Lecture 19: Automated Testing

Strategy for automated tests

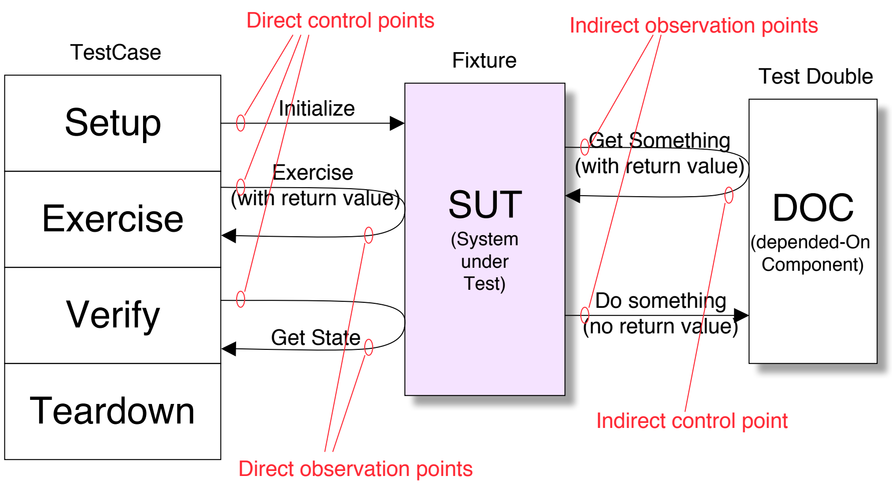
JUnit and family

Testing GUI-based software

Test coverage for Object-Oriented Systems

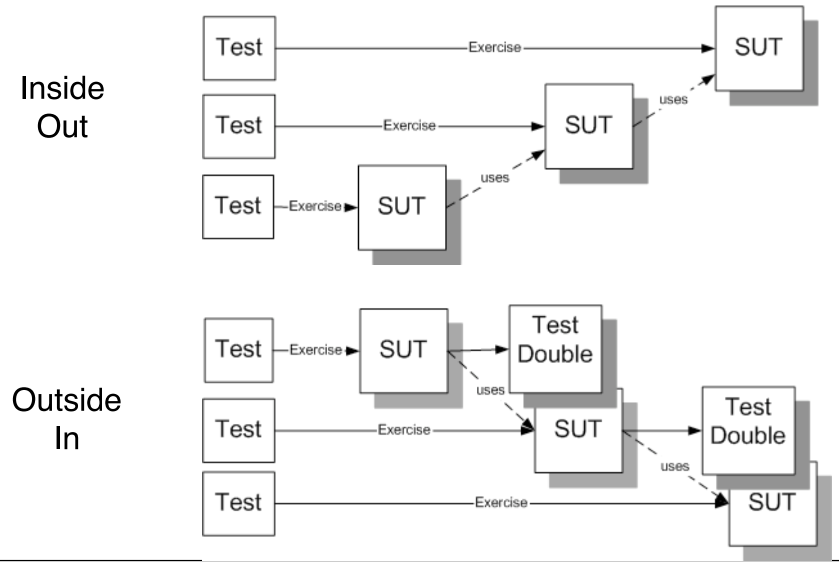
Automated Testing Strategy

Source: *Adapted from Meszaros 2007, p66*


Direct control points
Indirect observation points
TestCase
Fixture
Test Double
Setup
Exercise
Verify
Teardown
Initialize
Exercise (with return value)
Get State
SUT (System under Test)
Get Something (with return value)
Do something (no return value)
DOC (depended-On Component)
Indirect control point
Direct observation points

