**Software Qualities**

- **Maintainer**
  - Good Documentation
  - Readable Code
  - Good Design
- **User**
  - Functionality
  - Ease of use
  - Ease of learning
- **Customer**
  - Reliability
  - Correctness
  - Efficiency
  - Low Cost
  - Portability
  - Increased productivity

**Costs of Poor Quality**

- Increased time to find and fix problems
- Increased cost to distribute modifications
- Increased customer support
- Product liability
- Failure in the market place

**Software Reviews**

- Individuals read and comment on the software artifacts
- Very human intensive
- Overriding evidence shows that it
  - improves quality and productivity
  - reduces cost
- It is usually one of the first activities to be dropped when schedules get tight

**Software Reviews (cont.)**

- Applicable to all software artifacts
  - code inspections
  - requirements and design reviews
  - walk-throughs
- Recent research shows that
  - particular kind of review, size of team, etc. doesn’t matter
  - need at least one good, dedicated person doing the review
**Typical Review Team**

- Developer -- presents the material
- Moderator -- keeps the review on track
  - makes sure everyone abides by the process
- Secretary -- takes minutes, documents problems found
- Optionally
  - Apprentice -- learning about the project
  - Domain expert -- familiar with the domain and can verify assumptions

**Software Review Guidelines**

- Review the artifact
  - don’t attack the developer
- Stick to an agenda
- Limit debate
  - watch out for “religious” issues
  - watch out for stylistic issues that don’t affect maintainability
- Identify problems, not solutions
- Keep accurate notes
- Establish and follow evaluation guidelines
- Limit number of participants

**Technical Review Guidelines (cont.)**

- Prepare beforehand
  - both developers and reviewers
- Allocate resources for reviews
  - people and time
- Possible outcomes
  - accept product as is
  - reject until modified
  - reject product outright
  - accept product provisionally

**Sample evaluation Guidelines: Code Inspection**

- Has the design been correctly translated to code?
- Are language features used appropriately?
- Are coding standards followed?
  - be careful! make sure the standard makes a difference
- Are documentation standards followed?
- Are there misspellings or typos?
- Are the comments accurate and unambiguous?

**Sample evaluation Guidelines: Code Inspection (cont.)**

- Are data types and declarations appropriate?
- Are all constants correct?
- Are all variables initialized before being used?
- Are there overly complex conditions?
- Is there unreachable code?
- Are there obvious inefficiencies?

**QA Terminology**

- Correctness
- Reliability
- Testing
- Debugging
- Failure
- Fault
- Error
- Verification
- Validation
- V&V
**Terminology**

- **Correctness**: artifact is consistent with its specification
  - Specification could be wrong or incomplete
  - Rarely is software known to be correct
- **Reliability**: probability that the software is correct
  - Statistical measure based on past performance
    - e.g., mean time to failure

**More terminology**

- **Testing**: entails executing the software on selected test cases
  - Evaluate the results (oracle)
  - Evaluate the performance
  - Evaluate the ease of use
- **Common Wisdom**: Testing reveals bugs but does not guarantee the absence of bugs
  - How should you select test cases?
  - How do you know when to stop testing?

**More terminology**

- **Failure**: an erroneous result
  - Incorrect outputs/response for given inputs/stimuli
  - Fails to meet real-time constraints
- **Error**: incorrect concept
  - May cause failures if not corrected
- **Fault**: the cause of one or more failures
  - Discovered after release

**More terminology**

- **Debugging**: the process of finding the cause of a “bug” and a way to fix it
  - w/o introducing additional bugs!
- **Verification**: process of proving, using mathematical reasoning, that a program is “correct”
  - Proofs vs. model checking
  - Is expensive and is not always possible
  - Is not foolproof

**More terminology**

- **Validation**: process associated with showing that the software performs reasonably well
- **V & V**: verification & validation?
  - More typically equated with validation

**Many different kinds of testing**

- **Unit testing**: test individual components
  - Test stubs simulate called components
  - Test harness simulates “outer” context and maintains stubs
- **Integration testing**: combine components and test them
  - Follows build plan
- **System testing**: test whole system
More kinds of testing

- **Acceptance testing**: testing to determine if the product is acceptable
- **Regression testing**: retesting after the system has been modified
  - determine "old" test cases that must be re-executed
  - determine what new test cases are required

Testing is hard work

- Typically 50% of software development effort goes into testing
  - up to 85% for life-critical software
- How to identify "good" test cases?
  - high probability of finding a new error
  - hits "boundary" conditions
  - "weirdo" cases
    - often reveal bad assumptions and/or lack of rigor
- Objective is to find errors
  - test case is "successful" if it finds a new error

Testing Principles

- Tests should be traceable to requirements
- Tests should be planned long before testing begins
- Exhaustive testing is not possible
  - 80% of all errors typically occur in 20% of the modules
  - test cases should be chosen to maximize likelihood of finding an error

Testing

- **CAN**:
  - Uncover errors
  - Show specifications are met for specific test cases
  - Be an indication of overall reliability
  - Increase reliability (why??)
- **CANNOT**:
  - Prove that a program is error-free
  - Serve as verification (why??)
Testing Principles (cont.)

