Lecture 19: Automated Testing

Other testing strategies:
- Quicktests
- Exploratory Testing

Automated testing
- JUnit and family
- Testing GUI-based software

Test coverage for Object-Oriented Systems

When to stop testing

Quick Tests

A quick, cheap test
- e.g. Whittaker "How to Break Software"

Examples:
- The Shoe Test (key repeats in any input field)
- Variable boundary testing
- Variability Tour: find anything that varies, and vary it as far as possible in every dimension

Whittaker's QuickTests

Explore the input domain

1. Inputs that force all the error messages to appear
2. Inputs that force the software to establish default values
3. Explore allowable character sets and data types
4. Overflow the input buffers
5. Find inputs that may interact, and test combinations of their values
6. Repeat the same input numerous times

Explore the outputs

7. Force different outputs to be generated for each input
8. Force invalid outputs to be generated
9. Force properties of an output to change
10. Force the screen to refresh

Explore stored data constraints

11. Force a data structure to store too many or too few values
12. Find ways to violate internal data constraints

Explore feature interactions

13. Experiment with invalid operator/operand combinations
14. Make a function call itself recursively
15. Force computation results to be too big or too small
16. Find features that share data

Vary file system conditions

17. File system full to capacity
18. Disk is busy or unavailable
19. Disk is damaged
20. invalid file name
21. vary file permissions
22. vary or corrupt file contents

Interference Testing

Generate Interrupts

- From a device related to the task
- From a device unrelated to the task
- From a software event

Change the context

- Swap out the CD
- Change contents of a file while program is reading it
- Change the selected printer
- Change the video resolution

Cancel a task

- Cancel at different points of completion
- Cancel a related task

Pause the task

- Pause for short or long time

Swap out the task

- e.g. change focus to another application
- e.g. load processor with other tasks
- e.g. put the machine to sleep
- e.g. swap out a related task

Compete for resources

- e.g. get the software to use a resource that is already being used
- e.g. run the software while another task is doing intensive disk access

Exploratory Testing

Start with idea of quality:
- Quality is value to some person

So a defect is:
- something that reduces the value of the software to a favoured stakeholder
- or increases its value to a disfavoured stakeholder

Testing is always done on behalf of stakeholders
- Which stakeholder this time?
- e.g. programmer, project manager, customer, marketing manager, attorney…
- What risks are they trying to mitigate?

You cannot follow a script
- It's like a crime scene investigation
- Follow the clues…
- Learn as you go…

Kaner's definition:

Exploratory testing is

…a style of software testing

…that emphasizes personal freedom and responsibility

…of the tester

…to continually optimize the value of their work

…by treating test-related learning, test design, and test execution

…as mutually supportive activities

…that run in parallel throughout the project

Test Ideas

Function Testing: Test what it can do.

Domain Testing: Divide and conquer the data.

Stress Testing: Overwhelm the product.

Flow Testing: Do one thing after another.

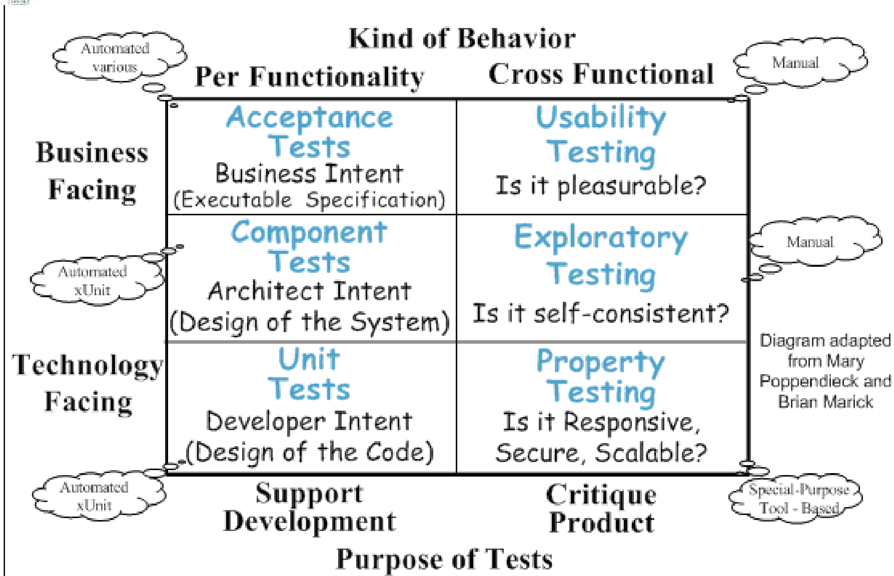
Scenario Testing: Test to a compelling story.

