Nathan Quirk  
CSC 402  
Midterm  
Submitted 10.18.18

Question 1: Personal Software Process

PSP - Overview

The Personal Software Process, or PSP, is a software engineering process designed to accumulate and leverage data to guide and improve the software development. Created by Watts Humphrey of Carnegie Mellon University, PSP is a personal process designed to be implemented by individual developers in order to produce high quality, defect-free programs. In Watt’s earlier work, *Managing the Software Process*, he lays out specifications for the Capability Maturity Model, which provides a framework to assess the performance characteristics of an organization’s software processes. The Personal Software Process complements and extends the ideas put forth in the CMM to the individual software engineer.

In creating the PSP, Humphrey sought to address the historically low level of quality and reliability of software products. “Historically,” he says, “few software organizations have reliably met their cost and schedule commitments.” He goes on to cite some of the better-known software engineering disasters, some of which resulted in loss of life.

Humphrey’s solution is to implement the process level by level, one at a time, in an evolutionary process that gives the programmer a chance to adopt one strategy as a basis for the next. The first level – PSP0 starts by stressing the importance of measurement – both of time spent programming and defects created. PSP1 introduces personal estimation and planning, and PSP2 stresses forethought to improve quality and overall design. Humphrey also meticulously details exactly how all this should be documented by including process scripts, forms, measures, and templates.

The first level – personal measurement – is meant to give the programmer a basis on which to form a plan for improvement. By gathering data on a series of initial projects, engineers can then evaluate their performance and gain real insights on how best to proceed. Programmers are encouraged to record the time they spend on each of a project’s phases (design, code, compile, testing) along with any interruptions that took them away from their work. Participants are also asked to carefully record any defects in the code, how these defects were injected along with how long it took to remedy them.

Armed with knowledge about his/her tendencies, the programmer can now begin to estimate how future projects will progress and how large of an undertaking it will be. Because detailed records now exist about each phase of the process, he/she can extrapolate those findings and make accurate predictions.

Finally, personal quality comes into focus with the implementation of design reviews and code reviews. Here Humphrey stresses the importance of minimizing the impact of defects in code - his primary aim being their avoidance altogether. His rationale here is that finding and fixing defect (not “bugs”!) is a time-consuming and expensive process. As such, any effort that can be made to keep them from happening is worthwhile. To this end, Humphrey urges adherents of PSP to rely on historical data to learn how the individual is likely to make mistakes. Doing this allows programmers to project the rate and density of defects injected into each phase of subsequent projects. If a defect is not prevented, Humphrey argues that it is essential to identify it as soon after injection as possible. Code-reviews are to be performed often. By examining code and identifying errors line-by-line, many issues can be resolved before they are passed on to another phase of development. Humphrey includes checklists designed to aid reviewers.

PSP - Pros

My initial reaction to the PSP was very negative. Forms upon forms, checklists, stopwatches… it all seemed designed to turn my love for programming into a bureaucratic nightmare. However, after reading Mr. Humphrey’s reasoning, there are many aspects of PSP that have merits. In fact, I think that *all* PSP’s suggestions have merit… just not all the time.

If code quality and project overruns are indeed a problem in the software industry, then it seems that programmers have one of two options: keep doing the same thing repeatedly expecting different results, or assess what has happened, what needs to change, and document those changes and their efficacy. Without an understanding of what precisely has been going wrong, efforts to enact change are just like fumbling in the dark. For this reason, I think that measurement of time management and defect injection are a great idea.

Additionally, Humphrey’s insistence on frequent code reviews also makes a lot of sense. He is certainly correct that a defect caught early is much less costly than one found during testing or production. I often miss mistakes in my own code because my eyes seem to correct the code as I read. A checklist here makes sense as it forces you to examine your own code objectively.

Additionally, PSP could eliminate friction between developers and managers over deadlines. With detailed information about prior projects, a developer could effectively make his or her case about the time and resources needed to perform their upcoming tasks. For example, if my lead developer came to me and asked if we could complete X feature in a week’s time, I could respond with: 1.) “No problem boss!” 2.) “Nope, not happening.” or, 3.) “Actually, after looking over my notes from the last time I worked on a similar project, I can see that even with minimal errors, a feature of this size and scope will take at least two weeks to implement.” Option 1, while probably what my lead wants to hear, might set him/her up for disappointment when crunch time hits. Option 2 makes me seem like I’m incompetent or lazy, or perhaps both. Option 3 keeps me sane and shows my superior that I’m basing my estimate off sound reasoning and he/she will be more likely to accept my revised timeframe. Additionally, having detailed metrics such as these could be an invaluable tool for me should I be in the market for a new job. PSP allows me to demonstrate how I’ve improved as an engineer and precisely what kind of work they can expect from me in the future.

