An Exploration of the Personal Software Process (PSP)

Brandon Maharrey
2323 Shelby Center
Auburn University
Auburn, Alabama 36849
e-mail: *maharbk@auburn.edu*

Abstract

A software process is nothing without the individual programmer. The Personal Software Process, or PSP, is a flexible, historical data-driven process tailored to teaching individuals about their own unique programming styles and even help software engineers further develop their skills in writing quality software with few defects. In this paper, I explain the Personal Software Process and how it helps individuals and teams meet deadlines while continuing to produce software which is high in quality and conforms to user specifications.

Introduction

The art of writing software has evolved quite significantly since the first computers were developed. At first, ad-hoc methods of writing software were rampant as there was no known methodology or process at that time to garner any knowledge of how to write software "better." Consequently, projects experienced a lot of variability as far as time and schedule management were concerned. Researchers and industrial leaders began to realize that software process, plans and methodologies for producing software, could help to produce accurate project deadlines, help keep software projects on budget and teach programmers to be more productive and knowledgeable of their own programming styles.

According to Humphrey, process improvement can be driven by senior management and process staffs, but beyond Carnegie Mellon University's (CMU) Software Engineering Institute's (SEI) Capability Maturity Model (CMM), improvement requires that engineers undertake software processes on an individual basis [2]. In [4], W.S. Humphrey quotes a few statistics of interest as far as successful software projects are concerned. He explains that more than half of all software projects are seriously late and over budget, and that nearly one-quarter of them are canceled without being finished. Humphrey further offers a few reasons given to him by software developers who have an inside look at their own organization's structure why they feel software projects are unsuccessful.[1] Unrealistic schedules, inadequate resources and unstable requirements are at the top of this list. After researching software processes for several years, Humphrey realized that when programmers approach software management problems in the proper way, they can generally solve these problems [4]. That is why he came up with the Personal Software Process — as a framework of techniques to help engineers and their organizations improve their performance while simultaneously increasing product quality.

The next sections introduce the notion of a software process and the rest of the paper details the Personal Software Process and how it relates to the Capability Maturity Model and the Team Software Process.

What is a Software Process?

A process, in its most basic form, is a sequence of steps required to do a job. A software process, on the same token, is a sequence of steps required to write a piece of software with known characteristics and to ensure that engineers consistently perform quality work. A software process defines methods and tasks which allow the programmer to make the most efficient use of his/her time. It helps the programmer separate routine tasks from complex tasks and establishes the criteria for starting and finishing each process step [4]. With a process, abstract tasks become more structured and more easily analyzed. The more taska are defined, the more a programmer can focus and improve in one particular aspect. A precise process will help engineers focus on the most promising areas for personal improvement [4].

The PSP is a self-improvement process that helps the programmer control, manage and improve the way he/she works. It is a structured framework of forms, guidelines and procedures for developing software, and, if properly used, can provide the data engineers need to make and meet commitments. It also makes routine elements of work more predictable and efficient.

When an engineer follows a software process, he/she is able to track his/her individual performance on a software project and understand issues related to his/her performance. If engineers take measurements at various stages of the process these measurements can be used to assess his/her abilities in a quantifiable way. A programmer, through review of his/her performance measurements, can get a good understanding of the areas in which he/she should make performance improvements.

A software process can also help individuals and teams manage their work more efficiently. If a well-defined process is in place before a project is fully underway, then this process can help guide programming tasks from one task to the next in an efficient manner. In effect, a software process enables one to quickly switch gears at the end of one task and get back to work performing the next task.

[1] The Standish Group International, Inc., 586 Olde King's Highway, Dennis, MA 02638; www.standishgroup.com.

Not only is a software process great for providing the methods and tools necessary for improving the quality of work performed by engineers, it also gives upper management and software project managers the statistics they need to ensure, or accurately predict, project deadlines. By following a process, engineers, when estimating project size and duration, can make more accurate predictions because they have recorded their individual performance characteristics and can make reasonable judgments based on past performance. With a sound process, programmers are able to make more accurate detailed plans and precisely measure and report the status of their work.

Why a Personal Software Process?

