Software Engineering Testing

Practical Considerations

The Toolbox

- Earlier lectures fill up a "toolbox" with useful criteria for testing software
- To move to Beizer's level 3 (reducing risk) or level 4 (mental discipline of quality), testing must be integrated into the development process
- Most importantly :
  - In any activity, knowing the tools is only the first step
  - The key is utilizing the tools in effective ways
- Topics :
  - Integrating software components and testing
  - Integrating testing with development
  - Test plans
  - Checking the output

Outline

1. **Regression Testing**
2. Integration and Testing
3. Test Process
4. Test Plans
5. Identifying Correct Outputs

Regression Testing

<u>Definition</u>

The process of re-testing software that has been modified

- Most software today has <u>very little new</u> development
  - – Correcting, perfecting, adapting, or preventing problems with existing software
  - – Composing new programs from existing components
  - – Applying existing software to new situations
- Because of the deep interconnections among software components, changes in one method can cause problems in methods that seem to be unrelated
- Not surprisingly, most of our testing effort is regression testing
- Large regression test suites accumulate as programs (and software components) age

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 are really worth running?

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 !!

Evolving Tests Over Time

- Changes to external interfaces can sometimes cause all tests to fail
  - Modern capture / replay tools will not be fooled by trivial changes like color, format, and placement
  - Automated scripts can be changed automatically via global changes in an editor or by another script
- Adding one test does not cost much – but over time the cost of these small additions start to pile up

Choosing Which Regression Tests to Run

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 CIA 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

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

Outline

1. Regression Testing
2. **Integration and Testing**
3. Test Process
4. Test Plans
5. Identifying Correct Outputs

Big Bang Integration

Throw all the classes together, compile the whole program, and system test it

- The polite word for this is risky
  - Less polite words also exist …
- 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

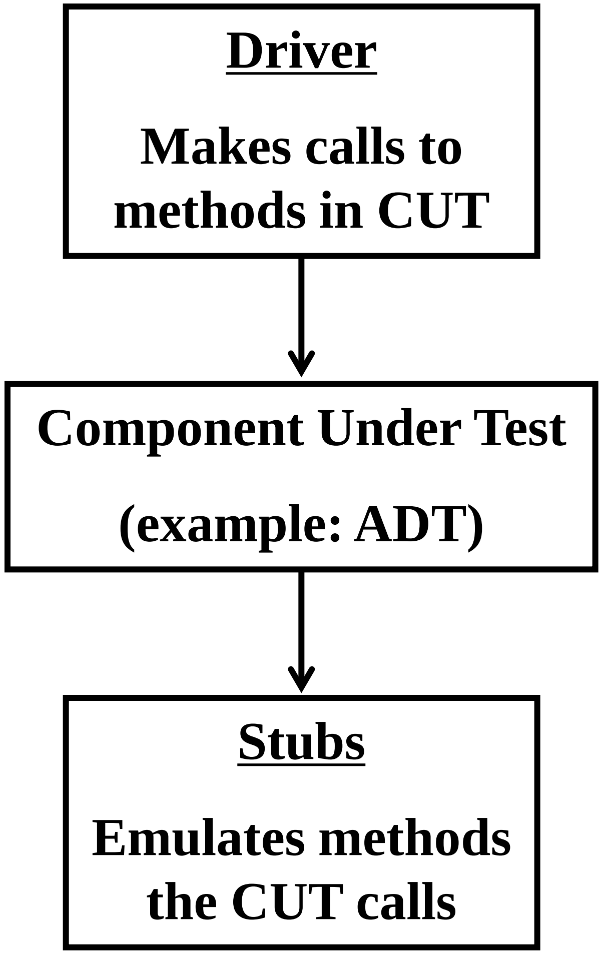
- <u>Integration</u> can be done at the <u>method</u> level, the <u>class</u> level, <u>package</u> level, or at <u>higher</u> levels of abstraction
- Rather than trying to use <u>all the words</u> in every slide …
- Or <u>not</u> using any specific word …
- We use the word <u>component</u> in a generic sense

- *A <u>component</u> is a piece of a program that can be tested independently*

- <u>Integration testing</u> is done in several ways
  - Evaluating <u>two specific components</u>
  - Testing integration aspects of the <u>full system</u>
  - Putting the system together "<u>piece by piece</u>"

