A Survey of Plan-Driven Development Methodologies

Plan-driven methodologies have been utilized by organizations for many years. In this chapter, we provide an overview of three prominent, modern plan-driven methodologies:

- the Personal Software Process
- the Team Software Process
- the Rational Unified Process

Software methodologists incorporate the general characteristics of a software development model (a *software development model* is a *simplified, abstracted description of a software development process*) into specific software development processes that adhere to the spirit of these models. A *software development process* is the *process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes installing and checking out the software for operational use. Note: these activities might overlap or be performed iteratively.* (IEEE, 1990) While software development models have general characteristics, such as "having strong documentation and traceability mandates across requirements, design and code" (Boehm and Turner, 2003), software development processes have *specific practices* (a *software development practice* is a *disciplined, uniform approach to the software development process* (IEEE, 1990)) that need to be followed, such as code inspection. You can think of the relationship between software models and software processes as an abstract superclass–subclass relationship. The model specifies generally the common techniques and philosophies, while the methodology "overrides" and specializes these general specifications with details of specific practices.

In this chapter, we will describe the practices of the most prominent methodologies that have plan-driven characteristics[1]: the Personal Software Process (PSP) (Humphrey, 1995), the Team Software Process (TSP) (Humphrey, 2000), and the Rational Unified Process® (RUP®).(Kruchten, 2004) The class diagram in Figure 1 shows these methodologies in the spirit of the superclass–subclass relationship. For each of these methodologies, we will present an overview, the main roles involved in the methodology, the documents and artifacts produced, the development process, and a final discussion.

[1] Personal Software Process, PSP, Team Software Process, and TSP are all service marks of Carnegie Mellon University. Rational Unified Process and RUP are registered trademarks of IBM, Corp.

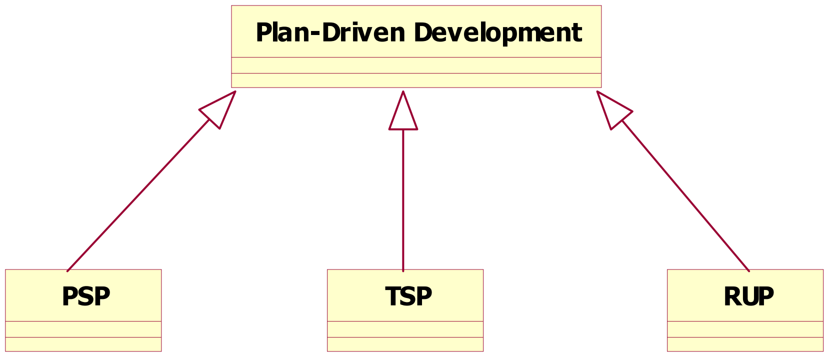


Figure 1: Plan-Driven Methodologies

Before beginning, it is important to understand that there is not a sharp dichotomy between plan-driven and agile software development methodologies. So, these three methodologies have some elements of agility to them or can be slightly modified to incorporate agility. The PSP is probably the most plan-driven. The TSP could be used to structure team development of agile teams. Finally, versions of the RUP have been created that could distinctly be classified as agile; this will become more clear when we discuss RUP in Section 3. Let us review some common characteristics of plan-driven methodologies, though these guidelines are more relaxed in smaller projects (Boehm and Turner, 2003):

- Focus on repeatability and predictability
- Defined, standardized, and incrementally improving processes
- Thorough documentation
- A defined software system architecture defined up-front
- Detailed plans, workflow, roles, responsibilities, and work product descriptions
- Process group containing resources for specialists: process monitoring, controlling, and educating
- On-going risk management
- Focus on verification and validation

1 Personal Software Process^{SM} (PSP^{SM})

The PSP (Humphrey, 1995; Humphrey, 1997) is a process to be followed by an individual programmer, not a team of programmers—hence, the name *Personal* Software Process. While many software processes are followed by a whole team, an individual programmer can practice the PSP even if his teammates are not using those practices. It certainly helps when others are using the practices as well, though.

1.1 Overview

The PSP is a structured framework of forms, guidelines, and procedures developed by Watts Humphrey of the Software Engineering Institute.[2] The framework guides an

[2] The Software Engineering Institute (http://www.sei.cmu.edu/) is a federally funded research and development center sponsored by the U.S. Department of Defense. The SEI's core purpose is to help others make measured improvements in their software engineering capabilities.

engineer in using a defined, measured, planned, and quality controlled process. However, another purpose of the framework is to help engineers understand their own skills so they can modify the process to meet their personal needs and preferences and to improve their own personal performance.

PSP training follows an evolutionary improvement approach. An engineer learning to integrate the PSP into his or her process begins at Level 0 and progresses in process maturity to Level 3 (See Figure 2). Each level incorporates new skills and techniques into the engineer's process—skills and techniques that have been proven to improve the quality of the software process. Each level has detailed scripts, checklists, and templates to guide the engineer through required steps. The scripts, checklists, and templates are only a starting point, however. The PSP provides measurement-based feedback that helps each engineer improve her own personal software process. Thus, Humphrey encourages the customization of these scripts and templates as the engineer receives this feedback and gains an understanding of his or her own strengths and weaknesses.

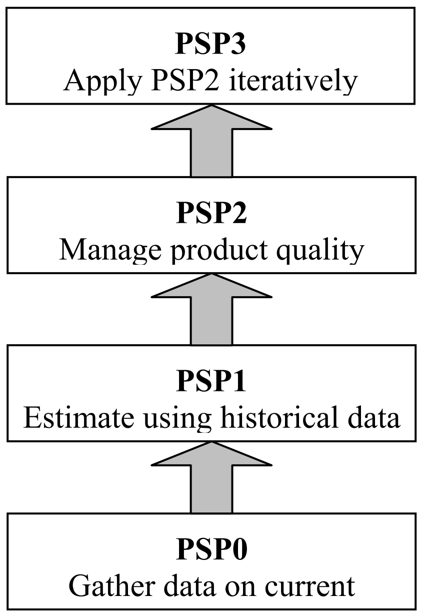


Figure 2: The levels of the personal software process.

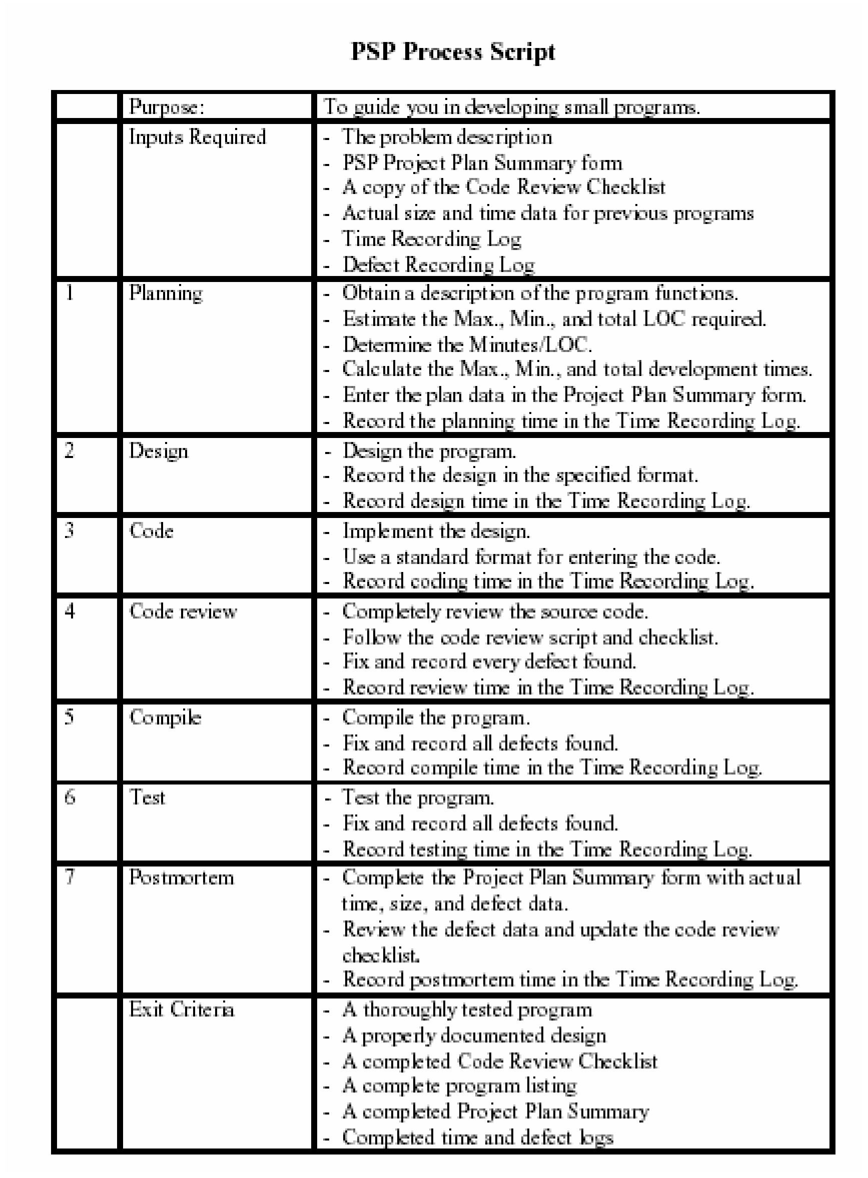
PSP has several strong tenets. The first is that the longer a software defect remains in a product, the more costly it is to detect and remove it. Therefore, thorough design and code reviews are performed for efficient defect removal. The second philosophy is that defect prevention is more efficient than defect removal. Careful designs are developed and data is collected to give additional input on where the programmer should adjust their own personal software process to prevent future defects.

