).

Performance measurement tutorial

After completion of 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. This software application is part of an open-source initiative to support developers in using PSP or Team Software Process (TSP). Process Dashboard© offers essential data collection, planning, tracking, analysis, and export functionalities.

Table 2: Origin of questionnaire questions

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

As part of the tutorial, participants worked on a programming exercise to practice the capturing of process measurements with Process Dashboard©. I started the tutorial with a discussion of the 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. 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 the completion of the entire tutorial. This included instructor-led explanations and discussions. No data (for research purposes) was collected during this step.

Programming exercise

After the tutorial, the participants completed an individual programming exercise. The purpose of the exercise was to:

- Use Process Dashboard© to capture process measurement data while doing a programming assignment following the PSP2.1 process script.
- 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.
- 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 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.

Post-activity questionnaire

After this exercise, the participants had to complete a post-activity questionnaire with primarily open-ended questions. This questionnaire aimed 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.

Focus group discussion

As the final step of data collection, a focus group discussion was included to gather narrative data on the participants' development processes. Initially planned with all six participants, only three attended despite

offering rescheduling options to those who declined. The discussion, conducted the day after the participants completed the first five activities, focused on the actual processes each participant followed to solve the programming problem.

Data analysis

For data analysis, Microsoft Excel was utilised for numerical data, while narrative data was analysed in NVivo 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 the study 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 to evaluate the collected data.

Result and discussion

This section presents the findings from the experimental case study, integrating both the quantitative process measurement data and the qualitative narrative feedback from participants.

Instructor observations

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

- Participants searched the Internet 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" phases.
- 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.

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 had been 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 the 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. However, the actual testing time would 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.

Differences between perceived and actual software development processes

Table 3 summarises and compares the perceived and actual design practices followed by the participants during the experimental case study.

Table 3: 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 on design than initially perceived. Despite the substantial amount of perceived design time (ranging from 10% to 20%), most of the participants spent 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 used (none) and the total lack of formal design documentation. Other design studies also noted a similar lack of designs (Eckerdal et al., 2006; Lotus et al., 2011). In Eckerdal et al.'s (2006) study, 80% of the students created either no designs or made no significant progress toward design. The only exception in this study was Participant 1, who 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 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). Except for Participant 5, most participants in this study completely underestimated the time that they usually 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 much of their coding occurred due to 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 cannot log data in the correct prescribed PSP phases. The higher-than-expected testing times could also indicate that participants did more "fixing than coding" instead of the code-and-fix strategy that most of them believed they typically used.

Table 4: 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 5 summarises and compares the participants' perceived and actual defect removal strategies, and their perceived

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

Table 5: 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 corresponded with their use thereof during the exercise. Participants 3, 5, and 6 indicated that they typically used other strategies (design review and code review) but only used debugging during the exercise. Of interest is the large amount of perceived time these three participants believed they were using for design reviews and code reviews. This might directly influence the time they thought they spent on testing/debugging because they believed they would pick up defects earlier in the life cycle (with design reviews and code reviews). The low perceived design review times might indicate the participants' inability to create reviewable designs (Humphrey, 2000). However, it is still questionable that some participants indicated they used design reviews and even assigned a substantial amount of time to it. At the same time, they did not do any designs. Since even small programming exercises require some code to be written, there will always be at least some code to review. In Towhidnejad and Salimi's (1996) study, students 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 they have developed from the first small program they have written, and it is not easy to convince them to adopt better practices (Humphrey, 1999).

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 to identify specific problems experienced by the participants in following these guidelines/practices. Since adopting 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.

PSP0: Software development process and basic measurements

Four potentially influencing attributes were identified using PSP0 as the first lens for this reflection.

Understanding of development phases

One of the PSP quality improvement practices states: "To do high-quality work, you must measure and manage the quality of your development process" (Humphrey, 2005: 157). Given defects' negative impact 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: 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. Therefore, they spent most of their actual

development time in the planning, coding and testing phases. The code-and-fix model does not make provisions 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 needed 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 indicates that Participant 5 did not know exactly what to do in each phase.

Technical programming skills

One of the code review principles of PSP is that one must 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, which 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: 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. 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 distinguishing 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.

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. 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.

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: 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: 185). All the participants in this study indicated some formal design modelling techniques that they typically used, but none of them attempted 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 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 6 summarises the participants' responses in this regard.

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: 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, and 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 and 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.

Table 6: 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).