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



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:
  - Return constant values from the stub
  - Return random values
  - Return values from a table lookup
  - Return values entered by the tester during execution
  - Processing formal specifications of the stubbed method

More costly / more effective $\downarrow$

Drivers

- Many good programmers add drivers to <u>every class</u> as a matter of habit
  - Instantiate objects and carry out <u>simple testing</u>
  - <u>Criteria</u> from previous lectures can be implemented in drivers
- Test drivers can easily be created <u>automatically</u>
- Values can be <u>hard-coded</u> or <u>read</u> 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

Outline

1. Regression Testing
2. Integration and Testing
3. **Test Process**
4. Test Plans
5. Identifying Correct Outputs

Test Process

We know <u>what</u> to do … but now … <u>how</u> can we do it?

Testing by Programmers

- The <u>important issue</u> is about quality

- Quality cannot be “<u>tested in</u>”!

Changes in Software Production

- Teamwork has changed
  - *1970*: we built log cabins
  - *1980*: we built small buildings
  - *1990*: we built skyscrapers
  - *200X*: we are building integrated communities of buildings
- We do more maintenance than construction
  - Our knowledge base is mostly about testing new software
- We are reusing code in many ways
- Quality vs efficiency is a constant source of stress
- Level 4 thinking requires the recognition that quality is usually more crucial than efficiency
  - Requires management buy-in !
  - Requires that programmers respect testers

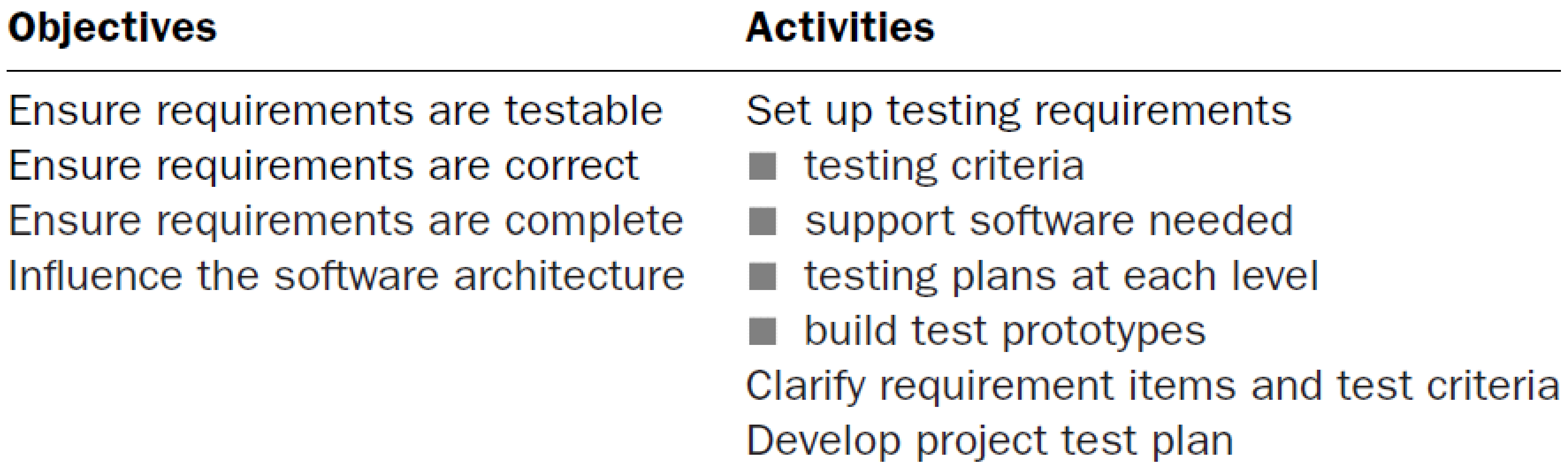
Test Activities

Software requirements $\longrightarrow$	**Define test objectives (criteria)** **Project test plan**
System design $\longrightarrow$	**Design system tests** **Design acceptance tests** **Design usability test, if appropriate**
Intermediate design $\longrightarrow$	**Specify system tests** **Integration and unit test plans** **Acquire test support tools** **Determine class integration order**
Detailed design $\longrightarrow$	**Create tests or test specifications**