1.2 Documents and Artifacts

The artifacts in the PSP are the scripts, forms, templates, standards, and checklists. Each of these is discussed in this section.

Script. The scripts provide an orderly structure of steps the engineer should go through to complete the process step and refer the engineer to the relevant standards, forms, templates, guidelines, and measures. The PSP has six different kinds of scripts. In many cases, there is a different version of the script for every relevant level (PSP0, PSP0.1, PSP1, PSP1.1, PSP2, PSP2.1, and PSP3).

- Process script—lays out the inputs and the steps to complete a process. By following the script, the engineer can ensure she has all the required inputs and understands the requirements of the job. The completeness of the process is checked via stated exit criteria. A sample process script is shown in Figure 3.
- Planning script—provides instructions for planning development activities, such as making size and time estimates.
- High-level design script—provides instructions for creating a high-level design (for PSP3 only).
- High-level design review script—provides instruction for reviewing the PSP3 high level design.
- Development script—outlines the steps for design, code, compile, and test.
- Postmortem script—provides the instructions for analyzing and summarizing the data collected in the forms and templates.



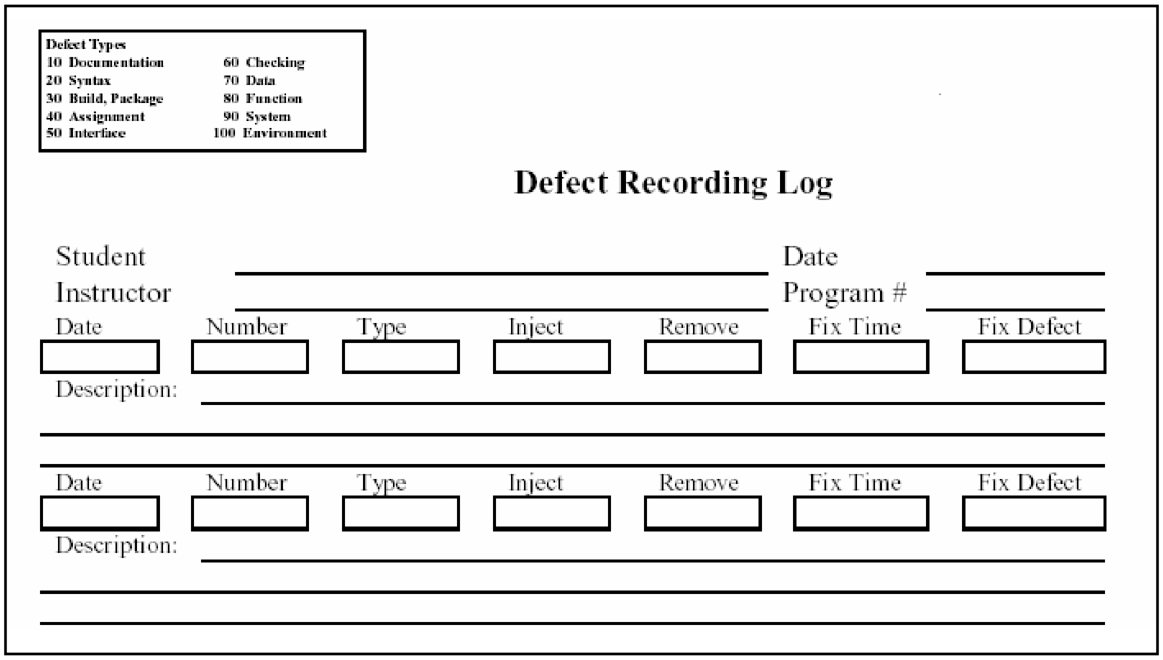
PSP Process Script

	Purpose:	To guide you in developing small programs.
	Inputs Required	- The problem description - PSP Project Plan Summary form - A copy of the Code Review Checklist - Actual size and time data for previous programs - Time Recording Log - Defect Recording Log
1	Planning	- Obtain a description of the program functions. - Estimate the Max., Min., and total LOC required. - Determine the Minutes/LOC. - Calculate the Max., Min., and total development times. - Enter the plan data in the Project Plan Summary form. - Record the planning time in the Time Recording Log.
2	Design	- Design the program. - Record the design in the specified format. - Record design time in the Time Recording Log.
3	Code	- Implement the design. - Use a standard format for entering the code. - Record coding time in the Time Recording Log.
4	Code review	- Completely review the source code. - Follow the code review script and checklist. - Fix and record every defect found. - Record review time in the Time Recording Log.
5	Compile	- Compile the program. - Fix and record all defects found. - Record compile time in the Time Recording Log.
6	Test	- Test the program. - Fix and record all defects found. - Record testing time in the Time Recording Log.
7	Postmortem	- Complete the Project Plan Summary form with actual time, size, and defect data. - Review the defect data and update the code review checklist. - Record postmortem time in the Time Recording Log.
	Exit Criteria	- A thoroughly tested program - A properly documented design - A completed Code Review Checklist - A complete program listing - A completed Project Plan Summary - Completed time and defect logs

Figure 3: A sample process script.

Forms. Forms are used to guide thorough, complete data collection. Forms are used when the amount of data you collect is fixed. Three fundamental forms are used in the PSP:

- Defect recording—An important activity of the PSP is collecting data about the defects that are injected and removed from the project. A form such as the one shown in Figure 4 is used to collect that data; this form is used by students learning the PSP. Each defect is assigned a unique defect number and is classified according to the type of defect using a defect classification scheme developed by IBM Research. (The defect classification scheme is the Orthogonal Defect Classification scheme or ODC (Chillarege, Bhandari et al., November 1992).) The engineer records the development phase in which she believes the defect was injected, the phase in which the defect was found/removed, and how long it took to fix the defect. Finally, if it is believed this defect was injected while fixing another defect, the number of that initial defect is recorded.



Defect Types	
10 Documentation	60 Checking
20 Syntax	70 Data
30 Build, Package	80 Function
40 Assignment	90 System
50 Interface	100 Environment