Claims Testing: Verify every claim.

User Testing: Involve the users.

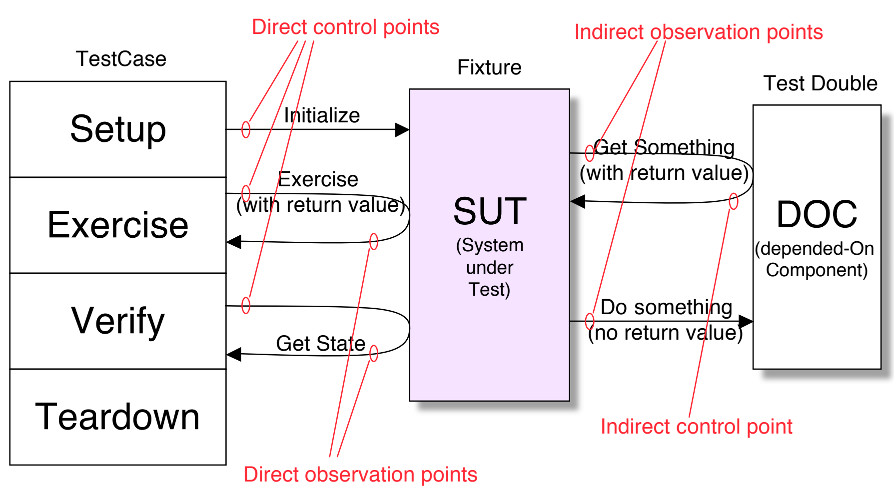
Risk Testing: Imagine a problem, then find it.

Automatic Testing: Write a program to generate and run a zillion tests.