Test Activities (2)

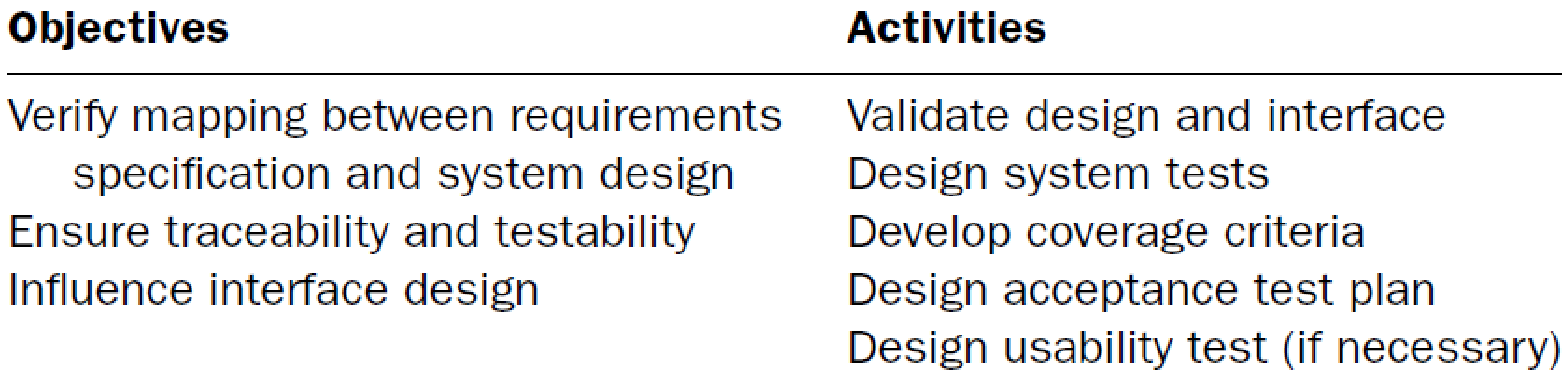
Implementation	Create tests Run tests when units are ready
Integration	Run integration tests
System deployment	Apply system test Apply acceptance tests Apply usability tests
Operation and maintenance	Capture user problems Perform regression testing

Testing objectives and activities during requirements analysis and specification



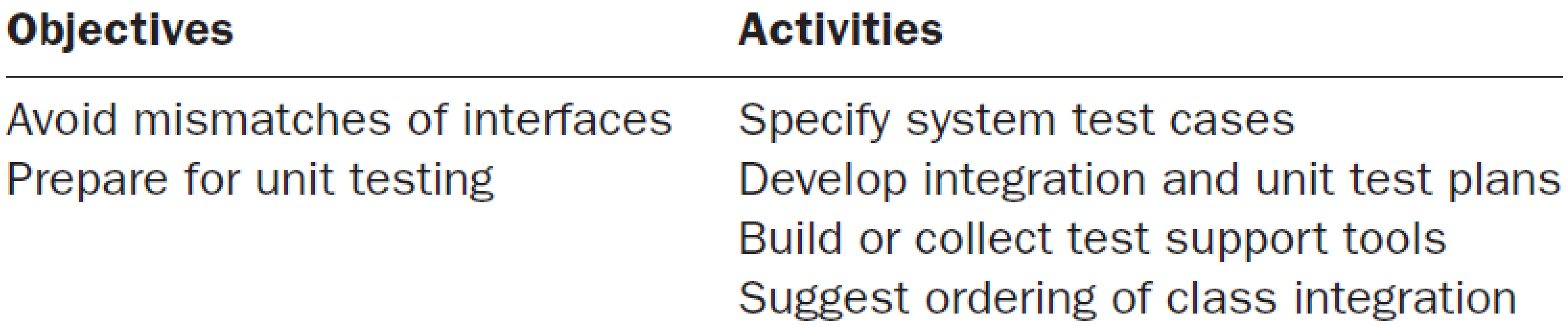
Objectives	Activities
Ensure requirements are testable	Set up testing requirements
Ensure requirements are correct	■ testing criteria
Ensure requirements are complete	■ support software needed
Influence the software architecture	■ testing plans at each level
	■ build test prototypes
	Clarify requirement items and test criteria
	Develop project test plan

