Design & Code Reviews

Outline

- Review of PSP Levels
- Introduction
- Why Review?
- Review Principles
- Design Review Principles
- Review Measures
- Checklists
- Reviewing Before vs. After Compiling
- Reviews & Inspections
- Homework #6 - Part 2
Review of PSP Levels (Humphrey, 1995, p. 11)

- PSP0
  - Current process
  - Time recording
  - Defect recording
  - Defect type standard

- PSP1
  - Size estimating
  - Test report

- PSP1.1
  - Task planning
  - Schedule planning

- PSP2
  - Code reviews
  - Design reviews

- PSP2.1
  - Design templates

- PSP3
  - Cyclic development

Quality Mgt
Planning
Baseline

Introduction (cf. Humphrey, 1995, p. 231)

- “Design and code reviews… [provide] more improvement… than… any other single change you can make in your personal software process.”

- “Doing reviews is the most important step you can take to improve your software engineering performance.”
Three Types of Reviews
(cf. Humphrey, 1995, p. 231-233)

- Inspection - team review
  - Prepare at initial meeting
  - Inspect separately, then in meeting
  - Author repairs, report is made, track to closure

- Walkthrough - less formal team review
  - Author makes presentation
  - Developers & users can participate
    - ID omissions & misunderstandings
    - Educate
  - Little advance preparation or follow-up is necessary

- Personal review - ID/fix as many defects as possible before compile, inspection, compile, or test
  - This was the standard practice before PC’s, fast compilers, and integrated graphical environments became the norm.
  - They save time later

Products to Review
(cf. Humphrey, 1995, p. 233)

- All SW products can be reviewed
- Reviewing early products provide most benefit.
  - Early products are even more critical for the whole SW development process.
  - They are easier and cheaper to review.

Products:
- Analysis
- Design
- Code
- Documentation
- Development plans
- Test cases / plans
- ...

AU INSY 560, Singapore 1997, Dan Turk
Humphrey Ch. 8 - slide 5

- The secret to good writing is re-writing.
- Many beginning PSP-users spend more than 33% of their development time on compiling and testing. At the end of the A-series programs students spend about 10% (or less).

**Conclusion:**
- Reviews improved time, efficiency, predictability, and quality
- cf. student data graphs, Fig. 8.1 & 2, p. 234


- The biggest single problem with reviews is convincing yourself of their value.
- It doesn’t seem worthwhile when you have a powerful compiler / debugger to find (some) defects for you…

- The only way to convince yourself is to collect data and see.
  - Table 8.1, p. 235, shows 8-12 times more time for unit test fix vs. code review, and 16-60 times for post unit-test fix…!
  - Fig 8.3, p. 236 shows 3-5 times more defects per hour for code review than test.
Code reviews are more efficient than testing:

- **Reviews**
  - Defects are found directly
  - You build a mental model of the program
  - Thus it's easier to fix errors when they are found
- **Testing**
  - Only symptoms of defects are found
- **Debugging**
  - You must search for the causes of the defects which were found in testing
- **Examples:**
  - Three months searching vs. 2 hours inspection: inspection found the error plus 71 others!
  - Three days searching for one misplaced semicolon after a for statement....

Debuggers are good for stepping through program logic and checking parameter values.

- This is helpful if you know what the values should be.
- In order to know this you have to understand the program logic.
- Conclusion: Why not thoroughly check the logic ahead of time since you need to know it anyway?!

Most professional programmers have about 100 defects / KLOC.

- Before using reviews, PSP students found approximately 50% of their defects in compile.
- Thus 50% were left for test.

You must decide the most efficient way to find them.

Collect personal data to convince yourself.

- Establish review goals
- Follow a defined review process
- Measure & improve your review process


- Ex:
  - 100% defect removal before first compile
- Reality:
  - Most people will achieve 50-80%
Review Principles: Follow Defined Process

- A defined process will include for each activity:
  - Entry & exit criteria
  - Tasks to perform
  - cf. Table 8.2, Code Review Script (Design script is very similar)
  - cf. Table 8.3, Checklist
- Keep script and checklist separate
  - Facilitates planning
  - Easier to update

Review Principles: Measure & Improve Your Process
(cf. Humphrey, 1995, p. 243)

- You measure reviews in order to improve their quality
- A high-quality review finds the most defects in the least amount of time
- In order to track this you must know:
  - Review time
  - Number of defects found
  - Number of defects found after review
Review Principles: Keep Design & Code Reviews Separate
(cf. Humphrey, 1995, p. 243)

- Keeping design and code reviews separate helps:
  - Make designs more understandable
  - Save implementation time
  - Avoid missing product defects
  - Spot possible design improvements

- When design & code reviews are kept separate you are more likely to:
  - Look for design alternatives
  - Look for ways to make the design neater and/or cleaner

Four Design Review Principles
(cf. Humphrey, 1995, p. 244-247)

- Produce reviewable designs
- Follow an explicit review strategy
- Review the design in stages
- Verify that the logic correctly implements the requirements
Design Review Principles: Reviewable Designs
(cf. Humphrey, 1995, p. 245)

- For a design to be reviewable:
  - It’s purpose and function must be explicitly stated.
    - Explicitly list program’s required functions and constraints, conditions, standards.
  - The design description must be complete and precise.
    - System issues that affect the design should be noted.
    - Ex: performance, memory, usability
  - The design must be segmented into logical elements.
    - This facilitates limited reviews at one time.
    - Rule of thumb: One page of text.

- Gather data and find out what works best for you.
  - Have we seen this theme before?!