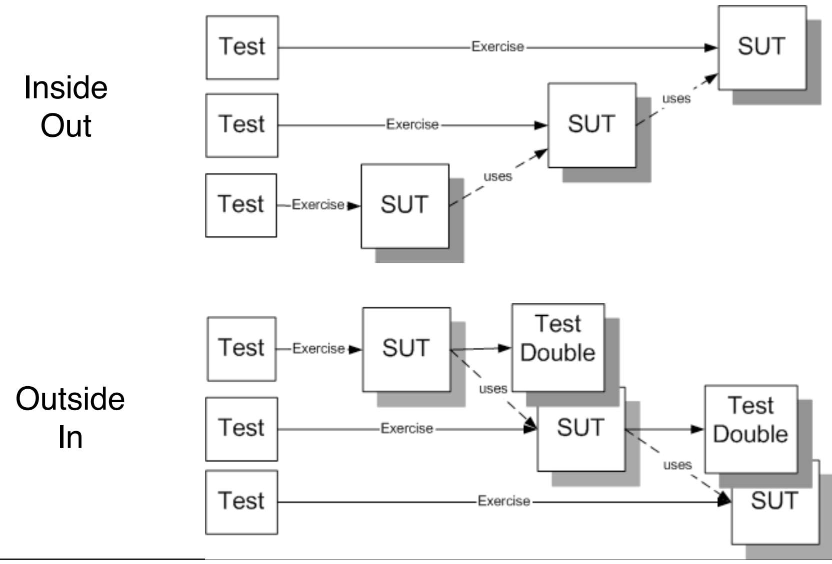
Automated Testing Strategy

Source: *Adapted from Meszaros 2007, p66*



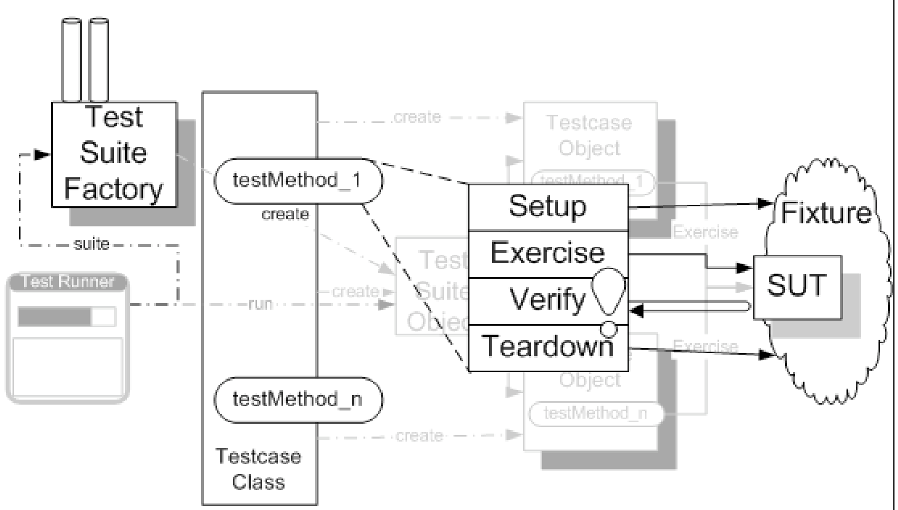
Test Order?

Source: *Adapted from Meszaros 2007, p35*



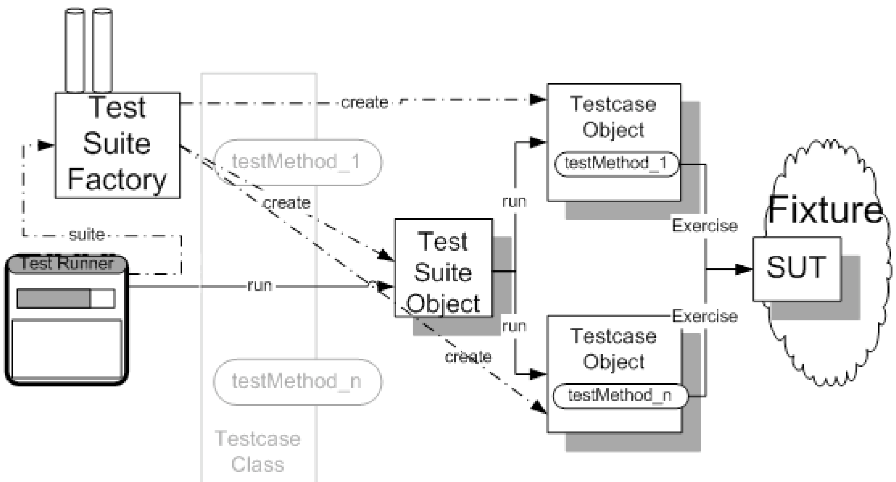
How JUnit works

Source: *Adapted from Meszaros 2007, p77*



How JUnit works

Source: *Adapted from Meszaros 2007, p77*



Assertion methods in JUnit

Source: *Adapted from Meszaros 2007, p365*

Single-Outcome Assertions

fail;

Stated Outcome Assertions

assertNotNull(anObjectReference);
assertTrue(booleanExpression)

Expected Exception Assertions

assert_raises(expectedError) {codeToExecute };

Equality Assertions

assertEqual(expected, actual);

Fuzzy Equality Assertions

assertEqual(expected, actual, tolerance);

Principles of Automated Testing

Source: *Adapted from Meszaros 2007, p39-48*

Write the Test Cases First

Design for Testability

Use the Front Door First
- test via public interface
- avoid creating back door manipulation

Communicate Intent
- Tests as Documentation!
- Make it clear what each test does

Don't Modify the SUT
- avoid test doubles
- avoid test-specific subclasses
- (unless absolutely necessary)

