Regression Testing

Regression testing

- Changes can break code, reintroduce old bugs
  - Things that used to work may stop working (e.g., because of another “fix”) – software regresses
- Usually a set of cases that have failed (& then succeeded) in the past
- Finding small regressions is an ongoing research area – analyze dependencies

“. . . as a consequence of the introduction of new bugs, program maintenance requires far more system testing. . . . Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such regression testing must indeed approximate this theoretical idea, and it is very costly." - Brooks, *The Mythical Man-Month*
Regression testing (Automation and Tool Support)

- Regression tests must be automated
  - Too many tests to be run by hand
  - Tests must be run and evaluated quickly
    - Often overnight, or more frequently for web applications
    - Testers do not have time to view the results by inspection

- Types of tools:
  - Capture / Replay – Capture values entered into a GUI and replay those values on new versions
  - Version control – Keeps track of collections of tests, expected results, where the tests came from, the criterion used, and their past effectiveness
  - Scripting software – Manages the process of obtaining test inputs, executing the software, obtaining the outputs, comparing the results, and generating test reports

- Tools are plentiful and inexpensive (often free)
Managing Tests in a Regression Suite

- Test suites accumulate new tests over time
- Test suites are usually run in a fixed, short, period of time
  - Often overnight, sometimes more frequently, sometimes less
- At some point, the number of tests can become unmanageable
  - We cannot finish running the tests in the time allotted
- We can always add more computer hardware
- But is it worth it?
- How many of these tests really need to be run?
Policies for Updating Test Suites

- Which tests to keep can be based on several policies
  - Add a new test for every problem report
  - Ensure that a coverage criterion is always satisfied
- Sometimes harder to choose tests to remove
  - Remove tests that do not contribute to satisfying coverage
  - Remove tests that have never found a fault (risky!)
  - Remove tests that have found the same fault as other tests (also risky!)
- Reordering strategies
  - If a suite of N tests satisfies a coverage criterion, the tests can often be reordered so that the first N-x tests satisfies the criterion – so the remaining tests can be removed
When a Regression Test Fails

- Regression tests are evaluated based on whether the result on the new program P is equivalent to the result on the previous version P-1
  - If they differ, the test is considered to have failed

- Regression test failures represent three possibilities:
  - The software has a fault – Must fix the fix
  - The test values are no longer valid on the new version – Must delete or modify the test
  - The expected output is no longer valid – Must update the test

- Sometimes hard to decide which!!
Choosing Which Regression Tests to Run

- Using Change Impact Analysis (How does a change impact the rest of the software)
  - When a small change is made in the software, what portions of the software can be impacted by that change?

- More directly, which tests need to be re-run?
  - Conservative approach: Run all tests
  - Cheap approach: Run only tests whose test requirements relate to the statements that were changed
  - Realistic approach: Consider how the changes propagate through the software

- Clearly, tests that never reach the modified statements do not need to be run

- Lots of clever algorithms to perform Change Impact Analysis have been invented
  - Few if any available in commercial tools
Rationales for Selecting Tests to Re-Run

- **Inclusive**: A selection technique is inclusive if it includes tests that are “modification revealing”
  - Unsafe techniques have less than 100% inclusiveness

- **Precise**: A selection technique is precise if it omits regression tests that are not modification revealing

- **Efficient**: A selection technique is efficient if deciding what tests to omit is cheaper than running the omitted tests
  - This can depend on how much automation is available

- **General**: A selection technique is general if it applies to most practical situations

- Test Suite Minimization.
- Test Case Selection.
- Test Case Prioritization.
Summary of Regression Testing

- We spend far more time on regression testing than on testing new software.

- If tests are based on covering criteria, all problems are much simpler
  - We know why each test was created
  - We can make rationale decisions about whether to run each test
  - We know when to delete the test
  - We know when to modify the test

- Automating regression testing will save much more than it will cost
Integration and Testing

• Throw all the classes together, compile the whole program, and system test it
  • This is risky

• The usual method is to start small, with a few classes that have been tested thoroughly
  • Add a small number of new classes
  • Test the connections between the new classes and pre-integrated classes

• Integration testing: testing interfaces between classes
  • Should have already been tested in isolation (unit testing)
Methods, Classes, Packages

- Integration can be done at the method level, the class level, package level, or at higher levels of abstraction.
- We use the word component in a generic sense.
- A component is a piece of a program that can be tested independently.
- Integration testing is done in several ways:
  - Evaluating two specific components
  - Testing integration aspects of the full system
  - Putting the system together “piece by piece”
Software Scaffolding

- Scaffolding is extra software components that are created to support integration and testing.