As the abstract reads, a software process is nothing without an individual programmer. It is mostly the individual programmer which is affected by process and who must practice an organization's processes. After all, it is the individual programmers that comprise the teams which are responsible for producing quality products on predictable schedules. Therefore, everything these individuals do must be planned, managed and quality-controlled. This means that almost everyone associated with software development must know how to do disciplined engineering work. This is where the Personal Software Process enters. When engineers use the disciplined approach to software development encompassed by PSP, they learn to become more efficient engineers producing quality software products on schedule. It is because of individual programmers that this *Personal* Software Process is so important — because it deals with the individual and the quality of his/her work.

Relationship to the Capability Maturity Model (CMM)

What is the Capability Maturity Model?

The Capability Maturity Model is service mark by Carnegie Mellon University (CMU) and refers to a development model elicited from actual observed data rather than from theory [5]. The writers of the Wikipedia article in [5] explain that the data were collected from organizations contracted with the U.S. Department of Defense who funded the research. When the CMM is applied to or integrated with an existing organization's software development process, it has the ability to improve that organization's software quality.

According to [5], the CMM was originally developed as a tool for objectively assessing the ability of government contractors' *processes* to perform a contracted software project and is used as a general model to aid in improving organizational business processes. The CMM is based on the process maturity framework first described in the 1989 book *Managing the Software Process* by Watts Humphrey.

A maturity model, according to [5], can be described as a structured collection of elements that describe certain aspects of maturity in an organization. A maturity model may provide, for example:

- a place to start,
- the benefit of a community's prior experiences,
- a common language and a shared vision,
- a framework for prioritizing actions and
- a way to define what improvement means for the organization.

The Capability Maturity Model involves the following aspects:

- *Maturity Levels*: The CMM is broken down into five levels. Level five is the uppermost level where processes are systematically managed by a combination of process optimization and continuous process improvement. Level one is the bottommost level where processes are hardly tracked and managed and hardly any statistical data is collected. The five levels of the CMM are summarized as follows:
  1. *Initial (Chaotic)*
     - This level is characteristic of processes that are undocumented and in a state of dynamic change, tending to be driven in an ad-hoc, uncontrolled and reactive manner by users or events.
  2. *Repeatable*
     - This level is characteristic of processes that are repeatable, possibly with consistent results. The process is not very likely to be rigorous, but gives enough structure to ensure that existing processes are maintained during times of stress.
  3. *Defined*
     - This level is characteristic of processes that have defined and documented standard processes and subject to some degree of improvement over time.
  4. *Managed*
     - This level is characteristic of processes that use process metrics so that management can effectively control the process and identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications.
  5. *Optimized*
     - This level is characteristic of processes that focus on continually improving process performance through both incremental and innovative technological changes/improvements.

- *Key Process Areas*: a Key Process Area identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered to be important.
- *Goals*: the extent to which the goals have been accomplished is an indicator of how much capability the organization has established at that maturity level. The goals signify the scope, boundaries and intent of each key process area.
- *Common Features*: common features include practices that implement and institutionalize a key process area. There are five types of common features: Commitment to Perform, Ability to Perform, Activities Performed, Measurement and Analysis and Verifying Implementation.
- *Key Practices*: the key practices describe the elements of infrastructure and practice that contribute most effectively to the implementation and institutionalization of the key process area.

PSP and CMM

It is easy to see that the Capability Maturity Model is not exactly a methodology or a specific process, but rather a framework to assess the performance characteristics of any given process. CMM is a model for *managing* a software process, ensuring that the proper steps are taken in the "correct" sequence.

The Personal Software Process, on the other hand, is a specific actualization or implementation of the CMM. For instance, a software producer that uses PSP as their software development process can be rated anywhere from level one to level five of the CMM. The activities contained within PSP are modeled after the requirements specified in the CMM and will help an organization to attain any level of the CMM that is desired as long as PSP is used consistently and religiously in the organization.

Organization Adoption Phases of PSP

The Personal Software Process has the programmer in mind. Instead of dumping all of the process on the programmer at once, Humphrey came up with a novel process-introduction sequence that gives the programmer a solid foundation from which to further assess his/her skill set. The process introduction iterations are defined and described in the following sections.

