Learning and Applying the Personal Software Process

Robert Bentall shares his experiences from learning to measure his own performance.

In the mid-1990s, I spent a couple of years as a postgraduate student at the Royal Northern College of Music in Manchester (UK). I was focused on becoming a professional horn player, and the accepted route is to go through a conservatoire training.

As I did this, it became clear that the most important part of my training was learning to listen – to myself, my teachers, my peers, and those in the profession I wished to emulate. This relationship between between practice, observations, and goals is shown if Figure 1.

It's this awareness that is the foundation of performance improvement. Without it,

- I couldn't tell how well I was doing
- it became much harder for me to refine my goals
- my practice was less effective.

As a musician, I figured out how well I was doing by listening. But as a software engineer, I don't have access to such direct feedback. We engineers need tools to gather data on how well we are doing, models to help us interpret that data, and techniques we can use to solve common problems.

That is the essence of the PSP training. It teaches the fundamentals of performance management as applied to the individual software engineer. By undergoing the training, we engineers can learn how to

- gather data about our performance
- compute measurements from that data
- interpret those measurements
- use different techniques to improve performance.

Once we've grasped these concepts, we are in a position to determine which techniques and methods work best for us and to continually improve our performance.

I began the PSP training in July 2011 and completed it about six months later. The course was both challenging and rewarding, and it has already started to pay dividends. The techniques I learned can be applied to good effect on a day-to-day basis, and the breadth of the course has provided a solid base for my future development.

History

PSP was the brainchild of Watts Humphrey (1927–2010).[1] He spent most of his career at IBM as a senior executive, and after retiring from IBM, he moved to the Software Engineering Institute (SEI) at Carnegie Mellon University. While at SEI, he was instrumental in the development of the capability maturity model. He started applying the model's principles to writing software, and over time, this grew into the PSP. His obituary, found on the SEI website, contains a wealth of information on his life.[2] An oral history, recorded with Grady Booch, gives us an insight into the breadth and depth of his expertise.[3]

ROBERT BENTALL

Robert Bentall is a software engineer for an oilfield services business. He works on the development of reservoir engineering simulators using Microsoft C# and C++ / CLI. He can be contacted at robertbentall@supanet.com

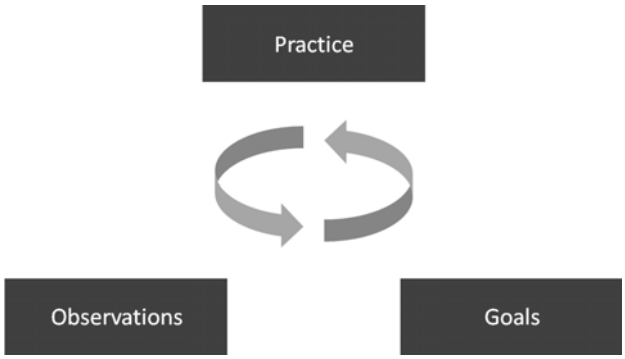


Figure 1

His intention was to apply personal quality management techniques to the development of 'module-sized' programs so that the work of each individual software developer would be of a very high quality.

Course structure

The course is structured as two 1-week parts, with report assignments at the conclusion of each part. Part 1 covers measurement, planning, and estimation. Part 2 addresses software quality, design, defect detection and removal techniques, and process management.

Each course includes lectures, programming exercises, and data analysis. The mix of theory, practice, and analysis is crucial to learning to apply the techniques and figuring out how to interpret the results.

The PSP course is based on a set of training processes; each process version builds techniques incrementally. Participants must write one or more programs using each process and undertake data analysis exercises at the mid-point and end of the course.

The processes have a simple linear structure comprising a sequence of up to eight steps:

- planning
- design
- design review
- coding
- code review
- compilation
- test
- postmortem.

This provides the framework to introduce the different techniques that are taught on the course (see Table 1).

The course covers a lot of ground in a short period of time, but because participants are always practising the techniques learned in previous versions of the process, it doesn't take too long to master them.