Defect Recording Log

Student ______________________ Date __________

Instructor ______________________ Program # __________

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect

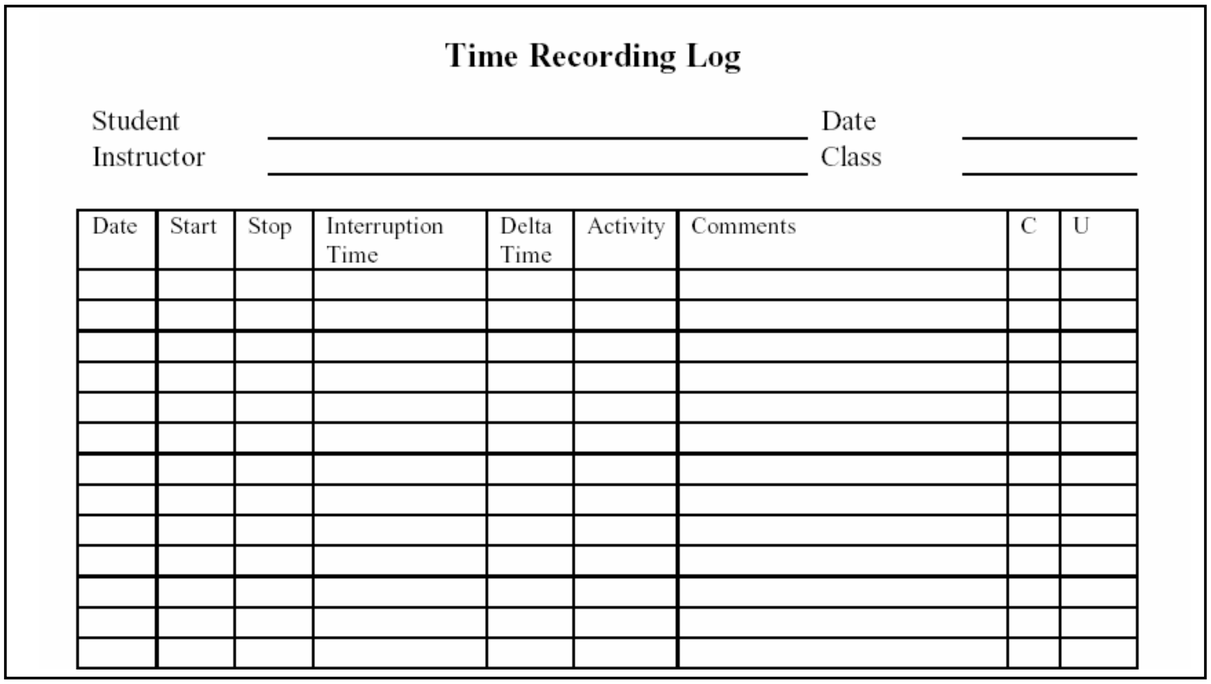
Description: ______________________

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect

Description: ______________________

Figure 4: PSP defect recording log.

- Time recording log—another important PSP activity is recording how much time is spent on development activities. This data is recorded in a form like the one shown in Figure 5 which was also designed for students learning PSP. When a developer starts to work, he records the date, the start time, and the development phase. If the developer is interrupted during work, the number of elapsed minutes of the interruption is recorded. When activity is completed, the stop time and any comments are noted.



Time Recording Log

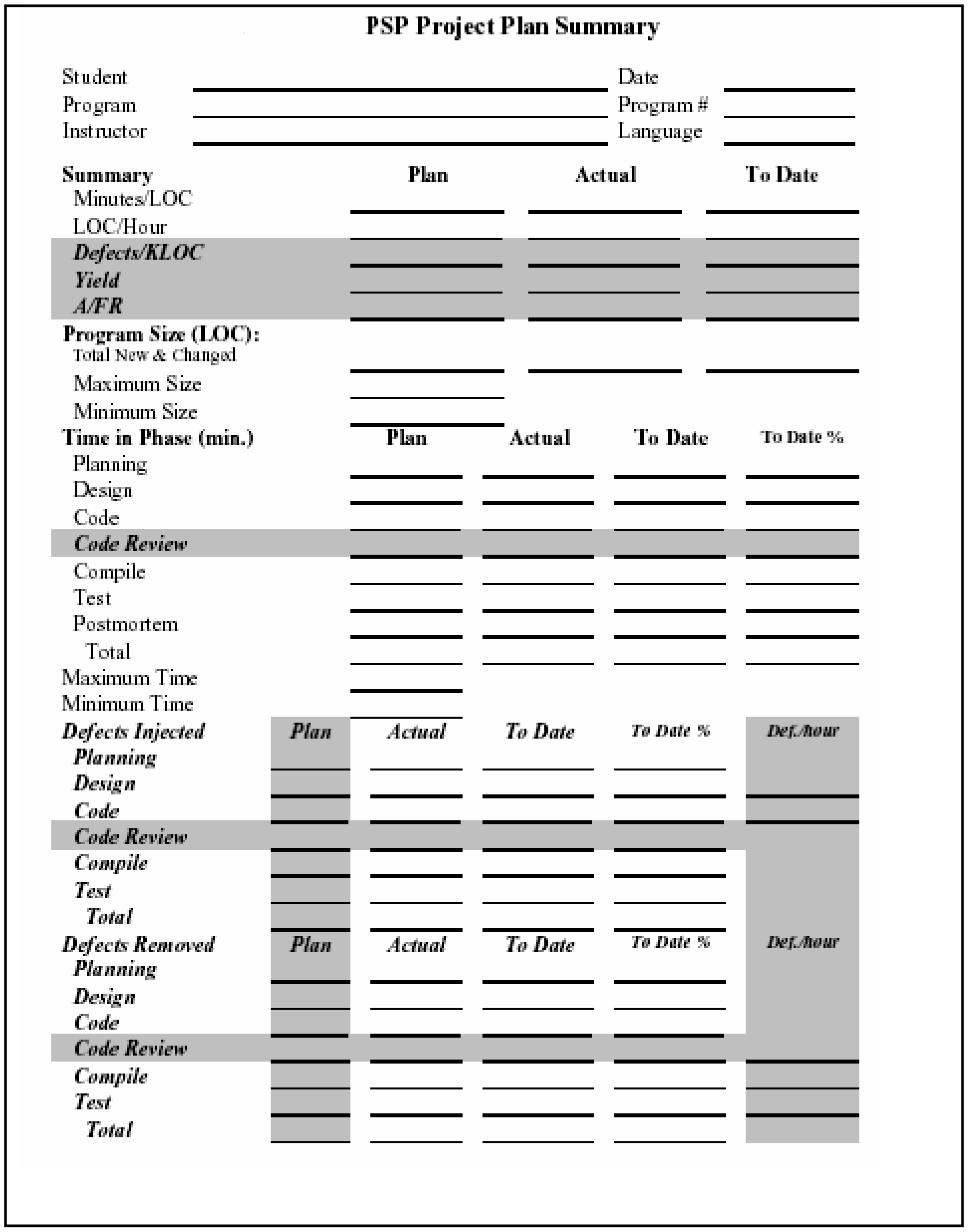
Student ______________________ Date ________
Instructor ______________________ Class ________

Date	Start	Stop	Interruption Time	Delta Time	Activity	Comments	C	U

Figure 5: PSP time recording log.

- Project plan—a summary form that is used for the summarization, analysis, and utilization of the data that has been entered, as shown in Figure 30.5. Often, the time and data logging and the completion of the project plan form can be automated via available PSP tools.[3]

[3] See http://www.ipd.uka.de/PSP/ for helpful PSP information and links to available PSP tools.



PSP Project Plan Summary

Student		Date	
Program		Program #	
Instructor		Language	

Summary	**Plan**	**Actual**	**To Date**
Minutes/LOC			
LOC/Hour			
Defects/KLOC			
Yield			
A/FR			
Program Size (LOC):			
Total New & Changed			
Maximum Size			
Minimum Size			

Time in Phase (min.)	**Plan**	**Actual**	**To Date**	**To Date %**
Planning				
Design				
Code				
Code Review				
Compile				
Test				
Postmortem				
Total				
Maximum Time				
Minimum Time				

Defects Injected	***Plan***	***Actual***	***To Date***	***To Date %***	***Def./hour***
Planning					
Design					
Code					
Code Review					
Compile					
Test					
Total					

Defects Removed	***Plan***	***Actual***	***To Date***	***To Date %***	***Def./hour***
Planning					
Design					
Code					
Code Review					
Compile					
Test					
Total					

Figure 6: Project plan template.

Templates. Templates are important tools for being complete in development activities. The PSP has defined templates for test reporting, size estimating, task planning, schedule planning, issue tracking log, and for creating operational scenarios and functional, state, and logic specifications.

Checklists. Checklists help you to completely follow a procedure. PSP suggests an initial version of design and a code review checklist. The intent is that the developer learn about the kinds of defects he typically injects and continually adapts these checklists to surface those "typical" defects. Here are some examples of code review checklist items for a program written in C:

- Verify that the code covers all the design

- Verify that includes are complete
- Verify the proper use of ==, =, ||, and so on.

1.3 Roles

PSP has only one role, the individual software engineer.

1.4 Process

PSP training is based on four levels of personal process: PSP Levels 0 through 3, as shown in Figure 2. Skills at one level are mastered before the engineer moves to the next level of personal process improvement.

Level 0 (Personal Measurement): The input to PSP is the requirements; requirements elicitation is assumed to have been completed and a requirements document delivered to the engineer. The PSP0 has three waterfall-like phases: planning, development (including design, code, compile, and test), and a postmortem. In the postmortem, the engineer ensures all data for the projects has been properly recorded and analyzed.

The software engineer begins by establishing a personal baseline of her current development process by basic measurements, such as the time spent on a program (using the form shown in Figure 5), the defects injected and removed in each development phase (using the form shown in Figure 4), and the size of the program (in lines of code), and creating some initial reports. This level is then improved by adding a coding standard, a size measurement, and the development of a personal process improvement plan (PIP). In the PIP, the engineer records ideas for improving her own process. The improvements constitute PSP0.1.

