**Topic X**

**Personal software process (PSP)**

**DAAD Project**

“Joint Course on Software Engineering”

Humboldt University Berlin, University of Novi Sad, University of Plovdiv, University of Skopje, University of Belgrade, University of Nis, University of Kragujevac

Parts of this topic use material from the textbook


---

**Literature**

Watts S. Humphrey

“A Discipline for Software Engineering: The Complete PSP Book”

Watts S. Humphrey

“Introduction to the Personal Software Process”

---

**PSP: A Self-Improvement Process for Software Engineers published**

Watts S. Humphrey

“A Self-Improvement Process for Software Engineers”
Addison Wesley Professional, 2005.

---

**X. Personal software process (PSP)**

- a) PSP overview
- b) PSP0 – PSP0.0 & PSP0.1
- c) PSP1 – PSP1.0 & PSP1.1
- d) PSP2 – PSP2.0 & PSP2.1
- e) PSP3
- f) PSP usage and results
What makes you better?

Why are you a better programmer than a first year student?

Why are some organizations better than others?
(deliver software on time; deliver with high quality and few defects)

Do some of the same things for a personal level scale up to organizations?

PSP - Personal software process

“Software improvement must begin at the individual level.”

Watts S. Humphrey

-arrow

Awareness

Best practices

-Commitment

Software Engineering Institute (SEI)

- An applied research laboratory situated as a college-level unit at Carnegie Mellon University.
- Supported by industry and the US government.
- Established in 1984.
- Offices in Arlington, Virginia (USA), Pittsburgh, Pennsylvania (USA), and Frankfurt (Germany).
- Mission is to foster improvement of software processes.

Watts S. Humphrey

- A fellow at the SEI of Carnegie Mellon University, which he joined in 1986.
- Established the Process Program, led initial development of the CMM, introduced the concepts of Software Process Assessment and Software Capability Evaluation, and most recently, the PSP and TSP.
- From 1959 to 1986 he was associated with IBM Corporation where he was director of programming quality and process.
- Holds a bachelor's degree in physics from the University of Chicago, a master's degree in physics from the Illinois Institute of Technology, and a master's degree in business administration from the University of Chicago.
- Was awarded an honorary Ph.D. degree in software engineering by Embry Riddle Aeronautical University in 1998.
Watts Humphrey awarded prestigious National Medal of Technology

Watts S. Humphrey has been awarded the prestigious 2003 National Medal of Technology for his contributions to the software engineering community.

- The National Medal of Technology is the highest honor awarded by the President of the United States to America's leading innovators.
- A formal ceremony took place March 14, 2005, at the White House.

The software problem

- Poor software quality in delivered systems is expensive
  - expensive service and enhancement
  - potential for accident or loss of life.
- Organizational progress with process improvement is limited because
  - process improvement takes time / is hard to sell.
- The PSP addresses these problems by
  - providing convincing evidence of the benefits of process improvement
  - exposing the engineers to the benefits of using effective processes in their work
  - teaching the engineers effective process improvement methods
  - providing the historical data to better manage cost, schedule, and quality.

Objectives for process improvement

- schedule (delivery date)
- functionality (features)
- quality (defects)

History of process models

- 1980s DoD Std 2167, AQAP 13 (NATO), industry standards (nuclear)
- 1983 IEEE 730 (software quality assurance plans)
- 1987 ISO 9000
- 1988 TickIT (ISO 9001)
- 1990 ISO 9000-3 (guideline for software)
- 1991 Capability Maturity Model (CMM)
- 1992 SPICE (ISO 15540)
from 1970s, W. Edwards Deming and J. M. Juran convinced U.S. industry to focus on improving the way people did their jobs.

Principal quality management & improvement methods:
- Most software communities rely on test-and-fix.
- Michael Fagan introduced software inspection (1976).
- Capability Maturity Model (1987): focuses on the management for the software organizations.
- Personal Software Process: individual engineers.

What is PSP? (1)

PSP is a framework of techniques to help software engineers improve their performance.
PSP is a self-improvement process designed to help control, manage, and improve the way you work.
It is a structured framework of forms, guidelines, and procedures for developing software.

(PSP as introduced in 1995 by its author, Watts S. Humphrey, SEI)

- PSP takes those large scale project methods and practices which can be used by individuals,
- structures them so that they can be introduced gradually, and
- provides a framework for teaching them.
What is PSP? (2)

- Provides engineers with a disciplined personal framework for doing software work.
- Provides a set of methods, forms, and scripts that show how to plan, measure, and manage the work.
- Emphasizes effort estimation & software quality.
- Designed for use with any programming language or design methodology.
- Can be used for any aspect of software development.
- Provides a rapid way to infuse each engineer with software engineering skill and experience.
- Goal – produce zero defect products on schedule and within planned costs.
- Key – ability to recognize and handle undue pressures.

How was the PSP developed?

- How to apply CMM to small software teams or individuals? --- personally use CMM principle
- Humphrey developed 62 programs and defined 15 PSP process versions, proved 12 KPAs of CMM were applicable to individuals.
- Humphrey wrote the manuscript.
- Howie Dow taught the first PSP course to graduates in 1993 (University of Massachusetts).
- Humphrey published the PSP textbook in later 1994.
- How to apply the same principles to engineering teams? --- Team Software Process (TSP)
- The First course to train instructors to teach PSP in industry (Jim Over & Neil Reizer)

Planning and quality principles of PSP (1)

- **PSP principles:**
  - System quality depends on the quality of its worst components.
  - Component quality depends on individual developers.
- **Every engineer is different:** to be most effective, engineers must plan their work and they must base their plans on their own personal data.
- To consistently improve their performance, engineers must personally use well-defined and measured processes.
- To produce quality products, engineers must feel personally responsible for the quality of their products. Superior products are not produced by mistake; engineers must strive to do quality work.

Planning and quality principles of PSP (2)

- It costs less to find and **fix defects** earlier in a process than later.
- It is more efficient to **prevent defects** than to find and fix them.
- The right way is always the fastest and cheapest way to do a job.
- **PSP applies a CMM-like assessment for individual work**
  - Measurement & analysis framework to help you characterize your process.
  - Self-assessment and self-monitoring.
  - Prescribes defined programming steps, data collection forms and standards.
  - Assumes individual scale & complexity.
  - Well-defined individual tasks of short duration.
What is TSP?

- A measurement driven framework that helps engineering teams more effectively develop software-intensive products.
- Provides structure, organization and discipline to running a team-based project.
- TSP is built upon PSP.
- TSP is an instance of a CMMI level 5 process for software teams.

TSP and PSP flow


Three process perspectives

PSP overview

- 7 upward compatible steps and four levels:
  - PSP0.0 & PSP0.1
  - PSP1.0 & PSP1.1
  - PSP2.0 & PSP2.1
  - PSP3

- Each level contains all the activities in the prior level plus some new ones.
- Starting point is level PSP0 - current programming practice.
- When learning PSP, software engineers write 10 module-sized programs using the PSP steps.
- They gather and analyze data on their work.
- Based on these analyses, they improve their working methods.

PSP evolution

- Cyclic Personal Process
  - PSP3
    - Cyclic development
  - PSP2
    - Design templates
  - PSP1
    - Size estimating
    - Test report
  - PSP0
    - Coding standard
    - Process improvement proposal (PIP)

- Personal Quality Management
  - PSP1.1
    - Task planning
    - Schedule planning

- Personal Planning Process
  - PSP1
    - Size estimating
    - Test report

- Baseline Personal Process
  - PSP0
    - Current process
    - Time recording
    - Defect recording
    - Defect type standard

- Process elements

- Process flow

PSP courses

- The “full” PSP course:
  - industry course: 3 intensive weeks (60 hours per week)
  - academic course: 12 to 15 weeks (3 hours per week)
  - includes 10 programming assignments and 5 reports
  - course results similar in industry and academia
  - uses *A Discipline for Software Engineering* [Humphrey 1995].

- The “introductory” PSP course:
  - does not teach the complete PSP
  - is often offered as part of a project management course
  - may start in the 1st year of a computing degree
  - uses *Introduction to the Personal Software Process* [Humphrey 1997].

X. Personal software process (PSP)

a) PSP overview
b) PSP0 – PSP0.0 & PSP0.1
c) PSP1 – PSP1.0 & PSP1.1
d) PSP2 – PSP2.0 & PSP2.1
e) PSP3
f) PSP usage and results

PSP0 – the baseline personal process

- Provides a convenient structure for small-scale tasks.
- Has three phases: planning, development (which includes design, code, compile, and test), and postmortem.
- Uses scripts and logs to measure the normal software development life cycle (design, code, compile, test).
- Forms are used to ensure a consistent and complete reporting of needed information.
- Basic measurements include the time for each phase, and the defects found and removed.
- The net result, other than the actual software, is a Plan Summary which compares the planned effort to the actual effort, and contains defect data.

PSP0 - tasks

- Define current process (PSP0.0)
- Time recording (PSP0.0)
- Defect recording (PSP 0.0)
- Defect type standard (PSP0.0)
- Code standard (PSP0.1)
- Size measurement (PS0.1)
- Process improvement proposal or PIP form (PSP0.1)
**PSP0 – current process script**

- **Planning:**
  - produce a requirement statement
  - estimate the required development time
  - estimate the total LOC (PSP0.1)
  - enter initial project data in the project plan summary
  - enter initial project data in the time recording log.
- **Development:**
  - 4 steps (PSP0.1), collect time recording log data.
  - The last step: *postmortem phase* (PM)
    - enter the plan summary form with actual data
    - complete the PIP (PSP0.1).
- **Delivery** of the finished product with forms.

**PSP0 - software plan**

- The software plan is key to the PSP.
- Primary contents of a software plan include:
  - Job size – size of the product, and time to create it
  - Job structure – how will work be done? In what order?
  - Job status – is the project on time and on schedule?
  - Assessment – how good was the plan? What lessons can be learned for next time?
  - Commitment – what will be delivered, when, and for how much money?
  - What will the product quality be? Will it meet the needs of the customer?
- Scope of the plan must be clear, or it can’t be accurate.

---

**PSP0.0 – time recording log**

<table>
<thead>
<tr>
<th>Date</th>
<th>Start</th>
<th>Stop</th>
<th>Interruption Time</th>
<th>Delta Time</th>
<th>Phase</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>7/1</td>
<td>7:58</td>
<td>8:45</td>
<td>3</td>
<td>44</td>
<td>Plan</td>
<td>Phone call</td>
</tr>
<tr>
<td></td>
<td>8:47</td>
<td>10:29</td>
<td>2</td>
<td>100</td>
<td>Design</td>
<td>Create and review design</td>
</tr>
<tr>
<td></td>
<td>7:49</td>
<td>8:59</td>
<td>70</td>
<td>Code</td>
<td>Code</td>
<td>Coded main functions</td>
</tr>
<tr>
<td></td>
<td>9:15</td>
<td>9:46</td>
<td>31</td>
<td>Compile</td>
<td>Compile and linked</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9:47</td>
<td>10:10</td>
<td>23</td>
<td>Test</td>
<td>Test</td>
<td>Ran tests A, B and C</td>
</tr>
<tr>
<td></td>
<td>4:33</td>
<td>4:51</td>
<td>18</td>
<td>Postmortem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Defects: basic quality measure (1)**

- Defects are not important to the user as long as they do not:
  - affect operations
  - cause inconvenience
  - cost time or money
  - cause loss of confidence in the program’s results.
- Low defect content is essential to a quality software process. (Experienced software engineers typically inject around 100 defects per KLOC.)
- Defects are injected at the ‘PSP level’ and this is where the engineers should: remove them, determine their causes and learn to prevent them.
Defects: basic quality measure (2)

- If you want a quality product out of test, you must put a quality product into test:
  - testing removes only a fraction of the defects
  - test is only meant to find defects - correction goes to developers.
- Data show that it is much more efficient to find defects in reviews than in testing:
  - in unit test, typically only about 2 to 4 defects are found per hour
  - code reviews typically find about 10 defects per hour
  - experienced reviewers can find 70% or more of the defects in a product
  - unit test rarely exceeds a 50% yield.
- PSP data show that reviews find 2 to 5 times as many defects per hour as unit test.

Defect injection

- Data show that it is much more efficient to find defects in reviews than in testing:
  - in unit test, typically only about 2 to 4 defects are found per hour
  - code reviews typically find about 10 defects per hour
  - experienced reviewers can find 70% or more of the defects in a product
  - unit test rarely exceeds a 50% yield.

PSP0.0 – defect recording log

<table>
<thead>
<tr>
<th>Table C18 Defect Recording Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
</tr>
<tr>
<td>Instructor</td>
</tr>
<tr>
<td>Date</td>
</tr>
<tr>
<td>Description:</td>
</tr>
</tbody>
</table>

| Date | Number | Type | Inject | Remove | Fix Time | Fix Defect |
| Description: |

| Date | Number | Type | Inject | Remove | Fix Time | Fix Defect |
| Description: |

| Date | Number | Type | Inject | Remove | Fix Time | Fix Defect |
| Description: |

| Date | Number | Type | Inject | Remove | Fix Time | Fix Defect |
| Description: |

PSP0.0 – defect type standard

<table>
<thead>
<tr>
<th>Type Number</th>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Documentation</td>
<td>comments, messages</td>
</tr>
<tr>
<td>20</td>
<td>Syntax</td>
<td>spelling, punctuation, typos, instruction formats</td>
</tr>
<tr>
<td>30</td>
<td>Build, Package</td>
<td>change management, library, version control</td>
</tr>
<tr>
<td>40</td>
<td>Assignment</td>
<td>declaration, duplicate names, scope, limits</td>
</tr>
<tr>
<td>50</td>
<td>Interface</td>
<td>procedure calls and references, I/O, user formats</td>
</tr>
<tr>
<td>60</td>
<td>Checking</td>
<td>error messages, inadequate checks</td>
</tr>
<tr>
<td>70</td>
<td>Data</td>
<td>structure, content</td>
</tr>
<tr>
<td>80</td>
<td>Function</td>
<td>logic, pointers, loops, recursion, computation, function defects</td>
</tr>
<tr>
<td>90</td>
<td>System</td>
<td>configuration, timing, memory</td>
</tr>
<tr>
<td>100</td>
<td>Environment</td>
<td>design, compile, test, or other support system problems</td>
</tr>
</tbody>
</table>
Why make plans?