Keep tests Independent
- Use fresh fixtures
- Avoid shared fixtures

Isolate the SUT

Minimize Test Overlap

Verify One Condition Per Test

Test Concerns Separately

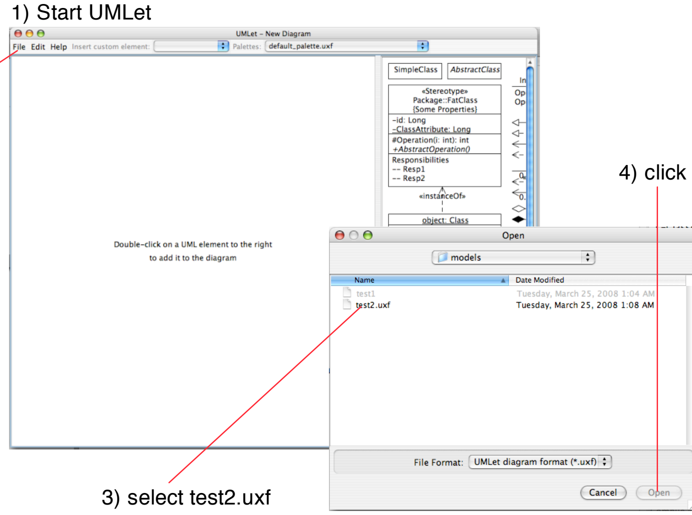
Minimize Untestable code
- e.g. GUI components
- e.g. multi-threaded code
- etc

Keep test logic out of production code
- No test hooks!

Testing interactive software

1) Start UMLet

2) Click on File -> Open



3) select test2.uxf

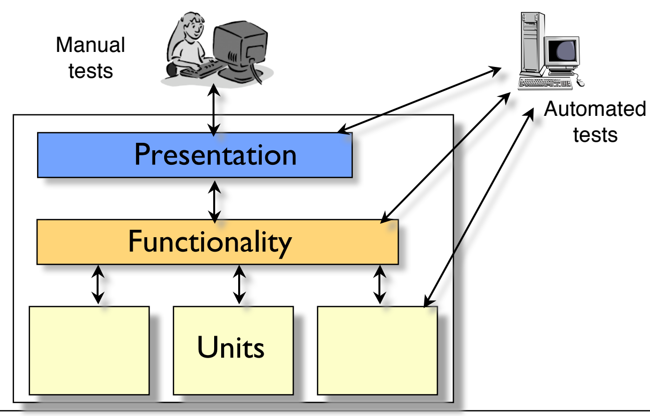
4) click Open

Automating the testing

Source: Adapted from Zeller 2006, p57

Challenges for automated testing:

- Synchronization - How do we know a window popped open that we can click in?
- Abstraction - How do we know it's the right window?
- Portability - What happens on a display with different resolution / size, etc



Presentation Layer

Source: Adapted from Zeller 2006, chapter 3

Script the mouse and keyboard events

- script can be recorded (e.g. "send_xevents @400,100")
- script is write-only and fragile

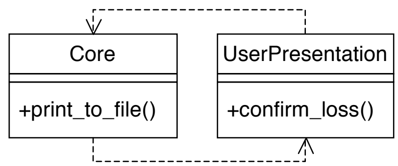
Script at the application function level

- E.g. Applescript: **tell** application "UMLet" **to** activate
- Robust against size and position changes
- Fragile against widget renamings, layout changes, etc.

Write an API for your application…

Circular Dependencies

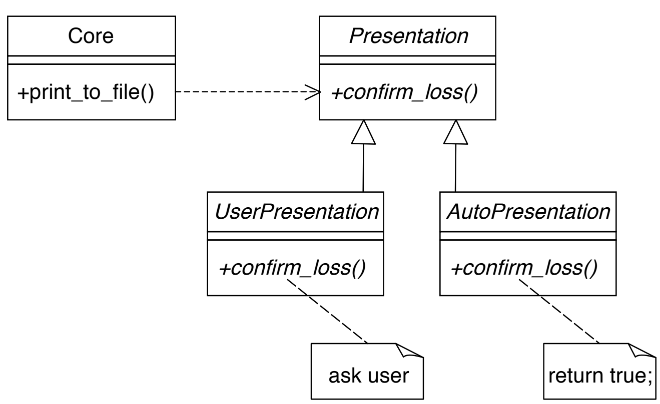
Source: *Adapted from Zeller 2006, chapter 3*




```
void print_to_file(string filename)
{
    if (path_exists(filename)) {
        // FILENAME exists; ask user to confirm overwrite
        bool confirmed = confirm_loss(filename);
        if (!confirmed)
            return;
    }
    // Proceed printing to FILENAME...
}
```


Revised Dependency

Source: *Adapted from Zeller 2006, chapter 3*



Testing Object Oriented Code

Encapsulation

- If the object hides it's internal state, how do we test it?
- E.g. add methods only to be used in testing, which expose internal state
- But: how do we know these extra methods are correct?