Level 1 (Personal Planning): Based upon the baseline data collected in PSP Level 0, the engineer estimates how large a new program will be and prepares a test report (PSP1). Accumulated data from previous projects is used to estimate the total time. Each new project will record the actual time spent. This information is used for task and schedule planning and estimation (PSP1.1).

Level 2 (Personal Quality): Defect prevention and removal are the focus at the PSP Level 2. Engineers construct and use checklists for design and code reviews (PSP2). PSP2.1 introduces design specification and analysis techniques. Engineers learn to evaluate and improve their process by measuring how long tasks take and the number of defects they inject and remove in each phase of development.

Level 3 (Scaling Up): In the final level, the programmer employs an incremental model of development for larger projects by dividing the problem into smaller sections, and then iteratively applies the PSP principles as each section is implemented.

1.5 Discussion

The main advantages of PSP have been demonstrated by several studies, including (Hayes and Over, 1997; Humphrey, May 1996; Ferguson, Humphrey et al., May 1997):

- Improved size estimation and time estimation

- Improved productivity
- Reduced testing time
- Improved quality

The possible drawbacks of PSP are as follows:

- Some people are not receptive to the detailed data recording.
- The longevity of the PSP requires discipline. Several studies, including (Webb, 1999), have noted that engineers stop using the PSP over time unless they work on TSP teams (discussed in the next section) that are competently coached and managed.

2 Team Software Process^{SM} (TSP^{SM})

The PSP, discussed in Section 1, is used to help an individual software engineer. Rarely, though, do engineers work alone. Most often, software engineers work as part of a team. The TSP (Humphrey, 2000; Hilburn and Humphrey, September/October 2002) provides a structure for self-directed teams to plan and track their work, to establish goals, and to create and own their processes and plans. It also provides guidance to individual software engineers on how to perform as an effective team member. To make a distinction, there is an industrial TSP for professional teams of up to 150 engineers who work on large, possibly multi-year projects. The material in this chapter comes from the Introductory Team Software Process (TSPi), a defined framework specifically developed for graduate or upper level undergraduate students. The process described here is the academic TSPi version of the TSP. The industrial strength TSP follows the same general principles described here, but it has many important differences. An overview description of industrial TSP principles can be found in *Winning with Software: An Executive Summary*. (Humphrey, 2002)

2.1 Overview

Watts Humphrey, the author of the PSP, also created the TSP and the TSPi. The TSP supports the development of industrial strength software through the use of team building, planning, and control. The overall structure of the TSPi is shown in Figure 7. The project starts with a product needed by a customer. A software development project to address this need is divided into overlapping, iterative development cycles. The team produces part of the product each cycle until the need is fulfilled with a finished product. Each of the cycles is a "mini waterfall" consisting of a cycle launch, strategy, planning, requirements, design, implementation, test, and postmortem. To some extent, the TSPi relies upon all the individual engineers using the PSP. However, TSPi is flexible enough that you could apply many of the principles and techniques if the individual engineers are not using PSP.

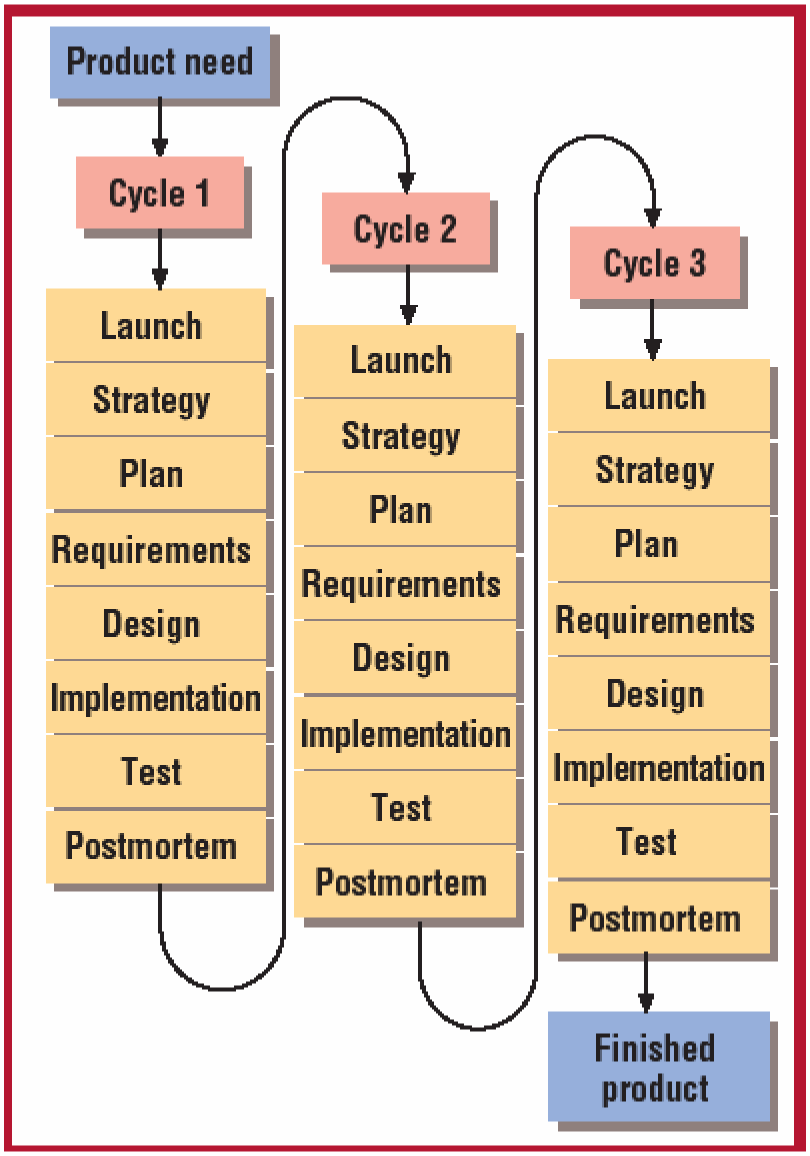
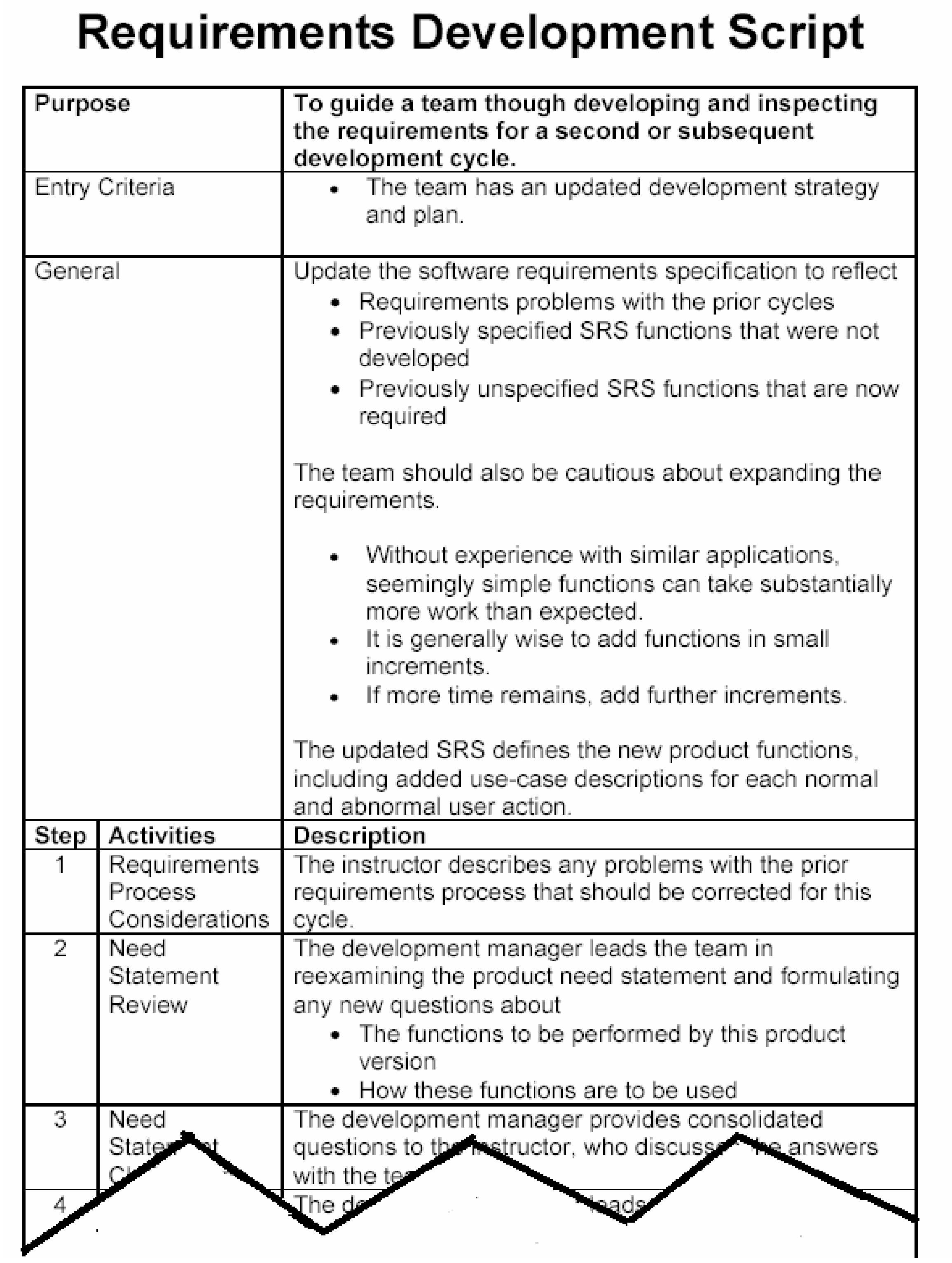