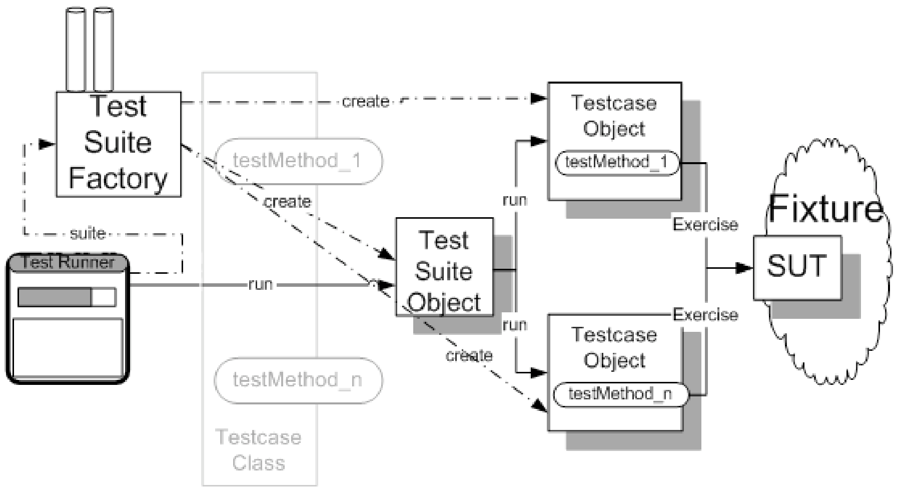
Test Order?

Source: *Adapted from Meszaros 2007, p35*



How JUnit works

Source: *Adapted from Meszaros 2007, p77*



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

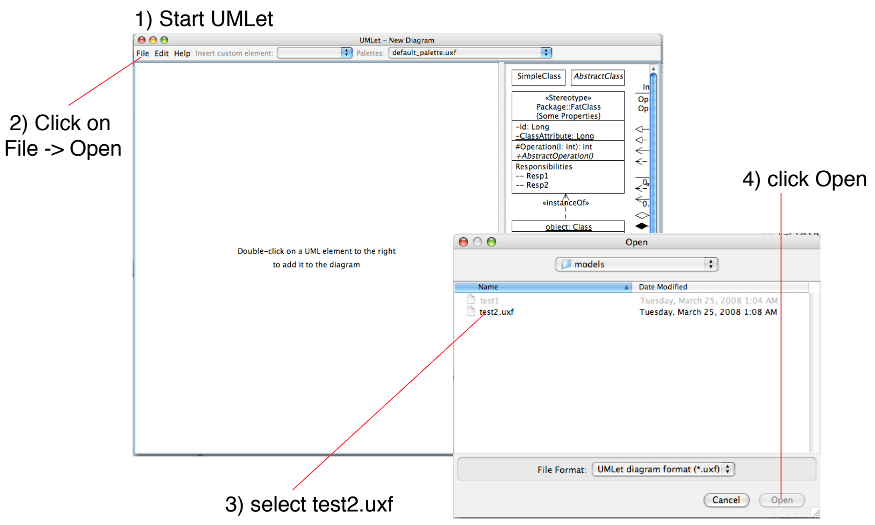
Verify One Condition Per Test

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

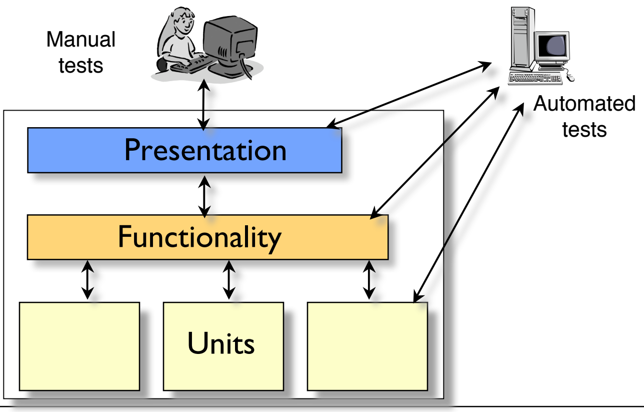

1) Start UMLet
2) Click on File -> Open
3) select test2.uxf
4) click Open
UMLet – New Diagram
File Edit Help Insert custom element: Palettes: default_palette.uxf
SimpleClass
AbstractClass
«Stereotype» Package::FatClass {Some Properties}
-id: Long
-ClassAttribute: Long
#Operation(i: int): int
+AbstractOperation()
Responsibilities
-- Resp1
-- Resp2
«instanceOf»
object: Class
Double-click on a UML element to the right to add it to the diagram
Open
models
Name
Date Modified
test1 Tuesday, March 25, 2008 1:04 AM
test2.uxf Tuesday, March 25, 2008 1:08 AM
File Format: UMLet diagram format (*.uxf)
Cancel
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