Testing objectives and activities during system and software design



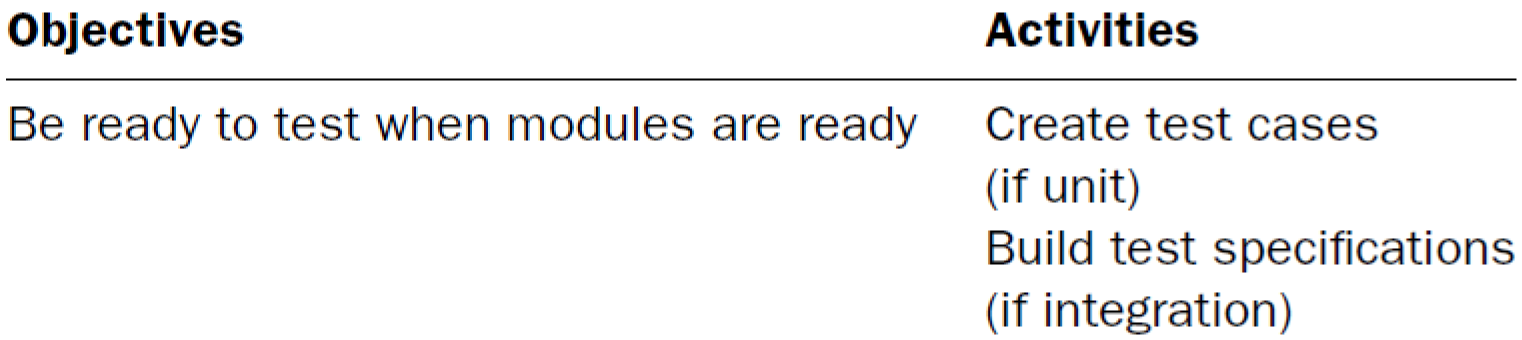
Objectives	Activities
Verify mapping between requirements specification and system design	Validate design and interface
	Design system tests
Ensure traceability and testability	Develop coverage criteria
Influence interface design	Design acceptance test plan
	Design usability test (if necessary)

Testing objectives and activities during intermediate design



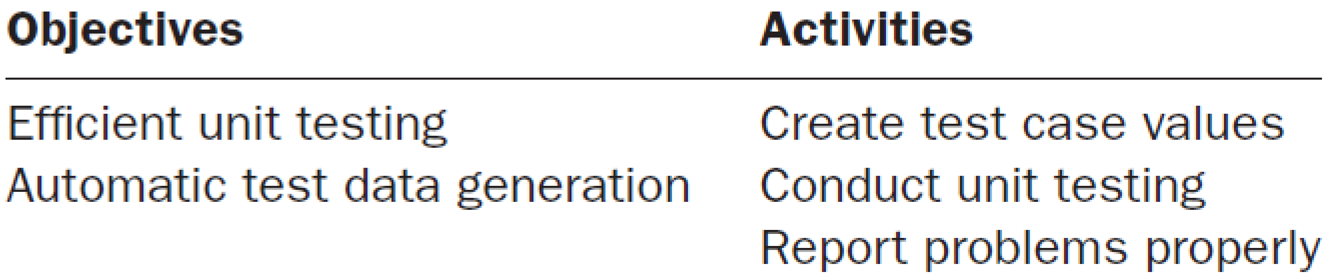
Objectives	Activities
Avoid mismatches of interfaces	Specify system test cases
Prepare for unit testing	Develop integration and unit test plans
	Build or collect test support tools
	Suggest ordering of class integration

Testing objectives and activities during detailed design



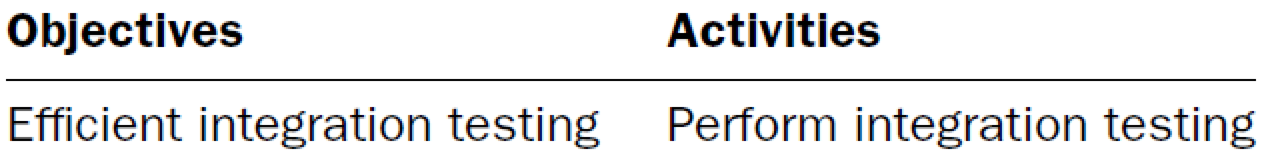
Objectives	Activities
Be ready to test when modules are ready	Create test cases (if unit) Build test specifications (if integration)