- A stub emulates the results of a call to a method that has not been implemented or integrated yet.

- A driver emulates a method that makes calls to a component that is being tested.

Diagram:

- **Driver** makes calls to methods in CUT.
- **Component Under Test** (example: ADT) is the CUT called.
- **Stubs** emulate the methods the CUT calls.
Stubs

- The first responsibility of a stub is to allow the CUT to be compiled and linked without error
  - The signature must match
- What if the called method needs to return values?
- These values will not be the same the full method would return
- It may be important for testing that they satisfy certain limited constraints

Approaches:
1. Return constant values from the stub
2. Return random values
3. Return values from a table lookup
4. Return values entered by the tester during execution
5. Processing formal specifications of the stubbed method

More costly / more effective
Drivers

- Many good programmers add drivers to every class as a matter of habit
  - Instantiate objects and carry out simple testing
- Test drivers can easily be created automatically
- Values can be hard-coded or read from files
Class Integration and Test Order (CITO)

- Old programs tended to be very hierarchical
- Which order to integrate was pretty easy:
  - Test the “leaves” of the call tree
  - Integrate up to the root
  - Goal is to minimize the number of stubs needed
- OO programs make this more complicated
  - Lots of kinds of dependencies (call, inheritance, use, aggregation)
  - Circular dependencies: A inherits from B, B uses C, C aggregates A
- CITO: Which order should we integrate and test?
  - Must “break cycles”
  - Common goal: least stubbing
- Designs often have few cycles, but cycles creep in during implementation
Identifying Correct Outputs

- Oracle Problem: Does a program execute correctly on a specific input?
- With simple software methods, we have a very clear idea whether outputs are correct or not.
- But for most programs it’s not so easy.
- Four general methods for checking outputs:
  1. Direct verification
  2. Redundant computation
  3. Consistency checks
  4. Data redundancy
Direct Verification

- Using a program to check the answer
- Requires more programming
- Verifying outputs is deceptively hard
  - One difficulty is getting the post-conditions right
- Not always possible – we do not always know the correct answer
  - Any problem where approximations are made based on models and guesses, we don’t know the correct answers!
Direct Verification Example

- Consider a simple sort method

- Post-condition: Array is in sorted order

<table>
<thead>
<tr>
<th>Input</th>
<th>8</th>
<th>92</th>
<th>7</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Output</td>
<td>92</td>
<td>14</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

- Post-condition: Array sorted from lowest to highest and contains all the elements from the input array

<table>
<thead>
<tr>
<th>Input</th>
<th>87</th>
<th>14</th>
<th>14</th>
<th>87</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>87</td>
</tr>
</tbody>
</table>

- Post-condition: Array sorted from lowest to highest and is a permutation of the input array
Direct Verification Example

Input: Array A
- Make copy B of A
- Sort A
  // Verify A is a permutation of B
  Check A and B are of the same size
  For each object in A
    Check if object appears in A and B the same number of times
  // Verify A is ordered
  for each index i except the last in A
    Check if A [i] <= A [i+1]

- This is almost as complicated as the sort method under test.
- We can easily make mistakes.
Redundant Computation

- Write two programs – check that they produce the same answer
- Very expensive!
- Problem of coincident failures
  - Both programs fail on the same input
  - “independent failure assumption” is not valid in general.
- This works best if completely different algorithms can be used
  - Not clear exactly what “completely different” means
- Consider regression testing
  - Current software checked against prior version
  - Special form of redundant computation
  - Clearly, independence assumption does not hold
    - But still extremely powerful
Consistency Checks

- Check part of the answer to see if it makes sense
- Check if a probability is negative or larger than one
- Check assertions or invariants
  - No duplicates
  - Cost is greater than zero
  - Internal consistency constraints in databases or objects
- These are only partial solutions
Data Redundancy

- Compare results of different inputs
- Check for “identities”
  - Testing sin (x) : \( \sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b) \)
    - Choose a at random
    - Set \( b=x-a \)
    - Note failure independence of \( \sin(x), \sin(a) \)
    - Repeat process as often as desired; choose different values for a
    - Possible to have arbitrarily high confidence in correctness assessment
      - Inserting an element into a structure and removing it
- These are only partial solutions