11

PERSONAL SOFTWARE DESIGN PROCESS PORTFOLIO

Reference Materials

Watts Humphrey. *Introduction to the Personal Software Process*. Addison Wesley Professional. 1996.

Online Resources

www.gnu.org: GNU Foundation. *GForth Reference Manual*
Software Engineering Institute: http://www.sei.cmu.edu/tsp/psp.html
PSP Resources Page developed by the Software Engineering Group at the University of Karlsruhe: http://www.ipd.uka.de/mitarbeiter/muellerm/PSP/

The purpose of the Personal Design and Software Process Portfolio (PDSPP) portion of the course is to introduce professional concepts of personal time management and continuous improvement. My experience is that students (and faculty, including me) often have poor time management skills. This is due partly to difficulties in budgeting time and partly to the nature of research. Regardless, we all can benefit from occasionally taking strict accounting of our time to discover ways we can improve.

There is a very practical aspect to this exercise. This text is meant for seniors in computer science who will soon be looking for a job. I have had many students tell me that showing recruiters a portfolio drawn from their classes is an extremely effective tool in job interviews. Professionally, a complete portfolio of a year's work is a powerful message at annual review time.

The outline of a portfolio is described in this chapter. The goal is to present practical advice on developing yourself as a professional in computer science. Many of the ideas for this chapter come from the concepts put forth in Humphrey's *Personal Software Process* or PSP as it is generally referred to (Humphrey 1996). However, PSP does not address issues relating to system design. My experience has been that undergraduate students

rarely have experience in developing an entire system and hence need help in this area. For this reason, the text must address both the software process and the design process. This chapter develops time management portions of PDSP. Most milestones will have unique design concepts to be practiced in the milestone.

11.1 TIME MANAGEMENT FOR COMPUTER SCIENCE STUDENTS

It is a simple fact that there are not twenty-four usable hours in a day, that a three-credit university course is more like three hours a week, and you can't sit down Thursday night to write a program due Friday morning.

11.1.1 There Are Not Twenty-Four Hours in a Day

Although the clock says there are twenty-four hours in a day, you cannot work twenty-four hours a day. For health's sake, you need seven to eight hours of sleep a day and about three hours for personal hygiene, meals, etc. It really looks like you can only have fourteen to fifteen hours per day to spend as you want. In a perfect world, you would have $7 \times 15 = 105$ hours to spend as you like.

Anyone who has been associated with college students knows that students do not spend Friday night, Saturday, or most of Sunday doing school work. And that is as it should be. However, this effectively removes thirty more hours from your time budget: you now have $5 \times 15 = 75$ hours per week to allocate.

11.1.2 A Three-Credit Course Is Nine Hours

A three-credit university course is really nine hours because you are expected to spend three hours in the classroom and two hours per hour of class preparing for class. The preparation may be reading or homework to hand in or it may be working on a program. Therefore, a fifteen-hour load requires that you spend forty-five hours per week. Although the "standard" work week is forty hours, you can expect to spend a bit more than that each week. Notice, though, that you have seventy-five disposable hours per week.

Unfortunately, most of us cannot be so efficient as to use all the time in-between class. So, let's say that you spend your days in class (fifteen hours) and that leaves thirty to forty-five hours at night; that's six to seven hours per night. My experience is that you cannot profitably do that, either. After a long hard day, you are not an efficient user of time. The suggestion is that you find a block of time during some days to get the nighttime load down to something manageable, say four hours a night.

Notation	Explanation
nnn	Class number
CP(nnn)	Preparing for Class nnn
C(nnn)	Attending Class nnn
D(nnn)	Designing programs for Class nnn
CT(nnn)	Coding and testing for Class nnn
U(nnn)	Documenting programs for Class nnn

Figure 11.1 Notations for Documenting TUL Form

11.1.3 The Key Is Finding Out How You Spend Your Time

The key to any science is the metrics used to measure activities. The Time Utilization Log (TUL)* is to be used to record how you're using your time. It has the class times† already accounted for between 8:00 a.m. and 5:00 p.m. The rest of the day is blocked off in half-hour increments. The assumption is that you stay up late and get up late. Early risers should modify the grid.