Scripts are used to guide workflow and these provide operational definitions [4] of each version of the process. This helps prevent important tasks from being missed and makes it much easier to apply the techniques. It also helps ensure consistency of data capture.

Table 1

Techniques	Process Version					
	Process discipline and measurement		Estimation and planning		Quality management and design	
	PSP 0	PSP 0.1	PSP 1.0	PSP 1.1	PSP 2.0	PSP 2.1
Process measurement	✓					
Coding standard process improvement proposals, size measurement		✓	✓	✓	✓	✓
Size estimation, test reports			✓	✓	✓	✓
Task planning, schedule planning				✓	✓	✓
Design reviews, code reviews					✓	✓
Design templates						✓

The first program I wrote used the PSP 0 process. This was intended to be a relatively small step from my existing development approach, and it introduced the basics of personal process measurement. By the time I got to PSP 2.1, the process had become a little more complex. But because the techniques had been built incrementally, it was still straightforward.

A common mistake when looking at the PSP is to think that it reduces the developer to 'development by checklist' and that it enforces the use of a waterfall. Both concerns are misplaced. It's straightforward to work iteratively within each version of the PSP, and while the use of scripts to guide workflow may seem alien, I found it liberating – I no longer had to worry about forgetting to do something. It freed me up to concentrate on the problem I should be solving.

What are the options for taking the course?

There are three options for people wanting to take the course:

- attend one of the SEI courses [5]
- obtain the materials and self-teach [6]
- work with a coach.

For me, the prospect of an SEI course was remote, so I opted to work with a coach. Working with a coach proved to be very interesting and worthwhile. Although anyone can just follow the book and materials, it is much easier with support and external feedback.

The course materials are readily available. [6] The course textbook, *PSP – A Self-Improvement Process for Software Engineers*, [1] provides all the theory needed, and an open-source tool, the Process Dashboard, makes metrics collection easier. [7]

Completing the course took me about 300 hours, including the time I spent writing extra programs and reworking several of my analyses. The target time for completing both parts of the course is 150 hours, and there are significant benefits to be gained just by completing the first part. This means that for an investment of 50 to 100 hours, we can gain much of the technique and insight needed to introduce personal quality management into our work.

Lessons learned during the PSP training

The course imparts a large amount of valuable information, and I know I will continue to use the techniques I've learned. The lessons I have learned can be divided logically into two groups:

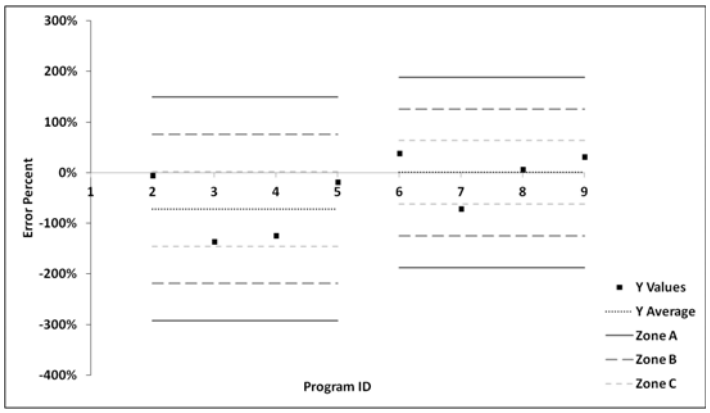
- techniques
- conclusions about my performance.

Techniques

The course provides an opportunity to practise new techniques in a safe environment:

- a structured approach to estimating the size, quality, and cost of software deliverables

Figure 2



- effective project tracking and status reporting
- classification and analysis of defects
- performing personal design and code reviews
- program verification
- measuring and managing quality.

Each one of these can be applied in isolation to target a specific problem. This means that once I'd learned the techniques I could introduce them progressively into my daily work. Although all the techniques are valuable, the design and code reviews stand out as surprisingly powerful.

