Testing

Lecture 31
Definition of Testing

☐ What is “testing”?
  ■ A process whereby we *increase our confidence* in an implementation by *observing* its *behavior*

☐ Fundamental point:
  ■ Testing can detect the *presence of mistakes*, never their *absence*!

☐ Fail a test case ===> 

☐ Pass *all* test cases ===>
Importance of Testing

- Despite limitations, testing is the most practical approach for large systems
- Donald Knuth quotation:
  “Warning: I’ve only proven this algorithm is correct... I haven’t tested it!”
Theory

- 3 levels of abstraction in functionality
- Want: the idea
- Have: an implementation
- “Testing” requires comparing it against something, but what?
Theory (II)

- Ideal: test against our “idea”
  - But the idea is usually too fuzzy
- So make it concrete by writing a specification
  - Defines desired mapping from input to output
Example: Sorting a List

- Idea: Function takes a list and puts it in order
- Too fuzzy!
- Questions:
  - Does it modify the list or return a new one?
  - Does it require the list to be non empty?
  - Does it sort in increasing or decreasing order?
  - What kind of items can be in the list?
Example: Sorting a List

- Specification: Describe how inputs map to outputs
- Recall software I/II contracts
  requires
  \[ |\text{list}| \leq 65535 \]
  ensures
  \[ \forall i: 0 \leq i < \text{list}.\text{length}: \text{list}[i] \leq \text{list}[i + 1] \]
Example: Sorting a List

- Specification: Describe how inputs map to outputs
- Recall software I/II contracts
  requires
  \[ |\text{list}| \leq 65535 \]
  ensures
  \[
  \forall i: 0 \leq i < \text{list}.\text{length} - 1: \text{list}[i] \leq \text{list}[i + 1]
  \]
Example: Sorting a List

- Specification: Describe how inputs map to outputs
- Recall software I/II contracts
  requires
  \[ |list| \leq 65535 \]
  ensures
  \[ \forall i: 0 \leq i < list.length - 1: list[i] \leq list[i + 1] \]
  \[ list = permutation(#list) \]
Initial and Final Values

- Often the final value of a parameter depends on its initial value
  - SW I/II: "Updates" parameter mode
  - Example: mutator method sort!

- Consequence: Specification includes old value (e.g., #list) in ensures clause

- Sometimes the final value is independent of its initial value
  - "Replaces" and "Clears" parameter modes
  - Example: mutator methods fill!, clear!
Relational Specifications

- A function maps each element in its domain to a *single* element in its range.
- A relation maps each element in its domain to *at least* one elt in its range.
- For specifications:
  - Function = deterministic behavior
  - Relation = nondeterministic behavior
- Examples
  - `find_factor n` returns *some* prime factor
  - `shuffle` scrambles elements of array
Frame of Mind for Validation

- Tests should be written to *break* a program
  - Not to show it works!
- When a test reveals an error, that’s *success*!
  - Failed test case is a *positive* thing
- Good approach: have *someone else* test your code
Importance of Indep't Testing

□ See IEEE Computer, Oct ‘99
  ■ study at NASA Langley
  ■ had two groups working in parallel
□ The group with independent testers found:
  ■ more faults overall (critical and non-critical)
  ■ found these faults earlier in the process
  ■ fixed these faults with less effort
Figure 1 from Arthur article

The bar chart shows the number of critical faults in the IV&V group and the Non-IV&V group across different phases:

- Requirements: 16 in IV&V, 20 in Non-IV&V
- HLD: 0 in IV&V, 2 in Non-IV&V
- LLD: 31 in IV&V, 8 in Non-IV&V
- Code/UT: 24 in IV&V, 34 in Non-IV&V
- I&T: 6 in IV&V, 14 in Non-IV&V
- Total: 97 in IV&V, 58 in Non-IV&V
Figure 2 from Arthur article

![Bar chart showing effort in minutes for IV&V group and Non-IV&V group across Noncritical, Critical, and Combined fault categories.](chart.png)