PSP0 – *Personal Measurement*

The main objective of PSP0, the baseline process, is to provide a general structural framework for writing first PSP programs and for gathering statistical data on this software that is written. The data from these initial PSP programs

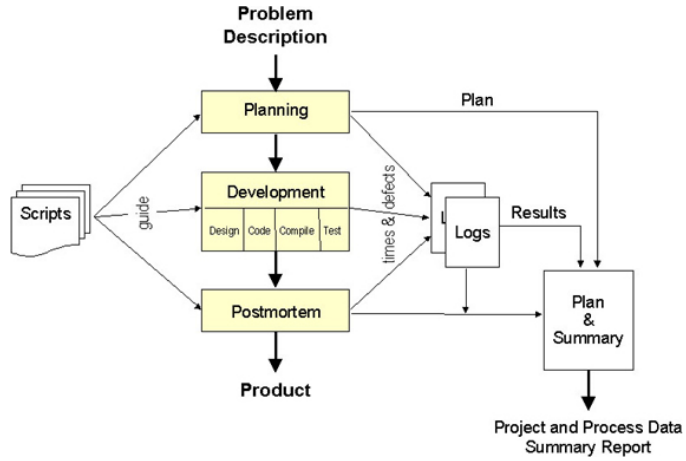

Figure 1: A simplified diagram view of PSP0's process flow. [4]

provides a comparative baseline for determining the impact of the PSP methods on future work. The PSP0 process is shown in figure 1. In the planning step in figure 1, programmers produce a plan to perform work. The four development steps come next: design, code, compile and test. The last step, postmortem, allows programmers to tally his/her actual performance and compare those values with his/her plan and produce a summary report. These steps help the programmer build a small library of statistical data from which to draw conclusions on his/her own performance characteristics. Once enough statistical data is gathered, typically after a few programs have been written or after a few projects, programmers can analyze their own performance data to manage and improve their individual personal process. Humphrey advises programmers to use the postmortem phase to think about their own performance data and determine where and if any improvements can or should be made.

Humphrey, in [4], painstakingly details several forms which he believes the PSP programmer should find useful. He explains that forms are helpful because they provide guidance, even for mundane tasks. With a planning form or a checklist, the programmer does not need to guess what step he/she should perform next — it is all laid out in the planning form. Humphrey further explains that good forms are hard to develop and that they should be periodically reviewed to ensure that they still meet the needs of the programmers in the organization.

PSP0 has two measure which programmers will initially use to compare performance on various different projects: the time spent per phase and the number of defects found per phase.

PSP0 Time Measurement

The time spent per phase is a record of the actual clock time spent in each part of the PSP process — designing, coding, compiling and testing — and consists of some basic time-

recording metrics. Those basic metrics should include, at a minimum, the following:

- the project or program on which the programmer is working
- the process phase for the task at hand
- the date and time the programmer started and finished working on the current task
- any interruption time
- the delta (or net) time the programmer has worked on the current task
- any comments

With this data, the programmer can visually see and even possibly graphically represent the time he/she spends in various stages of the process. Armed with this data, the programmer can focus on problem areas and prepare for improvement. In [4], Humphrey stresses the importance of keeping accurate timing in each phase, even giving the suggestion of using a stopwatch to record interruptions in each phase. Small increments of interruption time in a project can accumulate and invalidate the tallied results in the postmortem phase. This is why accurately maintaining the time spent in each phase is a critical aspect of PSP0.

PSP0 Defect Recording

If a programmer does not know his/her problem areas, how will he/she advance his/her own skills as a software developer? Defect recording is a way to determine problem areas for a programmer. Like other aspects of PSP, defect recording comes with its own form. In this form, a programmer can easily input pertinent data pertaining to defects found during development. The form detailed by Humphrey in [4] contains the following useful constructs to allow the programmer to enter defects:

- *Date*: This field should contain date the defect is found.
- *Number*: This field should contain the defect number. Each new defect should receive its own unique number.
- *Type*: This field should contain the defect type. The defect types are defined thus:
  - *Documentation*: comments, messages
  - *Syntax*: spelling, punctuation, typos, instruction formats
  - *Build, Package*: chance management, library, version control
  - *Assignment*: declaration, duplicate names, scope, limits
  - *Interface*: procedure calls and references, I/O, user formats
  - *Checking*: error messages, inadequate checks
  - *Data*: structure, content
  - *Function*: logic, pointers, loops, recursion, computation, function defects
  - *system*: configuration, timing memory
  - *Environment*: design, compile, test, or other support system problems
- *Inject*: This field should contain the phase in which the programmer believes he/she injects the defect.
- *Remove*: This field should contain the phase in which the programmer removes the defect.
- *Fix Time*: This field should contain the length of time it takes for the programmer to find and fix the defect.
- *Fix Reference*: If a defect were fixed improperly, this field should contain the number of that improperly fixed defect.
- *Description*: This field should contain a description of the defect found and fixed. When the programmer reviews his/her own data in the postmortem phase, this information will be very useful in determining where and how the programmer's personal process can be improved.

With this data, the programmer can get a better understanding of when and where he/she is most likely to inject defects into development.

PSP1 – *Personal Planning and Estimation*

When developing software, especially when developing very large software systems, it is very important to have a plan before diving in. Plans help to keep projects on schedule and programmers on task. Planning, when done properly, can save future time and effort, and almost equivalently, money and other resources. Most of the time, management of software projects wants projects to be delivered by a particular deadline and on budget, but how do software engineers give accurate estimates of needed time and resources? This is the reason accurately estimating software project size is extremely important, for continued job security for both programmers and management.

In [4], Humphrey discuses proxy-based estimation and how it relates to software estimation. In its most basic form, proxy-based estimation involves estimating by making comparisons of the planned project with previous completed projects. When a software project is broken down into its most basic parts, these parts can be compared with other previously written parts to get an idea of the size and effort required to produce this new part. Historical data is used as the proxy to estimate the project size and required effort.

Since estimation is such an important part of any software development organization, it is important for the size estimate to be as close as possible to the completed project's

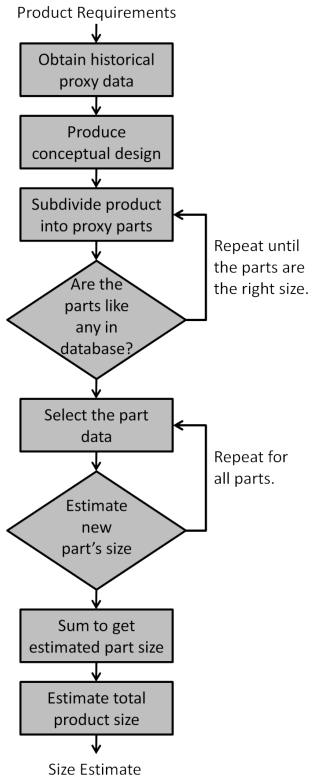

Figure 2: A state chart for estimating software size using a proxy. [4]

actual size. Since historical data is usually a good indicator of future performance, this is a good proxy to use for size estimation.

In figure 2, a state chart for estimating software size using a general proxy is given. For a software project, product requirements are fed into the figure from the top as input, and, after possibly several traversals of the state chart, an estimate of the size of the software project is given at the bottom of of the figure as output. In the figure, "database" refers to the programmer's repository of personal historical performance data gathered from completed time measurement forms. To help with estimating project size, proxies should have a close relationship to the resources required to develop the product [4]. This is why historical data is assumed to be a good proxy because comparing future parts with historical parts can help the programmer to better understand the planned product and make more accurate judgments about its size.

Estimation Bias

Estimation bias is one root cause of estimation errors. Firstly, a programmer may always estimate a certain percentage higher or lower than the actual amount of work he/she performs. Secondly, estimation accuracy will fluctuate around a mean by a certain standard deviation. Therefore, it is desirable to eliminate this estimation bias to gain more accurate size estimations. This is another reason why historical data is very important. If a programmer knows that he/she under or over estimates, then his/her reported estimate to management may be adjusted accordingly. Over time and with enough experience, programmers' estimates should gradually improve.

