Chapters 14 - 17

Summary & Highlights
Software Quality

• In 2005, *ComputerWorld* [Hil05] lamented that
  – “bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction.

• A year later, *InfoWorld* [Fos06] wrote about the
  – “the sorry state of software quality” reporting that the quality problem had not gotten any better.

• Today, software quality remains an issue, but who is to blame?
  – Customers blame developers, arguing that sloppy practices lead to low-quality software.
  – Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated.
Quality

• The *American Heritage Dictionary* defines *quality* as
  — “a characteristic or attribute of something.”
• For software, two kinds of quality may be encountered:
  — *Quality of design* encompasses requirements, specifications, and the design of the system.
  — *Quality of conformance* is an issue focused primarily on implementation.
  — *User satisfaction* = compliant product + good quality + delivery within budget and schedule
The Software Quality Dilemma

- If you produce a software system that has terrible quality, you lose because no one will want to buy it.
- If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway.
- Either you missed the market window, or you simply exhausted all your resources.
- So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete. [Ven03]
Quality Issues

• **Cost:** A balancing act
  – Loss of market share & eroded reputation due to bad software
  – Loss of market share due to late-arriving software
  – Recoverability of costs to improve/fix the software

• **Risk**
  – End use of software may have literal life-death consequences
  – Negligence/Liability: in calculating taxes, e.g., or estimating bridge cable tensile strength

• **Security**
  – Exploitation of vulnerabilities
  – Consequences range from benign (defaced website) to catastrophic (atomic weapon launch codes)

• **A stitch in time...**
  – Fixing an *error* ahead of time is relatively cheap
  – Releasing a product revision due to *defects* is expensive
Errors vs Defects

• Errors and defects
  – *Error*—a quality problem found *before* the software is released to end users
  – *Defect*—a quality problem found only *after* the software has been released to end-users

• We make this distinction because errors and defects have very different economic, business, psychological, and human impact

• However, the temporal distinction made between errors and defects in this book is *not* mainstream thinking
Improving SW Quality

• Engineering practices
  – Requirements Planning & Management
  – Architectural & Software Reviews
  – Sustainable Process Model

• Testing
  – Unit Level Testing
  – Functional Testing
  – Exploitation Testing/Challenge
  – Regression Testing with each “release”
  – Sustainable QA Process Model
(Formal Technical) Reviews

• Requirements Review
  – Code walk-throughs
  – Functional walk-throughs
  – Test-Design planning (TDD)

• Technical Focus
  – Should be ego-neutral
  – Should be detailed exam of code, function, design
  – Usually only a few developers and/or testers

• Short & Sweet
  – 2 hours prep, 2 hours duration
  – 3-5 people total
  – Keep detailed meeting minutes

• Develop METRICS based on the FTRs
  – Errors and type (major (e.g., bad calc); minor (typos))
## Review Options Matrix

<table>
<thead>
<tr>
<th>IPR *</th>
<th>WT</th>
<th>IN</th>
<th>RRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>trained leader</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>agenda established</td>
<td>maybe</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>reviewers prepare in advance</td>
<td>maybe</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>producer presents product</td>
<td>maybe</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>“reader” presents product</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>recorder takes notes</td>
<td>maybe</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>checklists used to find errors</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>errors categorized as found</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>issues list created</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>team must sign-off on result</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

*IPR—informal peer review  WT—Walkthrough  IN—Inspection  RRR—round robin review*
Defect Statistics

• According to the text:
  – a software process that does NOT include reviews,
    • yields 94 errors at the beginning of testing and
    • Releases 12 latent defects to the field
  – a software process that does include reviews,
    • yields 24 errors at the beginning of testing and
    • releases 3 latent defects to the field
  – A cost analysis indicates that the process with NO reviews costs approximately 3 times more than the process with reviews, taking the cost of correcting the latent defects into account
The effort required to correct a minor model error (immediately after the review) was found to require 4 person-hours.

The effort required for a major requirement error was found to be 18 person-hours.

Examining the review data collected, you find that minor errors occur about 6 times more frequently than major errors. Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 6 person-hours.