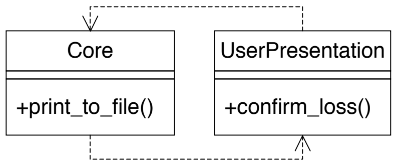
Script at the application function level

- 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

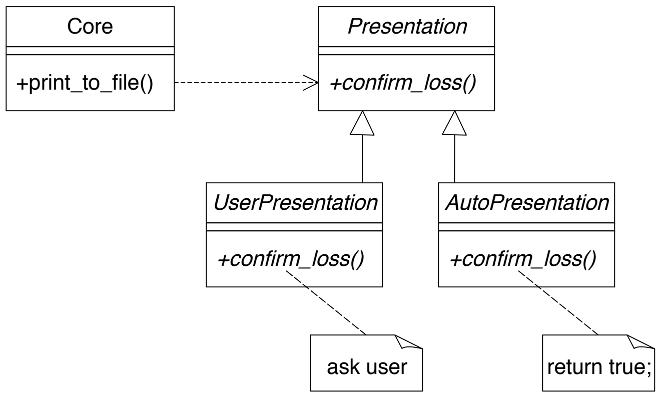
Source: *Adapted from Zeller 2006, chapter 3*



```
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

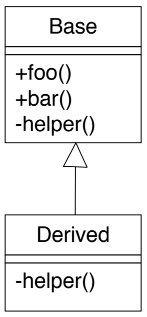
- When a subclass extends a well-tested class, what extra testing is needed?
- e.g. Test just the overridden methods?
- But with dynamic binding, this is not sufficient
- e.g. other methods can change behaviour because they call over-ridden methods

Polymorphism

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

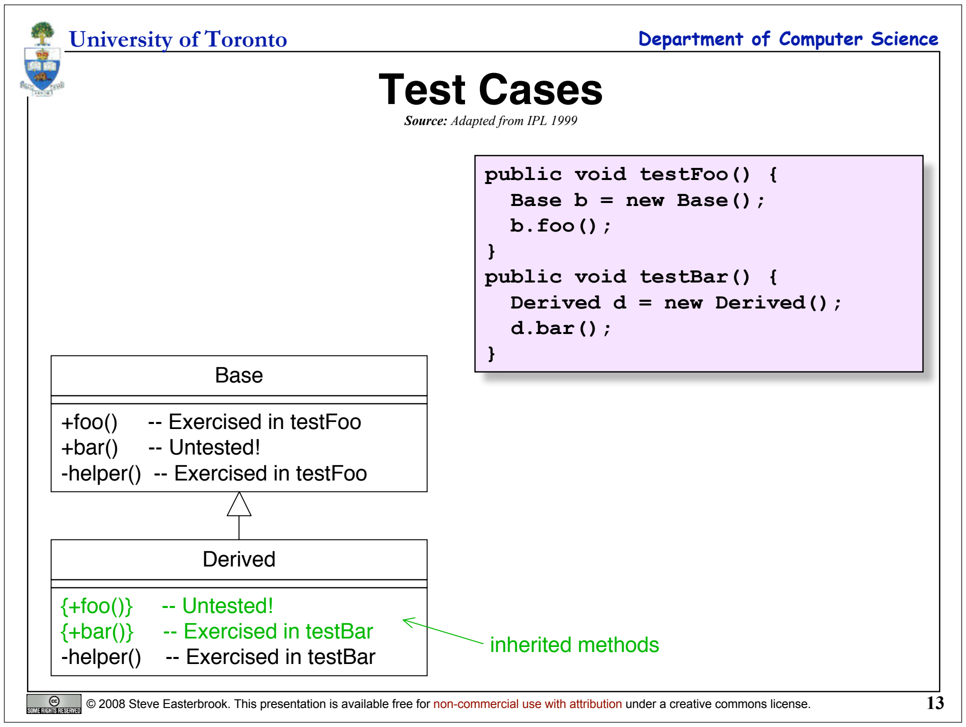
Consider this program…

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();
}
```