CASE 59. TRACKING YOUR TIME USAGE

You are to insert your schedule onto the spreadsheet using the codings shown in Figure 11.1. On the right side is a place to list major activities that you typically have during the week that are not accounted for by the set notation with a code of your own. I would suggest that you code activities that can take up to a half-hour.

During the week, keep track of your time and activities and enter them into the log. If you skip a class, be honest, and remove it from your schedule for that week. However, skipping class is a slippery slope to disaster. In my classes, you are allowed to skip a maximum of the number of hours for the course. After that, it's an automatic F.

11.1.4 Keeping Track of Programming Time

The Time Recording Template (TRT) (note that this form is one of many in the Excel packet) is to be used for recording time spent "programming," which includes time to design, code, test, and document programs. As with

* This is not a standard part of PSP. However, I find most students cannot account for their time and this is a good place to start.

† Clemson has two schedules: Monday–Wednesday–Friday and Tuesday–Thursday. Alter this schedule to fit the local situation.

the TUL, the TRT keeps track of your time but the TRT is used to determine how you use your time when developing programs. You should have an entry for every time you program for this class. The columns should be self-explanatory except the phase column. The phase designations are listed on the right of the sheet. You may add your own phase codes and explanations if you think them necessary.

11.2 DISCOVERING HOW LONG IT TAKES TO WRITE A PROGRAM

It is an unfortunate truth that developing programs that are rigorously tested is a time-consuming business. Experiments in my classes have validated a long-held folk theorem: programmers produce approximately 2.5 debugged lines of code per hour. The best I have seen in my classes is right at 3 lines per hour. In this course, you may well have several milestones that are 100 to 120 lines of code. At 2.5 lines of code per hour, you will spend approximately 40 to 48 hours to complete such a milestone. Even if given 2 weeks to complete the milestone, you need 20 to 24 hours per week.

In order to properly plan for your work, you must know how many lines of code a particular project requires *before* you start. How can we estimate how many lines of code a problem may take?

To begin with, what constitutes a line of code (LOC)? Certainly not blank lines and comments. Clearly, there are two categories of lines: data declaration and algorithmic. Fortunately, data declarations don't change much once they're written; however, they do take time. For very simple declarations, the time is negligible. You must learn to judge where the breakpoint is between *simple* and *complicated* data declarations.

Algorithms are a different problem entirely. The problem is that most real-life algorithms are very long and complex (see the 100- to 120-line milestone quoted above). Any long and involved algorithm must be decomposed into smaller blocks of code until one can accurately "guesstimate" how many lines are needed. An organized way to analyze an algorithm is to produce a work breakdown structure (WBS) (Figure 11.2).

Task Number	Description	Inputs From	Outputs To	Planned Value	Earned Value	Hours	Cum. Hours	Date

Figure 11.2 Work Breakdown Structure Headings

11.2.1 Problem Decomposition

Before explaining the WBS, we should first explore how one goes about decomposing a problem. This process relies on your having gained a certain level of expertise: you should be able to determine what parts of the problem you understand how to accomplish and what you must work harder on. Here's an example:

> Write a program in your favorite language that reads in a series of floating point numbers, one to a line, and determines the minimum and the maximum. At the end of file, print out the minimum first and the maximum second.

Below is the transcript of a session that decomposes the above problem into components:

> This is a full program, so there are three phases: input, process, and output. The process part is a while loop based on the "until end-of-file" clause. There's no range specified, so I have to initialize the two variables, call them min and max, but I don't have any particular value. OK, then read in the first one ... Oops, note that there might not be a first one ... init with the first one. Now you can have a loop with three statements: (1) check min changes, (2) check max changes, and (3) read next one.
>
> Now the conditions on the while loop: need to have a variable—call it num_read—set by the input statement. Use num_read not zero to continue.
>
> After the while loop, print out the two values. Oops, almost forgot the null case. Hm ... there's no spec on what to do there; print out nothing, I guess.