Figure 7: The TSPi process structure (taken from (Hilburn and Humphrey, September/October 2002)).

2.2 Documents and Artifacts

The structure of the TSPi is similar to that of the PSP; TSPi is also a structured framework of scripts, forms, and standards. Specifically, there are 21 process scripts and 10 role scripts (role scripts are discussed in the next section), 21 forms, and 3 standards. As stated above, TSP works best when the individual engineers on the team are using the PSP for their own personal development. As such, the scripts, forms, and standards of the PSP of the PSP guide the work and process improvement of each engineer on the team. The scripts, forms, and standards of the TSP guide the structure, organization, and measurement of the team as a whole.

Scripts. The scripts lay out detailed steps to guide the teams through launching and running their projects so that each engineer can see how to do what she needs to do. A sample of a partial TSPi process script is shown in Figure 8. This is a script for updating requirements in the cycles after the first cycle. Each script shows the entry criteria, some general information, detail process steps, and finally exit criteria. There are scripts for design, development, implementation, team launch, development plan, postmortem, requirements development, configuration management, development strategy, integration and system test, unit test, and the weekly meeting.

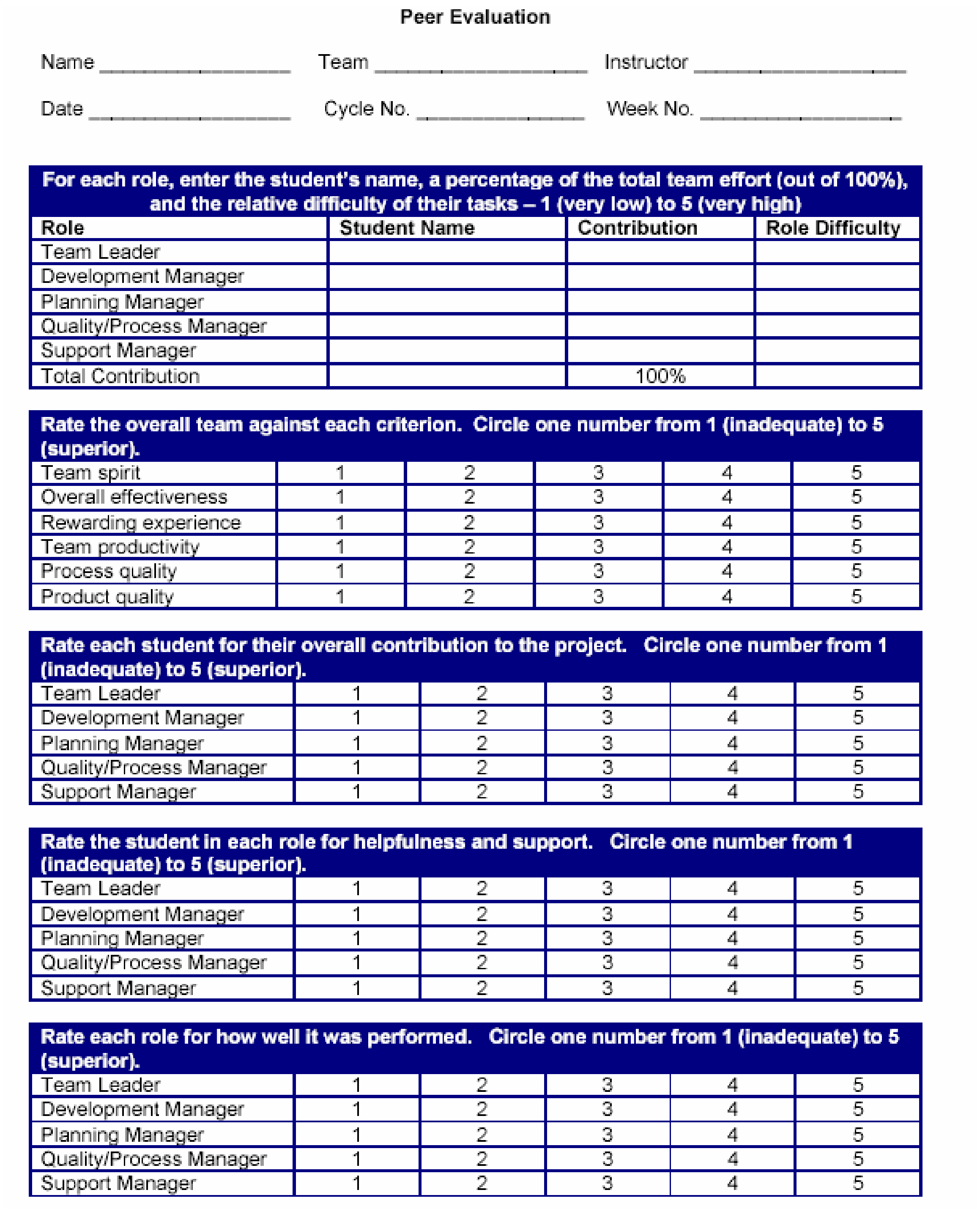


Requirements Development Script

Purpose		**To guide a team though developing and inspecting the requirements for a second or subsequent development cycle.**
Entry Criteria		• The team has an updated development strategy and plan.
General		Update the software requirements specification to reflect • Requirements problems with the prior cycles • Previously specified SRS functions that were not developed • Previously unspecified SRS functions that are now required The team should also be cautious about expanding the requirements. • Without experience with similar applications, seemingly simple functions can take substantially more work than expected. • It is generally wise to add functions in small increments. • If more time remains, add further increments. The updated SRS defines the new product functions, including added use-case descriptions for each normal and abnormal user action.
Step	**Activities**	**Description**
1	Requirements Process Considerations	The instructor describes any problems with the prior requirements process that should be corrected for this cycle.
2	Need Statement Review	The development manager leads the team in reexamining the product need statement and formulating any new questions about • The functions to be performed by this product version • How these functions are to be used
3	Need State C	The development manager provides consolidated questions to th structor, who discuss answers with the te
4		The d leads

Figure 8: TSPi process script (requirements).

Forms. As with the PSP, the TSPi provides forms to guide in thorough, complete data collection. Some of the forms, such as the Defect Recording Log and the Time Recording Log, are identical to the PSP. Many of the forms, however, are for higher level data collection and analysis where the plans and data from the whole team are summarized. An example of a form is shown in Figure 9. This form is for student evaluation of their team member peers. When students work in a team, some team members might contribute less than their fair share of the work while others go above and beyond what is expected (sometimes to make up for the students who are not doing their share). The form shown in Figure 9 is used to provide feedback to the instructor on the contributions of team members. Other forms included in the TSPi collect necessary team information, such as configuration change and status, inspections, issues, and testing. Still others provide the format for team summary information, such as planning, weekly status, and task planning.



