



Lecture 18: Automated Testing

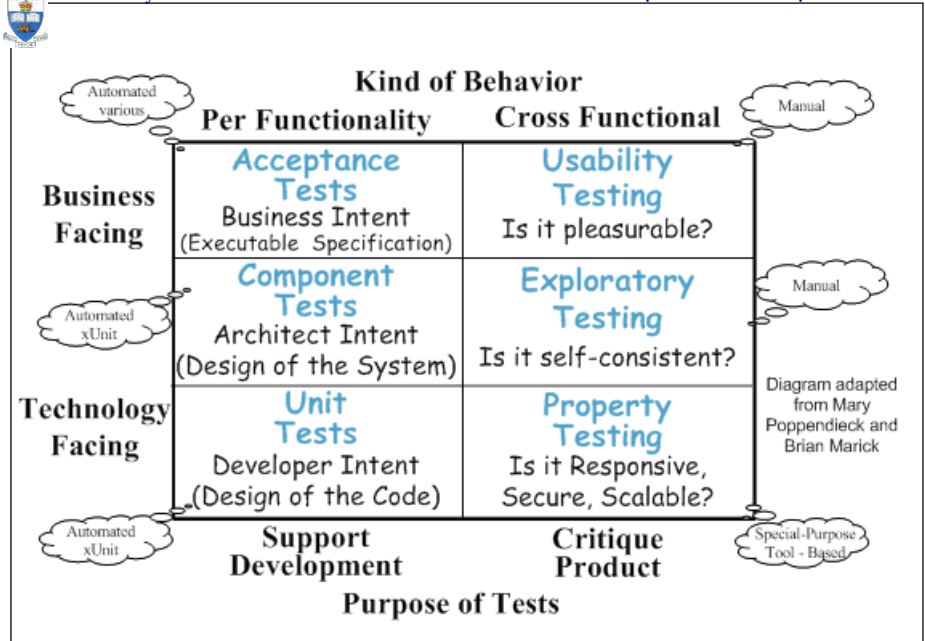
Automated testing

JUnit and family

Testing GUI-based software

Testing Object-Oriented Systems

When to stop testing





Automated Testing

Source: Adapted from Liskov & Guttag, 2000, pp239-242

Where possible, automate your testing:

- tests can be repeated whenever the code is modified (“regression testing”)
- takes the tedium out of extensive testing
- makes more extensive testing possible

Will need:

- test drivers** - automate the process of running a test set
 - sets up the environment
 - makes a series of calls to the **Unit-Under-Test (UUT)**
 - saves results and checks they were right
 - generates a summary for the developers

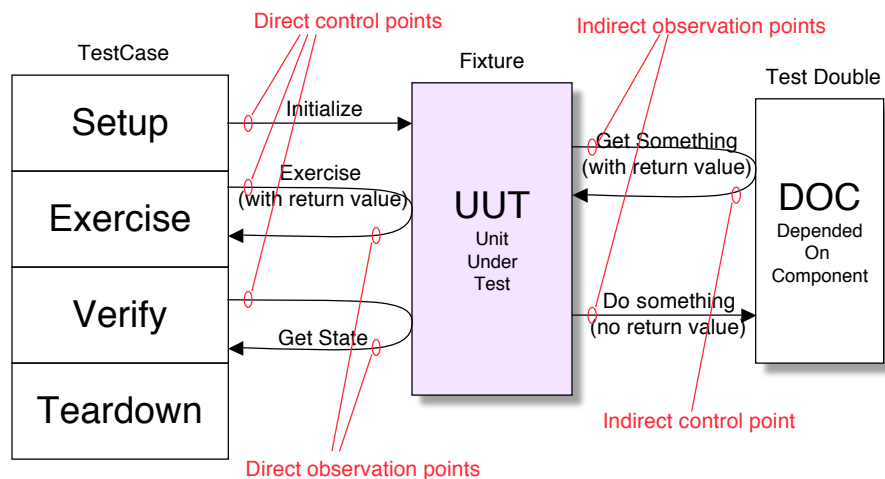
May need:

- test stubs** - simulate part of the program called by the unit-under-test
 - checks whether the UUT set up the environment correctly
 - checks whether the UUT passed sensible input parameters to the stub
 - passes back some return values to the UUT (according to the test case)
 - (stubs could be interactive - ask the user to supply return values)