My conclusions about my performance

It's important to be cautious about the conclusions we draw with respect to our own performance. I was learning new techniques and drawing conclusions from small sets of data. Both factors should cause us to be tentative in our interpretations, so please keep this in mind as you read my conclusions.

I found that my size and time estimation accuracy improved a little during the course (Figure 2, which shows size-estimating errors for Programs 2 to 9 and Figure 3, which shows time-estimating error for Programs 1 to 9).

I've used control charts [8] to show changes in my performance during the course. These were invented in the 1920s by Walter Shewhart and made popular by his pupil W. Edwards Deming. They can be used to understand the variability within a time series dataset, which enabled me to identify where significant changes have occurred. At least 5 points per process are needed, so these results are intended only for illustration. For more details on use of control charts, refer to the book by Don Wheeler. [9]

I have split the data so that Programs 1–4 are considered as one process and Programs 5–9 are considered as a separate process. The logic for this is that the earlier programs were written using different versions of the PSP but Programs 5–9 were all written using one version, PSP 2.1. I calculated the process mean and control limits using the normal rules. Zone A corresponds to ± 3 standard deviations, Zone B to 2, and Zone C to 1.

For the size estimation, I made these tentative conclusions:

- I was going from continually underestimating by a large amount to estimates balanced around zero.

Figure 3

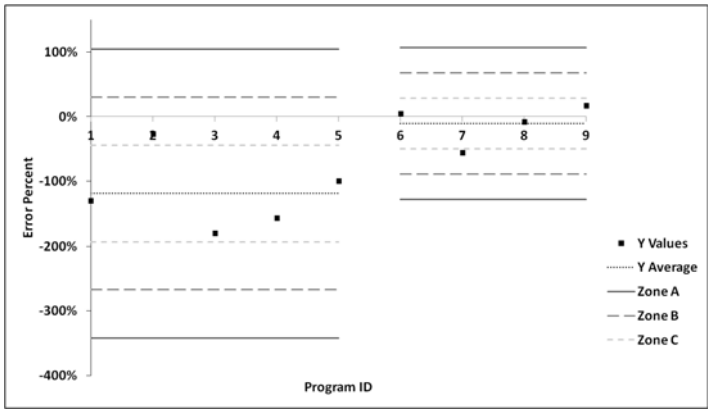
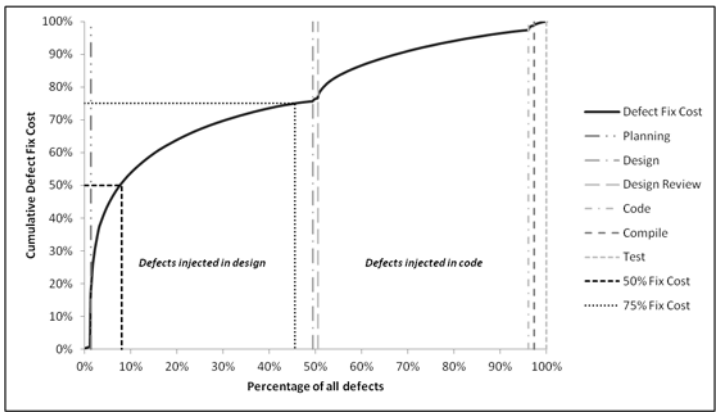


Figure 4



- The variability of estimates was improving slightly: Zone C was moving from ±85% to ±60%.

For time-estimating accuracy, the tentative conclusions were that I was improving a little:

- My process average improved from -120% in the first five programs to -10% in the second four programs.
- The variability of estimates improved: Zone C is ±75% for first five programs and ±40% in last four programs.

These results do not mean that my size- and time-estimating accuracy will continue to improve in the future, merely that I improved as I progressed through the programs.

There were some interesting patterns in the defects I injected. I rarely made significant errors in the structure of my programs (e.g., class design, relationships between classes, etc.), but the most expensive ones were design errors, typically related to algorithm structure. I also found that 50% of the total defect fix cost was due to only 8% of the defects (as seen in Figure 4).