Peer Evaluation

Name ________________ Team ________________ Instructor ________________

Date ________________ Cycle No. ______________ Week No. ________________

For each role, enter the student's name, a percentage of the total team effort (out of 100%), and the relative difficulty of their tasks – 1 (very low) to 5 (very high)			
Role	**Student Name**	**Contribution**	**Role Difficulty**
Team Leader			
Development Manager			
Planning Manager			
Quality/Process Manager			
Support Manager			
Total Contribution		100%	

Rate the overall team against each criterion. Circle one number from 1 (inadequate) to 5 (superior).					
Team spirit	1	2	3	4	5
Overall effectiveness	1	2	3	4	5
Rewarding experience	1	2	3	4	5
Team productivity	1	2	3	4	5
Process quality	1	2	3	4	5
Product quality	1	2	3	4	5

Rate each student for their overall contribution to the project. Circle one number from 1 (inadequate) to 5 (superior).					
Team Leader	1	2	3	4	5
Development Manager	1	2	3	4	5
Planning Manager	1	2	3	4	5
Quality/Process Manager	1	2	3	4	5
Support Manager	1	2	3	4	5

Rate the student in each role for helpfulness and support. Circle one number from 1 (inadequate) to 5 (superior).					
Team Leader	1	2	3	4	5
Development Manager	1	2	3	4	5
Planning Manager	1	2	3	4	5
Quality/Process Manager	1	2	3	4	5
Support Manager	1	2	3	4	5

Rate each role for how well it was performed. Circle one number from 1 (inadequate) to 5 (superior).					
Team Leader	1	2	3	4	5
Development Manager	1	2	3	4	5
Planning Manager	1	2	3	4	5
Quality/Process Manager	1	2	3	4	5
Support Manager	1	2	3	4	5

Figure 9: TSPi peer evaluation form.

Standards. The TSPi includes standards for defects types, the project notebook, and quality criteria. The defect type standards provide the defect classification that is used on the defect recording log, as discussed in Section 1.2 and appear in the upper left corner of Figure 4. The project notebook standard provides useful guidance on what should be included in a project notebook. Official project information, such as copies of work products, weekly team status reports and test plans, is archived in the team's project notebook, which is maintained by the team leader. Finally, the quality criteria provide helpful standards that can be used to establish team quality goals. Some examples of these standards are having fewer than 10 unit test defects/KLOC or inspecting fewer than two pages of requirements/hour.

2.3 Roles

A very important factor for effective team building is defining clear team roles for each team member. Then, each engineer has clearly identified responsibilities and understands what she is expected to do in terms of planning, tracking, quality, support, and leadership tasks. The TSPi defines five team roles and provides a very thorough guide for each role. When possible, the assignment of the person to the role should be based upon the person's interest in that role. In addition to the responsibilities outlined in these roles, each software engineer on the team would have technical responsibilities for product development. Via these roles, each engineer shares in the essential "overhead" of product development in addition to making a technical contribution to the working product.

In the TSPi, each role has its own chapter, complete with the role's goals, helpful skills and abilities, principle activities, and project activities. The main responsibilities and goals of each of these is briefly described here (Hilburn and Humphrey, September/October 2002):

- *Team leader*: leads the team and ensures that engineers report their progress data and complete their work as planned.
  - *Goal 1*: Build and maintain an effective team.
  - *Goal 2*: Motivate all team members to work aggressively on the project.
  - *Goal 3*: Resolve all the issues team members bring to you.
  - *Goal 4*: Keep the instructor fully informed about the team's progress.
  - *Goad 5*: Perform effectively as the team's meeting facilitator.
- *Development manager*: leads and guides the team in product design, development, and testing.
  - *Goal 1*: Produce a superior product (documented and meeting all functional and operational objectives and quality criteria).
  - *Goal 2*: Fully utilize the team members' skills and abilities.
- *Planning manager*: supports and guides the team in planning and tracking its work.
  - *Goal 1*: Produce a complete, precise, and accurate plan for the team and for every team member.
  - *Goal 2*: Accurately report team status every week.
- *Quality/process manager*: supports the team in defining the process needs and establishing and managing the quality plan.
  - *Goal 1*: Ensure that all team members accurately report and properly use TSPi process data.
  - *Goal 2*: Make certain that the team faithfully follows the TSPi quality plan and produces a quality product.
  - *Goal 3*: Make all team inspections properly moderated and reported.

    - *Goal 4*: Make sure that all team meetings are accurately reported, and the reports put in the project notebook.
- *Support manager*: supports the team in determining, obtaining, and managing the tools needed to meet its technology and administrative support needs.
    - *Goal 1*: Provide the team with suitable tools and methods to support its work.
    - *Goal 2*: Prevent unauthorized changes to baseline products.
    - *Goal 3*: Keep the risk-tracking system functional to keep the team's risks and issues recorded and reported each week.
    - *Goal 4*: Ensure that the team meets its reuse goals for the development cycle.

An important TSP role that is not in the TSPi (and is not in the preceding list) is the *team coach*. In the industrial form of the TSP, the team coach role is assumed by someone who is trained and qualified by the SEI and generally launches and coaches multiple projects. The team coach is not involved in the technical details of the project itself, as is even the team leader. In the industrial form of TSP, this role is assumed by the manager the team reports to or by some other person who is very knowledgeable on software process and might coach multiple projects. In academia, the teaching staff assumes this role. The job of this person is to motivate the team and to maintain a relentless focus on quality and excellence. This requires daily interaction with the team and an unceasing requirement that the process be followed, the data be gathered, and the results be analyzed. The team coach and the team meet regularly to review their performance and to ensure that the work meets the standards.

2.4 Process

Figure 7 lays out the process structure of the TSPi. TSPi uses multiple development cycles. Each cycle starts with a launch followed by seven iterative process steps: development strategy, development plan, requirements, design, implementation, test, and postmortem. The cycles can and should overlap. Each cycle should produce a *testable* version that is a subset of the final product.

The launch and the seven process steps are now briefly described. The TSP has a detailed script for the tasks of the launch and each of the seven process steps.

Launch. The customer (or instructor) describes the overall product objectives. Teams are formed and team structure is established (the engineers assume the roles as just described). The team establishes a meeting schedule and reporting structure. The team sets measurable goals and measurements. An example of a team goal and measurements is as follows (Hilburn and Humphrey, September/October 2002):

Team Goal: Produce a quality product

- Measure 1: More than 80 percent of the defects will be found before the first compile.
- Measure 2: No defects will be found during system test.
- Measure 3: At project completion, all product requirements will be correctly implemented.

Strategy. The development strategy specifies the order in which product functions are defined, designed, implemented, and tested. A high level conceptual design of the product is developed. Preliminary, high level size and time estimates are developed. Risks are assessed. Finally, a strategy is established for how the product will be enhanced in each cycle.

Plan. The team develops a comprehensive plan that includes the following:

- A list of the products to be produced with their estimated sizes;
- A list of tasks to be completed, each assigned a responsible team member and each complete with an estimation of how long it will take to complete the task, balancing team workload;
- A week-by-week schedule of tasks and available work hours;
- A quality plan that estimates the quantity of defects injected and removed by development phase; and
- A template summarizing the product's estimated and actual size, effort, and defect data.

These plans are often documented and updated via a TSPi spreadsheet tool.

Requirements. The team produces a clear and unambiguous description of what the product is to be. This is documented in Software Requirements Specifications (SRS) such as those described in Chapter 5 *Requirements Engineering and Elicitation*. Each cycle the SRS is re-examined and evolved.

Design. The principle objective of the design process step is to produce a precise, complete, high quality specification of how the product is to be built.

Implementation. The code is implemented utilizing standards, such as coding standards (which ensures the team's code looks the same person-to-person) or a defect classification standard, as previously discussed. All code is unit tested and is reviewed via formal inspections during the implementation phase.

Test. The product is integrated and tested. The purpose of this testing is to assess the product, not to fix it. If the assessment determines the product is not of high quality, testing should cease and the product should repeat any necessary requirements/design/implementation steps. Integration testing is done to verify that the product is properly built, all the parts are present, and that they function together. System testing is done to validate that the product does what the system requirements call for.

Postmortem. Similar to the postmortem in the PSP, the teamwork is reviewed to make sure all the tasks are complete and all the required data are recorded. The team also carefully reviews how the cycle went—to learn what went right and wrong and to brainstorm how they can do a better job in the next cycle or on the next project.

2.5 Discussion

In general, the advantages and disadvantages of the TSPi are similar to those of the PSP. It should be noted, however that the principal benefit of both the TSP and the academic TSPi is that it shows teams of students or engineers how to produce quality products for planned costs and on aggressive schedules. They do this by showing teams how to manage their work and by making them owners of their plans and processes. (Humphrey, 1998)

3 Rational Unified Process (RUP)

The RUP (Kroll and Kruchten, 2003; Kruchten, 2004) is very different from the prescribed PSP and TSP from two perspectives. First, the RUP specifically embeds object-oriented techniques and uses UML as its principle notation. Secondly, the RUP is a customizable process framework. Depending upon the project characteristics, such as team size, project size, and so on, the RUP can be tailored or extended to match the needs of an adopting organization.

