BUILDING ROBUST SOFTWARE

Ross Collard
Collard & Company
Current State of the Practice

• **What is the Most Robust Software?**
  – MTBF: 40 years; MTTR: 4 hours
  – Average time to trigger a failure: 900 years
  – Defect delivery ratio of 99% to 1

• **How Much Does it Cost?**
  – Testing is 90% of budget
  – Instrumentation is 70% of code
TKQ (the key question)

• The key question to ask during software development and modification is: What could go wrong???

• This implies we need a conscious fault model or theory of errors.
Major Causes of Defects: (1) Specifications

- Failure to understand who the client is
- Failure to address the right issues
- Lack of sufficient client/user involvement
- Lack of readability
- Ambiguity
- Inconsistency
- Omissions
Major Causes: (1) Specs

- Imprecision; vagueness and lack of detail
- Unstated or buried assumptions
- Factual errors
- Unrealistic assumptions
- Technical feasibility (i.e., specifying features which cannot easily be built)
- Volatility; lack of change control on specs
Major Causes of Defects:
(2) Design

- Lack of fit to functional specifications
- Expansion of the scope; over-engineering
- Poor change control
- Not modular and top-down
- Structure is not well engineered; e.g., fan-in or fan-out is too high
- “Spaghetti”, entangled linkages of components
Major Causes: (2) Design

- Technically too aggressive or technically obsolete
- Insufficient detail on which to build a product
- Product “illities” not sufficiently addressed (e.g., maintainability usability)
- Internal and external interfaces are not adequately defined
- Resource use constraints are not defined
Major Causes: (2) Design

- Few methods to prevent, detect or recover from defects
- Testing & maintenance needs not met
- Software design is not visible, flexible, robust nor fault tolerant
- Lack of integration with the existing technical environment and business operations
High-Level Design Validation

• *Does the system design fulfill the requirements?*

• *Can we walk through the design, step by step, to show how each individual requirement be satisfied, and how?*

• *Is the design over-engineered (more than the requirements call for)?
High-Level Design Validation

• Will the design meet performance, reliability, back-up and recovery, maintainability, security and control, scalability and usability goals?

• Can this design be built? Is it feasible in the technical environment and with the resources (people, tools, etc.) available?

• Does the design adhere to commonly accepted standards for design quality? (i.e., modularity, coupling, etc.)
High-Level Design Validation

• Are the interfaces among the subsystems and among the components (control flows, data flows and shared data such as tables), specified correctly?

• Are all components labeled and identified so we know what they do?
High-Level Design Validation

- *Is the overall purpose of each individual component clearly described, and is it appropriate?*

- *Is the design visible, i.e., can we trace through the design to review how the system works in performing some overall user function?*

- *Is the system testable? (See later for a discussion of designing for testability.)*
Low-Level Design Validation

- Does this component-level design match the high-level design, and serve its functions?
- Does each component deliver its required functions? Is the description of the internal component algorithm or processing correct?
Low-Level Design Validation

• Can existing components be adapted and re-used, instead of creating new ones?

• Does each component comply with the internal inter-component interfaces, as described in the high-level design?
Low-Level Design Validation

- Are the data structures defined correctly and used correctly by the components?
- Is the design under change control, with all new or modified components and interfaces identified clearly?
Low-Level Design Validation

- Does the detailed design provide sufficient information for the programmers to build or modify the system? Is this information readable, understandable and unambiguous in the eyes of the programmers?
Design Rules of Thumb

- Cyclomatic complexity of components should not exceed 10.
- Depth of nested-if decision logic <= 7.
- The depth of inheritance chains <= 7.
- The length of module calling chains <= 7.
- The average fan-out of the modules in a design <= 7.
Contradictions in Managing Complexity

• There are several inherent contradictions in the rules of thumb which are used to manage the complexity of a software architecture.

