

Testing

John Reekie
University of Technology, Sydney

Contributors
John Leaney, Lian Loke, Lizveth Robles,
University of Technology, Sydney

What is "testing"?

- ◆ Simulating conditions of operation in order to increase confidence that the software will perform "as expected" under actual use.
- ◆ There are many forms of testing. In this subject we will focus on two:
 - Unit testing. Test a single class or a small group of classes. Easily automated.
 - System testing. Test the whole system, typically using the user interface. Hard(er) to automate.

"You cannot test in quality"

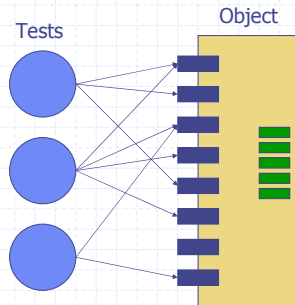
- ◆ Testing, by itself, doesn't improve software quality
- ◆ Instead, testing is part of an overall strategy to improve quality
 - A suitable process
 - Appropriate design and documentation
 - Formal inspections (design and code reviews)
 - A culture of getting things right the first time
 - And... testing.

Test-driven development

- ◆ Write the test case first. Then write the code that passes it.
 - An approach popularized by agile development methodologies
 - It ensures that test cases will exist for each key feature
 - Requires discipline and good tool support
 - Greatly improves confidence in new code
 - Requires automated testing support
- ◆ Related: write a test case to demonstrate a bug in existing code. Then the bug is fixed when the test case passes.

Unit testing

- ◆ A class has operations
- ◆ A test function constructs an object
- ◆ Then it makes a sequence of operation calls
- ◆ Then it checks that the object's state is what is expected (by calling other operations).
- ◆ A set of these is called a *test suite*.



Unit testing (2)

- ◆ The percentage of code in a class that is exercised by a test suite is called the *code coverage*.
 - Calculated automatically by suitable testing tools.
 - Expressed as a percentage figure: 40%, 95%, 100%
 - Even 100% code coverage is not exhaustive testing (why?)
- ◆ Code coverage figures are used as an *indication* of the extent to which the code in a class is being tested:
 - They provide a useful *guideline* to aim for.
 - They highlight classes that definitely **don't** have adequate test coverage

Designing unit tests (1)

- ◆ To start with, write test cases to test a single operation:
 1. Create new object
 2. Initialize to appropriate state
 3. Call the operation of interest
 4. Check the object state
- ◆ How do we decide what to do in item 3?
 - Answer: look at the operation's **contract**.
- ◆ As our test suite becomes more sophisticated, we can write a sequence of operations and then test the state

Designing unit tests (2)

- ◆ The pre-condition:
 - States the limits on input that we need to provide (we do not / should not provide data that violates the precondition)
 - Can also tell us something about the "boundary" conditions. Eg if $\text{pre } x >= 0$, use $x = 0$, $x = 1$, and $x = 600000000$
- ◆ The post-condition:
 - Tells us what we should check for. That is, for any input that satisfies the pre-conditions (as we provide in our test cases), check that the output meets the post-conditions.
- ◆ Notes on defensive programming
 - Preconditions are simpler (typically, always true)
 - Postconditions are more complex (since we have to "deal" with every possible input)

Boundary conditions

- ◆ State or input values which are on the "edge" of decision-making choices
- ◆ Use of boundary conditions is often considered to be "white box" testing, since you must "look inside" the object to see what to test (but, as noted, sometimes you can also see them from pre-conditions and post-conditions)
- ◆ Suppose that a class has this method:

```
public int doThis(int x) {
    if (x > 10)
        statementA;
    else
        statementB;
```
- ◆ In this case, the input data to test the boundary conditions would be $x=10$, and $x=11$

A first-in, first-out, buffer

```
getCount() : integer
  pre: true
  post: returns the number of elements

put(x:Object)
  pre: x is not null
  post: count = count'+1
       x is at tail of list

getFront() : Object
  pre: none
  post: count = count'
       if count = 0 then returns null
       else returns first object in list

getAndRemoveFront() : Object
  pre: count > 0
  post: count = count' - 1
       returns object at head of list
```

Testing it...