This chart is derived by taking the cost of fixing each defect and ordering it by phase injected and descending fix cost. The fix cost values are then plotted cumulatively as a percentage of the total fix cost for all defects. Vertical lines mark phase boundaries. Fix cost is the time taken to fix a defect, from the point at which the defect is first found to the point at which the fix is completed.

This profile is really interesting, because once I recognized the pattern, it became possible to design filters to reduce or even eliminate the issue. For example, I found that in the earlier programs, I was not specifying error behaviour accurately enough to allow me to code it without error. I introduced a check into my design review to prompt me to validate the expected error behaviour. This went some way to trapping these defects earlier in the process.

More generally, the PSP training emphasizes the use of individual developer reviews of design and code as a very powerful defect-detection technique. Personal defect data provide an excellent basis for tuning reviews to find the kinds of defects typically injected.

Although individual developer reviews are not as effective as properly conducted team-based inspections, they can still find a very significant percentage of the defects in a software artefact. A 2006 study by Cisco Systems found that individual developers inspecting their own code would find approximately 50% of the defects that were found by the team review. [10]

In PSP, 'Yield' is used to measure the effectiveness of defect-removal processes and is the ratio of defects found divided by total defects in the system at that point in time. I found that I was consistently getting 70% to 80% yield before the compile stage, meaning that only 20% to 30% of the defects were being found during compile and test (Figure 5).

In PSP, a defect is counted each time an artefact from a previous phase needs to be corrected. This could be an error in a design document, in test data, or in code. This chart therefore tells us the percentage of errors that are corrected before compile. A similar chart can also be plotted using the fix cost of each defect, which allows derivation of a fix cost profile for the development process.

Again I've used a control chart to show changes in my performance during the course. Looking at this data, my performance seemed to stabilize fairly quickly. I've opted to treat Programs 3–9 as one process, purely because that seems the most natural separation based on the precompile yield.

I had less success with the design verification techniques. Although they are good if they can be mastered, I found they didn't significantly increase my defect detection rate but did consume significant effort. I suspect that with further practice on my part I will become a lot more efficient and effective.

PSP has a rich set of metrics that provide insight into our development habits. The examples given are just a small subset that I've used to illustrate some of the lessons I have learned.

How I apply lessons learned

The wonderful thing about the techniques is that they are all independent. I can employ just the techniques I need, when I need them. This allows me to introduce them to my work progressively. For example,

- I started tracking most of my projects using the techniques taught on the PSP course as soon as I had learned them. They provided a simple graphical approach to demonstrate progress against plan.
- I developed and used a process for guiding defect resolution with a client. This facilitated clear communication, more accurate plans, and better management of the project.
- when I'm estimating cost, size, and schedule, I use the PSP techniques to do the estimates.

Going beyond the PSP: the team software process

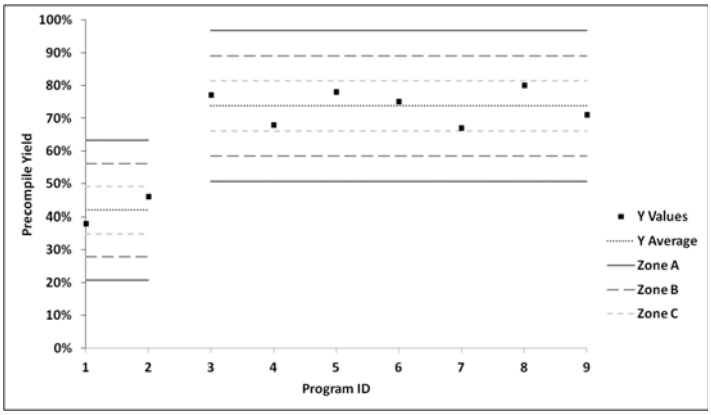
One of the interesting points identified by Watts Humphrey was that after completing the course, most PSP trained developers tended not to apply techniques in their day-to-day work. This was primarily because of the levels of self-discipline required and the environmental challenges faced by the developer. [11] To address this finding, he went on to develop the Team Software Process[SM]. [12] This process took the techniques of the PSP and applied them to the work of software teams. When applied correctly, the techniques have improved the quality, cost, and schedule record of software development teams. [13]