• If the complexity of components is capped at 10, this means that there will a greater volume of components, so that it will be more difficult to meet the other guidelines.
Basics of Effective Design

• Readability
• Completeness
• Visibility
• Modularity
• Cohesiveness
• Decoupling
Basics of Effective Design

- Traceability
- Necessity
- Consistency
- Feasibility
- Efficiency
- Portability
- Re-use
Designing for Robustness

• Where can an error occur in the use of a system?

• At each particular point in operation, what kinds of errors could occur?

• Which possible errors are most critical versus merely a nuisance?

• How can these errors be detected?
Designing for Robustness

• What alternative actions could be taken in reaction to each possible error?
• How can errors be contained and not propagate: their side effects are kept to a minimum?
Designing for Robustness

• Controlled use of common routines and working storage areas?
• Separate or decoupled processes for activities which operate independently?
• Simplicity of design, the most straightforward performance of functions?
• Simplicity of documentation, access and understanding?
Designing for Robustness

- Simplicity of the interfaces?
- Continual monitoring and recording logs?
- Automatic error detection & diagnosis?
- Built-in error recovery?
- Visibility of actions?
Types of Error Handling

• *Error avoidance* tries to detect and neutralize embedded, "sleeping" errors before they become activated.

• *Error masking* uses redundant information to cross-check and deliver the correct service regardless of errors.
Types of Error Handling

• Back-up processes periodically preserve a known correct state of a system for possible later use in recovery.

• Roll-back mechanisms can be used to return to this correct state, and the recovery process then proceeds.
Error Recovery Mechanisms

- Automatically logging a copy of every transaction and before-and-after snapshots of updated stored data.

- Suspending an input transaction for later manual review, correction and re-submission.
Error Recovery Mechanisms

• Re-starting a system from a checkpoint.
• Checking the automatic switch-over to redundant back-up systems.
• Ensuring that the error messages and procedures are clear and usable by the people who have to work with them.
Software vs. Hardware Reliability

• Software reliability differs from hardware reliability, as software failures do not occur because of physical components age and wear out.
  – Hardware degrades over time because chips fry, pins are broken off connectors, cables fray and become electrically conductive, and so on.
  – Software cannot physically break.
Software vs. Hardware
Reliability

– Another important difference is that hardware tends to be highly stable after manufacturing, with little change. By contrast, software continues to grow and evolve.

– As software ages, the primary cause of failure becomes the modifications made to the software. (This concept is called software entropy.)
Fault Tolerant Systems

- Planned-in Redundancy
- Concurrent Parallel Processes
- Monitoring Processes
- Roll-Back Mechanisms
- Fault Tolerant Communications
Fault Tolerant Systems

- Fault Tolerant Data Bases (e.g., Mirroring)
- Self-Tuning Systems
- Self-Healing (Autonomic) Systems
Designing For Scalability

• Scalability is the capability of a system to continue to expand or contract as the needs change, and to provide acceptable service as the demand or resources change.
Designing For Maintainability

• (1) Quality of documentation.
• (2) User demands for enhancements.
• (3) Competing demands for the time of the software engineers assigned.
• (4) Meeting scheduled commitments.
• (5) Inadequate training.
• (6) Turnover in the organizations.
Designing For Maintainability

• According to a study by the Software Engineering Institute, programmers introduce inadvertent new errors in 20% to 50% of all systems changes.

• (To be fair to maintenance programmers, many of these defects are minor and are caught almost immediately in compilation and unit testing.)
Advantages in Working with Existing Software

– Static code analyzers (path analyzers) can be used to analyze the existing software.
– The system has already been heavily "field tested" in production use and with real users.
– The prior operational experience with the system and prior defects are known.
– Test data and facilities should be available.
Disadvantages easily can Outweigh Advantages

– The people who really understood the system have disappeared years ago.
– Seemingly simple and small changes to existing systems can have unanticipated and devastating side effects.
– Existing systems are often poorly understood, a source of mystery to the people charged with maintaining them and even to the day-to-day users.
Disadvantages