1. Test empty buffer

1. Create new buffer => x
2. x.getCount() => 0
3. x.getFront() => null

2. With one element

1. Create new buffer => x
2. Add (non-null) element *e*
3. x.getCount() => 1
4. x.getFront() => *e*

Testing (2)

1. With two elements

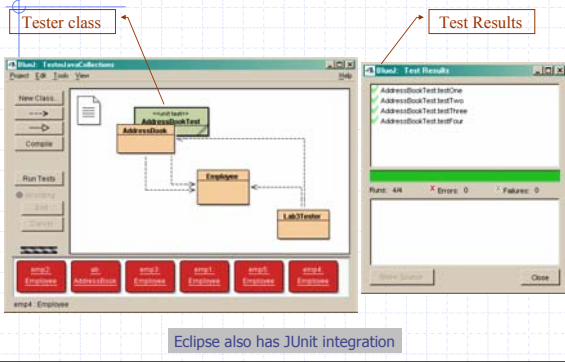
1. Create new buffer => x
2. Add (non-null) element *e1*
3. Add (non-null) element *e2*
4. x.getCount() => 2
5. x.getFront() => *e1*
6. x.getAndRemoveFront() => *e1*
7. x.getCount() => 1
8. x.ge

Yes, there is some redundancy here.
Is this "defensive testing"??

Testing with JUnit

- ◆ Thought experiment:
 - Fact 1. Tests are hard work to write.
 - Fact 2. Code changes.
 - Conclusion. You must write tests with the aid of a tool that helps to automate them.
- ◆ If tests are run regularly in order to check for introduced errors ("it worked last week"), then this is called *regression testing*
- ◆ JUnit is a simple framework written in Java to help you write tests and to run regression tests using them. There's nothing magic about it, but it is useful. It is an example of what is known as a *test harness*.
- ◆ There's nothing to stop you writing your own test harness. But, JUnit has the advantage that many IDEs (BlueJ and Eclipse, for example) have integration support for it

Test Driver (BlueJ) and JUnit



Automated testing

- ◆ Testing is tedious...
- ◆ So, automate it.
 - A *regression test* is run regularly against the code base, in order to ensure that test results don't "go backwards."
 - Typically, a subset of tests run nightly. The complete suite of tests run less frequently.
 - Breakages in the regression tests should be fixed immediately. Don't let the code deteriorate!!!
- ◆ Not everything can be automated, though...
 - GUI testing is harder to automate (but doable)
 - Tests that involve distributed systems are harder

System Testing

- ◆ System testing is the testing of the system as a "whole."
- ◆ It is concerned with demonstrating that the system **requirements** have been met successfully
- ◆ The system is treated as a "black box"
 - We don't **know** how it works inside, but we want to be sure that it works the way that we *expect* it to work
 - We don't **care** how it works inside, as long as...
- ◆ System tests are typically harder to automate than unit tests.
- ◆ In a real development organization, system tests may be the responsibility of a different group than the development team (who "can't see the forest for the trees").

Designing system tests

- ◆ The "obvious" place to look for system tests is in the use cases. Why? Because:
 - Use cases capture the user view of the system functionality (remember: end to end testing), but also
 - Are sufficiently detailed to provide the criteria that we use to decide whether or not a test **fails**
- ◆ System tests are **concrete** and **specific**
 - Need actual, specific data values to check against
 - They are prescriptive – do this, then that, check that x equals y ...

Asking the right questions

- ◆ Exhaustive testing is impossible
 - State space is very large
 - The intellectual effort to identify all possible test cases is gigantic
- ◆ Testing is therefore a matter of asking the right questions
 - Which classes are most likely to fail?
 - Which use cases will have the most severe consequences if they fail?
 - Which use cases have customers complained most about?
 - Which areas of the system have caused most reliability problems in the past?

Issues with testing

- ◆ Test cases are code. More code = more bugs. (In other words, how do you know that the test cases themselves don't have bugs?)
- ◆ Code changes break tests. Programmers don't necessarily like to update test cases when making (legitimate) changes to existing code...
 - Is there such a thing as over-testing?
- ◆ Tests take time (and therefore money) to write. What is the best balance between reliability and cost? Or similarly, between new features and reliability?
 - No point having super-reliable software if no-one buys it.
 - No point having great features if you get sued for negligence.
- ◆ Testing methods are sometimes controversial
- ◆ Not *finding* bugs doesn't mean that there *are* no bugs...

Alternatives to testing

- ◆ Code reviews
 - A structured form of "walkthrough" in which program code is examined a line at a time. Can find *potential* bugs that testing cannot.
 - Effect is weakened if code is subsequently changed.
- ◆ Program analysis
 - Sophisticated program analysis and visualization tools can find bugs that might otherwise go undetected. While many are "style" issues, these are often potential sources of error injection during later maintenance.