- **Effort in minutes**
  - **Noncritical**
    - IV&V group: 4 minutes
    - Non-IV&V group: 7 minutes
  - **Critical**
    - IV&V group: 21 minutes
    - Non-IV&V group: 11 minutes
  - **Combined**
    - IV&V group: 7 minutes
    - Non-IV&V group: 20 minutes
Writing Good Tests (Inputs)

- Too many possible inputs to test them all
  - Space is defined by requires clause
  - Choose inputs wisely
- Test boundary conditions
  - eg 0, empty array, empty string
- Test different categories of input
  - eg positive, negative, and zero
- Test different categories of behavior
  - eg each menu option, each error message
- Test “unexpected” input
  - eg nil, last name includes a space
- Test representative “normal” input
  - eg random, reasonable values
How To Create Expected Output

- By hand
  - Error-prone and tedious

- With another program
  - Also error-prone
  - Often just redoing the implementation, and making the same mistakes!

- Work backwards
  - Inverse may be easier to calculate
  - Eg start with a sorted list then permute it
Alternative: Validating Output

- **Steps:**
  - Keep a copy of the input
  - Run the program
  - Validate the actual output against input

- **Example: sorting**
  - write two helper functions:
    - 1.
    - 2.
      - copy the input
      - run program and check

- **Helper functions may be easier to get right than the unit under test**
Dangers with Testing

- “Expected output” is wrong
- Testing program is wrong
  - Extra code means more chances to make mistakes
  - E.g. `permute(a,b)` always returns true
- With these errors, there are 2 dangers:
  1. Spurious test failures (passes when shouldn't)
  2. False positives (fails when it shouldn't)
- Which is worse?
Another Danger with Testing

- A third, more subtle, potential error: The specification is wrong
- How can this be?

- When testing drives implementation, this kind of error will not be exposed
- To increase the chances of finding these problems, have *someone else* test your code!
Levels of Testing

- Different kinds of testing, aimed at identifying different kinds of errors
  1. Unit tests
  2. Integration tests
  3. System tests
Unit Tests

- Individual components tested in isolation
  - UUT: Unit Under Test

- Often uses a *test fixture*
  - Configuration, values, objects which are set up before running all the tests

- Flavors of unit testing:
  - *Black box*: testing based *only* on specification (tester does not look at code)
  - *White box*: testing based on code structure (tester looks at code to make sure every branch of a switch statement is followed)
Integration Tests

- Modules tested in combination in order to check the *interfaces*
- Best done incrementally
# Bottom-up vs Top-down Testing

<table>
<thead>
<tr>
<th>Bottom-up</th>
<th>Top-down</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Start with most basic modules</td>
<td>- Start at top (main)</td>
</tr>
<tr>
<td>- Easy to exercise all the features</td>
<td>- Test interfaces early</td>
</tr>
<tr>
<td>- Write a “driver” for higher-level modules</td>
<td>- Write “stubs” for lower level modules</td>
</tr>
</tbody>
</table>

Often these two occur simultaneously, in tandem
System Tests

- Verify that system as a whole meets the requirements and specifications
- Three flavors:
  - alpha: By developers, before release
  - beta: By “friendly customers”, before general release
  - acceptance: By end customer, to decide whether or not to hire you next time!
TDD: Test-Driven Development

- In dynamic languages, testing is even more important
  - Load-time errors << compile-time errors
- Extreme position: "If it isn't tested, assume it doesn't work"
- TDD: Write tests first (before code)
  - Recall "client-view first" in Software I/II
  - Development cycle: "red, green, refactor"
    - Write tests, watch them fail
    - Write (only) enough code for tests to pass (may need to refactor)
    - Repeat
Summary

- Nature of testing
  - Specification, implementation, test cases
  - Initial values matter too

- Importance of the right frame of mind
  - Write tests to break code
  - TDD: write tests to guide development

- Levels of Testing
  - Unit tests
  - Integration tests
  - System tests