Unit Testing, continued
8.3 Unit Testing
Proving Code Correct

- Formal proof techniques
- Symbolic execution
- Automated theorem-proving
Proving correctness

- assertions (propositions) form pre-conditions and post-conditions
- formally (mathematically) prove that the **pre-condition** being true directly leads to the **post-condition**
- Hoare logic example
  - [https://en.wikipedia.org/wiki/Loop_invariant](https://en.wikipedia.org/wiki/Loop_invariant)
- Issues?
• proving the **algorithm** correct, not the code
• how to construct the assertions (what are the pre and post conditions?)
• hard to keep track of all of them
• how to formalize requirements
• completely unrealistic for good sized systems
8.3 Unit Testing
Testing versus Proving

- Proving: hypothetical environment
- Testing: actual operating environment
8.3 Unit Testing

Steps in Choosing Test Cases

- Determining test objectives
- Selecting test cases
- Defining a test
Black box testing: have access to inputs, can see what outputs are

- how can I break this code?
- partition the values of inputs along different dimensions
  - “typical” values
  - “atypical” values
  - boundary values
  - “can never happen” values
Black box testing a sorting algorithm

What test cases should you use?

- input is randomly ordered items
- input is already sorted
- input is reverse sorted
- input are all unique
- input are all the same
- input have some duplicates, not all
- 0 items to be sorted
- 1 item
- “typical” number of items
- lots, lots, lots, lots of items
- heterogeneously typed items
Choosing Test Cases: White Box

White Box testing: Can see the code, so more opportunities to try to break it

- find control structures (conditionals, loops, exceptions)
- partition values for each part of the conditional
White box testing Quicksort

http://www.algolist.net/Algorithms/Sorting/Quicksort
8.3 Unit Testing
Test Thoroughness

- Statement testing
- Branch testing
- Path testing
- Definition–use testing
- All–uses testing
- All–predicate–uses/some–computational–uses testing
- All–computational–uses/some–predicate–uses testing
Branch Flow Analysis at HP

- do lexical analysis of source code
- identify branch points, insert code to tally when that branch is taken
- create a “branch analyzed” source
- compile and run that during testing
- report on number of branches exercised during testing
Unit Testing
Branch Flow Analysis
Recall difference between a path and a branch

see chalkboard example
8.3 Unit Testing
Comparing Techniques

- Fault discovery Percentages by Fault Origin

<table>
<thead>
<tr>
<th>Discovery Techniques</th>
<th>Requirements</th>
<th>Design</th>
<th>Coding</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototyping</td>
<td>40</td>
<td>35</td>
<td>35</td>
<td>15</td>
</tr>
<tr>
<td>Requirements review</td>
<td>40</td>
<td>15</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Design Review</td>
<td>15</td>
<td>55</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Code inspection</td>
<td>20</td>
<td>40</td>
<td>65</td>
<td>25</td>
</tr>
<tr>
<td>Unit testing</td>
<td>1</td>
<td>5</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>
### Effectiveness of fault-discovery techniques

<table>
<thead>
<tr>
<th></th>
<th>Requirements Faults</th>
<th>Design Faults</th>
<th>Code Faults</th>
<th>Documentation Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviews</td>
<td>Fair</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Good</td>
</tr>
<tr>
<td>Prototypes</td>
<td>Good</td>
<td>Fair</td>
<td>Fair</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Testing</td>
<td>Poor</td>
<td>Poor</td>
<td>Good</td>
<td>Fair</td>
</tr>
<tr>
<td>Correctness Proofs</td>
<td>Poor</td>
<td>Poor</td>
<td>Fair</td>
<td>Fair</td>
</tr>
</tbody>
</table>
8.3 Unit Testing

Sidebar 8.4 Fault Discovery Efficiency at Contel IPC

- 17.3% during inspections of the system design
- 19.1% during component design inspection
- 15.1% during code inspection
- 29.4% during integration testing
- 16.6% during system and regression testing
- 0.1% after the system was placed in the field
8.4 Integration Testing

- Bottom-up
- Top-down
- Big-bang
- Sandwich testing
- Modified top-down
- Modified sandwich
8.4 Integration Testing

Terminology

- **Component Driver**: a routine that calls a particular component and passes a test case to it
- **Stub**: a special-purpose program to simulate the activity of the missing component
8.4 Integration Testing

View of a System

- System viewed as a hierarchy of components
8.4 Integration Testing
Bottom-Up Integration Example