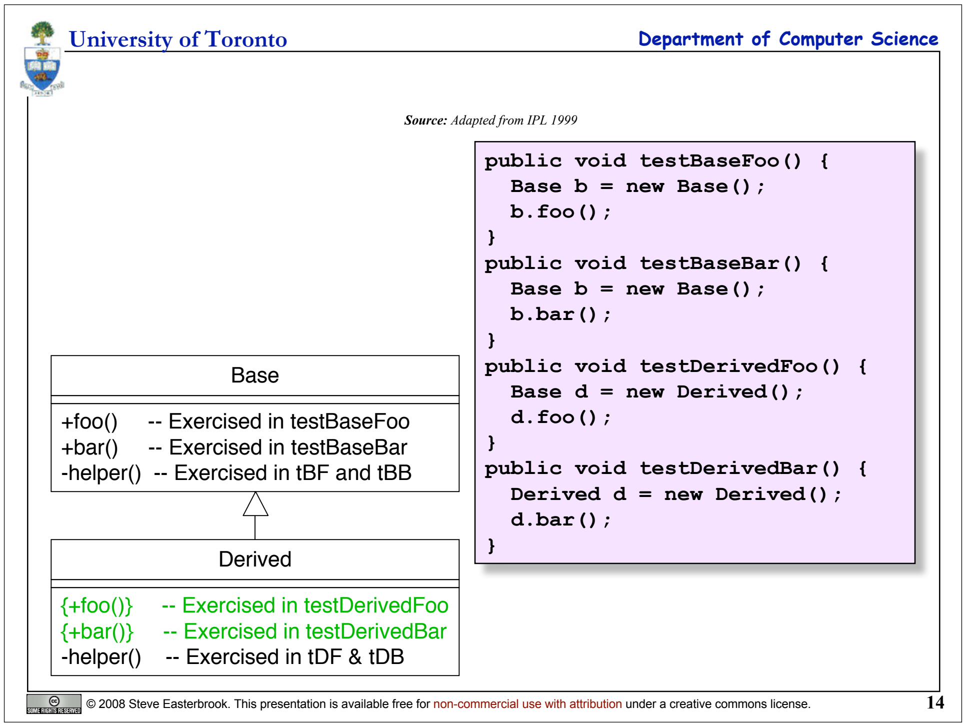
Base

+foo() -- Exercised in testFoo
+bar() -- Untested!
-helper() -- Exercised in testFoo

Derived

{+foo()} -- Untested!
{+bar()} -- Exercised in testBar
-helper() -- Exercised in testBar