- To make commitments you can meet.
- To provide a basis for agreeing on the job.
- To guide your work.
- To help track your progress.
- To project completion.

Project planning framework

1. Define the requirements
2. Produce the conceptual design
3. Estimate the product size
4. Estimate the resources
5. Produce the schedule
6. Develop the product
7. Analyze the process
8. Resources available
9. Productivity database
10. Size database
11. Schedule data
12. Developed product
13. Deliver product
14. Customer need

PSP0.1 - product size estimation

- To make better plans.
- To assist in tracking development.
- To normalize other measures:
  - development resources
  - defect rates.
- Size data is used in estimating development time and the expected number of defects.
- There are a number of criteria for good size measures:
  - has good correlation with effort
  - has a precise definition
  - can be counted automatically
  - is suitable for planning
  - is sensitive to language, design, and development method.
- LOC measure satisfies most of those criteria.

C++ LOC versus development time

CMU 94 data

LOC (lines of code) vs. time (minutes)
Screen LOC versus development time

CMU 94 data

Time (hours)

Screen LOC

Measurement precision

- When 2 people measure the same thing will they get the same result?
- To do so requires a precise measurement definition.
- The measure must also be properly applied:
  - Pascal LOC do not equate to assembler LOC
  - new LOC do not equate to modified LOC
  - logical LOC do not equate to physical LOC.

Machine countable
- Size measurement is time consuming and inaccurate.
- Automated counters can only work on definable program characteristics.
- Counters can be complex:
  - size definition selected
  - counting method.

Precision and accuracy

Imprecise and inaccurate

Precise and inaccurate

Imprecise and accurate

Precise and accurate


Suitable for early planning

- The issue is: what can you visualize early?
  - for a house, square feet predict cost
  - few people can visualize a house in terms of square feet of living space
  - numbers of rooms are more intuitive.
- Needed for good plans:
  - intuitive size measures
- Function points
  - intuitive
  - not directly measurable.
- LOC
  - not intuitive
  - directly measurable.
Selecting a size measure

- Start with product development data
  - resources required
  - product characteristics
  - any special development conditions.
- Rank products by resources required.
- See what characteristics distinguish those products that took the greatest effort from those that took the least.
- See if these differences are measurable.
- There may be no single best measure.
- If you are better at estimating resources than program size, size estimation will not improve your planning.
- If you estimate resources directly, you must:
  - keep accurate records
  - build a large database
  - use an estimating guru.

The SEI measurement framework

- Logical versus physical lines
- Statement specifications:
  - executable
  - nonexecutable
  - counted statement types.
- Application:
  - language and code type
  - origin and usage.

Counting program size

- Logical lines
  - invariant to editing changes
  - correlate with development effort
  - uniquely definable
  - complex to count.
- Physical lines
  - easy to count
  - not invariant
  - not uniquely definable.
- The PSP
  - uses a physical line for each logical line
  - uses a defined coding standard
  - this standard must be faithfully followed.
- Then physical line counting equals logical line counting.

The PSP counting standard

- Count all statements:
  - begin, end, if, then, else, etc.
  - {, }, ;, ., etc.
  - count declarations, directives, headers, etc.
- Do not count blanks, comment lines, automatically generated code, or reused code.
- Count new and changed code for measuring and estimating development productivity.
- For small programs, size tracking can be done manually, but it requires care.
- For larger programs, size tracking requires an accounting system.
- LOC accounting provides an orderly and precise way to track LOC changes through multiple program versions.
LOC type definitions

- **LOC accounting**
  - LOC of actual program changes during development. LOC are added, deleted, modified, or reused.
  - Productivity is then the number of LOC divided by the number of hours needed to produce it.

- **Code metrics**
  - **(B)** Base – LOC from a previous version
  - **(D)** Deleted – Deletions from the Base LOC
  - **(M)** Modified – Modifications to the Base LOC
  - **(A)** Added – New objects, functions, procedures, or any other added LOC
  - **(R)** Reused – LOC from a previous program that is used without modification
  - **(N)** New & Changed – The sum of Added and Modified LOC
  - **(T)** Total LOC – The total program LOC
  - **Total New Reused** – New or added LOC that were written to be reusable

### LOC accounting example

<table>
<thead>
<tr>
<th>New and changed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>170</td>
</tr>
<tr>
<td>Deleted</td>
<td>42</td>
</tr>
<tr>
<td>Modified</td>
<td>17</td>
</tr>
<tr>
<td>Added</td>
<td>79</td>
</tr>
<tr>
<td>Reused</td>
<td>51</td>
</tr>
<tr>
<td><strong>Total LOC</strong></td>
<td><strong>396</strong></td>
</tr>
<tr>
<td><strong>Total New Reuse</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

\[ W. S. Humphrey, "A Discipline for Software Engineering", 1995 \]

---

### Table C14  PSP0 Project Plan Summary

<table>
<thead>
<tr>
<th>Student</th>
<th>Date</th>
<th>Program</th>
<th>Program #</th>
<th>Instructor</th>
<th>Language</th>
<th>Time in Phase (min.)</th>
<th>Plan</th>
<th>Actual</th>
<th>To Date</th>
<th>To Date %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Postmortem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Defects Injected</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Total Development</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Defects Removed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Total Development</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>After Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ W. S. Humphrey, "A Discipline for Software Engineering", 1995 \]

---

### Table C3  PSP0.1 Project Plan Summary Example

<table>
<thead>
<tr>
<th>Student</th>
<th>Date</th>
<th>Program</th>
<th>Program #</th>
<th>Instructor</th>
<th>Language</th>
<th>Time in Phase (min.)</th>
<th>Plan</th>
<th>Actual</th>
<th>To Date</th>
<th>To Date %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Defects Injected</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Total Development</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Defects Removed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Total Development</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ W. S. Humphrey, "A Discipline for Software Engineering", 1995 \]
PSP0.1 – process improvement proposal (PIP)

- “The process is your servant. If it does not help you, you must change it.”
- Describe problems encountered on this project.
  - List difficulties, problems, and their impact.
- Describe proposals for process improvement.
- Add overall comments about the project.
  - Lessons learned.
  - Conditions you need to remember to determine why the process worked particularly well or poorly.

Some terms

- **Actual Time in Phase:** the sum of Delta time for a phase in a project
- **Time in phase To Date:** The sum of Actual Time in Phase for all completed projects.
- **Estimating Accuracy:** The degree to which the estimate matches the result, for size and time
  \[ \%\text{Error} = 100 \times \frac{(\text{Actual} - \text{Estimate})}{\text{Estimate}} \]
- **Review rate:** LOC reviewed per hour
  \[ = 60 \times \text{New and Changed LOC}\text{/review minutes} \]

PSP0 process script

<table>
<thead>
<tr>
<th>Purpose</th>
<th>To guide you in developing module-level programs.</th>
</tr>
</thead>
</table>
| Inputs Required | Problem description
  - PSP0 project plan summary form
  - Time and defect recording logs
  - Defect type standard
  - Stop watch (optional) |
| Planning | - Produce or obtain a requirements statement.
  - Estimate the required development time.
  - Enter the plan data in the project plan summary form.
  - Complete the time log. |
| Development | - Design the program.
  - Implement the design.
  - Compile the program and fix and log all defects found.
  - Test the program and fix and log all defects found.
  - Complete the time recording log. |
| Postmortem | Complete the project plan summary form with actual time, defect, and size data. |
| Exit Criteria | - A thoroughly tested program.
  - Completed project plan summary with estimated and actual data.
  - Completed defect and time logs. |

X. Personal software process (PSP)

a) PSP overview
b) PSP0 – PSP0.0 & PSP0.1
c) PSP1 – PSP1.0 & PSP1.1
d) PSP2 – PSP2.0 & PSP2.1
e) PSP3
f) PSP usage and results
**PSP1 – personal project management**

- PSP1.0 introduces size and effort estimating, and test report to PSP0.1.
- PSP1.1 adds resource and schedule estimation.
- Accumulated more project data, the estimation will become progressively more accurate.

*Tasks:*
- size estimating (PSP1.0)
- test report (PSP1.0)
- task planning (PSP1.1)
- schedule planning (PSP1.1).

**PSP1 – process script**

- **Planning**
  - Produce or obtain a requirements statement.
  - Estimate the software size and required development time (PSP1.0).
  - Complete the task plan (PSP1.1).
  - Complete the schedule plan (PSP1.1).
  - Enter initial project data in the project plan summary.
  - Enter initial data in the time recording log.

- **Development**
  - Design, Implement, Compile, Test.
  - Collect test report data (PSP1).
  - Collect time recording log data.

- **Postmortem**
  - Complete the project plan summary with actual time, defect, and size data.
  - Complete the PIP.

**Personal planning summary**

- The PSP shows students how to estimate and plan their work.
- As students gain experience, they learn to make better estimates and plans.
- The keys to making better estimates and plans are to use
  - relevant historical data
  - statistically sound methods
  - a defined estimating and planning process.

**Metrics**

- Measuring the output of a process is the first step to analyzing the process.
- Automobile companies measure how many manufacturing hours go into producing a car.
- As with any measurement, there are problems. You are trying to summarize a great deal of information into a single or small number of values.
- How much does the high and low temperatures really tell you about the weather, or the barometric pressure, or the wind direction.
- Looking at any one of these measures it is easy to find flaws, and short comings.
  - Should the temperature be recorded at ground level? Or a top a central tower? In the shade or in the sunlight?
Lines-of-code

- What unit of measure should be used to measure software output?
- The traditional and much maligned metric is called lines-of-code or LOC.
- A summary of concerns with LOC (Capers Jones):
  1) Should the code count include non-executable code, such as comments, or explanatory information?
  2) How are differences between high-level and low-level languages represented?
  3) Whether non-delivered code should be represented in a LOC count?
- So, it appears that LOC counts may not be the best way to measure software development output.

Function points

- If not LOC, then what?
- Many other metrics have been developed. Most of these metrics have major flaws.
- Albrecht developed Function Point Analysis (FPA) in the late 70's.
- This method assumes that “the amount of function to be provided by the application (program) can be estimated from an itemization of the major components of data to be used or provided by it”.
- Albrecht believes that the complexity of a software project is determined by the functions of the software.

Why estimate size?

- To make better plans:
  - to better size the job
  - to divide the job into separable elements.
- To assist in tracking progress:
  - can judge when job scope changes
  - can better measure the work.
- Value for the PSP:
  - learn estimating methods
  - build estimating skills.

Estimating background

- Estimating models in other fields
  - large base of history
  - in wide use
  - generate detailed planning data
  - require a size estimate as input.
- Software size estimating experience
  - 100% + errors are normal
  - few developers make estimates
  - fewer still use orderly methods.
Size estimating principles

- Estimating is an uncertain process
  - no one knows how big the product will be
  - the earlier the estimate, the less is known
  - estimates can be biased by business and other pressures.
- Estimating is an intuitive learning process
  - ability improves with experience and data
  - some people will be better at estimating than others.
- Estimating is a skill
  - improvement will be gradual
  - you may never get very good.
- The objective, however, is to get consistent
  - you will then understand the variability of your estimates
  - you seek an even balance between under and over estimates.

Time estimating accuracy - % error

<table>
<thead>
<tr>
<th>Program Number</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -100</td>
<td></td>
</tr>
<tr>
<td>2 0</td>
<td></td>
</tr>
<tr>
<td>3 100</td>
<td></td>
</tr>
<tr>
<td>4 200</td>
<td></td>
</tr>
<tr>
<td>5 300</td>
<td></td>
</tr>
<tr>
<td>6 400</td>
<td></td>
</tr>
<tr>
<td>7 500</td>
<td></td>
</tr>
<tr>
<td>8 Max</td>
<td></td>
</tr>
<tr>
<td>9 Class</td>
<td></td>
</tr>
<tr>
<td>10 Min</td>
<td></td>
</tr>
</tbody>
</table>

Project planning framework

- Define the requirements
- Produce the conceptual design
- Estimate the product size
  - Size database
  - Resources available
- Produce the schedule

W. S. Humphrey,
"A Discipline for Software Engineering", 1995
Estimating approaches

- Fuzzy logic
- Function points
- Standard components
- Wideband-Delphi

Fuzzy logic size estimating

- Gather size data on previously developed programs.
- Subdivide these data into size categories:
  - very large, large, medium, small, very small
  - establish size ranges
  - include all existing and expected products.
- Subdivide each range into subcategories.
- Allocate the available data to the categories.
- Establish subcategory size ranges.
- When estimating a new program, judge which category and subcategory it most closely resembles.