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 their own intelligence and abilities could ultimately influence their individual motivations and behaviours. Given the development processes followed by the participants in the programming 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: 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, and 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 s. 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 7 provides a summary of the identified skills and behavioural characteristics of each participant, mapped to the identified attributes.

Table 7: 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

Conclusion

This study investigated the discrepancies between novice programmers' perceived and actual software development processes, with a particular focus on their use of QATs within a PSP framework. By analysing actual process measurement data alongside narrative feedback from participants, we aimed to identify attributes that influence the adoption of QATs among novice programmers.

The findings revealed significant gaps between what the participants believed they practiced and what they actually did during the programming exercise. Despite reporting the use of design, design reviews, and code reviews in their perceived development processes, most participants did not implement these practices when faced with an actual programming task. Instead, they predominantly engaged in coding and debugging activities, allocating minimal time to design and review phases. This discrepancy suggests that while novice programmers may understand the theoretical importance of QATs, they struggle to integrate these practices into their actual workflows.

The following difficulties were identified that could potentially hinder the effective use of QATs among novice programmers:

- 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.

The study's limitations include a small sample size and the use of participants from a single institution, which may affect the generalisability of the results. Future research should consider larger, more diverse populations to validate these findings. Longitudinal studies could also provide insights into how novice programmers' use of QATs evolves over time with continued practice and education. Additionally, exploring the impact of different teaching methodologies on the adoption of QATs could offer valuable guidance for educators.

By identifying the critical factors that influence novice programmers' use of QATs, this study contributes valuable insights to the field of computer science education. Addressing the identified challenges through targeted educational interventions can promote the effective adoption of QATs, leading to improved software development practices among novice programmers. Ultimately, fostering these skills and behaviours is essential for preparing students to meet the quality demands of the software industry and to excel in their future professional endeavours.

Funding

This research paper received no internal or external funding.

Acknowledgments

This publication is based on research conducted under the supervision of Prof JC Cronje, in fulfilment of the requirement for the Doctoral Degree 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.

ORCID

Guillaume Nel https://orcid.org/0000-0003-1750-0960

References

1. 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.
2. 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.
3. Carrington, D., McEniery, B., & Johnston, D. (2001). PSPSM 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
4. Contreras-Vas, J.Á., Arias-Masa, J., Hidalgo-Izquierdo, V., Martín-Espada, R. (2021). Personal Software Process: A Study from an Academic Perspective. In: Costa, A.P., Reis, L.P., Moreira, A., Longo, L., Bryda, G. (eds) Computer Supported Qualitative Research. WCQR 2021. *Advances in Intelligent Systems and Computing*, vol 1345 (pp. 148-154). Springer. https://doi.org/10.1007/978-3-030-70187-1_11
5. Dweck. C. (2000). *Self-Theories: Their role in motivation, personality, and development* [Kindle edition]. Taylor & Francis.
6. Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., & Zander, C. (2006). Can graduating students design software systems?. *ACM SIGCSE Bulletin*, 38(1), 403-407). https://doi.org/10.1145/1124706.1121468
7. Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 182-211.
8. Grove, R. F. (1998). 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
9. Hou, L., & Tomayko, J. (1998). 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
10. Hu, C. (2016). 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

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

12. 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.

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

14. Humphrey, W. S. (2005). *PSP: A Self-Improvement Process for Software* Engineers. Pearson Education Inc.

15. Kusakabe, S., Araki, S., Katamine, K., & Umeda, M. (2020). *Analyzing motivation in personal software process education course with a qualitative approach*. In *2020 9th International Congress on Advanced Applied Informatics (IIAI-AAI)* (pp. 298-303). IEEE. https://doi.org/10.1109/IIAI-AAI50415.2020.00066

16. 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

17. 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). ACM.

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

19. Pando Soto, B., & Rodríguez Rafael, G. (2020). Personal Software Process (PSP) skills for the software industry in Latin America. *Industrial Data*, 23(1), 237-244.

20. Plowright, D. (2011). *Using Mixed Methods: Frameworks for an Integrated Methodology*. SAGE Publications.

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

22. Quinn, N. (2023). Adjustment of Models and Procedures for Web Development. *Infotech Journal Scientific and Academic* , *4*(2), 01-17.

23. 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

24. 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. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7883322&isnumber=7883261

25. Schach, S. R. (2011). *Object-Oriented and Classical Software Engineering* (8th ed.). McGraw-Hill.

26. 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.

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