- Testing should be done by someone other than the developers
  - Developers do original testing
  - SQA does independent testing
    - usually black box testing
- Automated testing tools should be used
  - Reduce testing costs
  - Reduce likelihood of human error

Testability

- Simple software is easier to test
  - minimize coupling, maximize cohesion
- Output is sufficient to determine correct behavior
- Performs its own tests for internal errors
  - raises meaningful exceptions
- All code is reachable
- Independent modules can be tested in isolation
- Documentation is complete and accurate

Quality is an on-going concern

- You can’t build quality into a system after the fact
- Quality should be a consideration during every phase of development
- Plan for testing / validation in all phases
  - requirements -> functional text cases
  - design -> functional and structural test cases
  - code -> enhanced func & struct test cases
  - maintenance -> further enhanced func & struct test cases

Debugging

- Find the cause of a failure and fix it
  - an art, not a science
- Debugging is difficult because
  - symptom may appear long after the fault occurs
  - symptom may be difficult to reproduce
  - symptom may be intermittent
- Unit testing helps localize errors

SQA Summary

- U.S. software costs $200 billion/year
- Need to
  - improve software quality
  - reduce costs
    - V&V is over 50% of the cost
  - Improving V&V should reduce costs significantly while improving quality

Introduction to Formal Verification

How many tests do you have to do to show $d=n^2$, always?? (or that the loop even works…)

\[
\begin{align*}
i & = 0 \\
d & > 0 \\
di & \neq n \\
i & = i + 1 \\
d & = d + 2
\end{align*}
\]

Need a way to “prove” properties in general.

Proofs, Model Checking

Formal, Mathematically based
**Example: Proving A Loop Correct**

$$i = 0$$
$$d = 0$$
$$\text{do } i \neq n$$
$$i = i + 1$$
$$d = d + x$$
$$\text{od}$$

We'll do this using an invariant and a theorem about loops.

**Loop Theorem**

For an integer function $t$ bounded by 0, and

$$u \equiv \text{do } x \text{ while } t \geq 0$$

Then:

$$t \geq 0 \rightarrow \text{after } u$$

Point 2 means $P$ is invariant and the integer function decreases each time through the loop.

**Weakest Precondition**

This is a program statement, like an if, or do, or assignment

$$wp(P, S)$$

$\text{wp}()$ is a function that produces a predicate. The predicate $wp(P)$ produces describes a set of states. If $S$ starts in one of these states, $P$ will be true when it finishes.

**Example:**

$$wp(i := i + 1, i > 1, j = 5) = i > 0 \land i < 4$$

**Requirement #1**

So, let $P$ be:

$$\text{Direct substitution produces:}$$

$$0 \leq i \leq n \land d = i \Rightarrow n > i$$

**Requirement #2 / "t" part**

This must go down

$$i = 0$$
$$d = 0$$
$$\text{do } i \neq n$$
$$i = i + 1$$
$$d = d + x$$
$$\text{od}$$

The loop replaces $i$ with $i+1$, so that's what we do.

**Requirement #2 / invariant part**

Need to show that $P \land d$ before the loop.

$$i = 0$$
$$d = 0$$
$$\text{do } i \neq n$$
$$i = i + 1$$
$$d = d + x$$
$$\text{od}$$

We use the same trick again, and sub $i+1$ for $i$ and $d+x$ for $d$ and prove $P$ is still true.
**Showing Invariant Holds**

- \(0 \leq i \leq n \land d = ix \Rightarrow 0 \leq i + 1 \leq n \land d = (i+1)x\)

  \[
  \begin{align*}
  0 \leq i \leq n & \Rightarrow 0 \leq i + 1 \leq n \\
  \text{But } B \text{ is true, so } i \neq n & \\
  0 \leq i < n & \Rightarrow 0 \leq i + 1 \leq n
  \end{align*}
  \]

  \[
  \begin{align*}
  d = ix & \Rightarrow d + x = (i+1)x \\
  d = ix & \Rightarrow d + x = ix + x \\
  d = ix & \Rightarrow d = ix
  \end{align*}
  \]

**Payoff**

So we have \(P \Rightarrow \nu P(D_0, P \land \neg B)\) by the theorem.

We know that we can make \(P\) true before the loop, so we have a set of states such that \(P \land \neg B\) is true.

\[
\begin{align*}
0 \leq i \leq n \land d = ix \land i = n & \quad \text{\(P\)} \\
\text{\(\neg B\)} & \\
\sum &
\end{align*}
\]

\[
\begin{align*}
\begin{array}{ll}
i = 0 & \\
d = 0 & \\
i \neq n & \\
i = i + 1 & \\
d = d + x & \\
\text{end}
\end{array}
\end{align*}
\]

And, we are done.