- The sequence of tests and their dependencies
8.4 Integration Testing
Top-Down Integration Example

- Only A is tested by itself
8.4 Integration Testing
Modified Top-Down Integration Example

- Each level’s components individually tested before the merger takes place.
8.4 Integration Testing
Bing–Bang Integration Example

• Requires both stubs and drivers to test the independent components
8.4 Integration Testing
Sandwich Integration Example

- Viewed system as three layers
8.4 Integration Testing

Modified Sandwich Integration Example

- Allows upper-level components to be tested before merging them with others
## 8.4 Integration Testing

### Comparison of Integration Strategies

<table>
<thead>
<tr>
<th></th>
<th>Bottom-up</th>
<th>Top-down</th>
<th>Modified top-down</th>
<th>Bing-bang</th>
<th>Sandwich</th>
<th>Modified sandwich</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integration</strong></td>
<td>Early</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Early</td>
<td>Early</td>
</tr>
<tr>
<td><strong>Time to basic working program</strong></td>
<td>Late</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Early</td>
<td>Early</td>
</tr>
<tr>
<td>Component drivers needed</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Stubs needed</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Work parallelism at beginning</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Ability to test particular paths</td>
<td>Easy</td>
<td>Hard</td>
<td>Easy</td>
<td>Easy</td>
<td>Medium</td>
<td>Easy</td>
</tr>
<tr>
<td>Ability to plan and control sequence</td>
<td>Easy</td>
<td>Hard</td>
<td>Easy</td>
<td>Hard</td>
<td>hard</td>
<td>hard</td>
</tr>
</tbody>
</table>
8.4 Integration Testing

Sidebar 8.5 Builds at Microsoft

- The feature teams synchronize their work by building the product and finding and fixing faults on a daily basis.
Questions at the Beginning of Testing OO System

- Is there a path that generates a unique result?
- Is there a way to select a unique result?
- Are there useful cases that are not handled?
8.5 Testing Object-Oriented Systems

Easier and Harder Parts of Testing OO Systems

- OO unit testing is less difficult, but integration testing is more extensive.
8.5 Testing Object-Oriented Systems
Differences Between OO and Traditional Testing

• The farther the gray line is out, the more the difference
8.6 Test Planning

- Establish test objectives
- Design test cases
- Write test cases
- Test test cases
- Execute tests
- Evaluate test results
8.6 Test Planning

Purpose of the Plan

• Test plan explains
  – who does the testing
  – why the tests are performed
  – how tests are conducted
  – when the tests are scheduled
8.6 Test Planning

Contents of the Plan

- What the test objectives are
- How the test will be run
- What criteria will be used to determine when the testing is complete
8.7 Automated Testing Tools

• Code analysis
  – Static analysis
    • code analyzer
    • structure checker
    • data analyzer
    • sequence checker

• Output from static analysis
8.7 Automated Testing Tools (continued)

- Dynamic analysis
  - program monitors: watch and report program’s behavior
- Test execution
  - Capture and replay
  - Stubs and drivers
  - Automated testing environments
- Test case generators
8.8 When to Stop Testing
More faulty?

- Probability of finding faults during the development
8.8 When to Stop Testing
Stopping Approaches

• Coverage criteria

• Fault seeding
  \[
  \frac{\text{detected seeded faults}}{\text{total seeded faults}} = \frac{\text{detected nonseeded faults}}{\text{total nonseeded faults}}
  \]

• Confidence in the software, \( C \)
  \[
  C = 1, \quad \text{if } n > N
  \]
  \[
  = \frac{S}{S - N + 1}, \quad \text{if } n \leq N
  \]
8.8 When to Stop Testing
Identifying Fault-Prone Code

- Track the number of faults found in each component during the development
- Collect measurement (e.g., size, number of decisions) about each component
- Classification trees: a statistical technique that sorts through large arrays of measurement information and creates a decision tree to show best predictors
  - A tree helps in deciding which components are likely to have a large number of errors
8.8 When to Stop Testing
An Example of a Classification Tree
Information Systems Example
Piccadilly System

- Using data-flow testing strategy rather than structural
  - Definition–use testing
8.10 Real-Time Example
The Ariane-5 System

• The Ariane-5’s flight control system was tested in four ways
  – equipment testing
  – on-board computer software testing
  – staged integration
  – system validation tests

• The Ariane-5 developers relied on insufficient reviews and test coverage
8.11 What this Chapter Means for You

- It is important to understand the difference between faults and failures
- The goal of testing is to find faults, not to prove correctness