Advanced testing topics
Integration testing
Transitive verification
Testing patterns
Integration testing

The process whereby the whole product is built from its parts, testing along the way.
Key principle: use the success of past tests to condition the success of future tests.

Two kinds:

**bottom-up**: start with *leaves* of module/call diagram, integrate upward.

**top-down**: start with *root* of module diagram, integrate downward.

Both kinds:

attempt to localize defects in the smallest testable contexts.
rely upon prior tests to assure preconditions required for later tests.
Appropriate testing exhibits a "snowball" effect; tests that we have done so far reduce future work.
Bottom-up testing is natural
verify a leaf,
then verify what calls it.
use a **scaffolding** to test each leaf.

Top-down testing is difficult to understand
"fake" the functions of leaves, test top level.
Add leaf functions after top-level seems to work.
Use **stubs** as placeholders for untested leaves.
A scaffolding
   provides the environment(s) the unit expects.
   performs a variety of tests.
   reports the results.

Scaffoldings are complex!
 A scaffolding for equivalence partition testing might be 10 times as complex as the unit under test!
 (I tell my students in implementation courses that the test program is 10 times the length of their program.)
A "stub" is a placeholder for a function that has not been written yet. When you first compile a top-level module, one way to compile it is to "stub out" the functions it needs.

Stubs are simple!
Some stubs might do nothing at all, e.g., stubs for a database model. Other stubs might return values that are reasonable but incorrect.
Applying scaffolding: Testing libraries of functions.

Every function gets its own test suite. A scaffolding program runs all tests. There is no reasonable meaning to "top-down" in this context, because there are several tops!
A natural application of stub-testing: user interfaces
Stubs return "conforming" but "incorrect" values. Can test the user interface before one tests the correctness of its data feeds.

Stub testing is typically very obvious and easy.
Obvious uses for bottom-up (scaffolded) testing: libraries or reusable code of other kinds.
situations in which it is obvious -- at lower levels -- what correct behavior should be.

Obvious uses for top-down (stubbed) testing: complex, data-driven user environments.
situations in which the top-level code is needed to detect correctness of the lower-level code.
A typical case history
I was writing a complex MVC environment.
I needed the environment in order to validate the model!
So, I stubbed the model part and verified the interface.
Every stub returned a constant value.
Updates to data didn't do anything.
Once the user interface seemed to work, I plugged in the real model.
Now I could see model botches by using the interface.
(These are invisible unless I have the view working!)

Transitive claim is reasoning of the form
"If A is correct, then B is correct."
"A is correct."
-------------------------------------
"B is correct."
Snowballing in testing
The best test plan
Builds upon prior successes.
Makes best use of the software to test itself (minimizes scaffolding)
Is inexpensively repeatable after any change.
The dream of testing:
   test once, works everywhere.

This is just a dream.
   portability testing is always needed.
   even if the application is written to be portable.

An industry joke: Java is a "write once, debug everywhere" language.

A possible solution to the quandary:
standards bases: define preconditions for the application as a whole within its environment
Why transitive verification is needed
Testing something completely is expensive. Without further machinery, must test it on every platform on which it executes.
With transitive verification
Do something expensive once. Further testing is cheap.
A simple example of transitive verification: Android

Every android App is constrained to interact with its environment only in specific ways
- Use library functions.
- Don't open system files.
The Android environment is constructed so that -- whether it's running on top of Windows or Linux -- this environment exists and works.

So, if you write your application in a manner that is **entirely compliant** with Android best practices,
- The fact that it works in a linux-based Android means that it works in a Windows-based Android!

Test once; works everywhere!

Application A, Tests T, Environment E.
V(E) = assertion that the environment is compliant (e.g., supports Android functions).
V(A) = assertion that the app is compliant.
T(A,E) = assertion that A passes the tests T in environment E.
if $V(E)$ and $V(A)$ and $T(A,E)$,
then if $V(E')$, we can conclude that $T(A,E')$
The Linux Standard Base
Developed for vendors writing open-source linux.
Goal: test once, works everywhere.
Technology: transitive verification claims.

Design goals of LSB
Works even if the source code is unavailable; analyzes object code.
Reason: vendors (e.g. Corel Draw, Mathematica, etc) won't release their source code.
Uses static analysis of the execution environment and the program. Doesn't execute the program.
Leverages potentially expensive testing in one environment to avoid testing in another.
LSB components
- an **application A** to be tested, in **compiled** form.
- an **environment E** where it is to be tested, e.g., RedHat 5 or Ubuntu or....
- a **test suite** $T(A,E)$ that tests that application A works in environment E.
- an **environment verifier** $V(E)$ that checks that proper libraries and versions are present in an environment E.
- an **application verifier** $V(A)$ that checks that library functions are only called with the proper types of arguments, and that files are only opened in documented and standard locations.

**This works on compiled (closed-source) code A!**

**LSB Logic:**

For an application A and two environments E and E', if $V(A)$ and $V(E)$ and $T(A,E)$, then for all $E'$ with $V(E')$, there is evidence that $T(A,E')$. In other words, "test once, (potentially) works everywhere".

![Diagram](attachment://diagram.png)
Inside the LSB validator

Inside the LSB application validator checks:
- names of files that are opened.
- how system calls are used.
- which system calls are used.