– Not only are the design and code inscrutable, the person who made the last few years' worth of patches before disappearing apparently was a devotee of voodoo.
– Because of the demand for fast turn-around times for fixes or enhancements, there may be little time to plan and develop tests.
– Existing documentation, such as the systems technical description and the prior history of changes, often is close to unusable.
Designing For Usability

- User-Centered Design
- Ease of Learning
- Ease of Use
- Error Processing
- Usability Error Checklists
Designing For Testability

• To be testable, a system has to be observable and controllable.

• Just as systems can be designed to exhibit desirable characteristics such as maintainability and usability, they can be designed for testability.
Software Re-Use

• *In its simplest form, software re-use is the process of assembling and adapting existing components.*

• *Extensions of the useful life:*
  – Software Re-Engineering
  – Data Re-engineering
  – Refactoring
Major Causes Of Defects: (3) Programming

- Unstructured, highly coupled code
- Lack of fit to specifications (difficult to avoid if the specs. are poor)
- High complexity
- Use of obscure language features
- Violations of programming standards
Major Causes: (3) Programming

- Hard-coded data values
- Insufficient change & version control, and quick, ill-considered patches
- Lack of fault tolerance and robustness
- Inflexibility (code was built without consideration for system maintenance)
- Computations and comparisons which use inconsistent data
Major Causes: (3) Programming

- Data initialization (failure to explicitly initialize or re-set)
- Shared data (accessed or updated by more than one module or program)
- Pointers and indexes that could exceed their expected ranges
- Possible field overflows (e.g., result field smaller than largest value)
Major Causes: (3) Programming

- Incorrect interface assumptions (e.g., a wrong set of parameters is passed)
- Entangled or “sloppy” loops
- Unintended fall-through conditions
- Nested conditions (e.g., ifs)
- Memory leaks
The Personal Software Process (PSP)

• “Developing software products involves much more than just stringing programming instructions together and getting them to run on a computer. It requires meeting customer requirements at an agreed cost and schedule....[PSP] shows you how to do this.” Watts Humphrey
The PSP

• **Personal responsibility (e.g., committing to and meeting deadlines)**

• **Personal commitment to quality**

• **Time management skills in the individual programmer, including:**
  – Analyzing time consumption. (“Where did the time go?”.)
  – Personal planning. (“What do I need to do to accomplish this goal?”.)
  – Estimation. (“How big is this task?.)  

• **Defect prevention skills**
Incident Analysis

• "Those who do not understand history are condemned to repeat it." George Santayana

• An organized method for learning from defects in order to prevent the occurrence of future similar defects.
Incident Analysis

• An analysis is performed and circulated for each defect, and this analysis addresses the following questions:

• How & when was the defect found?
• When was the defect made?
• Who made the defect? (this should only be asked if the environment is a trusting one.)
• Why and how was the defect caused?
Incident Analysis

• What triggered the detection of the defect, or initiated or enabled its occurrence?

• Why was the defect not discovered earlier?

• How could the defect have been prevented?

• How can we prevent similar defects in the future?

• Where else might this same defect or similar defects be embedded, and how can we find and remove them?
Walkthrough Guidelines

• Walkthroughs occur continually throughout a project
• No walkthrough is longer than 2 hours
• Do homework prior to walkthrough
• Egoless peer review: criticize work product, not author
• Moderator keeps focus & pace
Advantages of Walkthroughs

• Can find defects very early
• Communication device within the team
• Expectation of peer review increases task quality
• Helps track & ascertain project status
• Builds involvement in the project
• Can help build team spirit
Advantages of Walkthroughs

• *Promotes consistency*
• *Shares expertise*
• *Trains people early in the functionality and design*
• *Identifies opportunities for improved practices*
Next Steps

• Which ideas are valuable to you?

• How can you apply them?

• What support do you need?