Inheritance

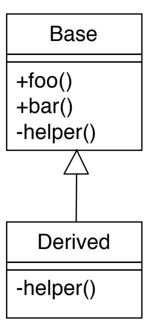
- When a subclass extends a well-tested class, what extra testing is needed?
- e.g. Test just the overridden methods?
- But with dynamic binding, this is not sufficient
- e.g. other methods can change behaviour because they call over-ridden methods

Polymorphism

- When class A calls class B, it might actually be interacting with any of B's subclasses…

Consider this program…

Source: *Adapted from IPL 1999*



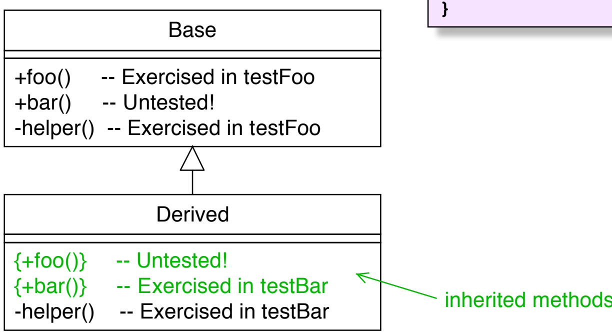

```
class Base {
  public void foo() {
    … helper(); …
  }
  public void bar() {
    … helper(); …
  }
  private helper() {…}
}

class Derived extends Base {
  private helper() {…}
}
```

Test Cases

Source: *Adapted from IPL 1999*

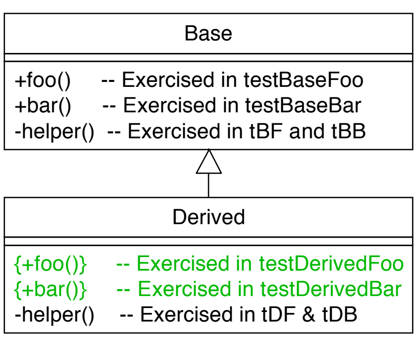
```
public void testFoo() {
  Base b = new Base();
  b.foo();
}
public void testBar() {
  Derived d = new Derived();
  d.bar();
}
```



Extend the test suite

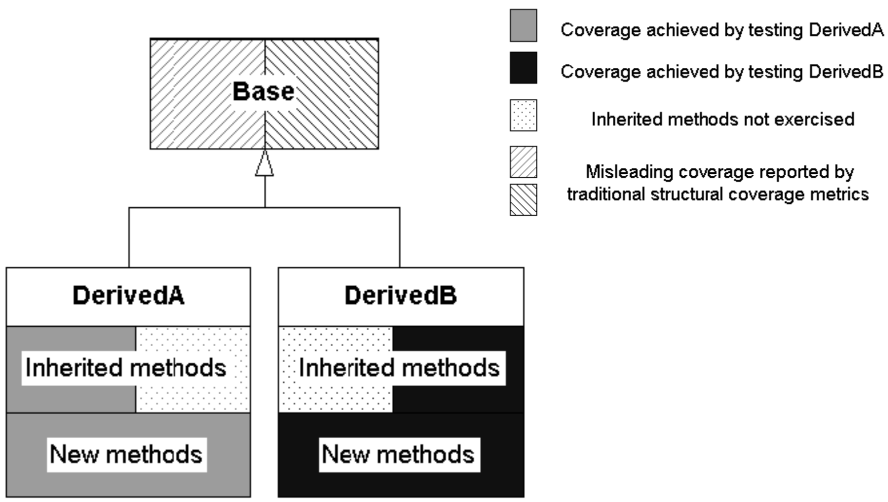
Source: *Adapted from IPL 1999*

```
public void testBaseFoo() {
  Base b = new Base();
  b.foo();
}
public void testBaseBar() {
  Base b = new Base();
  b.bar();
}
public void testDerivedFoo() {
  Base d = new Derived();
  d.foo();
}
public void testDerivedBar() {
  Derived d = new Derived();
  d.bar();
}
```