Fuzzy logic size estimating: (dis)advantages

+ based on relevant historical data
+ easy to use
+ requires no special tools or training
+ provides reasonably good estimates where new work is like prior experience

- requires a lot of data
- the estimators must be familiar with the historically developed programs
- only provides a crude sizing
- not useful for new program types and programs much larger/smaller than the historical data

Function point estimating

- A function point is an arbitrary unit
  - based on application functions
    - inputs, outputs, files, inquiries
  - scaled by simple, average, complex.
- For job complexity:
  - adjust a further +/- 35%
- Procedure:
  - determine numbers of each function type in the application
  - judge the scale and complexity of each function
  - calculate function point total
  - use historical data on development cost per function point to make the estimate
  - multiply function points times rate to get the estimate.
**Function point estimating: (dis)advantages**

- usable in the earliest requirements phases
- independent of programming language, product design, or development style
- there exists a large body of historical data
- it is a well documented method
- there is an active users group

- you cannot directly count an existing product's function point content
- difficult to improve estimating skill without historical data
- function points do not reflect language, design, or style differences
- function points are designed for estimating commercial data processing applications

**Standard component sizing**

- Establish the principal product size levels
  - components, modules, screens, etc.
  - determine typical sizes of each level.
- For a new product:
  - determine the component level at which estimation is practical
  - estimate how many of those components will likely be in the product
  - determine the maximum and minimum numbers possible.
- Calculate the size as the
  - number of components of each type
  - times typical sizes of each type
  - total to give size.
- Calculate for the maximum, minimum, and likely numbers of components.
- Calculate size as:
  - \[ \frac{\text{maximum} + 4 \times \text{likely} + \text{minimum}}{6} \]

**Standard component sizing: (dis)advantages**

- based on relevant historical data
- easy to use
- requires no special tools or training
- provides a rough estimate range

- must use large components early in a project
- limited data on large components

**Delphi size estimating**

- Uses several estimators
  - each makes an independent estimate
  - each submits estimate to a coordinator.
- Coordinator
  - calculates average estimate
  - enters on form: average, other estimates (anonymous), and previous estimate.
- When reestimates stabilize
  - average is the estimate
  - range is range of original estimates.
- Divergence generates discussion of issues.
- More accurate results are achieved.
Delphi size estimating: (dis)advantages

- can produce very accurate results
- utilizes organization’s skills
- can work for any sized product

- relies on a few experts
- is time consuming
- is subject to common biases

Size estimating proxies

- The basic issue
  - good size measures are detailed
  - early estimators rarely can think in detail.
- Alternatives
  - wait to estimate until you have the detail
  - make your best guess
  - identify a suitable proxy.
- A good proxy should correlate closely to development costs.
- A good proxy would be easy to visualize early in development.
- It should also be a physical entity that can be counted.
- Example proxies: function points, objects, product elements (components; screens, reports, scripts, files; book chapters).

Function points as proxies

- Data show that function point counts correlate well with development time.
- Function points can be visualized early in development.
- To use function points properly, trained estimators are required.
- Function points cannot directly be counted.
- Conversion factors are available for counting LOC and calculating function points from the LOC value.
- The function point users group (IFPUG) is refining the function point method.

Standard components as proxies

- Component count correlation with development depends on the components.
- A lot of development data is required.
- Component counts are hard to visualize early in development.
- Components are machine countable.
Objects as proxies

- Correlation with development hours
  - numbers of objects correlate reasonably well
  - object lines of code (LOC) correlate very closely
  - object LOC can be estimated using the standard component estimating method
  - then calculate LOC estimate from historical relationship between object LOC and program LOC.
- When objects are selected as application entities, they can be visualized early in development.
- Functions and procedures can often be estimated in the same way.
- Objects, functions, procedures, and their LOC can be automatically counted.

Example proxies - other

- Possible candidates
  - screens, reports, scripts, files
  - book chapters.
- If the number of items correlates with development, you estimate the number of items.
- With a suitable proxy size measure, you can often estimate proxy size.
**Size estimating overview**

- Obtain historical size data
- Produce conceptual design
- Subdivide the product into parts
- Do the parts resemble parts in the database?
- Select the database parts most like new ones
- Estimate the new part’s relative size
- Sum the estimated sizes of the new parts
- Estimate total product size

**PSP1.0 - size estimating with PROBE**

1. **Start**
   - Conceptual design
   - Identify objects
     - Number of methods
     - Object type
     - Relative size
     - Reuse categories

2. **Result?**
   - Calculate added and modified LOC
   - Estimate program size
   - Calculate prediction interval
   - Estimate

- PROBE = PROxy Based Estimating
- Requirements of Proxy: correlated with effort, can be estimated during planning, can be counted in the projects
  - OO languages: relative size of objects and their methods
  - procedural languages: functions or procedures
- Use relative size of Proxy to make initial estimate
- Use historical data to convert the relative size of Proxy into LOC
- Determine the overall size using linear regression.

**Conceptual design**

- A conceptual design is needed
  - to relate the requirements to the product
  - to define the product elements that will produce the desired functions
  - to estimate the size of what will be built.
- For understood designs, conceptual designs can be done quickly.
- If you do not understand the design, you do not know enough to make an estimate.

**Identify the objects (1)**

- Where possible, select application entities.
- Judge how many methods each object will likely contain.
- Determine the type of the object: data, calculation, file, control, etc.
- Judge the relative size of each object: very small (VS), small (S), medium (M), large (L), very large (VL).
- From historical object data, determine the size in LOC/method of each object.
Identify the objects (2)

- Multiply by the number of methods to get the estimated object LOC.
- Judge which objects will be added to the reuse library and note as “New Reused.”
- When objects do not fit an existing type, they are frequently composites.
  - Ensure they are sufficiently refined
  - Refine those that are not elemental objects
- Watch for new object types.

Statistically based estimates

- PROBE uses historical data and linear regression to relate estimates of object size to actual program size and actual development time.
- Linear regression provides the best fit, or minimum variance, of a line to these data.
- To use the regression method, you need
  - a reasonable amount of historical data
  - data that correlate.

Estimate program size (1)

- Total program size consists of
  - newly developed code (adjusted with the regression parameters)
  - reused code from the library
  - base code from prior versions, less deletions.
- Newly developed code consists of
  - base additions (BA) - additions to the base
  - new objects (NO) - newly developed objects
  - modified code (M) - base LOC that are changed.

Estimate program size (2)

- Calculate the new and changed LOC from the newly developed code
  - BA+NO+M
  - use regression to get new and changed LOC.

\[
New\&Changed = \beta_0 + \beta_1 \cdot (BA + NO + M)
\]

\[
y_k = \beta_0 + \beta_1 \cdot x_k
\]
Estimate program size (3)

- Calculate the regression parameters from data on each previously-developed program, using for the x values the sum of
  - the estimated new object LOC
  - the estimated base LOC additions
  - and the estimated modified LOC.

- For the y values, use
  - for size estimates, use the actual new and changed LOC in each finished program
  - for time estimates, use the actual total development time for each finished program.

Estimate program size (4)

- Code used from the reuse library should be counted and included in the total LOC size estimate.

- Base code consists of LOC from a previously-developed program version or modified code from the program library.

- While base code is a form of reuse, only unmodified code from the reuse library is counted as reused LOC in the PSP.

Completing the estimate

- The completed estimate consists of:
  - the estimated new and changed LOC calculated with the regression parameters
  - the 70% and 90% upper prediction interval (UPI) and lower prediction interval (LPI) for the new and changed LOC
  - the total LOC, considering new, base, reused, deleted, and modified code
  - the projected new reuse LOC to be added to the reuse library.

To make size estimates, you need several items

- Data on historical objects, divided into types.

- Estimating factors for the relative sizes of each object type.

- Regression parameters for computing new and changed LOC from:
  - estimated object LOC
  - LOC added to the base
  - modified LOC.
Historical data on objects

- Object size is highly variable
  - depends on language
  - influenced by design style
  - helps to normalize by number of methods.

- Pick basic types
  - logic, control
  - I/O, files, display
  - data, text, calculation
  - set-up, error handling.

Estimating factors for objects

- You seek size ranges for each type that will help you judge the sizes of new objects.

- To calculate these size ranges
  - take the mean
  - take the standard deviation
  - very small: VS = mean - 2*standard deviations
  - small: S = mean - standard deviation
  - medium: M = mean
  - large: L = mean + standard deviation
  - very large: VL = mean + 2*standard deviations.

Normal distribution with Si ranges

Regression parameters

- $X_{avg} =$ average of estimates
- $Y_{avg} =$ average of actual sizes

- Estimating linear progression parameters
  - $\beta_0 = Y_{avg} - \beta_1 \times X_{avg}$
  - $\sum (x_i \times y_i) - n \times x_{avg} \times y_{avg}$
  - $\beta_1 = \frac{\sum (x_i^2) - n \times (x_{avg})^2}{\sum (x_i^2) - n \times (x_{avg})^2}$

- Range = Get a calculator!!!
- Need 3 data points to get started.
Correlation

- In order for linear regression to give us meaningful results, the x and y data sets must correlate to each other (i.e., have a good “straight-line” fit).
- The degree to which two sets of data (x and y) “correlate” is given by the correlation coefficient (r).

\[ r(x, y) = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}} \]

Prediction interval (1)

- The prediction interval provides a likely range around the estimate
  - a 90% prediction interval gives the range within which 90% of the estimates will likely fall
  - it is not a forecast, only an expectation
  - it only applies if the estimate behaves like the historical data.
- It is calculated from the same data used to calculate the regression parameters.

Prediction interval (2)

- The lower prediction interval (LPI) and upper prediction interval (UPI) are calculated from the size estimate and the range where
  - LPI = Estimate - Range
  - UPI = Estimate + Range.

\[ \text{Range} = t(\alpha / 2, n-2)\sigma \sqrt{1 + \frac{1}{n} + \frac{(x_k - x_{avg})^2}{\sum (x_i - x_{avg})^2}} \]

Prediction interval (3)

- The t distribution is for
  - the two-sided distribution (alpha/2)
  - n-2 degrees of freedom.
- Sigma is the standard deviation of the regression line from the data.

\[ \sigma = \sqrt{\frac{1}{n-2} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2} \]
The t distribution

- The t distribution
  - is similar to the normal distribution
  - has fatter tails
  - is used in estimating statistical parameters from limited data.

- t distribution tables
  - typically give single-sided probability ranges
  - we use two-sided values in the prediction interval calculations.

Single-sided t distribution


Double-sided t distribution


t distribution values

- Statistical tables give the probability value p from minus infinity to x.

- For the single-sided value of the tail (the value of interest), take 1 - p.

- For the double-sided value (with two tails), take 1 - 2*(1 - p) = 2p - 1
  - look under p = 85% for a 70% interval
  - look under p = 95% for a 90% interval.
### Example: identify and size software objects

- Students first identify the objects/methods in their conceptual design.
- Then they judge the type and size of those objects.

#### Object/Method Types and Sizes

<table>
<thead>
<tr>
<th>Object/Method</th>
<th>Type</th>
<th>No. Meth.</th>
<th>Rel. Size</th>
<th>Obj LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input_Data</td>
<td>I/O</td>
<td>1</td>
<td>Large</td>
<td>22</td>
</tr>
<tr>
<td>List</td>
<td>Data</td>
<td>3</td>
<td>Medium</td>
<td>27</td>
</tr>
<tr>
<td>Calc_Mean</td>
<td>Calc</td>
<td>1</td>
<td>Medium</td>
<td>11</td>
</tr>
<tr>
<td>Calc_SD</td>
<td>Calc</td>
<td>1</td>
<td>Medium</td>
<td>11</td>
</tr>
<tr>
<td>Print_Result</td>
<td>I/O</td>
<td>1</td>
<td>Large</td>
<td>22</td>
</tr>
</tbody>
</table>

#### Example: estimate size

**CMU 94 data**

<table>
<thead>
<tr>
<th>Object/Method</th>
<th>Type</th>
<th>Obj LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input_Data</td>
<td>I/O</td>
<td>22</td>
</tr>
<tr>
<td>List</td>
<td>Data</td>
<td>27</td>
</tr>
<tr>
<td>Calc_Mean</td>
<td>Calc</td>
<td>11</td>
</tr>
<tr>
<td>Calc_SD</td>
<td>Calc</td>
<td>11</td>
</tr>
<tr>
<td>Print_Result</td>
<td>I/O</td>
<td>22</td>
</tr>
</tbody>
</table>

**Regression Parameters**

\[ \beta_0 = 38 \quad \beta_1 = 0.8 \quad r^2 = 0.8 \]

**Est N&C LOC**

- Est N&C LOC = \( \beta_0 + \beta_1 \cdot \text{Est obj LOC} \)
- Est N&C LOC = 38 + 0.8 * 93
- Est N&C LOC = 112 LOC
### Size estimating template

- **Guides the estimating process**
- **Holds the estimate data**

### PSP1 additions

- The PROBE script - already covered.
- The test report:
  - to report test plans and results
  - helpful for later regression testing.
- Project plan summary
  - LOC/hour - plan, actual, to date - to check estimates for reasonableness
  - size estimating calculations
  - actual size calculations.

### Test report

- Records the tests and results.
- Detailed enough to repeat and get the same result.

- It consists of:
  - test name and number
  - test objective
  - test description
  - any special configurations or timing conditions
  - expected results
  - actual results.

### Size estimating calculations

- When completing a size estimate, you start with the following data:
  - new and changed LOC (N): estimate
  - modified (M): estimated
  - the base LOC (B): measured
  - deleted (D): estimated
  - the reused LOC (R): measured or estimated.

- And calculate:
  - added (A): N-M
  - total (T): N+B-M-D+R.
### Actual size calculations

- When determining actual program size, you start with the following data:
  - the total LOC (T): measured
  - the base LOC (B): measured
  - deleted (D): counted
  - the reused LOC (R): measured or counted
  - modified (M): counted.

- And calculate:
  - added (A): T-B+D-R
  - new and changed (N): A+M.

### The PSP1.1 process

- The objectives of PSP1.1 are to introduce and practice methods for:
  - making resource and schedule plans
  - tracking performance against these plans
  - judging likely project completion dates.

- There are two new process elements:
  - task planning template
  - schedule planning template.

- Adds time estimation to size estimation.
- Determines tasks.
- Breaks out available project time by week.
- Plans the task sequence.
- Sets task schedule.

### Estimating accuracy

- Planning is a skill that must be developed
  - the PSP helps to build planning skill
  - even simple plans are subject to error
    - unforeseen events
    - unexpected complications
    - just plain mistakes.