The approach given here is technically called *talking a solution* and this technique is very useful in decomposing a problem. You can use this technique yourself by either writing out your thoughts or by using a voice-activated tape recorder.

11.2.2 Work Breakdown Structure

The WBS is just a form that aids the decomposition process. A logical way to begin is to decompose the problem and identify the components. It may be more natural to have a graphic representation of the components and their couplings (connections). Number each component and enter this as the "Task Number" in the form. Enter the component name and description. Fill in the "Inputs From" and the "Outputs To" columns based on the task number of the input and output components. Remember that at this level inputs may come from many different components and the outputs may go to many components. For all but the simplest projects, a graphical

representation of this information is a must. We are now ready to estimate the effort.

Now estimate the number of "Lines" per component. The estimated "Hours" should be computed by either your measured rate or the 2.5 lines per hour rate that is the historical average. Now sum the "Hours" column to produce the cumulative hours, "Cum. Hours."

We have now completed the planning portion. However, the WBS must be used during development to determine whether or not you can complete the project on time. In the next section we discuss *earned value*.

11.3 EARNED VALUE COMPUTATIONS

The WBS and scheduling planning template have a column for cumulative hours under the heading of "Earned Value" which stands for *cumulative earned value*. Earned value is a concept of project management used in many large companies and by all government contractors (Fleming and Koppelman 2000). The basic idea is that every project has a value but that value is not just a percent completion of a particular component because it is almost impossible to reliably estimate percent complete. Earned value is about costs of resources (time, people, equipment). For class, we will equate planned hours as planned earned value and actual hours as actual costs because that time is the only capital you have to invest.

11.3.1 Baseline and Schedule Variance Computations

For illustration, suppose we have six modules *A* to *F* that must be delivered in two weeks. Suppose we estimate the number of time units (say hours) that this task should take as 100 and the amount of time needed for each as shown in Figure 11.3, called the baseline. In the baseline, the tasks are presented with the planned time estimates.

As the programming work is performed, value is "earned" on the same basis as it was planned, in hours. Planned value compared with earned value measures the volume of work planned versus the equivalent volume of work accomplished. Any difference is called a *schedule variance*. In contrast to what was planned, Figure 11.4 shows that work unit *D* was not completed and work unit *F* was never started, or 35 hours of the planned

	A	B	C	D	E	F	Total
Planned	10	15	10	25	20	20	100

Figure 11.3 Planned Workload

	A	B	C	D	E	F	T otal
Planned	10	15	10	25	20	20	100
Earned	10	15	10	10	20	0	65
Schedule	0	0	0	−15	0	−20	−35

Figure 11.4 Earned Value Computation (The time invested in the various modules is compared with the planned time and the variance is computed. The goal is to have no variance.)

work was not accomplished. As a result, the schedule variance shows that 35 percent of the work planned for this period was not done.

Schedule variance really only makes sense if applied relative to completion when we also know exactly what must be done.

11.3.2 Rebudgeting

In this example, we're really not in too bad shape. Four modules are complete and we estimated them correctly but we have badly miscalculated how long it takes to do the other two (*D* and *F*). The work still has to be done, but the next milestone looms large. The only thing you can do at this point is to consider merging the effort for *D* and *F* with what has to happen in the next milestone.

There are many lessons here:

- If you cannot correctly estimate your effort and consistently provide fewer than necessary resources, you're doomed.
- Overestimation is just as bad as underestimation. From overestimation, you waste resources that could be used to do other things.
- You must keep track of how much resources you've expended in order to not over- or undercommit.
- Accurate estimation is the key to this whole process.