Testing objectives and activities during implementation



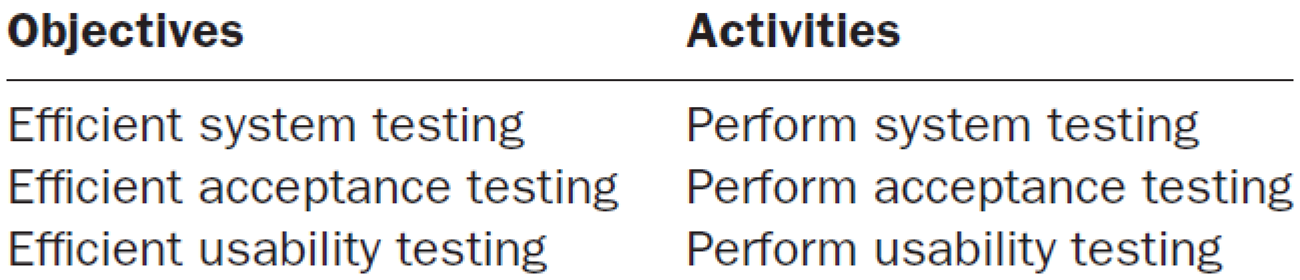
Objectives	Activities
Efficient unit testing	Create test case values
Automatic test data generation	Conduct unit testing
	Report problems properly

Testing objectives and activities during integration



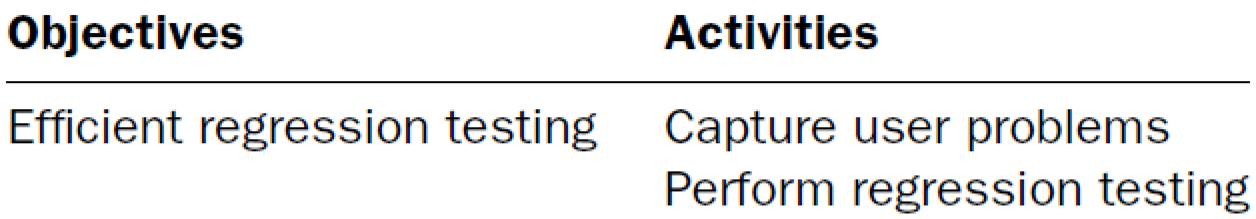
Objectives	Activities
Efficient integration testing	Perform integration testing

Testing objectives and activities during system deployment



Objectives	Activities
Efficient system testing	Perform system testing
Efficient acceptance testing	Perform acceptance testing
Efficient usability testing	Perform usability testing

Testing objectives and activities during operation and maintenance



Objectives	Activities
Efficient regression testing	Capture user problems
	Perform regression testing

Managing Test Artifacts

- Don’t fail because of lack of organization
- Keep track of :
  - Test design documents
  - Tests
  - Test results
  - Automated support
- Use configuration control
- Keep track of source of tests – when the source changes, the tests must also change

Professional Ethics

- Put quality first : Even if you lose the argument, you will gain respect
- If you can’t test it, don’t build it
- Begin test activities early
- Decouple
  - Designs should be independent of language
  - Programs should be independent of environment
  - Couplings are weaknesses in the software!
- Don’t take shortcuts
  - If you lose the argument you will gain respect
  - Document your objections
  - Vote with your feet
  - Don’t be afraid to be right!

Outline

Test Plans

- The most common question we hear about testing is

“ <u>How do I write a test plan</u>? ”

- This question usually comes up when the focus is on the document, not the contents
- It’s the contents that are important, not the structure
  - Good testing is more important than proper documentation
  - However – documentation of testing can be very helpful
- Most organizations have a list of topics, outlines, or templates

Standard Test Plan

- ANSI / IEEE Standard 829-1983 is ancient but still used

<u>Test Plan</u>

A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.