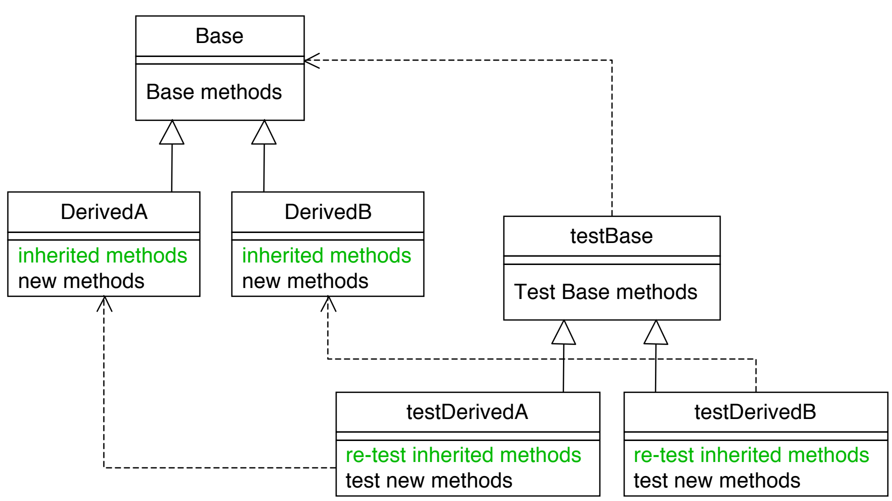
Inheritance Coverage

Source: *Adapted from IPL 1999*



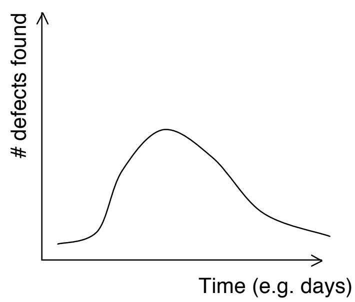
Subclassing the Test Cases

Source: *Adapted from IPL 1999*

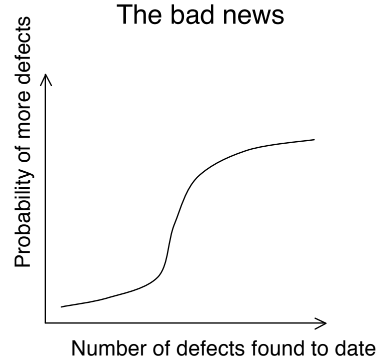


When to stop testing?

Typical testing results



The bad news



When to stop testing?

***Source:** Adapted from Pfleeger 1998, p359*

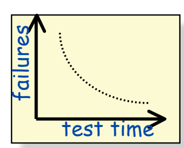
Motorola's Zero-failure testing model

- Predicts how much more testing is needed to establish a given reliability goal
- basic model:

empirical constants

$$\text{failures} = ae^{-b(t)}$$

testing time



Reliability estimation process

- Inputs needed:
  - fd = target failure density (e.g. 0.03 failures per 1000 LOC)
  - tf = total test failures observed so far
  - th = total testing hours up to the last failure
- Calculate number of further test hours needed using:

$$\frac{\ln(fd/(0.5 + fd)) \times th}{\ln((0.5 + fd)/(tf + fd))}$$

- Result gives the number of further failure free hours of testing needed to establish the desired failure density
  - if a failure is detected in this time, you stop the clock and recalculate
- Note: this model ignores operational profiles!

Fault Seeding

Seed N faults into the software

- Start testing, and see how many seeded faults you find
- Hypothesis:

$$\frac{\text{Detected seeded faults}}{\text{Total seeded faults}} = \frac{\text{Detected nonseeded faults}}{\text{Total nonseeded faults}}$$

- Use this to estimate test efficiency
- Estimate # remaining faults

Alternatively

- Get two teams to test independently
- Estimate each team's test efficiency by:

$$\text{Efficiency(team1)} = \frac{\text{\# faults found by team 1}}{\text{Total number of faults}} = \frac{\text{Faults found by both teams}}{\text{Total \# faults found by team 2}}$$

unknown