PSP – Cons

PSP does have some drawbacks, however. I believe that it introduces far too much process into the task of software engineering, and that it would bog me down as a developer. Furthermore, I think that such extensive documentation has diminishing returns. That is, I am not convinced that it needs to be done to such a degree after a sufficient performance baseline has been achieved.

There is also the worry that employers who require that their developers use the PSP might acquire the data and use it against their employees. If a developer were honest and noted that they took several breaks while working, or injected defects frequently into their code, employers may choose to leverage this information in an employee review or salary negotiation. This of course would lead to engineers falsifying their reporting, thus negating any benefit that might have been had. Humphrey stresses that PSP data is to be private and confidential, but the PSP is meant to be used in conjunction with TSP, or the Team Software Process which aggregates PSP information. It seems likely that some confidentiality would be lost in the process of gathering individual developer’s data.

Question 2: eXtreme Programming

Extreme Programming (XP) is a software development process that builds on the Agile methodology by taking some Agile practices further while implementing additional standards that some would consider “extreme.” Extreme Programming was first developed by Kent Beck in collaboration with Ward Cunningham and Ron Jeffries in 1996 during the development of a payroll system for the Chrysler Corporation.

XP has four central *values* that drive its philosophy: communication, simplicity, feedback, and courage. *Communication* is stressed in XP as a necessary way to facilitate the movement of ideas and to keep the members of the team engaged with one another always. Because the failure of any project can usually be traced back to some breakdown in communication, keeping information flowing in all directions is a healthy goal to maintain. *Simplicity* in XP means doing “the simplest thing that could possibly work.” This ideal aims to keep teams focused, objectives clear, and deadlines met. *Feedback* highlights the need to allow feedback from all the participants be they client, leader, or developer. Finally, *courage* refers to a certain degree of confidence necessary to fearlessly commit to these principles even though they may fly in the face of accepted wisdom.

True to its Agile roots, XP is both incremental and iterative. XP recommends short periods between releases so that feedback from stakeholders in the development process can be gathered and considered. The customer generates *user stories* which describe some aspect of functionality that the developers will need to include into the project. These requirements may change, and engineers are encouraged to welcome these changes and regard them as inevitable. Teams subject these small releases to constant testing both by the team itself and by the customer. Thus, the end-user is never surprised by the product or the direction development is taking.

Testing in eXteme Programming is extremely important. In fact, *code the unit test first* is another stipulation of XP. By first creating the test, the developers have firmly established what the code is meant to do, and while the software is being written, they have immediate feedback about how their work is progressing.

In order to facilitate the rapid iteration and testing schedule, XP suggests that a customer be present to work with developers. The *on-site customer* should be on hand at all times to provide a reality check for the development team. They have the first and last say with regard to design and can make time and money saving decisions regarding project direction and scope.

Team communication in an XP project happens often, perhaps most notably during the daily *stand-up meeting*. Stand-ups should be opportunities for the entire team to connect efficiently. Only members of the team who are stakeholders may speak and standing in a circle helps to keep discussion short and to the point. Each stake-holder should describe at least three things: what was accomplished yesterday, what will be attempted today, and what problems are causing delays (Wells).

XP developers *refactor mercilessly*; that is, they constantly revisit code to reduce repetition, enhance consistency, and reduce coupling (Jeffries). Keeping coupling at a minimum also ensures that once code has passed testing, it shouldn’t need to be revisited so that it can accommodate future designs. All this testing and refactoring means that code can be integrated successfully more often. *Continuous integration* ensures that there are no compatibility issues with new code lurking around the corner. By building early and often, teams find issues that escape unit testing.

Perhaps the most controversial aspect of XP is the demand that every line of production code be written by programmers working in pairs. *Pair programming* dictates that developers sit side by side at the same computer with one “driving” and the other sitting back and “navigating.” The hope here is that any defects generated by the driver will be immediately noticed by the navigator. Downtime is also decreased as two people are less likely to waste one another’s time.

In an XP project, code is owned by anyone and everyone on the team. *Collective code ownership* means that anyone can make changes to any part of the code base. The aim here is to avoid specialists who come to dominate one area of the project. Specialists create bottlenecks that stymie progress and decrease the resilience of a team in the wake of a sudden departure or setback.

XP stipulates that overtime should be avoided if at all possible, citing excessive hours as a drain on productivity.