- **Many organizations are required to adhere to this standard**
- **Unfortunately, this standard emphasizes documentation, not actual testing – often resulting in a <u>well documented vacuum</u>**

Types of Test Plans

- Mission plan – tells “why”
  - Usually one mission plan per organization or group
  - Least detailed type of test plan
- Strategic plan – tells “what” and “when”
  - Usually one per organization, or perhaps for each type of project
  - General requirements for coverage criteria to use
- Tactical plan – tells “how” and “who”
  - One per product
  - More detailed
  - Living document, containing test requirements, tools, results and issues such as integration order

Test Plan Contents – System Testing

- Purpose
- Target audience and application
- Deliverables
- Information included
  - Introduction
  - Test items
  - Features tested
  - Features not tested
  - Test criteria
  - Pass / fail standards
  - Criteria for starting testing
  - Criteria for suspending testing
  - Requirements for testing restart
  - Hardware and software requirements
  - Responsibilities for severity ratings
  - Staffing & training needs
  - Test schedules
  - Risks and contingencies
  - Approvals

Test Plan Contents – Tactical Testing

- Purpose
- Outline
- Test-plan ID
- Introduction
- Test reference items
- Features that will be tested
- Features that will not be tested
- Approach to testing (criteria)
- Criteria for pass / fail
- Criteria for suspending testing
- Criteria for restarting testing
- Test deliverables
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing & training needs
- Schedule
- Risks and contingencies
- Approvals

Outline

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
- This section presents four general methods for checking outputs:
  - Direct verification
  - Redundant computation
  - Consistency checks
  - Data redundancy

Using a program to check the answer

- Appealing because it eliminates some human error
- Fairly expensive – requiring 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
  - Flow calculations in a stream – the solution is an approximation based on models and guesses; we don’t know the correct answers !
  - Probability of being in a particular state in a Petri net – again, we don’t know the correct answer

Direct Verification Example

- Consider a simple sort method
- Post-condition : Array is in sorted order

Input	8	92	7	14
Output	1	2	3	4
Output	92	14	8	7

Oops !

Oops !

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

Input	87	14	14	87
Output	14	14	14	87

Oops !

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

Direct Verification Example – *Cont*.

```
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 but 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 in the verification methods

Computing the answer in a different way

- Write two programs – check that they produce the same answer
- Very expensive !
- What if they have similar mistakes… that is, they fail on the same inputs
  - Unfortunately, researchers have found the “independence assumption” is not valid
- 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

Redundant Computation (2)

- On a similar note:
- Sometimes a problem might have different algorithms to solve it
- Implementations of the different algorithms are
  - Excellent candidates for checking against each other
  - Even though the common failure problem still remains
- For example, consider searching algorithms
  - A binary search routine could easily be tested by comparing the result with a linear search

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 and does not always apply, but is very useful within those limits

Consistency Checks (2)

- Consistency analysis can also be internal
- Recall the RIP (reachability, infection, propagation) model for failures
- External checks can only examine the outputs, so the infection must propagate for the error to be detected
- Internal checks raise the possibility of identifying faulty behavior with only the first two (RI) properties

Consistency Checks (3)

- It is quite common for programmers to require certain relations to hold on internal structures
- For example, an object representation might require that a given container never hold duplicate objects
- Checking these “invariant” relations is an extremely effective way of finding faults
- Programmers trained in developing software under the contract model can produce the code for such checking in the course of normal development
- For object-oriented software, such checks are typically organized around object invariants
  - Both on the abstraction of the object and on its representation
  - As well as object method preconditions and postconditions

Compare results of different *inputs*

- Powerful method of evaluating correctness on a given input is to consider how the program behaves on other 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

Data Redundancy (2)

- <u>Inserting</u> an element into a structure and removing it
- They often work with classes
- For example, adding an element to a container and then removing the element from the container often has a well-defined effect on the container
- For some containers, such as bags, the result is no change at all
- For other containers, such as sets
  - the result might be no change, or
  - it might be a change of one element, depending on whether the item was originally in the container
- These are only partial solutions and does not always apply, but is very useful within those limits

Summary

- A major obstacle to the adoption of advanced test criteria is that they affect the process
  - It is very hard to change a process
  - Changing process is required to move to level 3 or level 4 thinking
- Most testing is actually regression testing
- Test criteria make regression testing much easier to automate
- OOP has changed the way in which we integrate and test software components
- To be successful, testing has to be integrated throughout the process
- Identifying correct outputs is almost as hard as writing the program

References

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008