Inside the LSB environment validator checks:
- versions of system calls and libraries and the kernel.
- where files are located in linux.
LSB Logic is based upon a **transitive verification claim**: For an application $A$ and an environment $E$, if $V(A)$ and $V(E)$ and $T(A,E)$, then for all $E'$ with $V(E')$, there is evidence that $T(A,E')$. In other words, "test once, (potentially) works everywhere".
V(A) and V(E) are static verifiers:
same code for all applications.
Really cheap to use.
Automatic; no human intervention required.

T(A,E) is a dynamic verifier:
Expensive to do.
Might require human effort, e.g.,
verifying a user interface.

We can skip something expensive (T(A,E'))
after doing one expensive thing (T(A,E)) and
two cheap things (V(A), V(E)).

"Test once, runs everywhere that is verified
as a suitable environment."
LSB is an imperfect solution
"Catches" common environment/configuration mistakes that impair function.

Does not deal with:
- coding defects that manifest when changing the architecture of a machine, e.g., SPARC versus X86 versus X86_64. (this wreaks havoc with uninitialized variables, which have different default values)
- defects that depend upon files outside the LSB's registered set of files. E.g., program crashes if there is not an /etc/motd!

But it is definitely better than nothing!
Testing patterns

Another way to save time: testing patterns.
Tried and successful ways to test specific kinds of code.
Several **special cases** of testing
  - client-server
  - interactive application
  - web application

Key to each one is in part **applying what we've learned already**.
  - equivalence partitioning
  - path basis testing
  - scaffolding
  - stubbing

Every testing pattern defines
  - Some concept of completeness of testing (path basis, equivalence partitioning, etc)
  - Some methodology for testing (scaffolding, stubbing, etc)
Client-server testing
Monday, November 23, 2009
2:15 PM

Client-server (distributed) programs
Client and server are loosely coupled
Client can run without server, and vice versa

Basic methodology: bottom-up
Test server without client (use scaffolding)
  (equivalence partitioning/state analysis)
Test client without server (use stubs)
  (equivalence partitioning)
Finally, test them together
  (equivalence partitioning, use cases, path analysis)
Testing state

Server functions are often persistent, i.e., there is some state change due to an action
Can test the **state transition diagram** for the service
Can each state be achieved?
Can each transition be accomplished?
Does every path work?

A key idea
Paths through a state transition diagram are like paths through code.
There is a dimension and a path basis!
Can test representative paths by testing the path basis!

This is a concept of exhaustive testing.
State machine testing
   Reach every state.
   Traverse every edge.
   Traverse common use cases.
   traverse representative paths.

Path traversal
   Either
      Select start and end states.
   or
      Define one universally reachable state as start and end.
Reach every state
Test every transition
Test representative paths

Nodes are states.
Edges are user actions.
Testing interactive applications

Interactive and web applications have a different structure than traditional programs

Flow charts are irrelevant.
Event-condition-action (ECA) structure.

State-machine model of behavior

condition C: what state are you in?
event E: what happened?
action A: what should program do?

An ECA list corresponds again to a state transition diagram.
The user is the one who makes state changes!
Testing strategies for interactive programs

"press every button"
"test common use cases as sequences of user actions"
"test representative sequences of user actions"

**What is a representative sequence?**
"press every button" is analogous to "execute every line"
"take every allowed path of interactions" is analogous to "test every path through the code"

But what we mean by taking a "path" here refers to the **user**.
Thus we can use **white-box** methods for modeling the actions of a **user** of a **black-box** component.
User state transitions

Monday, November 23, 2009
4:58 PM
Finding defects

Recall that any testing mechanism either checks requirements finds defects.

Example: uninitialized variable found by path basis testing.
Path basis testing for interactive programs

Describe user experience as a set of paths starting at a "start" node and ending at an "end" node, where "end" reconnects to "start".

Compute a path basis for the user.

Result: every page is loaded in every context.
Special considerations for web programs

Two parts: server and client.
Server entry points (e.g. URLs) do not depend upon client actions.
Can test them out of context.
Preconditions are not guaranteed.
Bottom-up test model:

Step 1: test that URLs do what they should
  scaffolding: HTML page that calls URL under various conditions.
  (or program that emulates a browser)
Step 2: test that one cannot "trick" URLs into behaving inappropriately
  scaffolding: HTML page (or program)
Step 3: Ensure that each page reference satisfies preconditions for the page.
  point at dummy page; exhibit arguments.
Step 4: Integration: press every button.
Step 5: Usability: perform standard use cases.
Step 6: Comprehensiveness: test alternative paths.
Step 7: Security: try illicit references to pages (e.g., without preconditions)
Top-down testing model

"Press every button"
"Try every use case"
"Try representative paths"
and when these fail, do unit testing!
Is path basis testing appropriate for web applications?

A very tricky question

Path basis testing exposes defects due to **conditions** arising from **different paths**.

Typical example: a pointer that can be uninitialized.

Web programs function entirely based upon invoking sub-programs.

The only uninitialized variable defects arise from uninitialized **cookies**.

If a page doesn't **initialize** a cookie, it can't possibly affect whether the cookie is **initialized**.

So largely, path basis testing isn't needed!

Important lesson: testing methodologies look for particular kinds of defects; if these can't happen, then you're wasting your time!
Testing patterns:

Everything we have learned so far is an instance of a testing pattern.

A testing pattern is a way to test that has been proven through experience.

So far, we know the following patterns:

- Scaffolding
- Stubbing
- Basis testing
- Equivalence partitioning
- State machine testing
Some more common testing patterns

**Scenario testing**: test function of use cases

**Pair testing**: two testers (people) work together to form a comprehensive test suite.

**Separate test interface**: create a set of methods intended only for testing private components of an object.