- The best strategy is to plan in detail
  - identify the recognized tasks
  - estimate based on similar experience
  - make judgments on the rest.

### Project planning framework

- Define the requirements
- Produce the conceptual design
- Estimate the product size
- Estimate the resources
- Produce the schedule
- Develop the product
- Size, resource, schedule data
- Analyze the process
- Tracking Reports

**W. S. Humphrey, “A Discipline for Software Engineering”, 1995**
Planning development time


Obtain historical data

Are there sufficient data for a regression calculation?

Yes

Estimating Choice A

Estimating Choice B

Calculate the time required

Calculate the prediction interval

Calculate the shortest and longest likely times

Calculate the hours required

Calculate historical productivity in LOC per hour

LOC Size Estimate

Yes

Estimating Choice C

Resource planning process

Start with a size estimate.

Identify available data.

Use regression when you have 3+ sets of data that correlate.

Use data for estimated LOC to actual hours where available.

Calculate the prediction interval.

Time estimation

Correlate actual time to estimated size

Assume linear relation

$$\beta_0 = \text{Hours}_{\text{avg}} - \beta_1 \times \text{LOC}_{\text{avg}}$$

$$\beta_1 = \frac{\sum(\text{LOC}_i \times \text{Hours}_i) - n \times \text{LOC}_{\text{avg}} \times \text{Hours}_{\text{avg}}}{\sum(\text{LOC}_i)^2 - n \times (\text{LOC}_{\text{avg}})^2}$$

We use this to “correct” our estimate for historical fluctuation. Again at least three old points are needed. Otherwise, use personal productivity (LOC/Hr).

$$\text{Time} = \beta_0 + \beta_1 \times \text{Size}$$

Estimating effort

Regression Parameters

$$\beta_0 = 110 \quad \beta_1 = 1.5 \quad r^2 = 0.7$$

CMU 94 data

<table>
<thead>
<tr>
<th>Object/Method</th>
<th>Type</th>
<th>Obj LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input_Data</td>
<td>I/O</td>
<td>22</td>
</tr>
<tr>
<td>List</td>
<td>Data</td>
<td>27</td>
</tr>
<tr>
<td>Calc_Mean</td>
<td>Calc</td>
<td>11</td>
</tr>
<tr>
<td>Calc_SD</td>
<td>Calc</td>
<td>11</td>
</tr>
<tr>
<td>Print_Result</td>
<td>I/O</td>
<td>93</td>
</tr>
</tbody>
</table>

Data estimation

$$y = 1.4507x + 109.8$$

$$R^2 = 0.6869$$

CMU 94 data

Estimated Object LOC vs. Actual Minutes

Est Time = $\beta_0 + \beta_1 \times \text{Est obj LOC}$

Est Time = 110 + 1.5 * 93

Est Time = 250 minutes
Regression in resource planning

- The regression method for resource planning is identical to that used for size estimating.
- If multiple estimates are combined from the same data
  - combine the parts, i.e., if 3 LOC estimates were made, combine their object LOC as input to one resource estimate
  - do the same for the prediction interval.

Using multiple proxies

- If you have size/hour data for several proxies
  - estimate each as before
  - combine the total estimates and prediction intervals as just described.
- Use multiple regression if
  - there is a correlation between development time and each proxy
  - the proxies do not have separate size/hour data
  - multiple regression is covered later.

Schedule estimating

- To make a schedule you need three things
  - the estimated direct project hours
  - a calendar of available direct hours
  - the order in which the tasks will be done.
- You then need to
  - estimate the hours needed for each task
  - spread these hours over the calendar of available hours.

Available direct hours

- Staffing schedule
  - new projects are not instantly staffed
  - you need a committed staffing plan.
- Produce a calendar spread of available hours
  - at 52 weeks a year and 40 hour weeks - one year = 2080 hours
  - with 3 weeks vacation and 10 holidays, one year = 1880 hours (90%)
  - with 10% for meetings, 5% for mail... one year = 1000 to 1400 hours (50 to 65%).
Task order

- The task order must be driven by the development strategy
  - you need a conceptual approach
  - each task needs completion criteria
  - must consider task interdependencies
  - also consider cost and cycle time priorities.

- Determine planned task order
  - task order will change with new knowledge
  - the initial task order provides a basis for planning.

Produce the schedule

- Estimate the hours for each task
  - what portion of total hours have such tasks historically taken?
  - will anything unusual affect this project?
  - to ensure tasks are not omitted, spread the task time for the entire project.

- Spread the task hours over the calendar
  - identify key project checkpoints
  - use a standard format.

Earned value

- The purpose of earned value is to
  - establish a value for each task
  - permit progress tracking against the plan
  - facilitate tracking even with plan changes.

- The principles behind earned value are
  - it provides a common value for each task
  - this value is the percent of total project hours this task is planned to take
  - no value is given for partial task completion
  - major plan changes require new plans.

Establishing the planned value

- On the task template
  - total the project hours
  - calculate the % each task is of the total hours
  - enter this % as the planned value (PV) for that task
  - calculate the cumulative PV for each task.

- On the schedule template
  - enter the cumulative planned value for the tasks to be completed each week.
**Example: task planning template**

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Hours</th>
<th>PV</th>
<th>CH</th>
<th>CPV</th>
<th>Date</th>
<th>Date</th>
<th>EV</th>
<th>CEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Plan</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5/1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Design</td>
<td>25</td>
<td>31</td>
<td>30</td>
<td>37</td>
<td>5/22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Code</td>
<td>15</td>
<td>19</td>
<td>45</td>
<td>56</td>
<td>5/28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Test</td>
<td>30</td>
<td>38</td>
<td>75</td>
<td>94</td>
<td>5/31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Post</td>
<td>5</td>
<td>6</td>
<td>80</td>
<td>100</td>
<td>6/2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>80</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


**Schedule planning**

- Records estimate and actual hours expended by calendar period.
- Contains:
  - week number for each week
  - calendar date for each week
  - planned hours on that week and the total hours
  - actual hours.
- Earned value methods can help track progress, to balance
  - the amount of work accomplished (the amount of value earned)
  - the effort used (labor hours)
  - the cost expended ($).

**Example: schedule planning template**

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>DH</th>
<th>CH</th>
<th>CPV</th>
<th>ADH</th>
<th>ACH</th>
<th>CEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5/1</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5/8</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5/15</td>
<td>10</td>
<td>25</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5/22</td>
<td>25</td>
<td>50</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5/29</td>
<td>30</td>
<td>80</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Projecting project completion**

- Assume that the project will continue to earn value at the rate it has in the past.
- Extrapolate to project completion by linearly extending the EV line until it reaches 100%.
- This is the likely project completion date unless
  - the rate of progress can be accelerated
  - the work for the remaining tasks can be reduced below the original plan.

Hours Avail = 40 * .25 = 10

### Changing the plan

- For small plan changes, the earned value amounts can be adjusted as follows:
  - Assume the change is a task addition.
  - Estimate the hours for the new task.
  - Determine the new task PV%.
  - Add this amount to the project total.
  - Proportionally reduce the value of every task by the ratio 100/(100 + new task PV).

- The plan is still tracked against the original planned value schedule.
- By adding a task, the value of all the completed and planned tasks is reduced.
- When tasks are deleted, the value of all the completed and planned tasks is increased.
- For major plan changes, you must produce a new plan.

### PSP1.1 additions

- The PSP is augmented to include:
  - Resource estimating: already covered.
  - Schedule estimating: already covered.
  - A new project plan summary.

- The project plan summary adds:
  - The cost performance index (CPI) as a measure of the degree to which projects are completed within planned cost.
  - Reuse data - reuse measures are % reused and % new reused.

### Process measurement principles

- **To be useful, measurements should be**
  - Gathered for a specific purpose.
  - Explicitly defined.
  - Properly managed.
  - Properly used.

- **We measure for the following reasons**
  - To understand and manage change.
  - To predict or plan for the future.
  - To compare one product, process, or organization with another.
  - To determine adherence to standards.
  - To provide a basis for control.

### Types of measurements

- We generally seek objective, absolute, and explicit measures.
- They may be dynamic or static, predictive or explanatory.
- We seek useful relationships that correlate, for example:
  - Program size versus development hours.
  - Cost distributions.
  - Defect densities.
- We also seek controlling or predictive parameters, for example:
  - Actions to reduce test defects.
  - Steps to improve review quality.
  - Means to improve productivity.
The Goal-Question-Metric (GQM) paradigm

- The GQM paradigm establishes a framework for gathering and using data.
- It starts with an explicit statement of data gathering goals.
- Next it defines the questions the data are to answer.
- And then defines the data metrics.

Measurement goals

- What are the goals for which data are required?
  - these may be personal, project, or business
  - they should be explicit.
- Be clear on whose goals these are
  - yours, your project’s, your management’s
  - try to understand the goals at all these levels.
- Relating your goals to other’s goals provides
  - a clearer context for the measurements
  - more likely support for your work.

Measurement questions

- What will it take to meet these goals?
  - plan the actions required
  - who must take these actions?
- Do people need to be convinced to act?
  - are data needed to convince them?
  - how will these data be used?
- What is needed to implement the actions?
  - what data are required?
  - how will these data be used?

Measurement metrics

- Precisely what data are needed?
  - define the data so others could gather them
  - establish specific fields and formats.
- How will these data be gathered?
  - provide data gathering forms
  - define the data gathering process.
- How will these data be retained and used?
  - define the specific calculations/analyses
  - work through some test cases.
Making the measurements

- You are now ready to start the data gathering process.

First
- communicate the entire GQM framework
- ensure that the process is understood
- conduct a test if necessary.

Start data gathering
- monitor data quality
- provide feedback.

GQM for the PSP

- The PSP has the basic goal of assisting you in improving your personal performance.

- This suggests some likely personal goals to
  - understand your personal process
  - determine steps to improve product quality
  - understand your personal productivity
  - establish benchmarks for measuring improvement
  - make better plans.

Some general PSP questions

- How good is my process?
- Where can it be improved?
- What is the most important improvement I should make now?
- What are others doing that works better?
- How can I learn from them?

How good is my process?

- The principal dimensions of process quality are
  - product quality
  - predictability
  - productivity
  - cycle time.

- Since improvement must start from the current process, the first step is to establish measures of current process quality.

- You can then ask more informed questions.
Measuring process quality

Start with building a basic understanding of your process
• what do you do?
• how much time do you spend?
• how is this time distributed?
• how predictable is your work?
• how stable is your work?
• what is the quality of the products you produce?

The PSP addresses these issues by gathering basic process data.

PSP measurements

• The basic PSP data are
  • program size
  • time spent by phase
  • defects found and injected by phase.

• Both actual and estimated data are gathered on every item.

• The derived measures then establish
  • data to support planning
  • measures of process quality.

PSP size measures

• The goals of the PSP size measures are to
  • define a consistent size measure
  • establish a basis for normalizing time and defect data
  • help make better size estimates.

• The questions asked are
  • what size program did I plan to develop?
  • how good was my size estimate?
  • what was the complete size description of the finished program?

PSP time measures

• The goals of the PSP time measures are to
  • determine how much time you spend in each PSP phase
  • help you to make better time estimates.

• The questions asked are
  • how much time did you spend by PSP phase?
  • how much time did you plan to spend by PSP phase?
PSP defect measures

- The goals of the PSP defect measures are to
  - provide an historical baseline of defect data
  - understand the numbers and types of defects injected
  - understand the relative costs of removing defects in each PSP phase.

- The questions asked are
  - how many defects did I make in each phase?
  - how many defects did I inject in each phase?
  - how much time did it take to find and fix each defect?

PSP derived measures

- To Date and To Date %
- LOC/hour
- CPI
- % Reuse and % New Reuse.

To Date and To Date %

- **Goal**
  - provide a basis for projecting the time distribution of a new project based on the time distribution of previous projects.

- The *questions* are
  - how much time have I spent in each PSP phase for all projects to date?
  - what is the % distribution of this time?

- Note - you may wish to restart these To Date data when your process changes.

LOC/hour

- **Goals**
  - provide a convenient basis for comparing plans with historical performance
  - provide a fall-back in case the regression method does not produce a reasonable result.

- The *questions* are
  - how many LOC per hour have I produced on the most recent project?
  - what has been my average LOC/hour on prior projects?
**Goal**
- plan and manage projects so they are generally completed at or near to the committed plan.

**The question is**
- what is the ratio of the planned to the actual time for all my PSP projects to date?

---

**Goals**
- understand the degree to which previously developed code are reused
- understand the degree to which new code are added to the reuse library.

**The questions are**
- how much code was reused?
- how much was planned for reuse?
- what percent of new development contributed to the reuse library?
- what are the percents to date for these data?

---

**Some other sources of questions**
- Your peers and coworkers
- Project management
- Senior management

---

**Your peers and coworkers**
- If my work depends on yours, will you be ready in time?
- Can you provide support in areas that I need?
- If I need your support
  - when can you provide it?
  - what is the likely quality of this support?
- Are you doing something that would help me to do a better job?
**Project management questions**

- What is the project’s status?
- How does this status compare with the plan?
- Will we meet the schedule?
- Where are we in trouble?
- Where are we ahead?

**General management questions**

- Is productivity improving?
  - where can we cut costs?
  - what can we do to reduce cycle time?
  - how do we compare with competition?
- Is quality improving, by how much, and how does it compare to competition?
- What has changed since the last report and is it an improvement?

**Data gathering considerations**

- Automation
- A personal process notebook
- Forms and templates
- Databases and spreadsheets
- Summary project report

**Establishing a process baseline**

- Are you getting better?
  - you need an objective basis to tell
  - measuring alone will not cause improvement.
- Statistical variation
  - your performance will fluctuate
  - improvement is a statistical question.
- Bolstering
- Clutching
Data distributions

- Data analysis requires knowledge of the distribution of the data.