11.3.3 Process

The process of developing an earned value plan is exactly the process we need to design a program. The most important step is the WBS. Every milestone will start with developing a WBS, which includes everything that is required to successfully complete the project. The key to success is to account for everything! For example, the following classes of work need to be accounted for:

- Class time
- Class preparation time

- Planning time
- Design time
- Programming time
- Test development time
- Debugging time

Note to the wise: Do not forget the effort for your other classes and extracurricular activities.

Because the design and implementation of each subprogram method and class structure take time, those times show on the WBS as well. If you are honest with yourself and honestly estimate your required effort, you will quickly see that you must be very disciplined. There is a sobering statistic: over the past 50 years, an accomplished programmer has been able to produce only 2.5 debugged statements per hour and this figure has been validated in several of the author's classes.

As a procedure to develop each milestone:

1. When we start a new milestone, develop a preliminary WBS that defines each task. Large tasks are broken into manageable work packages using task decomposition techniques. The WBS normally follows this hierarchy: the program that represents the solution to the milestone is broken into data structures (class variables) and algorithms (methods). (There is a well-defined OOPS design methodology: make use of it!) The WBS must contain, at a minimum, an entry for every method because every method takes time to write. This WBS will be its most accurate if it is based upon the very short, well-defined tasks.
2. The instructor determines the overall development budget based upon the instructor's estimate of the work. This will normally be given as a number of lines of code (LOC). I generally estimate code based on how many pseudo-code algorithm lines will be required. I tend not to count type statements unless there are a significant number of structure type statements. Estimating LOC is an art, not a science, and I shoot for an order of magnitude, say, to the nearest ten lines. $LOC/2.5$ is the number of hours.
3. You then will give an initial budget based upon the instructor's estimate. In the past, whenever the question of extension has come up, I generally made that decision based on my perception of how close the better students are. With this earned value discipline, I will have objective measurements.
4. The WBS is now the basic implementation plan.
5. As the milestone proceeds, you must update the plan. Based upon the more current estimates, time shifts may be possible only if reasonably documented and a general pattern of slippage is evident.

6. A new WBS/budget sheet with any variances will be due each Friday. Using the WBS/budget sheet as the input, a spreadsheet can be run by the instructor. The weekly WBS updates should be more detailed as design and implementation proceed.
7. Scheduling information is discussed each Friday in class.
8. Students should come prepared to explain their WBS status.
9. Plans are adjusted and actions taken to address any problems that are indicated by the reports.
10. You should assign planned start dates and completion dates to each subtask in accordance with the milestone schedule.
11. The final WBS will be handed in with the milestone reports.

11.3.4 Templates

The Personal Design and Software Process (PDSPP) documentation templates are provided as an Excel workbook. This section provides a key to that workbook based on the tab names in the Excel workbook. The original Excel workbook was retrieved online from the *PSP Resources* page at the University of Karlsruhe, Germany.* There are many such sites on the Internet. The *Time Utilization Log* was developed for the Clemson University time schedules.

The Karlsruhe packet contains all the PSP forms mentioned in Humphrey's book. The instructor should choose how much of the PSP discipline he or she is willing to impose. **Note:** Every form has a header that asks for what should be obvious information. For example, PSP0 asks for your name, due date, milestone, instructor, and language of implementation; this information should be obvious.

11.4 SUMMARY INFORMATION FOR PDSPP0

PDSPP0 forms are only used during the implementation phase of the course: otherwise, such fields as *milestone number* are meaningless.

- Header. This should include your name, due date, milestone, instructor, and language of implementation. Fill these out with the obvious.
- Phases. There are some standard phases that are always in play, such as planning and design, and should always be on the form. Add phases ("time sinks") for your unique situation. There are approximately 16 hours (960 minutes) of "usable" time during the day and approximately 180 minutes for meals. Therefore, you should

* http://www.ipd.uka.de/mitarbeiter/muellerm/PSP/#forms

account for about 750 minutes per day (3750 minutes per five-day week). Don't forget recreation and other obligations, such as civic activities. The sense I get from talking to students is that a 50- to 60-hour work week is not uncommon in computer science. Enter the planned and actual time in each phase. Planned values are estimates so they may well be a changing number as you understand more about the project.