Automated Testing Strategy

Source: Adapted from Meszaros 2007, p66

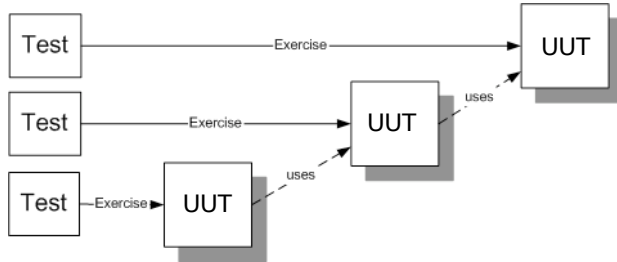




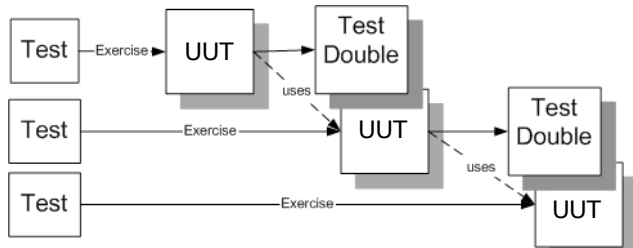
Test Order?

Source: Adapted from Meszaros 2007, p35

Inside Out

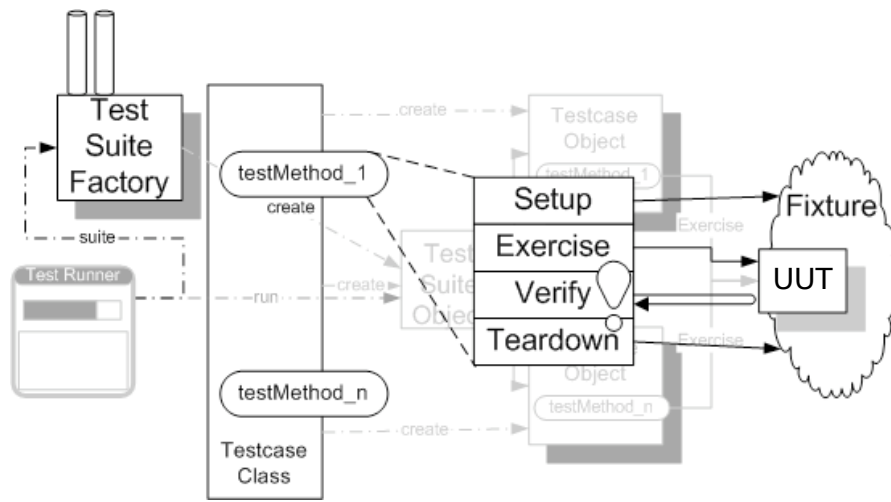


Outside In



How JUnit works

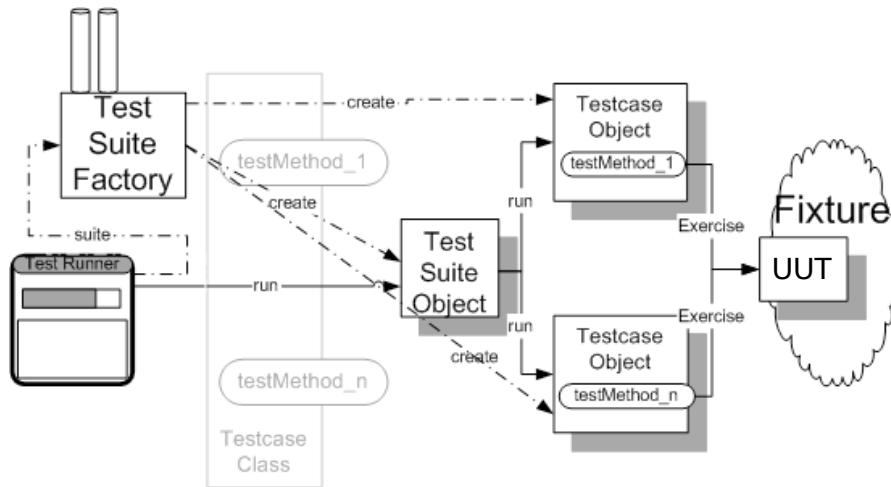
Source: Adapted from Meszaros 2007, p77





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 UUT

- avoid test doubles
- avoid test-specific subclasses (unless absolutely necessary)