Finally, the project should be given a *system metaphor*. Ideally, this is a fitting comparison that neatly encapsulates how a project is to be built or function. This metaphor can then inform decisions about how the team names and labels aspects of the build. For instance, suppose the plan resembles a train, with a central node acting as a locomotive pulling the rest of the modules along with it. The team might then conceptualize each of the subsequent nodes as “train cars” coupled by “links” that run on a “track.” Adopting this metaphorical framework gives everyone on the team a common language with which to communicate ideals about any one project, facilitating cohesion and focus.

XP – Pros

There are many aspects of eXtreme Programming that I like. For one, XP’s belief in a “whole team” mentality of peers who are all equal partners in the codebase seems like a fantastic way to look at things. In fact, this is precisely what good managers do in other business sectors. It is tempting as a manager to allow people to do what they like, which often tends to be what they are best at. Pushing them into other areas forces them to grow their skillset and makes team more resilient should someone leave. If one person “holds the keys” to a certain segment of project functionality, they can essentially hold the entire project hostage. It doesn’t matter how good someone is, they will eventually need help, and if they have been working on an intellectual island, it will take a while to get someone else caught up to speed sufficiently to dig them out of a mess.

I also happen to agree that writing tests before writing code makes sense. If you can’t sufficiently articulate what you’re testing for, you shouldn’t be attempting to write the code. Having a clear grasp of what is required of any code segment means that you are more likely to write clean functional programs.

Frequent iterations and frequent integrations are also a solid philosophy. How else can you avoid nasty surprises that send the team back to the drawing board? Finding out about issues sooner rather than later is always a good thing. Frequent integration also means that everyone is apprised of everyone else’s progress. If a particular project area is becoming a trouble spot, you’ll know it when you build.

Going along with this, daily stand-up meetings that involve only stakeholders is a great way to start the day. I believe that the unsaid mechanism here is that stand-ups generate accountability. If you report the same level of progress on Friday as you did on Monday, the team can’t help but take notice. Either that developer needs help or needs to step up to the plate. Either way, no surprises. I also loved that only stakeholders are allowed to speak at stand-ups, and commentary is limited. All meetings should be done this way!

XP – Cons

One of the frequent criticisms I see often regarding XP is the emphasis on so much iteration and constantly revising scope and goals that getting a clear consensus and vision for the project can be hard. With so much being said lately about the need for more security within software development, XP seems to be at odds with the need to design a robust, clear, and actionable security plan. Having a customer on site at all times seems also to invite chaos. Promising the customer a development process where they can interject new requirements at will could lead to a bloated mess. This also seems to conflict with the principles of “you ain’t gonna need it” and “do the simplest thing that could possibly work.”

Of course, no critique of eXtreme Programming would be complete without the contentious issue of pair programming. I have tried to examine my distaste for this idea, and if I’m honest – part of me simply doesn’t want to have to explain myself to another programmer or have my ideas come under constant fire. That part of me feels that pairing me with someone somehow insults or belittles me. And that’s bad. As a professional, I should be able to work with others and not get my feathers ruffled if my ideas aren’t adopted or my methods are questioned. And of course, I shouldn’t feel threatened by someone looking over my shoulder. However, I’m probably not alone when I say that it would be hard to keep from feeling that way from time to time. Also, in a purely non-childish sense, I still think that pair programming would inhibit my creativity. I couldn’t help but be worried about my idea not working and wasting my partner’s time. So, I would undoubtedly suggest the option least likely to rock the boat. Which, I suppose is more of that simplicity that XP loves.

Pair programming forces you to think in parallel, which for some problems is great. However, there are lots of issues that I feel benefit from having a more serial, sequential approach. I saw this recently in another class. We were charged with setting up a Linux machine and we were having trouble getting the networking settings right. With one person at the computer, we all provided suggestions and one by one, we undid each other’s suggestions and eventually muddled things up so badly that we eventually just reinstalled the OS. Later that night, I returned and hammered away at the problem, working my way methodically through an elimination process that successfully fixed the issue. Some obstacles just need steady and consistent work rather than concurrent input from more sources.

Finally, pair programming seems to be highly dependent on the personalities of the people involved. I know a few people at West Chester that I would have absolutely no problem programming with, but not everyone on the team is going to be able to take direction (in the case of the driver) or be vocal with suggestions as the navigator.

Works Cited

Jeffries, Ron. “What is Extreme Programming?” (2018). Ronjeffries.com. Retrieved 15 October 2018, from https://ronjeffries.com/xprog/what-is-extreme-programming/#whole

Wells, Don. "Extreme Programming: A Gentle Introduction." Extremeprogramming.org. N. p., 2018. Web. 15 Oct. 2018.

Humphrey, Watts S. *Introduction to the Team Software Process(Sm)*. Addison-Wesley, 2000.