- Enter the planned time in minutes you plan to spend in each phase.
- Enter the actual time in minutes you spent in each phase.
- Enter your best estimate of the total time the development will take.
- Time — To Date. Enter the total time spent in each phase to date. Since we're always working on a milestone, there is always something to charge time to.
- Time — To Date %. Enter the percent of the total "To Date" time that was spent in each phase. This form is a summary form. You can use Excel to compute these times for you. Your actual time usage should be consistent with the Time Recording log (TRT).
- Defects. Defects are not planned, so there is no way you can budget time for them, but you can count them and analyze why you are injecting them. This information comes from the Defect Log (DLT)
- Defects Injected and Defects Removed. Enter the actual numbers of defects injected and removed in each phase. Careful planning of Excel will simplify these calculations.
- Defects — To Date. Enter the total defects injected and removed in each phase to date.
- Defects — To Date %. Enter the percent of the total "To Date" defects injected and removed in each phase.

11.5 REQUIREMENTS TEMPLATE (RWT)

- Header. Enter the name, date, instructor, and milestone as on the summary.
- Item. Identification; could be a number or mnemonic.
- Inputs. What are the inputs to this constraint or criteria?
- Criteria or Constraints. Criteria are logical statements about what the system is supposed to do. Constraints are performance standards. Requirements are the first full step in the Dewey Problem Solving Paradigm.

11.6 TIME RECORDING LOG (TRT)

- Header. Name, date, instructor, and program number as in the summary.
- Date. Enter the current date.
- Start. Enter the time in minutes when you start a project phase.
- Stop. Enter the time in minutes when you stop work on a project phase, even if you are not done with that phase.
- Delta time. Enter the elapsed start to stop time less the interruption time.
- Phase. Note the phase on which you were working by using the phase name.
- Comments. Describe the interruption to the task you were doing and anything else that significantly affects your work.

11.7 SCHEDULE PLANNING TEMPLATE (SPT)

- Header. Enter the name, date, instructor, and milestone as on the summary.
- Number (No). This is the number of the work period.
- Date. Date the work period took place.
- Plan. How many hours do you plan to work this period and what are the total cumulative hours?
- Actual. Hours and Cumulative Hours as above. Cumulative EV and Adjusted EV are described below.

11.8 DEFECT LOG TEMPLATE (DLT)

A defect is anything in the program that must be changed for it to be properly developed, enhanced, or used.

- Header. Enter the name, date, instructor, and milestone as on the summary.
- Date. Enter the date when you found and fixed the defect.
- Number. Enter a unique number for this defect. Start each project with 1.
- Type. Enter the defect type from the defect type standard. See tab DCL.
- Inject. Enter the phase during which you judge the defect was injected.
- Remove. Enter the phase in which you found and fixed the defect.
- Fix time. Enter the time you took to fix the defect. You may time it exactly or use your best judgment.

- Fix defect. If this defect was injected while fixing another defect, enter the number of that defect or an X if you do not know.

The Defect Codes Legend (DCL) is a table of possible causes and defects. You may add to your own personal copy any defect information that will help you spot defect causes in your process.

Cause Code	*Defect Type*	*Description*
	10	Documentation comments, messages
ed (education)	20	Syntax spelling, punctuations, formats
co (communication)	30	Build, package change management, library, version control
ov (oversight)	40	Assignment declaration, duplicate names, scope, limits
tr (transcription)	50	Interface procedure calls and references, I/O, user formats
pr (process)	60	Checking error messages, inadequate checks
	70	Data structure, content
	80	Function login, pointers, loops, recursion, computation
	90	System configuration, timing, memory
	100	Environment design, compile, test, other support system problems