PSP2 – *Personal Quality*

PSP2 adds two new phases: design review and code review. Since product defects are the main problem with production software, defect prevention is the main goal of PSP2. At this stage, the organization's engineers are able to use historical data to estimate a project's size, and management is able to plan accordingly. Now the focus is on defect prevention to decrease development time and produce higher-quality software products. Humphrey designed the PSP phases to allow early detection and removal of defects. Humphrey's idea was that it is much more economical and effective to remove defects as close as possible to the time of injection, thus, engineers are usually encouraged to conduct personal code reviews in each phase to assess the quality of his/her work.

Reviews are very important to the PSP software process and come in many different flavors. An *inspection* is a structured team review of a software product. *Walk-throughs* are less formal than inspections and generally follow a presentation format with audience members pointing out issues and/or making comments. In a *personal review*, the individual programmer examines a product that he/she personally developed. The objectives of these different types of reviews are the same, however: to find as and to fix as many defects as possible. In essence, design and code reviews are to save as much future time and effort as is possible.

Design Reviews

Some problems are difficult to discern from reviewing source code alone. A great example of this phenomenon are security issues [4]. Design review, according to Watts, also increases the likelihood of producing smaller, neater and more understandable programs and decreases the time spent debugging incorrect logic. Reviewing designs before implementation allows the programmer to see and incorporate potential design improvements which can considerably save time and effort. In [4], Humphrey suggests a few helpful design review guidelines:

- Produce designs that can be reviewed.
  - What good is reviewing a design which can not be reviewed? The purpose of a design is to have an actual physical document which organization programmers can review for defects before any implementation begins to take place.
- Follow an explicit review strategy.

  - Humphrey suggests following the same review strategy for projects, changing the strategy if necessary for new projects. Following the same review strategy each time a design review takes place can help ease confusion. Humphrey details a Design Review Checklist which begins with the "big picture" and gradually gets more detailed.

- Review the design in stages.
  - When a design is reviewed in stages, programmers can ensure that the simpler design meets the needs of the project rather than examining the whole design at once and possibly missing some problem which would have been discovered had the design been reviewed in earlier stages of its development.

- Verify that the logic correctly implements the requirements.
  - Faulty designs beget faulty implementations and hence produce more defects. This is one of the main reasons why design reviews are such an important part of software engineering.

- Check for safety and security issues.
  - As stated above, some problems are difficult to discover from reviewing source code alone. Safety and security issues are more readily visible when viewed from the design perspective.

Code Reviews

The main goal of a code review is to be sure all of the details are correct. Like design reviews, code reviews also come with a form design by Humphrey. Code reviews require a complete design and a source-code listing of the program to be reviewed. It is also necessary for individual software engineer reviewers to possess the most up-to-date copy of the code for review. A C++ code review checklist designed by Humphrey is detailed as follows:

- *Complete*: Verify that the code covers all of the design
- *Includes*: Verify that all includes (included libraries) are complete.
- *Initialization*: Check variable and parameter initialization at program initiation, at the start of every loop and at every class/function/procedure entry
- *Calls*: Check function call formats.
- *Names*: Check spelling of variable names and their usages.
- *Strings*: Check that all strings are identified by pointers and terminated by NULL.
- *Pointers*: Check that all pointers are initialized NULL, deleted only after new and always deleted after use.
- *Output Format*: Check the output format such as line and character spacing.
- *() Pairs*: Ensure that all parentheses are properly matched.
- *Logic Operators*: Verify the proper use of ==, =, || and so on and check every logic function for parentheses.
- *Line-by-line check*: Check every line of code for instruction syntax and proper punctuation.
- *Standards*: Ensure that the code conforms to the coding standard.
- *File Open and Close*: Verify that all files are properly declared, opened and closed after use.

To build a checklist to suit an organization's programmers' needs, it may be necessary to change some of the items listed above to reflect those issues most relevant to the programming language used. Also, personal checklists can be developed that focus on a programmer's weaknesses. This is another method which can be used by programmers to improve his/her own performance.