Requirements related errors uncovered during testing require an average of 45 person-hours to find and correct. Using the averages noted, we get:

- Effort saved per error = $E_{testing} - E_{reviews}$
- $45 - 6 = 30$ person-hours/error

Since 22 errors were found during the review of the requirements model, a saving of about 660 person-hours of testing effort would be achieved. And that’s just for requirements-related errors.
• Effort expended with and without reviews
  • Higher upfront manpower expenditure,
  • Which is why managers don’t want to spend time for testing
Software Quality Assurance

• Everybody wants “quality” software
• Nobody wants to pay for it...
• Cultural tradition of very sloppy software
  – Banks with poor/little software security
  – Microsoft products with rampant security holes and bugs
  – Fuzzy concept of causes, consequences by general public
  – No clear incentives for high quality SW products
    • Litigation?
    • Damages/Awards?
    • User vs Vendor responsibilities
• ISO 9001:2000 Standard
  – 20 requirements for effective QA
  – Cover all aspects from management responsibilities to statistical techniques
Elements of SQA

- Standards
- Reviews and Audits
- Testing
- Error/defect collection and analysis
- Change management
- Education
- Vendor management
- Security management
- Safety
- Risk management
SQA Goals (see Figure 16.1)

• **Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow.

• **Design quality.** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements.

• **Code quality.** Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability.

• **Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high quality result.
Statistical SQA

- Information about software errors and defects is collected and categorized.
- An attempt is made to trace each error and defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
- Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the vital few).
- Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.
Software Reliability

• A simple measure of reliability is mean-time-between-failure (MTBF), where

MTBF = MTTF + MTTR

• The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

• Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as

\[ \text{Availability} = \left[ \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \right] \times 100\% \]
Software Testing

• Correct *errors* before they become *defects*
• Look at performance, requirements of total system
• Testers should be *independent* of developers
  – Not part of supervisory chain
  – Get SW deliveries just like customer does
  – Schedule adherence is important
  – Testers can be coders, but should not be doing acceptance testing of their own code
  – Developers should do *unit testing* of their own code
• Developers tend to use “expected” tests
• Independent testers tend to use “unexpected” tests
V & V

- **Verification** refers to the set of tasks that ensure that software correctly implements a specific function.
- **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:
  - **Verification**: "Are we building the product right?"
  - **Validation**: "Are we building the right product?"
Who Tests the Software?

*developer*

Understands the system but, will test "gently" and, is driven by "delivery"

*independent tester*

Must learn about the system, but, will attempt to break it and, is driven by quality
Strategic Issues

• Specify product requirements in a quantifiable manner long before testing commences.
• State testing objectives explicitly.
• Understand the users of the software and develop a profile for each user category.
• Develop a testing plan that emphasizes “rapid cycle testing.”
• Build “robust” software that is designed to test itself
• Use effective technical reviews as a filter prior to testing
• Conduct technical reviews to assess the test strategy and test cases themselves.
• Develop a continuous improvement approach for the testing process.
Unit Testing

module to be tested

interface
local data structures
boundary conditions
independent paths
error handling paths
test cases

These slides are designed to accompany Software Engineering: A Practitioner’s Approach, 7/e (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.
Regression Testing

• *Regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

• Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

• Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

• Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.
Smoke Testing

• A common approach for creating “daily builds” for product software
• Smoke testing steps:
  – Software components that have been translated into code are integrated into a “build.”
    • A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  – A series of tests is designed to expose errors that will keep the build from properly performing its function.
    • The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
  – The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    • The integration approach may be top down or bottom up.
WebApp Testing - I

• The content model for the WebApp is reviewed to uncover errors.
• The interface model is reviewed to ensure that all use cases can be accommodated.
• The design model for the WebApp is reviewed to uncover navigation errors.
• The user interface is tested to uncover errors in presentation and/or navigation mechanics.
• Each functional component is unit tested.
WebApp Testing - II

- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.
Hard-To-Find Bugs: Symptoms & Causes

- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent
Debugging Strategies

• Brute Force
  – Least efficient
  – Memory, run-time trace, examine stack, etc.

• Backtracking
  – Start at location of observed error
  – Work back through functions/modules in code
  – Not practical for large code base

• Cause elimination
  – Hypothesis-based approach
  – Induction/deduction on causes, each tested
  – Must know code very well

• Automated
  – IDEs often have tools to help trace errors
  – Other debugging tools that monitor state of variables, files, etc.

• People
  – An outside point of view can make the biggest difference in the shortest time
Correcting the Error

• Is the cause of the bug reproduced in another part of the program? In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.

• What "next bug" might be introduced by the fix I'm about to make? Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.

• What could we have done to prevent this bug in the first place? This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.