Keep tests Independent

- Use fresh fixtures
- Avoid shared fixtures

Isolate the UUT

Minimize Test Overlap

Check 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

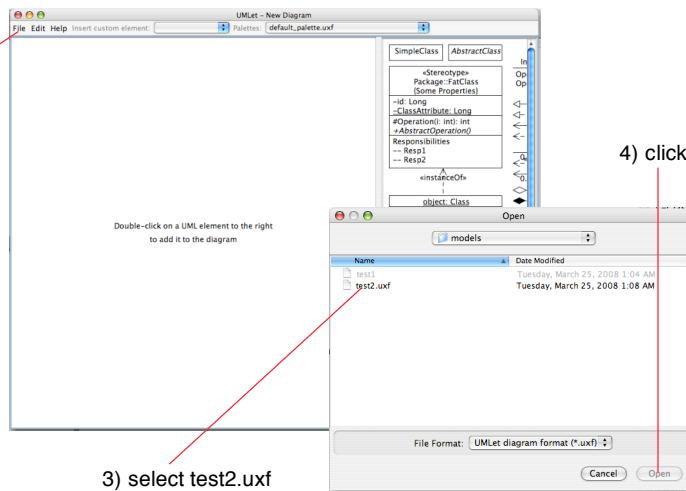
- No test hooks!



Testing interactive software

1) Start the application (e.g. UMLet)

2) Click on File -> Open



3) select test2.uxf

4) click Open



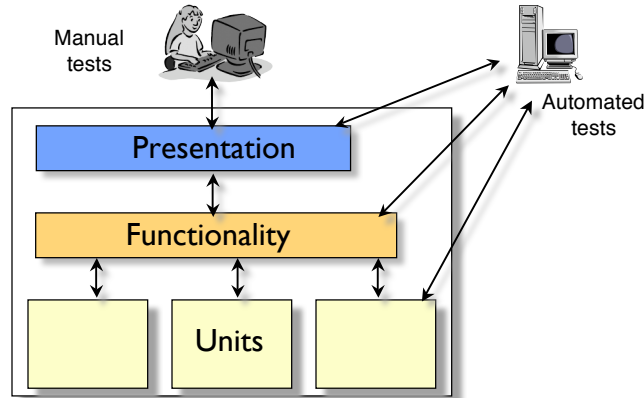


Automating the testing

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



Testing the 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.

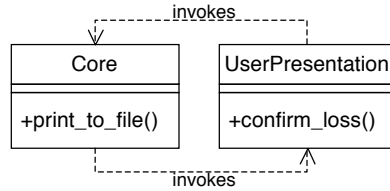
Write an API for your application...

- Allow an automated test to create windows, interact with widgets, etc.



Dealing with Circular Dependencies

Source: Adapted from Zeller 2006, chapter 3



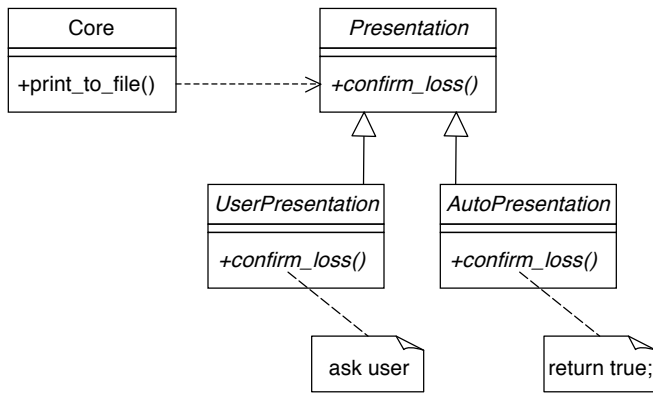
```

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





How to Test Object Oriented Code?

Encapsulation

If the object hides its internal state, how do we test it?
 Could add methods that expose internal state, only to be used in testing
 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

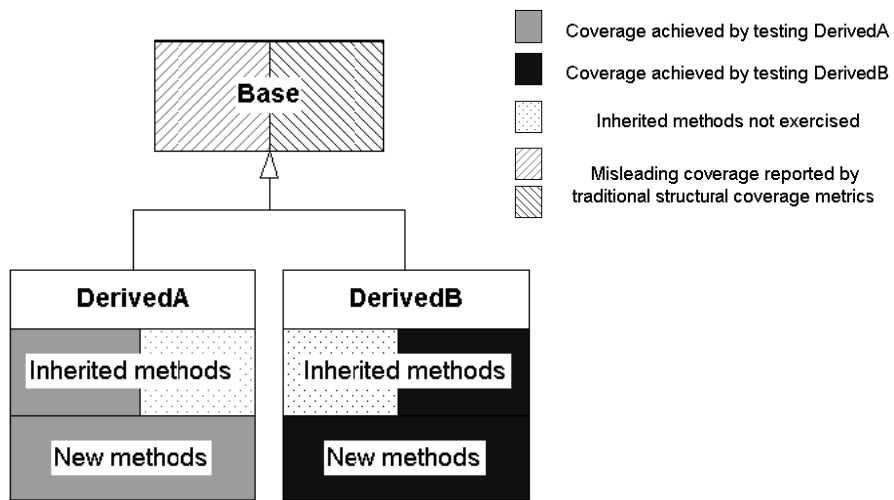
Polymorphism

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



Inheritance Coverage

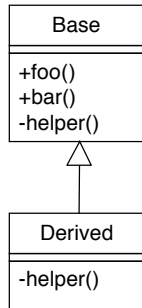
Source: Adapted from IPL 1999





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() {...}
}
  