- There are many types of distributions:
  - normal, log-normal
  - Poisson
  - Chi-squared
  - Student’s t
  - F.

- The PSP analyses assume the data are normal or log-normal.

Normal distribution

- Often occurs in nature:
  - heights or weights of people
  - measurement errors.

- Properties:
  - symmetrical
  - ranges from minus to plus infinity
  - the median equals the mean.

- In adding independent normal distributions:
  - the mean of the sums = the sum of the means
  - the variance of the sums = the sums of the variances.

- Error calculations:
  - standard deviation = square root of variance
  - about 68% of values are within +/- 1 standard deviation of the mean
  - about 95% are within 2 standard deviations of the mean.

PSP1 - process script

<table>
<thead>
<tr>
<th>Phase Number</th>
<th>Purpose</th>
<th>To guide you in developing module-level programs.</th>
</tr>
</thead>
</table>
| Inputs Required | Problem description
       | PSP Project Plan Summary form
       | Historical estimated and actual size and time data
       | Time and Defect Recording Logs
       | Defect Type Standard
       | Stop watch (optional) |
| 1 Planning | Produce or obtain a requirements statement.
       | Estimate the total new and changed LOC required and the prediction interval.
       | Estimate the required development time and the prediction interval.
       | Produce a schedule plan (if more than a couple days).
       | Enter the plan data in the Project Plan Summary form.
       | Complete the Time Recording Log. |
| 2 Development | Design the program, using design templates where appropriate.
       | Review the design and fix and log all defects found.
       | Implement the design.
       | Review the code and fix and log all defects found.
       | Compile the program and fix and log all defects found.
       | Test the program and fix and log all defects found.
       | Complete the Time Recording Log. |
| 3 Postmortem | Complete the Project Plan Summary form with actual time, defect, and size data. |
| Exit Criteria | A thoroughly tested program
       | Completed Project Plan Summary with estimated and actual data
       | Completed design templates
       | Completed Design Review Checklist and Code Review Checklist
       | Completed Test Report Template
       | Complete PIP forms
       | Complete Defect and Time Recording Logs |

X. Personal software process (PSP)

a) PSP overview
b) PSP0 – PSP0.0 & PSP0.1
c) PSP1 – PSP1.0 & PSP1.1
d) PSP2 – PSP2.0 & PSP2.1
e) PSP3
f) PSP usage and results
PSP2 – personal quality management

- “The way to write a perfect program is to make yourself a perfect programmer and then just program naturally.”
- Bugs are engineered in - they aren’t caught like a cold.
- **Goal**: find and remove all defects before the first compile -- yield = 100%
- **New steps**: design review & code review
- **Detailed process**
  - During planning, estimate the number of defects that will be injected and removed in each phase.
  - Use historical correlation between review rates and yield to plan effective and efficient reviews.
  - During development, control quality by monitoring the actual defects versus planned and by comparing actual review rates to established limits.

What are reviews?

- A review is a way to personally examine your own work.
- Needed to help find and eliminate defects before those mistakes get passed on to the rest of the life cycle.
- It is one of a family of methods
  - inspections
  - walkthroughs
  - reviews.
- Much more cost effective at finding defects than testing.

Inspections

- Inspections were introduced by Fagan at IBM in 1976.
- Inspections follow a structured process
  - done by peers
  - managers do not attend
  - each participant has a defined role
  - preparation is required
  - the objective is to find problems.
- Inspections are useful for requirements, designs, code, test cases, plans, etc.

Walkthroughs

- Walkthroughs typically follow a meeting format
  - developer leads the audience through the product
  - audience may include peers, managers, or other interested parties
  - objective is to communicate or obtain approval
  - no preparation is required.
- Walkthroughs are most useful for requirements and system design issues.
**Reviews**

- In a personal review
  - professional privately reviews his/her product
  - objective is to find defects before the first compile and test
  - reviews are most effective when structured and measured.

- Reviews can be used for requirements, designs, and code.

**Why do reviews? (1)**

- Data show that it is much more efficient to find defects in reviews than in testing
  - in unit test, typically only about 2 to 4 defects are found per hour
  - code reviews typically find about 6 to 10 defects per hour
  - experienced reviewers can find 70% or more of the defects in a product
  - unit test rarely exceeds a 50% yield.

- PSP data show that reviews find 2 to 5 times as many defects per hour as unit test.

**Defect removal rates - class**

![Defect removal rates graph](image)

**Quality vs. maintenance**

- There is clear evidence that errors found early in the development cycle are far less expensive to fix than later in the cycle.

- An error
  - found in design will cost $1000 to fix
  - found just before testing will cost $6500 to fix
  - found during testing cost $15,000 to fix
  - found after release will cost between $60,000 to $100,000 to fix!

- Can we afford $100,000 to fix a $1000 problem?
**Looking at it another way**

<table>
<thead>
<tr>
<th>Cost Per Error Vs Where Error Found</th>
<th>$1,000</th>
<th>$6,500</th>
<th>$15,000</th>
<th>$80,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>$0.00</td>
<td>$10,000</td>
<td>$20,000</td>
<td>$30,000</td>
</tr>
<tr>
<td>Development Phase</td>
<td>Design</td>
<td>Code</td>
<td>Test</td>
<td>Release</td>
</tr>
</tbody>
</table>

**Why do reviews? (2)**

- After unit test, defect removal becomes much more expensive
  - in integration and system test it takes 10 to 40 programmer hours to find and fix each defect
  - inspections typically take less than an hour per defect.
- The reason to eliminate defects as early as possible is because
  - every review, inspection, compile, and test finds only a fraction of the defects
  - thus, the cleaner the code entering a phase, the cleaner it will be on exit.

**Why are reviews efficient?**

- In testing
  - you start with a problem
  - then you must find the defect
  - next, you devise a fix
  - finally, you implement and test the fix.

- With reviews and inspections
  - you see the defect
  - then you devise a fix
  - finally, you implement and review the fix.

**PSP review strategy**

- The PSP objective is to produce the highest possible program quality at every process phase.
- The review strategy to achieve this is to
  - gather data on your reviews
  - study these data
  - decide what works best for you
  - adjust your process accordingly.
- This is a continuous learning process using data on your own work.
Review principles

- PSP reviews follow a disciplined process with
  - entry and exit criteria
  - a review structure
  - guidelines, checklists, and standards.

- The suggested PSP review goal is to find every defect before the first compile or test.

- To address this goal, you should
  - use coding standards
  - use design completeness criteria
  - measure and improve your review process.

Design review principles

- Produce designs that can be reviewed.

- Follow an explicit review strategy.

- Review the design in stages.

- Verify that the logic correctly implements the requirements.

Produce designs that can be reviewed

- A reviewable design has
  - a defined context
  - a precise representation
  - a consistent and clear structure.

- This suggests that
  - the design’s purpose and function be explicitly stated
  - you have criteria for design completeness
  - the design is structured in logical elements.

Follow a review strategy

- The review strategy specifies the order in which you review the design elements.
  - this depends on the product structure
  - the review strategy should thus be considered during design.

- The objective should be to produce a design that can be
  - reviewed in stages
  - tested in stages
  - integrated in stages.
Review the design in stages

- By reviewing in stages, you ensure that all selected topics are carefully checked.
- Suggested review stages are:
  1. check that all elements have been designed
  2. verify overall program structure and flow
  3. check the logical constructs for correctness
  4. check for robustness
  5. check the function, method, and procedure calls to ensure proper use
  6. check special variables, parameters, types, and files for proper use.

Verify that the logic correctly implements the requirements

- Review the requirements to ensure that each required function is addressed by the design.
- Check carefully for oversights and omissions.

Review measures

- Explicit measures
  - the size of the program being reviewed
  - the review time
  - the numbers of defects found
  - the numbers of defects not found: the escapes.
- Derived measures
  - review yield: %found
  - LOC reviewed per hour
  - defects found per KLOC
  - defects found per review hour
  - review leverage.

Review yield

- Yield
  - a measure of process quality
  - the percent of defects in the product at review time that were found by the review
  - measures the effectiveness of a process step
    - design and code reviews
    - the overall process - prior to test
    - the development process - including test.

\[
\text{yield (for a phase or the entire process)} = 100 \times \frac{\text{defects found}}{\text{defects found} + \text{not found}}
\]
Defect removal leverage: DRL

- DRL measures the relative effectiveness of a process step at removing defects.

- DRL is the number of defects removed per hour by a process step relative to a base process:
  - the usual base is unit test
  - \(\text{DRL}(X/UT)\) is the DRL for phase X with respect to unit test.

- DRL is calculated as follows:
  \[
  \text{DRL}(X/UT) = \frac{\text{defects/hour phase X}}{\text{defects/hour unit test}}
  \]

Process control

- To control your process, you need measures that are available while you are enacting the process.

- While yield and DRL are very helpful, they are not available until after process completion.

- The potential control parameters for yield are:
  - LOC reviewed per hour
  - defects found per hour
  - defects found per KLOC.

- While none are good, LOC/Hour is best.
Review considerations

- Checklists
- Reviewing before or after compile
- The relationship of reviews and inspections

Checklists

- When performing precise tasks, it is difficult to do more than one thing well at a time.
- The checklist defines the review steps in the order suggested for performing them.
- By checking off each item, you are more likely to perform it properly.
- Establish a personal checklist that is customized to your defect experience.

Reviewing before compile

- The PSP calls for code reviews before the first compile.
- It uses compile as a check on the quality of the reviews.
  - if too many defects are found in compile, another review is called for
  - if the compile is clean, it is likely that most of the defects have been found.

Compile vs. test defects – student 1

<table>
<thead>
<tr>
<th>Compile Defects</th>
<th>Test Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

CMU 94 data
Compile vs. test defects – student 20

CMU 94 data

Reviews and inspections

- The principal focus of inspections should be to find those requirements and design problems that you have missed.

- When programs have many simple defects, inspectors are distracted and often miss more important problems.

- Reviewing the code first
  - provides a quality product for the inspection
  - shows respect for the inspectors’ time
  - produces higher quality inspections.

PSP review process

- To have effective PSP personal reviews students must
  - follow the process
  - use a personal checklist that is designed to find the defects they make
  - devise a review strategy and use it
  - review one product component at a time
  - check for one topic at a time
  - treat each check as a personal certification that the product is free of this defect
  - measure their review (time and defects).

PSP2.0 – code reviews

- Check variable and parameter initialization
- Check name and spelling use
  - Is it consistent?
  - Is it within the declared scope?
- Check strings
  - Identified by pointers and NULL terminated
- Verify all files
  - Properly declared
  - Properly opened and closed
- …
Example: code review process

- Phases include:
  - review
  - correct
  - check.

- Each student should design their own checklist so that it supports their review process.

**Table C58 C++ Code Review Checklist**

<table>
<thead>
<tr>
<th>Phase Number</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review</td>
<td>Follow the code review checklist. Check off each item as it is completed. For multiple procedures or programs, complete a separate checklist for each.</td>
</tr>
<tr>
<td>Correct</td>
<td>Correct all defects.</td>
</tr>
<tr>
<td>Check</td>
<td>Check each defect for correctness. Re-review all design changes. Record any fixed defects as new defects and, where you leave the defective file number, enter it in the fix defect box. If you do not have a fix number, enter an * in the defect box.</td>
</tr>
</tbody>
</table>

| Exit Criteria | A fully reviewed source program one or more Code Review Guideline and Checklist with every line checked. All identified defects fixed. Completed defect and time logs. |

W. S. Humphrey, "A Discipline for Software Engineering" 1995

---

Example: code review checklist (C++)

- completeness
- includes
- initialization
- calls
- names
- strings
- pointers
- output format
- { } pairs
- logic operators
- line-by-line check
- standards
- file open and close

**Table C58 C++ Code Review Checklist**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Program Unit Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>As you complete each review step, check that item in the box to the right. Complete the checklist for one program before you begin the next.</td>
</tr>
<tr>
<td>Complete</td>
<td>Verify that the code covers the design.</td>
</tr>
<tr>
<td>Time and defect log forms</td>
<td>Check the output format: line stepping is proper, spacing is proper.</td>
</tr>
<tr>
<td>Stop watch (optional)</td>
<td>Verify the proper use of ==, =,</td>
</tr>
<tr>
<td>Initialization</td>
<td>Verify that initialization is: complete</td>
</tr>
<tr>
<td>Calls</td>
<td>Check function call formats: includes, initialization, calls, names, strings, pointers, output format, { } pairs, logic operators, line-by-line check, standards, file open and close.</td>
</tr>
<tr>
<td>Notes</td>
<td>Check name spelling and use: is it consistent? is it within declared scope?</td>
</tr>
<tr>
<td>Sources</td>
<td>Verify program logic: stacks, lists, etc. are in the proper order. Recursion unwinds properly.</td>
</tr>
<tr>
<td>Stacks, lists, etc.</td>
<td>Verify that all loops are properly initiated, incremented, and terminated.</td>
</tr>
<tr>
<td>Recursion unwinds properly</td>
<td>Protect against out of limits, overflow, underflow conditions.</td>
</tr>
<tr>
<td>All specified outputs are produced</td>
<td>Ensure &quot;impossible&quot; conditions are absolutely impossible.</td>
</tr>
<tr>
<td>Protect against out of limits, overflow, underflow conditions</td>
<td>Handle all incorrect input conditions.</td>
</tr>
<tr>
<td>All needed inputs are furnished</td>
<td>Verify function use: check every logic function for proper parameters.</td>
</tr>
<tr>
<td>All required includes are stated</td>
<td>Check instruction comes from the proper program.</td>
</tr>
<tr>
<td>All specified outputs are produced</td>
<td>Verify that all externally referenced abstractions are properly initialized, incremented, and terminated.</td>
</tr>
<tr>
<td>Protect against out of limits, overflow, underflow conditions</td>
<td>Check all special cases: ensure proper operation with empty, full, minimum, maximum, negative, zero values for variables.</td>
</tr>
<tr>
<td>Ensure &quot;impossible&quot; conditions are absolutely impossible</td>
<td>Handle all incorrect input conditions.</td>
</tr>
<tr>
<td>Handle all incorrect input conditions</td>
<td>Verify that all externally referenced abstractions are properly defined.</td>
</tr>
<tr>
<td>Ensure &quot;impossible&quot; conditions are absolutely impossible</td>
<td>Review the design for conformance to all applicable design standards.</td>
</tr>
</tbody>
</table>