3.1 Overview

The RUP combines the expertise from many sources and three decades of development and active use. Development started in Sweden by Dr. Ivar Jacobson (Jacobson, Christerson et al., 1992) in the late 1980s. Jacobson was first at Ericsson, and then he started his own company, Objectory AB. Jacobson's process was called Objectory, *which* is an abbreviation of "Object Factory." Objectory AB was acquired by Rational, Inc. in the fall of 1995. Objectory was integrated with Rational's process (into the Rational Objectory Process or ROP) and became an iterative process, focused on software architecture spanning the whole development lifecycle; ROP 4.0 was launched in the fall of 1996. Finally, the Rational Unified Process was released in 1998 (Kruchten, 2004) as a unification of development approaches and the work of many methodologists. The RUP integrated a wide range of seemingly independent tools developed by Rational. In 2003, Rational—with RUP and its associate tools—was acquired by IBM. Similar to the TSP/TSPi, there is both a professional and a student version of RUP. (Robillard and Kruchten, 2003; Kruchten, 2004) The student version is called the Unified Process for Education, abbreviated UPEDU and pronounced *yoopeedoo*. Only the most important features of the RUP process are retained to enable students to focus on the essential components of learning to use the disciplined process.

The idea behind the creation of the RUP was to capture many of the best practices in modern software development and to put them into a form from which they could easily be composed into a suitable process for a wide range of projects and organizations. The overall process structure of RUP is shown in Figure 10. The horizontal axis represents the timeline of a project lifecycle as it unfolds. The core process disciplines that logically group software engineering activities that are often performed simultaneously are shown on the vertical axis. The height of the area indicates how much work is spent on a discipline at a given point in time. The figure indicates that the disciplines need work simultaneously throughout various phases of development and that the software product is designed and built in a succession of incremental iterations.

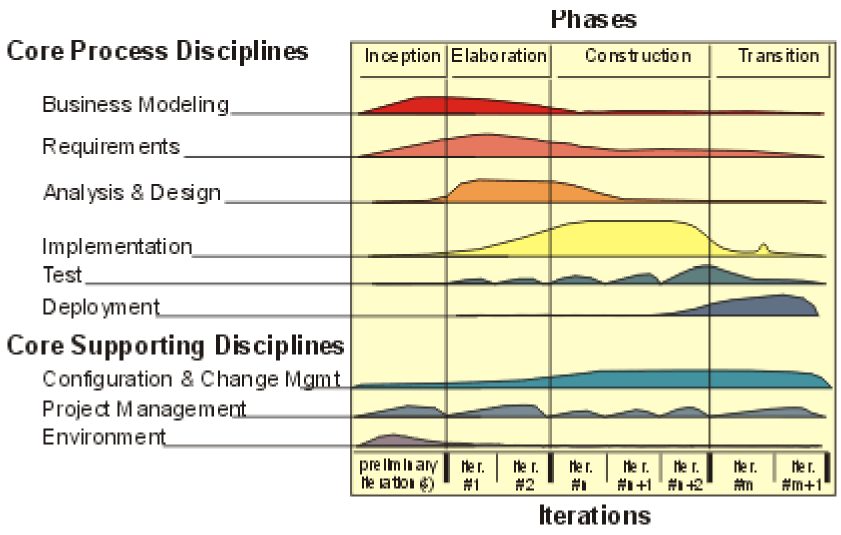


Figure 10: RUP process structure (from (Kruchten, 2004)).

Before describing the RUP, some basic definitions are needed so that you can understand the underlying structure. A *role* defines the responsibilities of team members who carry out their activities in their software process. The role of a team member evolves over time and is driven by the activities that are being performed at any given time. An *activity* is a piece of work (or "operation") that can be executed by one role. The granularity of an activity is usually a few hours to a few person-days. An activity is assigned to a particular role and must have a clear purpose, which is usually expressed in terms of creating or updating artifacts. An *artifact* is any piece of information or physical entity produced, modified, or used by the activities of the software engineering process. Artifacts are used as input to workers to perform an activity and are also the results of such activities.

RUP considers the software process as a collaboration among roles that perform activities on concrete, tangible artifacts. This association is shown as a class diagram in Figure 11.

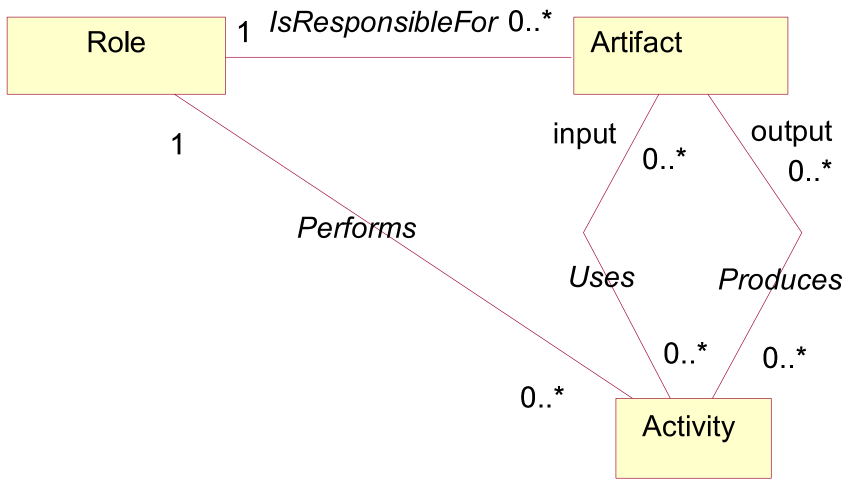


Figure 11: Association between roles, artifacts, and activities (from (Robillard and Kruchten, 2003)).

3.2 Documents and Artifacts

RUP is not just a defined, static process document. While there are published books on RUP (Jacobson, Booch et al., 1999; Kroll and Kruchten, 2003; Kruchten, 2004), most utilize the RUP as a compilation of modular, electronic resources including a versioned process definition (that is updated approximately twice per year) and a multitude of integrated support software. The RUP contains the activities, artifacts, guidelines, and examples necessary for modification and configuration by the adopting organization.

Each of the nine disciplines shown in Figure 10 is comprised of a set of specified activities. For example, in the UPEDU (the RUP for students) the requirements management discipline consists of five activities:

- Elicit stakeholder requests
- Find actors and use cases
- Structure the use-case model
- Detail a use-case model
- Review requirements

Each discipline has its own set of artifacts. The RUP defines about 30 top level artifacts. One role is responsible for each of these artifacts. For example, the UPEDU requirements management workflow has the following artifacts associated with it.

- Vision (provides stakeholders with a high level description of the product to be developed)
- Glossary (defines terms in the application domain that are needed to understand the requirements)
- Software requirements specification (functional requirements)
- Supplementary specification (nonfunctional requirements)
- Use case (describes the interactions between a user and a system)
- Use case model (all the actors and all the use cases, the totality of the functional behavior of the system)

Most often in RUP, artifacts are not paper documents. Instead, artifacts are contained within the tool that is used to create them, maintain them, and to help the developer make progress on the product. For example, a project requirements database in Rational RequisitePro is an example of an artifact.

Artifacts typically have associated guidelines and checkpoints, which present information on how to develop, evaluate, and use the artifacts. Some artifacts have concept pages associated with them. Additionally, templates (e.g. blank forms) are often associated with artifacts. Activities (or tasks) to accomplish process steps are defined in the RUP. The concept page for each artifact links the activities of the RUP by listing the activities that lead into the artifact and the activities to which the artifact contributes.

3.3 Roles

RUP defines the roles necessary to complete the disciplines. Roles have responsibility for particular activities and associated artifacts. A role in RUP may be performed by an individual or by a group of people. Equally, an individual or a group may perform several roles, even in the course of one day.

The recommended roles for a particular product are dependent upon the project characteristics (the bigger the project, the more roles are necessary). Some example roles are given here:

- Analyst, who is responsible for eliciting the needs from the stakeholders, and communicating the needs to the development team.
- Designer, who identifies and defines the design and makes sure it is consistent with the system architecture.
- Implementer, who is responsible for developing and testing software components.
- Reviewer, who reviews the software product or development activities.
- Test Designer, who defines the test approach to make sure the test is successfully implemented.
- Tester, who implements and runs the test suites.
- Integrator, who is responsible for planning and performing integration tasks.
- Project manager, who manages the resource and activities of the project to ensure the success of the outcome.
- Technical writer, who composes the communications from the developer for the product stakeholder.
- Software architect, who is responsible for the system architecture.
- User interface designer, who works with the developers to produce desired user interface.