```



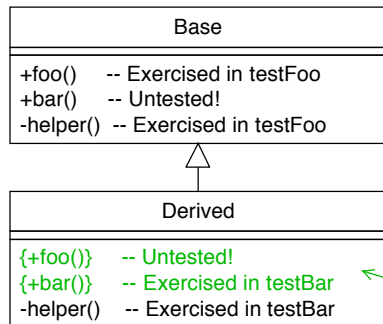
Test Cases

Source: Adapted from IPL 1999

```

public void testFoo() {
    Base b = new Base();
    b.foo();
}

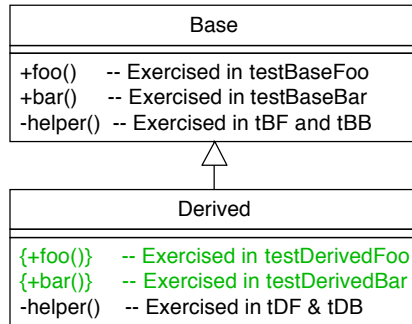
public void testBar() {
    Derived d = new Derived();
    d.bar();
}
  
```





Extend the test suite

Source: Adapted from IPL 1999



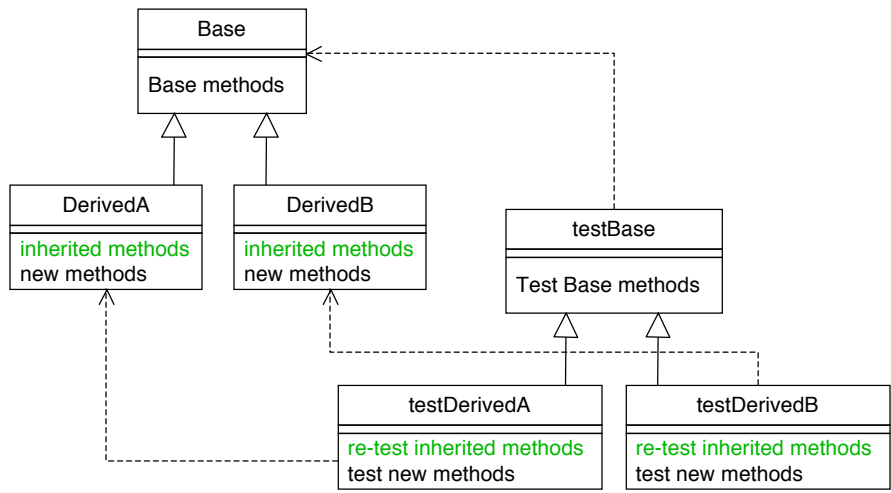
```

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();
}
  
```



Subclassing the Test Cases

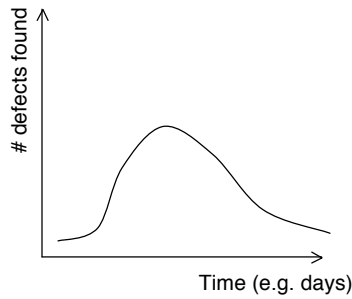
Source: Adapted from IPL 1999



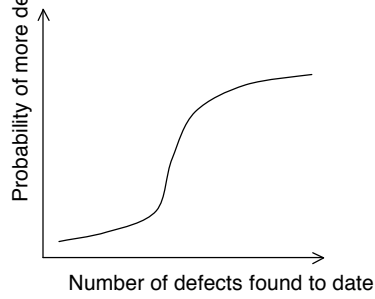


When to stop testing?

Typical testing results



The bad news



When to stop testing?

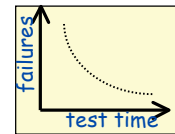
Source: Adapted from Pfleeger 1998, p359

Motorola's Zero-failure testing model

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

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

empirical constants 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:

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

unknown