W. S. Humphrey, "A Discipline for Software Engineering" 1995

---

**Table C57 C++ PSP2 Design Review Checklist**

<table>
<thead>
<tr>
<th>PROGRAM NAME AND #:</th>
<th>C++ PSP2 Design Review Checklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>To guide you conducting an effective design review.</td>
</tr>
<tr>
<td>General</td>
<td>As you complete each review step, check that item in the box to the right. Complete the checklist for one program before you begin the next.</td>
</tr>
<tr>
<td>Complete</td>
<td>Verify that the code covers the design.</td>
</tr>
<tr>
<td>Time and defect log forms</td>
<td>Check the output format: line stepping is proper, spacing is proper.</td>
</tr>
<tr>
<td>Stop watch (optional)</td>
<td>Verify the proper use of ==, =,</td>
</tr>
<tr>
<td>Initialization</td>
<td>Verify that initialization is: complete</td>
</tr>
<tr>
<td>Calls</td>
<td>Check function call formats: includes, initialization, calls, names, strings, pointers, output format, { } pairs, logic operators, line-by-line check, standards, file open and close.</td>
</tr>
<tr>
<td>Notes</td>
<td>Check name spelling and use: is it consistent? is it within declared scope?</td>
</tr>
<tr>
<td>Sources</td>
<td>Verify program logic: stacks, lists, etc. are in the proper order. Recursion unwinds properly.</td>
</tr>
<tr>
<td>Stacks, lists, etc.</td>
<td>Verify that all loops are properly initiated, incremented, and terminated.</td>
</tr>
<tr>
<td>Recursion unwinds properly</td>
<td>Protect against out of limits, overflow, underflow conditions.</td>
</tr>
<tr>
<td>All specified outputs are produced</td>
<td>Ensure &quot;impossible&quot; conditions are absolutely impossible.</td>
</tr>
<tr>
<td>Protect against out of limits, overflow, underflow conditions</td>
<td>Handle all incorrect input conditions.</td>
</tr>
<tr>
<td>All needed inputs are furnished</td>
<td>Verify function use: check every logic function for proper parameters.</td>
</tr>
<tr>
<td>All required includes are stated</td>
<td>Check instruction comes from the proper program.</td>
</tr>
<tr>
<td>All specified outputs are produced</td>
<td>Verify that all externally referenced abstractions are properly defined.</td>
</tr>
<tr>
<td>Protect against out of limits, overflow, underflow conditions</td>
<td>Review the design for conformance to all applicable design standards.</td>
</tr>
</tbody>
</table>

W. S. Humphrey, "A Discipline for Software Engineering" 1995

---

**PSP2.0 – design reviews**

- Ensure the requirements, specification and high-level design are completely covered
- Verify program logic
- Check all special cases
- Verify function use
- Verify names
- Review the design for conformance to all applicable design standards
- ...
What is software quality?

- **Basic definition**
  - meeting the users’ needs
  - needs, not wants
  - true functional needs are often unknowable.

- **There is a hierarchy of needs**
  - do the required tasks
  - meet performance requirements
  - be usable and convenient
  - be economical and timely
  - be dependable and reliable.

Dependable and reliable

- **To be used, the software must**
  - install quickly and easily
  - run consistently
  - properly handle normal and abnormal cases
  - not do destructive or unexpected things
  - be essentially bug free.

- **The latent bugs must**
  - be operationally insignificant
  - not be destructive
  - rarely be evident.

PSP quality focus

- **Defects are the basic quality measure.**
- **Bugs are not important to the user as long as they do not**
  - affect operations
  - cause inconvenience
  - cost time or money
  - cause loss of confidence in the program’s results.

- **The defect content of software products must first be**
  managed before other more important quality issues can be addressed.

- **Low defect content is an essential prerequisite to a quality**
  software process.

Tests and inspections (1)

- **Without inspections and a 50,000 LOC system**
  - 25+ defects/KLOC at test entry
  - that is 1250 defects
  - at the typical 10+ hours per defect, that is 12,500+ programmer hours
  - that is 6 programmer years.

- **If properly planned, these tests could be done in**
  12 to 15 months.

- **If unplanned, testing could take two years or more.**
**Tests and inspections (2)**

- With inspections and a 50,000 LOC system
  - inspections take about 10 programmer hours per 250 LOC, or about 2,000 hours
  - this is 1 programmer year
  - if done well, inspections can remove about 80% of the defects.

- This means, 250 defects would be left for test
  - this would take about 2,500 hours
  - or a saving of 8,000 hours
  - or a saving of 4 programmer years.

**Tests and inspections (3)**

- With the PSP
  - code quality will be sharply improved
  - several thousand hours could probably be saved.

- Inspection should still be done
  - the inspection focus should be on design.

- The principal advantages are
  - improved product quality
  - a more predictable schedule
  - time to focus on the important quality issues.

**The cost of quality (COQ) (1)**

- COQ is a way to measure process quality.
- COQ has the following components:
  - Failure costs
    - repair, rework, and scrap.
  - Appraisal costs
    - costs of inspecting for defects.
  - Prevention costs
    - finding and resolving defect causes
    - generally handled before projects start
    - should typically be a process and not a project activity.

**The cost of quality (COQ) (2)**

- A useful COQ measure is the ratio of appraisal to failure costs (A/FR):
  \[
  \text{(appraisal COQ)/(failure COQ)}
  \]

- A/FR experience
  - the A/FR measure is not widely used
  - if measured, most software organizations would be near zero
  - in the PSP, A/FR should exceed 2.0
  - high A/FR is associated with low numbers of test defects and high product quality.
The quality strategy (1)

- Identify your PSP quality objectives, i.e.
  - removing all defects before the first compile
  - achieving high productivity
  - producing accurate plans.
- Establish PSP process quality measures, i.e.
  - overall process yield
  - COQ appraisal vs. failure costs - A/FR
  - LOC reviewed per hour
  - Cost performance index – CPI.
- Examine the projects you have completed
  - determine their ratings on these measures
  - see what behaviors impacted these results.
- Based on these data, identify the most effective practices for your work.

The quality strategy (2)

- Incorporate these practices in your process
  - process scripts
  - checklists
  - forms.
- Identify measures that will reasonably predict process quality
  - establish these as control variables
  - set specifications for these variables.
- Track your performance against these specifications.
- Track your process to determine
  - if and when to change the specifications
  - actions to take to improve the process further.

Process benchmarking

- A method for tracking process improvement should
  - consider quality and productivity
  - provide means for comparing processes used by different people or organizations
  - be insensitive to project specifics.
- Industrial process benchmarking typically deals with the ability of the process to
  - produce products within specifications
  - withstand drift and perturbation.

Software benchmarking

- At present, software benchmarking techniques are process dependent.
- They are still useful, however as long as we
  - establish objective measures
  - track them over time
  - use them for improving the process for which they are designed.
- Comparisons should not be made among individuals or organizations using process sensitive benchmarks.
Using software benchmarks

- Establish a consistent set of measures for evaluating your process performance
  - take these measures on every project
  - compare individual projects to determine trends or problems.

- Establish and track short-term improvement goals against these measures.

- Establish and track long-term improvement goals against these measures.

Benchmarking data

- The following data are from various of the students in the PSP course at Carnegie Mellon University in the spring of 1994.

- The data are
  - yield by project
  - yield vs. A/FR
  - A/FR vs. test defects
  - productivity by project
  - yield vs. productivity
  - A/FR vs. productivity.

Yield – student 3

Yield – student 20
Yield vs. A/F ratio – student 3

CMU 94 data

Yield vs. A/F ratio – student 20

CMU 94 data

Yield vs. A/F ratio – all students, all programs

CMU 94 data

Test defects vs. A/F – student 3

CMU 94 data
**Test defects vs. A/FR – student 20**

**Test defects vs. A/FR – class**

**Productivity trend – student 3**

**Productivity trend – student 20**
Productivity vs. A/FR – student 20

Productivity vs. A/FR – all students

Defect removal strategies (1)

- Focus inspections and reviews on specialties
  - HLD reviews - structural issues
  - DLD reviews - logic correctness
  - code reviews - implementation details.

- To save time, do not address
  - system issues in DLD
  - design issues in code reviews.

- Do the reviews thoroughly the first time and then trust them.

Defect removal strategies (2)

- Do thorough unit tests
  - check all parameters at normal values, at limits, and outside limit values
  - check all loops and recursions for normal and abnormal termination
  - check all dependencies among procedures and objects.

- Then do thorough system level testing
  - integration
  - system
  - regression.
Defect prevention

- Defect prevention is important because
  - it is always expensive to find defects
  - if the defects can be prevented, these costs are avoided
  - the defect prevention analysis costs are incurred once but save time on every project.

- Defect prevention should follow an orderly strategy and a defined process.

- For the PSP, defect prevention actions include improved design methods and prototyping.

Defect prevention strategy

- Set priorities for the defects types that are the most
  - frequently found defects
  - troublesome defects
  - easily prevented defects.

- The defect prevention process has the following steps:
  1. follow an established schedule
  2. select one or two defect types for initial action
  3. track and evaluate the effectiveness of the defect prevention actions
  4. make needed adjustments and continue.

PSP2.1 - software design

- Review is quite effective for eliminating most defect, but better quality designs are needed.

- There are six new process elements:
  - PSP2.1 project plan summary
  - PSP2.1 design review checklist
  - operational scenario template
  - functional specification template
  - state specification template
  - logic specification template.

- Examines and documents the design from different perspectives.

Design framework

- Gather data on user requirements
- Analyze the requirements data
- Validate the design against the requirements
- Conceive of a high level design
- Refine and document the design
- Obtain answers to requirements questions
- Completed Design

### Development framework

- **Requirements**
- **Design**
- **Implementation**
- **Unit test**
- **Integration test**
- **System test**
- **Acceptance**
- **Use**

- User

---

### Design cycle

1. **Requirements definition**
2. **System specification**
3. **Product 1 specification**
4. **Product 1 high-level design**
5. **Component 1-n specification**
6. **Component 1-n high-level design**
7. **Module 1nk specification**
8. **Module 1nk detailed design**

---

### Design process

- Software design is the creative process of producing a precise and effective solution to an ill-defined problem.
- The design process cannot be:
  - reduced to a routine procedure
  - automated
  - precisely controlled or predicted.
- The design process can be structured to:
  - separate the routine from the creative activities
  - ensure that the design work is properly performed
  - identify potential design support tools and methods.
- It is important to separate two issues:
  - how to do the design
  - how to represent the design when it is completed.

---

### Poor design representations cause defects

- **Levels of design**
  - an obvious design concept may not be obvious during implementation
  - reconstructing the design context during implementation is time consuming and error prone
  - to save time and prevent defects, the design should be precisely recorded when it is conceived.
- **Design visibility**
  - complex designs are difficult to visualize
  - a poor representation compounds this problem
  - a well-represented design is unambiguous.
- **Design redundancy**
  - a redundant design is often inconsistent
  - inconsistency breeds errors and causes defects
  - a quality design has minimum duplication.
**Design representation - requirements**

- The design representation must
  - precisely define all significant design aspects
  - include all important details
  - communicate the designers’ intent
  - help identify design problems and omissions.

- Also
  - the design should be compact and easy to use
  - the design topics must be readily located
  - redundancy must be avoided.

**Users of the design (1)**

- The principal users of the design are
  - implementors
  - design reviewers
  - testers and test developers
  - documenters, maintainers, and enhancers.

- The users all need
  - a clear statement of the program’s logic
  - a description of all external calls and references
  - a list of all external variables, parameters, and constants
  - a specification for all related objects and classes
  - a description of all files and messages
  - the specification of all system constraints
  - the specification of all implementation constraints.

**Users of the design (2)**

- In addition, the design and code reviewers need
  - a picture of where and how the program fits into the system
  - a structural view of the product
  - a precise statement of the program’s external functions.

- The other users need
  - typical user scenarios
  - the specification of special error checks or conditions
  - the reasons for the design choices.

- This is potentially a large amount of material
  - not all of it is needed immediately
  - some can be obtained from other sources
  - it is wise to limit the design workload as much as possible.

**Design dimensions**

<table>
<thead>
<tr>
<th>Object Specification</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Attributes</td>
<td>Inheritance</td>
</tr>
<tr>
<td></td>
<td>Constraints</td>
<td>Class Structure</td>
</tr>
<tr>
<td></td>
<td>logic specification template</td>
<td>functional specification template</td>
</tr>
<tr>
<td>Dynamic</td>
<td>State Machine</td>
<td>Messages</td>
</tr>
<tr>
<td></td>
<td>state specification template</td>
<td>operational scenario template</td>
</tr>
</tbody>
</table>

**Design hierarchy**

- Program requirements: what the user needs
- Program specifications: what the program does
- Operational Scenario
- Logic specification
- State specification
- Functional specification
- Module/object specifications

**High-level design:** how the program works

---

**Implementation hierarchy**

- Module requirements: what the module needs
- Module specifications: what the module does
- Detailed design: how the module works
- Module source code
- Operational Scenario
- Logic specification
- State specification
- Functional specification

---

**Using design templates**

- Templates comprise one way to represent a design
  - their intent is to be precise, unambiguous, non-redundant, and complete
  - use the design templates with the PSP where you can.
- Other representations may be substituted if they are equally precise, unambiguous, non-redundant, and complete.
- Additional representations are acceptable.