inherited methods



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();
}
```

Base

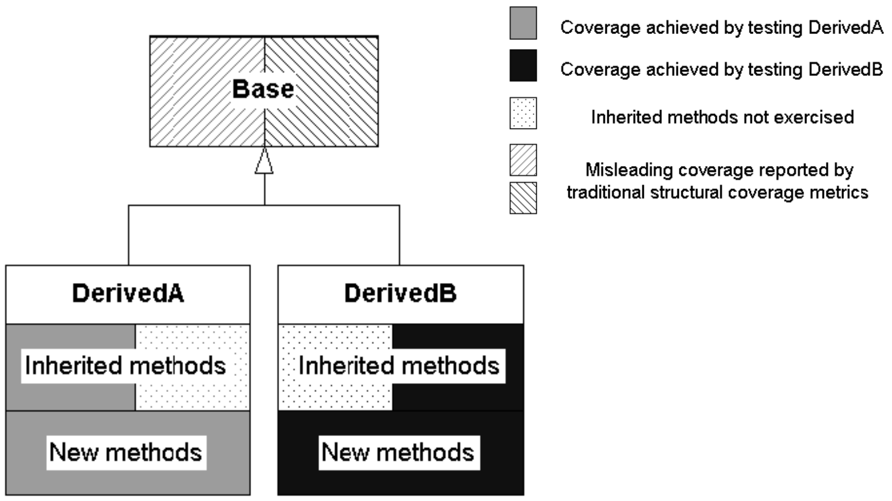
+foo() -- Exercised in testBaseFoo
+bar() -- Exercised in testBaseBar
-helper() -- Exercised in tBF and tBB

Derived

{+foo()} -- Exercised in testDerivedFoo
{+bar()} -- Exercised in testDerivedBar
-helper() -- Exercised in tDF & tDB

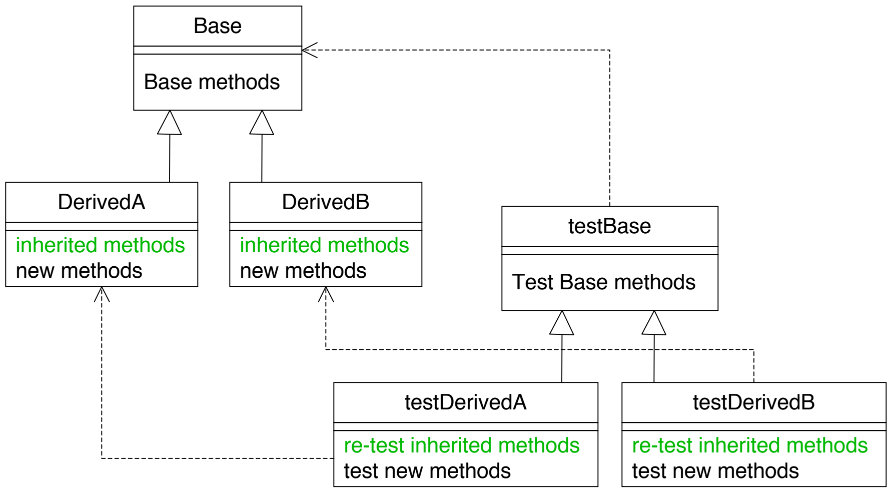
Inheritance Coverage

Source: *Adapted from IPL 1999*


Base
DerivedA
Inherited methods
New methods
DerivedB
Inherited methods
New methods
Coverage achieved by testing DerivedA
Coverage achieved by testing DerivedB
Inherited methods not exercised
Misleading coverage reported by traditional structural coverage metrics

Subclassing the Test Cases

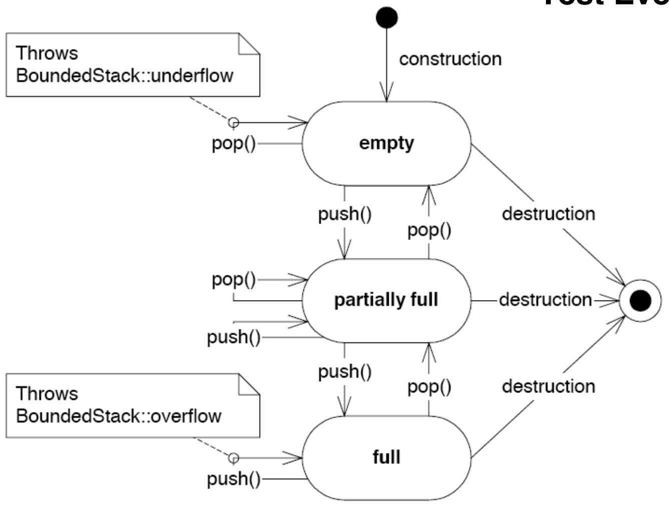
Source: *Adapted from IPL 1999*



State-based Context Coverage

Source: *Adapted from IPL 1999*

Test Every Transition!


Throws BoundedStack::underflow
construction
pop()
empty
push()
pop()
destruction
pop()
partially full
destruction
push()
push()
pop()
destruction
Throws BoundedStack::overflow
full
push()