Closing thoughts

It's been a big investment to complete the course. However, the course teaches something pretty fundamental. It's about learning to listen to ourselves as software engineers, and about understanding how we can use the information we hear to improve our performance. In some areas of my performance, I am starting to see the benefits, although it has taken time.

I know of no other training program that provides a consistent, complete grounding in pretty much all the tools needed to improve performance. In that sense, I suspect the course is unique.

Figure 5



Achieving 'master level' performance in any field takes a lot of effort. The oft-quoted number is 10,000 hours of practice over 10 years.[14] Viewed in that light, 300 hours doesn't seem so bad. Perhaps we shouldn't be surprised that becoming a good programmer takes a lot of time.

In one of his interviews, Watts Humphrey noted that he was taking techniques already in existence and scaling them to apply to the individual software engineer. [15]

What has been fascinating as I've gone through the training is that I've become much more aware of this. Many of the problems that software developers face on a day-to-day basis have already been solved. We just need to recognize this and understand how to apply the solutions.

My advice? Do the course.■

References

[1] Humphrey, Watts S.: *PSP: A Self-Improvement Process for Software Engineers*, Upper Saddle River, New Jersey, USA, Addison-Wesley Professional, (2005).

[2] Carnegie Mellon University: http://www.cmu.edu/news/archive/2010/October/oct28_wattshumphreyobit.shtml (accessed 10 May 2012).

[3] Pearson Education, Informit, An Interview with Watts Humphrey: http://www.informit.com/promotions/promotion.aspx?promo=137746 (accessed 10 May 2012).

[4] W. Edwards Deming, *Out of the Crisis* (1st ed.), Cambridge, MIT Press (2000), 276.

[5] Carnegie Mellon University: SEI Training, http://www.sei.cmu.edu/training/?location=main-nav&source=1358 (accessed 11 May 2012).

[6] Carnegie Mellon University: SEI Training, http://www.sei.cmu.edu/tsp/tools/student/?location=tertiary-nav&source=5784 (accessed 2 July 2012).

[7] The Software Process Dashboard Initiative, http://www.processdash.com/ (accessed 11 May 2012).

[8] Wikipedia, Control chart, http://en.wikipedia.org/wiki/Control_chart (accessed 11 May 2012).

[9] Wheeler, Don: Understanding Variation-The Key to Managing Chaos, Knoxville, Tennessee, USA, SPC Press Inc. (1993).

[10] Cohen, Jason: Best Kept Secrets of Peer Code Review, Beverly, Massachusetts, USA, SmartBear Software (2006), as cited in Oram, Andy and Wilson, Greg (eds.), *Making Software: What Really Works, and Why We Believe It*, Cepastopol, California, USA, O'Reilly Media Inc. (2010) 336.

[11] Pearson Education, Informit, An Interview with Watts Humphrey: http://www.informit.com/articles/article.aspx?p=1625324 (Accessed 2 July 2012)

[12] Carnegie Mellon Software Institute, Team Software Process, http://www.sei.cmu.edu/tsp/ (accessed 11 May 2012).

[13] Jones, C., *Software Engineering Best Practices*, New York, McGraw-Hill (2009) 293-298.

[14] Ericsson, K.A., Krampe, R. Th., and Tesch-Romer, C.: 'The Role of Deliberate Practice in Expert Performance', *Psychological Review* (1993) 103, 363-406.

[15] Pearson Education, Informit, An Interview with Watts Humphrey, Part 21: The Personal Software Process, http://www.informit.com/articles/article.aspx?p=1614506 (accessed 11 May 2012).