3.4 Process

The RUP process is based upon six best practices:

1. Develop software iteratively.
2. Manage requirements.
3. Use component-based architectures.
4. Visually model software.
5. Continuously verify software.
6. Control changes to software.

In the RUP, the work is broken into nine workflows, as denoted in the vertical axis in Figure 10. These nine disciplines are briefly explained:

- **Business modeling**: Describing the business processes of a company for which the software is to be developed.
- **Requirements management**. Eliciting, organizing, and documenting requirements.
- **Analysis and design.** Creating the architecture and the design of the software system.
- **Implementation.** Writing, debugging, building and unit testing source code.
- **Test.** Testing via integration, system, and acceptance testing.

- **Deployment.** Packaging the software, creating installation scripts, writing end user documentation, and other tasks necessary for delivering the software to customers.
- **Project Management.** Project planning and monitoring.
- **Configuration and Change Management.** Planning and utilizing version and release management and change request management.
- **Environment.** Adapting the RUP process to the needs of the environment and selecting, introducing, and supporting development tools.

The product cycle is divided into four types of phases; each phase may be run in one or more iterations. The iterations are show in Figure 10 as Elab #1, Elab #2, and so on. Each iteration builds upon the results of the previous iteration(s).

- **Inception.** Defining the objectives of the project, including the business case. This involves risk analysis, initial project plans, and resource requirements.
- **Elaboration.** Creating and validating the architecture of the software system, capturing the most important and critical requirements, and planning and estimating the rest of the project. The use cases developed in the inception phase are done in greater detail.
- **Construction.** Implementing the system based on the executable architecture created in the elaboration phase.
- **Transition.** Beta testing the system with some customers and preparing release candidates.

3.5 Discussion

RUP is a "generic" process framework that can be specialized for a large class of software systems, for different application areas, different types of organizations, different competence levels, and different project sizes. Organizations can modify, adjust, and expand the RUP to accommodate their specific needs, characteristics, constraints, culture, and domain. In doing so, the organizations can benefit from the years of experience of the many people who have contributed to its development.

The initial intent of the RUP was an organization that would be able to compose a set of best practices to meet the needs of a particular project—large or small. However, many still view the RUP as impractical for small, fast-paced projects. Some have worked to dispel this perception. Object Mentor has created RUP-XP and others, such as Hirsch (Hirsch, 2002), have surfaced the adaptable, agile aspects of the process. Hirsch emphasizes the need to carefully select the proper subset of artifacts and to keep the artifacts concise and free from unnecessary formalism.

Many plan-driven methods suffer from the difficulty in "tailoring down" just discussed. (Boehm and Turner, 2003) The experts who develop the methods provide guidance for most situations and make the methods "tailor down-able." However, the users of the process don't have the experience the expert-authors have and often "take it all" in terms of the full set of plan, specification, and standards and cannot confidently tailor down to their own projects. (Boehm and Turner, 2003) This must be taken into consideration when adopting a plan driven method.

4 Summary

Several practical tips for using plan-driven methodologies were presented throughout this chapter. The keys for successful use of choosing and using plan-driven methodologies are summarized in Table 4.

🔑	The PSP can be used by an individual software engineering to improve his or her ability to consistently produce a high-quality software product on time.
🔑	If a software engineer tracks the time he or she spends on their tasks, they can make better predictions for future commitments. Software engineers are often pressured to produce software according to an imposed schedule based upon a business objective date. By making sound commitments based on historical data, engineers can astutely and defensively fend off pressure to make unrealistic commitments. When an engineer makes an unrealistic commitment, often his or her personal life is impacted in an attempt to make the commitment.
🔑	Using team roles outlined in the TSP, a team can produce their product in an organized fashion.
🔑	In TSP, the Launch meeting provides the customer with the opportunity to state and/or re-state their product objectives. Additionally, the software engineers can devise realistic commitments for meeting these objectives.
🔑	The RUP was specifically designed to use the UML diagrams taught throughout this book.
🔑	The RUP can be customized to meet the needs of the team and the product.

Table 4: Key Ideas for Plan-Driven Methodologies

In this chapter, we provided an overview of three prominent plan-driven methodologies, PSP, TSP, and RUP. We emphasized that each of these methodologies also has agile aspects to it and/or can be adapted for a situation in which agility would be beneficial or for smaller projects. The PSP is a structured framework of forms, guidelines, and procedures that guides an engineer in using a defined, measured, planned, and quality controlled process. Additionally, the PSP helps engineers understand their own skills so they can modify the process to meet their personal needs and preferences and to improve their own personal performance. The TSP provides a structure for self-directed teams to plan and track their work, to establish goals, and to create and own their processes and plans. It also provides guidance to individual software engineers on how to perform as an effective team member. Finally, the RUP is a methodology that specifically embeds object-oriented techniques and uses UML as its principle notation. Additionally, the RUP is a customizable process framework whereby, depending upon the project characteristics, the RUP can be tailored or extended to match the needs of an adopting organization.

Acknowledgements
Many thanks to Watts Humphrey and Philippe Kruchten for reviewing this chapter.

Glossary of Chapter Terms

Word	Definition	Source
Software development practice (or technique)	a disciplined, uniform approach to the software development process	(IEEE, 1990)
Software development process (or methodology)	The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes installing and checking out the software for operational use. Note: these activities might overlap or be performed iteratively.	(IEEE, 1990)
Software process model	simplified, abstracted description of a software development process	

References

Boehm, B. and R. Turner (2003). Balancing Agility and Discipline: A Guide for the Perplexed. Boston, MA, Addison Wesley.

Chillarege, R., I. Bhandari, et al. (November 1992). "Orthogonal Defect Classification -- A Concept for In-Process Measurements." IEEE Transactions on Software Engineering **18**(11): 943-956.

Ferguson, P., W. S. Humphrey, et al. (May 1997). "Results of Applying the Personal Software Process." IEEE Computer **30**(5): 24-31.

Hayes, W. and J. W. Over, "The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers," Software Engineering Institute, Pittsburgh, PA CMU/SEI-97-TR-001, December 1997.

Hilburn, T. B. and W. S. Humphrey (September/October 2002). "Teaching Teamwork." IEEE Software **19**(5): 72-77.

Hirsch, M. (2002). Making RUP Agile. Conference on Object Oriented Programming Systems Languages and Applications, Seattle, WA, ACM Press.

Humphrey, W. S. (1995). A Discipline for Software Engineering. Reading, MA, Addison Wesley.

Humphrey, W. S. (1997). Introduction to the Personal Software Process. Reading, Massachusetts, Addison-Wesley.

Humphrey, W. S. (1998). "Three Dimensions of Process Improvement, Part III: The Team Process." Crosstalk **April**(4).

Humphrey, W. S. (2000). Introduction to the Team Software Process. Reading, Massachusetts, Addison Wesley.

Humphrey, W. S. (2002). Winning with Software: An Executive Strategy. Boston, Addison Wesley.

Humphrey, W. S. (May 1996). "Using a Defined and Measured Personal Software Process." IEEE Software **13**(3): 77-88.

IEEE (1990). IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.

Jacobson, I., G. Booch, et al. (1999). The Unified Software Development Process. Reading, Massachusetts, Addison-Wesley.

Jacobson, I., M. Christerson, et al. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach. Wokingham, England, Addison-Wesley.

Kroll, P. and P. Kruchten (2003). The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP. Boston, Addison Wesley.

Kruchten, P. (2004). The Rational Unified Process: An Introduction. Boston, Addison Wesley.

Robillard, P. N. and P. Kruchten (2003). Software Engineering Process with the UPEDU. Boston, Addison Wesley.

Webb, D., Humphrey, Watts (1999). "Using the TSP on the TaskView Project." Crosstalk **February 1999**(2).

Chapter Questions

1. Name at least five characteristics of plan-driven methodologies.
2. What are the factors that distinguish RUP from the PSP and TSP?
3. Explain the relationship between PSP and TSP.
4. Name and explain the team roles of the TSP and of RUP.
5. What is the purpose of having templates, scripts, and checklists in the TSP and the PSP?
6. Based on the layout of the chapter, we can summarize three elemental components in a software process. Identify these three components.
7. List the phases in RUP.
8. Describe the process of TSPi.
9. What are the three levels of PSP?
10. In RUP, software engineering process is organized by nine disciplines. Describe the nine disciplines in RUP