---

**Functional specification template**

- The purpose of the functional specification template is to unambiguously define all the external functional services provided by this product
  - the objects, classes, and inheritance
  - the externally visible attributes
  - the precise external functions provided by each object.
- Where possible, each function call and return should be specified in a formal notation.
- The functional specifications of related objects/classes should be grouped together in common templates.
### Functional specification example

- Specify first order logic postconditions of methods in a class.
- Define Class interfaces for public methods.
- Specified as \textit{condition:: action}.
- \& and \lor or

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Parent Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CData</td>
<td>CCollection</td>
<td>ListState (0-4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ListPosition (0-n)</td>
</tr>
</tbody>
</table>

#### Example:

- **bool empty()**: ListState==0::return(true)
  ListState!=0::return(false)

- **bool clear()**: ::set CData pointers to null
  (ListState==0 ^ ListPosition==0)::
  return(true)

### State specification template

- An object is a state machine when
  - identical inputs produce different responses
  - previous history is remembered by the states.
- The state specification template precisely defines the object’s states and the transitions among them.
- For each object state machine, the template specifies
  - the name of every state
  - the attributes that characterize each state
  - the attribute values for that state
  - a brief description of the state
  - the precise conditions that cause transitions from the state to itself and other states
  - the precise conditions that cause transitions from any other state to this state.

### Example state machine*

*Note: the transitions of a state to itself are not shown*

### A partial state specification

<table>
<thead>
<tr>
<th>State</th>
<th>Transition Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>First&amp;Only</td>
<td>the set has one member</td>
</tr>
<tr>
<td></td>
<td>( N = 1 )</td>
</tr>
<tr>
<td></td>
<td>ListState = 1 ListPosition = 1</td>
</tr>
<tr>
<td>EmptySet</td>
<td>Clear \lor Pop \lor (SubtractSet(D) &amp;&amp; D in ASet)</td>
</tr>
<tr>
<td>First&amp;Only</td>
<td>Reset \lor StepForward \lor StepBackward \lor (AddSet(D) &amp;&amp; D in ASet) \lor (SubtractSet(D) &amp;&amp; D not in ASet) \lor MemberSet \lor Empty \lor Last \lor Status \lor Position</td>
</tr>
<tr>
<td>FirstOfSeveral</td>
<td>Push \lor (AddSet(D) &amp;&amp; D not in ASet)</td>
</tr>
<tr>
<td>MiddleOfSeveral</td>
<td>Impossible</td>
</tr>
<tr>
<td>LastOfSeveral</td>
<td>Impossible</td>
</tr>
</tbody>
</table>

Logic specification template

- The logic specification template precisely defines the program’s internal logic.
- Describe the logic in a convenient notation
  - a pseudocode compatible with the implementation language is often appropriate
  - formal notation is also appropriate
  - the implementors must be fluent with the notation used.
- The logic specification template should specify
  - the logic for each method of each object and for the main program
  - the precise call to the program or method
  - the includes
  - special data types and data definitions
  - the project name, date, and developer.

Operational scenario template

- The operational scenario template is used to ensure that the users’ normal and abnormal interactions with the system are considered and defined both before and during the design.
- The operational scenario template can be used
  - to define test scenarios and test cases
  - to resolve development questions about operational issues
  - to resolve requirements discussions with users.
- The operational scenario template uses a scenario format.
- It contains
  - the principal user actions and system responses
  - the anticipated error and recovery conditions.

Logic specification example

- Pseudo-code for methods in classes.
- Shows all includes, type definitions needed by pseudocode.

Example:

Function: Subtract Set

INCLUDES: <iostream>
<string>

TYPE DEFINITIONS: char data[ARRAY_SIZE][STRING_LENGTH+1]

Declaration: int ASet::SubtractSet(data D)
#define ARRAY_SIZE 2
#define STRING_LENGTH 32

Reference:

<table>
<thead>
<tr>
<th>Logic reference number</th>
<th>Program Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Done = 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>If Empty`</td>
</tr>
</tbody>
</table>
| 2                      | if D==first item
|                        | delete first item |

Operational specification example

- Specify scenarios
  - interactions between user and system
  - error conditions.
- Each one accomplishes a goal.
- Operational scenario template
  - Describes operational behavior of program.

Example:

<table>
<thead>
<tr>
<th>Source</th>
<th>Step</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>1</td>
<td>Start Program</td>
<td></td>
</tr>
<tr>
<td>Program</td>
<td>2</td>
<td>Display Main Screen, Wait for input</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>3</td>
<td>Select New Project from menu</td>
<td></td>
</tr>
</tbody>
</table>
PSP2 – process script

- **Planning**
  - Produce or obtain a requirements statement.
  - Estimate software size and required development time (PSP1.0).
  - Complete the task plan (PSP1.1).
  - Complete the schedule plan (PSP1.1).
  - Enter initial project data in the project plan summary.
  - Enter initial data in the time recording log.

- **Development**
  - Design, implement, compile, test.
  - Add design review and code reviews (PSP2).
  - Use design template where appropriate (PSP2.1).
  - Collect test report data (PSP1.0).
  - Collect time recording log data.

- **Postmortem**
  - Complete the project plan summary with actual time, defect, and size data.
  - Complete the PIP.

X. Personal software process (PSP)

a) PSP overview

b) PSP0 – PSP0.0 & PSP0.1

c) PSP1 – PSP1.0 & PSP1.1

d) PSP2 – PSP2.0 & PSP2.1

e) PSP3

f) PSP usage and results

PSP3 – cyclic process

- Efficiently scales the PSP up to larger projects without sacrificing quality or productivity.

- Cyclic development strategy: decompose large programs into parts, develop and integrate them.

- **PSP3 introduces:**
  - high-level design and design review
  - cycle planning
  - development cycles based on the PSP2.1.

- **Incremental development.**

PSP3 - scaling up the PSP

- Large systems can be broken down into smaller logically separate parts, where the interaction among the parts is knowable.

- This can be repeated until the smaller parts are small enough to apply the PSP.

- **Scaling up the PSP**
  - scalability principles
  - handling software complexity
  - development strategies.
**What is scalability?**

- A product development process is scalable when the methods and techniques used will work equally well for larger projects.

- Scalability typically:
  - is limited to similar application domains
  - does not apply to unprecedented systems
  - does not work for poorly managed projects
  - is unlikely to apply where the engineering work is undisciplined.

- Scalability requires that the elements of larger projects behave like small projects.

- The product design must thus divide the project into separately developed elements.

**Scalability stages**

- stage 0 - simple routines
- stage 1 - the program
- stage 2 - the component
- stage 3 - the system
- stage 4 - the multi-system.

**Scalability stage 0**

- Stage 0 is the basic construct level.
- It concerns the construction of loops, case statements, etc.

- Stage 0 is the principal focus of initial programming courses.

- At stage 0, you consciously design each programming construct.

- When your thinking is preoccupied with these details, it is hard to visualize larger constructs.

**Scalability stage 1**

- Stage 1 concerns small programs of up to several hundred LOC.

- Movement from stage 0 to stage 1 naturally occurs with language fluency.

- You now think of small programs as entities without consciously designing their detailed constructs.

- As you gain experience at stage 1, you build a vocabulary of small program functions which you understand and can use with confidence.
Scalability stage 2

- Stage 2 is the component level.
- Here, multiple programs combine to provide sophisticated functions.
- Stage 2 components are typically several thousand LOC.
- The move from stage 1 to stage 2 comes with increased experience.
- You can now conceive of larger programs than you can possibly build alone.
- At stage 2, system issues begin to appear: quality, performance, usability, etc.

Scalability stage 3

- Stage 3 systems may be as large as several million LOC.
- Here, system issues predominate
  - the components must work together
  - the component parts must all be high quality.
- The move from stage 2 to stage 3 involves
  - handling program complexity
  - understanding system and application issues
  - working in a team environment.
- At stage 3, the principal emphasis must be on program quality.

Scalability stage 4

- Stage 4 multi-systems may contain many millions of LOC.
- Multiple semi-independent systems must work together.
- Quality is paramount.
- The move from stage 3 to stage 4 introduces large scale and distributed system issues as well as problems with centralized control.
- Stage 4 requires semi-autonomous development groups and self-directing teams.

Scalability conditions (1)

- To be scalable
  - the process must be managed
  - the project must be managed
  - the product must be managed.
- A managed process should
  - be defined
  - divide the work into separable elements
  - effectively integrate these elements into the final system.
- For a managed project
  - the work must be planned
  - the work must be managed to that plan
  - requirements changes must be controlled
  - system design and system architecture must continue throughout the project
  - configuration management must be used.
Scalability conditions (2)

- For a managed product
  - defects must be tracked and controlled
  - integration and system testing must be done
  - regression testing is used consistently
- Product quality must be high
  - module defects should be removed before integration and system test
  - the module quality objective should be to find all defects before integration and system test (i.e. miss less than 100 defects per MLOC).
- The scalability objective is to develop large products with the same quality and productivity as with small products.
- Scalability will only apply to tasks that were done on the smaller project.

The scope of scalability

Managing complexity (1)

- Size and complexity are closely related.
- While small programs can be moderately complex, the critical problem is to handle large programs.
- The size of large programs generally makes them very complex.
- Software development is largely done by individuals
  - they write small programs alone
  - larger programs are usually composed of multiple small programs.
- There are three related problems: ways to
  - develop high quality small programs
  - enable individuals to handle larger and more complex programs
  - combine these individually developed programs into larger systems.

Managing complexity (2)

- The principal problem with software complexity is that humans have limited abilities to
  - remember details
  - visualize complex relationships.
- We thus seek ways to help individuals develop increasingly complex programs
  - abstractions
  - architecture
  - reuse.
The power of abstractions (1)

- People think in conceptual chunks
  - we can actively use only 7 +/- 2 chunks
  - the richer the chunks, the more powerful thoughts.
- Software abstractions can form such chunks if
  - they are precise
  - we fully understand them
  - they perform exactly as conceived.
- Some potential software abstractions are
  - routines
  - standard procedures and reusable programs
  - complete sub-systems.

The power of abstractions (2)

- To reduce conceptual complexity, these abstractions must
  - perform precisely as specified
  - have no interactions other than as specified
  - conceptually represent coherent and self-contained system functions.
- When we think in these larger terms, we can precisely define our systems.
- We can then build the abstractions of which they are composed.
- When these abstractions are then combined into the system, they are more likely to perform as expected.

Architecture and reuse

- A system architectural design can help reduce complexity since
  - provides a coherent structural framework
  - identifies conceptually similar functions
  - permits isolation of subsystems.
- A well structured architecture facilitates the use of standard designs
  - application specifics are deferred to the lowest level
  - this enhances reusability through the use of standardized components
  - these standard components can then be used as high-level design abstractions.
  - scalability will more likely be achieved with high quality components.

Feature-oriented domain analysis

- Feature-Oriented Domain Analysis was developed by the Software Engineering Institute (SEI).
- It is an architectural design method that
  - identifies conceptually similar functions
  - categorizes these functions into classes
  - defines common abstractions for each class
  - uses parameters wherever possible
  - defers application-specific functions
  - permits maximum sharing of program elements.
**Development strategies**

- A development strategy is required when a system is too large to be built in one piece
  - it must then be partitioned into elements
  - these elements must then be developed
  - the developed elements are then integrated into the finished system.
- The strategy
  - defines the smaller elements
  - establishes the order in which they are developed
  - establishes the way in which they are integrated.
- If the strategy is appropriate and the elements are properly developed
  - the development process will scale up
  - the total development is the sum of the parts plus system design and integration.

**Some development strategies**

- Many development strategies are possible.
- The objective is to incrementally build the system so as to identify key problems at the earliest point in the process.
- Some example strategies are:
  - the progressive strategy
  - the functional enhancement strategy
  - the fast path enhancement strategy
  - the dummy strategy.

**The progressive strategy**

```
Cycle 1
In   1st Module   Out

Cycle 2
In   1st Module   1st Enhancement   Out

Cycle 3
In   1st Module   1st Enhancement   2nd Enhancement   Out
```

**Functional enhancement**

```
Core System

1st Functional Enhancement

3rd Functional Enhancement

4th Functional Enhancement

2nd Functional Enhancement
```
Fast path enhancement

The cyclic PSP flow

Team software process


1st Enhancement
2nd Enhancement
3rd Enhancement
4th Enhancement

The dummy strategy


Specifications
Product

Requirements & Planning
High-level Design
HLD Review
Cyclic Development
Post-mortem
Integration System test

Specify Cycle
Detailed Design & Design Review
Test Development and Review
Implementation and Code Review
Compile
Test
Reassess and Recycle

Core System
Function A
A
B
C

Core System
Function A
B
C

Core System
Function A
B
Function B

Core System
Function A
B
C
Function C

Team software process

- To further increase project scale, a team development process is typically required.
- This identifies the key project tasks
  - relates them to each other
  - establishes entry and exit criteria
  - assigns team member roles
  - establishes team measurements
  - establishes team goals and quality criteria.
- The team process also provides a framework within which the individual PSPs can relate, it
  - defines the team-PSP interface
  - establishes standards and measurements
  - specifies where and when to use inspections
  - establishes planning and reporting guidelines.
Inspections

- Inspections are the most cost-effective technique known for improving software quality and reducing development time and cost.
- Inspections help to
  - motivate better work
  - ensure effective team communication
  - maintain a dedication to excellence.
- The objectives of inspections are to
  - find defects at the earliest possible point
  - ensure that all parties agree on the work
  - verify that the work meets defined criteria
  - formally complete a task
  - provide data.

Inspection process

- Inspections follow a formally structured process.
- Checklists and standards are developed for each inspection type.
- The inspection is conducted by technical people for technical people. Managers do not attend.