Design Review Principles: Explicit Strategy
(cf. Humphrey, 1995, p. 245-246)

- Following a specific design / development sequence provides a context and the ability to coordinate and/or integrate designs.
Design Review Principles: Review in Stages
(cf. Humphrey, 1995, p. 246-247)

Guidelines:
- Check for all required program elements.
- Verify overall program structure and flow.
- Check correctness of logical constructs.
- Check logic for robustness. (Stress test.)
- Check function calls - parameter number, order, & type; valid values.
- Check special variables, data types, files.

Human vs. Compiler checking of names & types
- If you don’t have name / type defects then don’t worry about this during design review

Humphrey:
- During design review manually check global variables and state controlling parameters, and all specially declared types.
- Check all others during code review

Design Review Principles: Verify Logic vs. Requirements
(cf. Humphrey, 1995, p. 247)

Checking that the program’s logic meets the requirements is:
- Hard work
- The only way to check for oversights and/or omissions

- **There are 4 explicit review measures:**
  - Reviewed program size - LOC
    - PC and PI would help to have common size measure throughout
  - Review time - minutes
  - Number of defects found
  - Number of escapes - defects found later

- **Derived measures:**
  - Review yield = % defects found during review
  - Defects / KLOC design or code reviewed
  - Defects / Hour
  - LOC reviewed / Hour
  - DRL = defect removal leverage
    - relative rate of defect removal for any two process phases


- **Review yield**
  - Is the best measure of review quality
  - Is the % of defects in design or code at the time of review which were found by the review
  - You can’t calculate this precisely until later

- **cf. Table 8.4, Yield Calculation Ex.**
- **cf. Table 8.5, corresponding Defect Log**
- **cf. Table 8.6, Ex. defect summary (net escapes, ...) and formulas**
- **cf. Fig 8.5, Ex C++ Code Review Yield**
- **cf. Fig 8.6, Ex Student yield data**
Instant Review Measures
(cf. Humphrey, 1995, p. 251-256)

- You need measures which can be gathered at the current time which correlate with yield.
  - This tells how good you’re doing while you’re doing reviews.
  - % yield is not known until the end.

Examples:
- Defects / KLOC
  - Problem:
    - Is low yield due to superficial review or did you start with few defects?
    - Fig. 8.7, p. 253 doesn’t show strong correlation.
- Defects / Hour
  - 200 LOC / Hour optimal
  - cf. Fig 8.9, p. 255

Instant Review Measures: DRL
(cf. Humphrey, 1995, p. 256-257)

- DRL = Defect Removal Leverage
  - Measures relative effectiveness
  - Ratio of defects removed / Hour for any two phases

- Most used to compare test phase with some other phase

Examples
- cf. Table 8.7, Student PSP 10a data
- cf. Table 8.8 & Fig 8.11, Humphrey’s PSP data

- Checklists are very important
  - Example: airline pilots’ preflight checks

Using Checklists
- Review 1 topic at a time
- Review 1 program section at a time
- Design reviews are best performed top-down
- Code reviews are best performed bottom-up (unless you are unfamiliar with the code)


- Building Checklists
  - Review your defect data to see where you should focus
  - Start with the PSP0 defect standard (Tables 8.9 & 10) information the checklist
  - Modify the checklist based on your defects-found (Pareto) distribution
    - Categories not to worry about
    - Subcategories
    - cf. Fig 8.12, p. 261, Pareto distribution (sorted by frequency)
    - Focus on most-frequently found defect types, and see how you can improve your rate.
    - Don’t drop checking for low-frequency “found” review items, just those that you are not having.
      - You’re finding these!
      - If you drop them you’ll have to find them in test…
  - Check coding standard items in your reviews
Reviewing Before vs. After Compiling  
(cf. Humphrey, 1995, p. 263-264)

- This is not a simple issue
- Not 100% of syntax errors are caught by the compiler
  - 8.7-9.3% of Humphrey’s weren’t
  - These may actually be thought of as semantic, not syntax, errors: the code does not do what was intended.
- cf. Fig 8.13, p. 264, Defect types found / missed

Reviewing Before vs. After Compiling: Pros & Cons  
(cf. Humphrey, 1995, p. 264-265)

Compiling First:
- Compiling has 2x DRL for some defect types
- 90% of syntax & naming defects found
- Individual review effectiveness varies: may miss from 20-50% of syntax defects
- Syntax defects missed by compiler are easy to find

Reviewing First:
- Compiler misses about 9% of syntax defects
- Finding defects in review saves both compile time and makes it more predictable
- It generally takes longer to fix syntax errors in test than in review
- Unit testing generally finds about 1/2 of a program’s defects. If you find more defects before test then your total found is likely to go up.
- Later test phases are even less efficient than unit test
- Hard to do thorough job reviewing pre-compiled code because there are few defects. You lose interest…
- You won’t save any time by compiling first; reviewing first saves time in compile and in later test.
Reviewing Before vs. After Compiling: Objectives
(cf. Humphrey, 1995, p. 265-266)

- **What is your goal?**
  - Do you want to get to test as soon as possible, or do you want to remove the most defects?
- **Don’t confuse speed with progress!**
- **If you are trying to remove the most defects, then you might as well review where it is most effective.**

Reviews & Inspections
(cf. Humphrey, 1995, p. 267-268)

- You should perform (group) inspections in addition to your personal reviews
  - Include all involved people’s time in your Time Log
- **Question: Where to inspect?**
  - Review code before inspection?
  - Compile code before inspection?
- **Answers**
  - Give inspectors as clean code as possible - review it first: polite, they’ll focus better.
  - When improving your review process - inspect before compile.
  - When you have a good review process - compile before inspection.
  - Don’t unit test first.
Homework #6 - Part 2

- See “Homework Assignments” list and textbook instructions.