PSP as Part of the Team Software Process

The Team Software Process (TSP) was developed to address the commitment, control, quality and teamwork problems faced by most software development teams [4]. Dyer in [1] defines a team as a group of at least two people who are working toward a common goal for which each person has a specific role and the team members must all contribute and support each other to be successful. In [3] Humphrey notes that many studies have shown that successful teams share the following characteristics:

1. The team members are all skilled and capable of doing their job.
2. The team has an aggressive and important goal that the members must cooperatively finish.
3. The team members believe that the goal is achievable, and they each have a defined role in achieving that goal.
4. The team members have a common process and plan that guides them in doing the work and in tracking their progress.
5. The team leader supports and protects the team and keeps the members informed of management issues.

The TSP launch process helps groups develop these characteristics, but personal motivation is also a strong contributing factor.

TSP Launch Process

After PSP training has ended, TSP teams are guided through the TSP launch process by a trained and qualified TSP coach. During launch, management and the team hold a series of ten team, management and customer meetings. The details of the meetings are described as follows:

1. In meeting 1, management and the customer meet with the team to explain the scope of the project, what management expects from the team and what the customer wants the software to do. Management outlines their goals for the team and describes their critical needs for the project including any trade-offs the customer is willing to accept to consider the project successful.

2. In meeting 2, the team defines its goals and the members select their personal team roles. Roles may include planning, process, support, quality, requirements, design, implementation and testing, but there could be more or less depending on the size and need of the team.

3. In meeting 3, the team produces its initial product concept decides on a development strategy and defines the process needed to support the goals and strategy.

4. In meeting 4, the team produces an overall plan for the job. The team reviews the job and produces a high-level plan which covers all phases and project deliverables.

5. In meeting 5, the team produces a quality plan.

6. In meeting 6, the team members make detailed personal plans for the next several weeks. Each team member takes on responsibility for a particular aspect of the project and reviews every other team member's workload to produce a plan which balances the workload among the team members.

7. In meeting 7, the team produces a project risk assessment and assigns team members to monitor and mitigate key risks.

8. In meeting 8, the team prepares a plan presentation for management.

9. In meeting 9, the team, team leader and coach meet with the management group that attended meeting 1.

10. In meeting 10, the final meeting, postmortem occurs during which the team reviews the launch process, records improvement suggestions, ensures that all important decisions and data are recorded and assigns responsibility for any outstanding follow-up items.

By working together on a TSP, the programmers are able to giving management and the customer a good idea of how much effort would be involved in producing the desired software product. This can eliminate wasted time and effort because potential problems can be identified up front instead of later when time and resources have already been wasted.

TSP Coach

Team leaders are important, but a team coach will help programmers to hone in on their individual weaknesses, making the team as a whole much stronger. Coaches focus on supporting the team and the individuals on the team while maintaining independence. The TSP coach has three objectives:

1. to motivate superior performance,

2. to insist on a dedication to excellence and

3. to support and guide individual and team development.

Coaches are first instructed in PSP skills and best practices and then trained as a TSP coach. Humphrey suggests organizations have at least one TSP coach for every four or five teams that they support. He argues that this gives coaches enough time to both launch and support their teams and to spend time with each team member. Programmers new to TSP teams usually need a small amount of guidance from the TSP coach, but, as they become more knowledgeable of the process, the team members need the support of the TSP coach less and less.

Concluding Remarks

In this paper I have presented an overall summary of the uses and practices of the PSP and how it relates to the TSP. We have seen that the PSP is a flexible, historical data-driven approach to software programming which can be tailored to an individual programmer's programming style. Aside from being a very flexible process, PSP has also shown that processes alone can help shape the quality of a software project. PSP has shown us the importance of having a plan before actually implementing a system and has paved the way for using personal software processes in a team environment with a TSP coach as a helpful guide.

References

[1] J. L. Dyer. Team research and team training: A state-of-the-art review. *Human Factors Review*, pages 286, 309, 1984.

[2] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley, Reading, Mass., 1989.

[3] Watts S. Humphrey. *Winning with Software: An Executive Strategy*. Addison-Wesley, Boston, 2002.

[4] Watts S. Humphrey. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley, Upper Saddle River, NJ, 2005.

[5] Unknown. Capability maturity model. Published on the web, Retrieved April 2010. Retrieved from http://en.wikipedia.org/wiki/Capability_Maturity_Model.