Planning  Briefing Meeting  Preparation  Inspection Meeting  Follow-up
Manager  Moderator  Producer  Reviewers  Moderator  Producer
Producer  Reviewers  Moderator  Producer  Recorder  Producer

Inspection roles (1)

- The moderator
  - leads the inspection process
  - maintains a focus on problem identification rather than problem solution
  - ensures that the identified problems are resolved
  - submits the inspection report.
- The producer (the developer who did the work)
  - produces the review materials
  - answers questions
  - resolves the identified problems.

Inspection roles (2)

- The reviewers
  - attend the inspection kick-off meeting
  - review the work in advance
  - attend the inspection meeting
  - raise issues and questions about identified defects or other concerns.
- The recorder
  - documents the identified issues and notes the person responsible for resolving them
  - records all relevant data.
PSP3 – new elements

- **Cycle summary**
  - size / development time/ defects for each cycle
  - one copy to record cycle plans, another to enter actual cycle result.

- **Issue tracking log**
  - Documents the issues, problems and open questions that may affect future or completed cycles.

- **Project plan summary**
  - two phases are added:
    - high-level design
    - high level design review.

PSP3 – process script

- **Planning**
  - Produce or obtain a requirements statement.
  - Estimate software size and required development time (PSP1.0).
  - Complete the task plan (PSP1.1).
  - Complete the schedule plan (PSP1.1).
  - Enter initial project data in the project plan summary.
  - Enter initial data in the time recording log.

- **Development**
  - Design, implement, compile, test.
  - Add design review and code reviews (PSP2.0).
  - Use design template where appropriate (PSP2.1).
  - Use cyclic development with Cycle Summaries and Issue Tracking Log (PSP3).
  - Collect test report data (PSP1.0).
  - Collect time recording log data.

- **Postmortem**
  - Complete the project plan summary with actual time, defect, and size data.
  - Complete the PIP.

X. Personal software process (PSP)

- **a)** PSP overview
- **b)** PSP0 – PSP0.0 & PSP0.1
- **c)** PSP1 – PSP1.0 & PSP1.1
- **d)** PSP2 – PSP2.0 & PSP2.1
- **e)** PSP3
- **f)** PSP usage and results
**PSP evaluation**

- Humphrey - using PSP in SE courses:
  - Claims that PSP "substantially improves engineering performance in estimating accuracy and early defect removal while not significantly affecting productivity".
  - Provided charts to prove that as PSP exposure continues results, quality of work improves.
- Estimation accuracy increased considerably.
- Number of defects introduced per 1000 lines of code (KLOC) decreased by a factor of two.
- Number of defects per (KLOC) to be found late during the development decreases by a factor of three or more.
- Patchy, but promising use in industry.
- EASY TO LEARN!
- Still immature. Requires large overhead for data gathering.

**Personal quality management**

- In the PSP, students conduct personal design and code reviews.
- Engineers use a defined review process:
  - they design their own review checklists
  - they use a review script and rigorously follow the checklists
  - they collect data on reviews and use it to improve their review process.
- Engineers collect data, compute metrics from the data, analyze the metrics, and use the results to improve the quality of their process and products.

**PSP quality measures**

<table>
<thead>
<tr>
<th>Quality Measure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product Quality</strong></td>
<td></td>
</tr>
<tr>
<td>Total defects/KLOC</td>
<td>the number of defects found in development, per 1000 lines of code</td>
</tr>
<tr>
<td>Test defects/KLOC</td>
<td>the number of defects found in test, per 1000 lines of code</td>
</tr>
<tr>
<td><strong>Process Quality</strong></td>
<td></td>
</tr>
<tr>
<td>Yield</td>
<td>the percent of defects found before compile</td>
</tr>
<tr>
<td>Appraisal COQ (Cost of Quality)</td>
<td>the percent of total development time spent in design review and code review</td>
</tr>
<tr>
<td>Failure COQ</td>
<td>the percent of total development time spent in compile and test</td>
</tr>
<tr>
<td>Total COQ</td>
<td>Appraisal COQ + Failure COQ</td>
</tr>
<tr>
<td>A/F R</td>
<td>Appraisal COQ / Failure COQ</td>
</tr>
<tr>
<td>Review rate - LOC/hour</td>
<td>the number of lines of code reviewed per hour in a review (code review or design review)</td>
</tr>
<tr>
<td>Defect removal rate - defects/hour</td>
<td>the rate at which defects are removed in a defect removal phase (design review, code review, compile, test)</td>
</tr>
</tbody>
</table>

**Total defects/KLOC removed**

![Graph showing the reduction in defects per KLOC removed over assignment numbers](image-url)
Defect fix times

Defect fix times (in minutes) for an individual PSP student:

<table>
<thead>
<tr>
<th>Defects Injected</th>
<th>Defects Found</th>
<th>Design Review</th>
<th>Code Review</th>
<th>Compile</th>
<th>Test</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tot. Fix Time</td>
<td>21</td>
<td>5</td>
<td>3</td>
<td>109</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>Tot. Defects</td>
<td>17</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Avg. Fix Time</td>
<td>1.24</td>
<td>1.25</td>
<td>3.00</td>
<td>18.17</td>
<td>4.93</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tot. Fix Time</td>
<td>32</td>
<td>52</td>
<td>75</td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tot. Defects</td>
<td>29</td>
<td>42</td>
<td>15</td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg. Fix Time</td>
<td>1.10</td>
<td>1.24</td>
<td>5.00</td>
<td>1.85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tot. Fix Time</td>
<td>21</td>
<td>37</td>
<td>55</td>
<td>184</td>
<td>297</td>
<td></td>
</tr>
<tr>
<td>Tot. Defects</td>
<td>17</td>
<td>33</td>
<td>43</td>
<td>21</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>Avg. Fix Time</td>
<td>1.24</td>
<td>1.12</td>
<td>1.28</td>
<td>8.76</td>
<td>2.61</td>
<td></td>
</tr>
</tbody>
</table>

Test Defects/KLOC vs. A/F R

Test Defects vs. A/FR - Class
### Time invested per (new and changed) LOC

![Graph showing time invested per LOC over program number]


### Personal PSP implications (1)

- If you seek personal excellence, the PSP can help you to attain it.
- By defining and measuring your work, you gain the knowledge to improve your personal performance.
- The question is: “Do you want to improve?”
- As a software professional you need to
  - make commitments you can meet
  - deal with unreasonable commitment pressures
  - review status and plans with customers, managers, and coworkers.

### Personal PSP implications (2)

- The PSP involves change and change involves risk.
  - your methods may have sufficed in the past
  - no one else may use disciplined personal practices.
- But the problems of the future will be more challenging than those of today.
  - Will your current methods be adequate?
  - Do you know a better way?
- In using the PSP, you may face resistance.
  - Do you have a supportive environment?
  - Does your management agree with your interest in personal improvement?

### Making personal commitments

- The managers want the tightest plan that will actually deliver the product.
- Your most successful course is to strive to produce a good plan, regardless of the pressure. Then defend it.
- **A strategy for defending your plan:**
  - This is the best plan I can make to meet the requirements I was given. (Show assumptions, historical data, comparison with similar project.)
  - If you want to change the requirements or the assumptions, I will reexamine the plan.
  - This is minimum cost plan, if you are concerned about schedule, we could save time but at added cost.
The benefits of the PSP (1)

- **Insight into your talents and abilities**
  - you will better understand your strengths and weaknesses
  - you will be better able to maximize your assets
  - the PSP will help you to objectively deal with your weaknesses.

- **Improvement ideas**
  - by defining your process, you assert control over it
  - you will unconsciously observe your working self
  - you will see many ways to improve your process and your performance.

- **Improvement framework**
  - you can better see how the process parts relate
  - you can better focus on priority areas for improvement.

The benefits of the PSP (2)

- **Personal control**
  - you will have a planning framework
  - you will have the data on which to base your plans
  - your plans will be more reliable
  - you will be better able to track your status
  - you will be better able to manage your work.

- **Accomplishments and personal bests**
  - you will recognize your personal bests
  - you will better understand how to repeat and to surpass them
  - you will see where and how you have improved
  - you will have your own personal improvement goals
  - you will have the satisfaction that comes with knowing you are doing superior work.

The costs of the PSP

- **Labor intensive** - requires significant training, experience, and management support to realize benefits.
- **Tedious task**
- **The time investment**
  - process development takes about 1 to 2 hours per form and script
  - process updates will be needed at least every 3 months
  - data entry and analysis will take about an hour for each PSP-sized project.
- **Suggestions**
  - try to promptly enter your process data
  - analyze your data every few weeks.
- **The emotional investment** - occasional frustrations.
- **Lack of support tools** - requires good CASE tool support to record and gather metrics with minimal effort.

Return on investment (ROI)

<table>
<thead>
<tr>
<th></th>
<th>Without PSP</th>
<th>With PSP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Defects</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Defects</td>
<td>1,000</td>
<td>500</td>
</tr>
<tr>
<td>Found In Code Review</td>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td>Defects Remaining</td>
<td>1000</td>
<td>200</td>
</tr>
<tr>
<td>Found In Compile</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>Found In Unit Test</td>
<td>400</td>
<td>80</td>
</tr>
<tr>
<td>Defects Remaining</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td><strong>Time (hours)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Review Time</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>Compile Time</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>Unit Test Time</td>
<td>200</td>
<td>40</td>
</tr>
<tr>
<td>Personal Defect Removal Time</td>
<td>240</td>
<td>100</td>
</tr>
<tr>
<td><strong>Integration And System Test Time</strong></td>
<td>1000</td>
<td>200</td>
</tr>
<tr>
<td><strong>Total Defect-Removal Time</strong></td>
<td>1240</td>
<td>300</td>
</tr>
</tbody>
</table>
Teradyne example - benefits of using PSP & TSP

- The first organization to report their results after using both PSP and TSP was Teradyne.
- Their return on investment (ROI) analysis indicates that by using PSP and TSP on two projects totaling 112 KLOC, they saved $5.3 million in engineering time, or approximately 120 hours/KLOC in integration, system, and field testing.
- Quality levels improved 20 times over prior projects and actual effort and schedule were within 8% of plan (early).
- Teradyne estimates the cost of PSP training to be one month per engineer.

Data on the benefits of TSP & PSP (1)

This data is a compilation of results from 18 TSP/PSP projects completed by Boeing, AIS, Hill Air Force Base, and Teradyne.

Data on the benefits of TSP & PSP (2)

This data is a compilation of results from 18 TSP/PSP projects completed by Boeing, AIS, Hill Air Force Base, and Teradyne.

Data on the benefits of TSP & PSP (3)

This data is a compilation of results from 18 TSP/PSP projects completed by Boeing, AIS, Hill Air Force Base, and Teradyne.
Using the PSP in an organization

- Introducing the PSP into an organization involves 2 situations:
  - *the solo PSP performer* - you are the only person using the PSP in your organization
  - *the lone PSP team* - your team uses the PSP but they are the only team in the organization to do so.

- You will also need management support for PSP introduction.

The solo PSP performer

- It is hard to maintain personal discipline without the support of peers and managers.
- It is easy to get discouraged by a slow rate of personal progress.
- Your peers may kid you for wasting your time with the PSP.
- If you are not confident that the PSP helps you, it will be hard to withstand such criticism.
  - normal statistical fluctuations will seem like major disasters
  - instead of learning from your mistakes you may get defensive about them.
- Until you have data to support the benefits of the PSP, you would be wise to say little about it.

The lone PSP team (1)

- When your team has been trained in the PSP, you will have a powerful base of support.
- You will be able to
  - review each others’ work
  - share process improvement ideas and results
  - celebrate successes
  - get support when you need it.
- Be cautious about describing your results. Other groups may
  - critique your results
  - argue that they already do better work.

The lone PSP team (2)

- They are probably comparing their occasional best results with your normal performance.
- Without consistent data, such comparisons are meaningless and should be avoided.
- If your results are superior, others may feel defensive.
- Be careful not to seem critical of other peoples’ work
  - do not imply that your results apply to them
  - suggest they try the PSP for themselves.
- Concentrate on how the PSP has helped you to improve.
### Organizational PSP support

- To be most effective, you need organizational support.
  - education and training
  - database and analysis
  - process definition
  - tools.
- To get these, you will need management’s help.
- You may have trouble getting management support unless
  - they see your work as a prototype for the organization
  - you have data to demonstrate the benefits of the PSP for your team.

### Introducing the PSP (1)

- In getting management support, show enough of your own and other groups’ data to convince them that
  - there are important benefits
  - the costs are controllable.
- In introducing the PSP, it is essential that
  - it is introduced with a formal course
  - all professionals voluntarily participate
  - the engineers be given time to do the work
  - the managers provide weekly support and encouragement to their engineers to complete the PSP exercises
  - the engineers’ personal data be respected as their private property.

### Introducing the PSP (2)

- Where possible, do the PSP training by project team.
- Attempt to build clusters of PSP-trained teams that can reinforce and support each other.
- In selecting the initial projects, try to pick ones that are not in perpetual crisis.
- After PSP training, adapt the PSP to each project by
  - measuring and planning the current process
  - adjusting PSP2.1 or PSP3 to the project needs
  - testing each process change before general introduction
  - planning for continuous process improvement.

### The responsible professional

- As a responsible professional, you need to:
  - find and learn new methods
  - use these methods in your work
  - recognize your strengths and weaknesses
  - identify areas for improvement
  - practice, practice, practice
  - publicize the methods you find helpful
  - learn from history.
- This will assure your continuing improvement.
Coaching and improvement

- A coach can help you to improve your personal performance by
  - motivating superior performance
  - demanding personal commitment
  - insisting on a dedication to excellence
  - supporting and guiding your development.

- Seek coaching support from your peers and your managers.

Messages to remember

1. The PSP is designed to help you.

2. Use it to improve your performance and to help you to act professionally in difficult situations.

3. Emphasize what the PSP has done for you and urge others to find out for themselves what it will do for them.