

# Systematic Testing

# Unit testing

Also called *component testing*

Testing in isolation all operations associated with an object

Setting and querying all attributes (data members) of an object

Exercising the object in all possible states

# System testing

Also called *integration testing*

Testing the interaction of components

For CSC207H assignments this will usually be the whole program

For larger systems this may involve sub-components of the system

# Regression testing

Maintain a set of tests

Both unit and system tests

Every time a change is made to the system, run the tests to make sure everything that used to work still does

When you add a new feature to the system, add new regression tests

JUnit and its cousins make it easy to automate this form of testing

# Testing terminology

**Test or test case:** An action carried out

Must know the context and the expected outcome

**Fixture:** The context of the test

E.g. a data structure with a set of values

In the simplest Java case, a single initialized object one of the methods of which is to be called.

**Possible results:**

pass: test produced the expected outcome

fail: test ran but produced an incorrect outcome

error: test failed to produce an answer: there is something wrong with the test itself

# Unit testing

## Unit testing follows a pattern

Lots of small, independent tests

Reporting passes, failures, and errors

Some optional shared setup and teardown (creating the fixture)

Aggregation (combine tests into test suites)

## Design principles:

When you see a pattern, build a framework

Write shared code once

Make it easy for people to do things the right way

# JUnit

## JUnit testing framework

Written by Erich Gamma in 1997

Now hosted at <http://www.junit.org>

Has become the unofficial standard for Java testing

Supported by many IDEs

Widely imitated: C++, Perl, Python, .NET all have versions

Once you know one, you can easily use others

# Example

```
import static org.junit.Assert.*;

import org.junit.Test;

public class TestAdd {

    @Test public void testPositive() {
        assertEquals(2 + 2, 4);
    }

    @Test public void testNegative() {
        assertEquals((-2) + (-2), (-4));
    }
}
```

# How to use it

JUnit uses reflection to look up methods

(Reflection: extracting characteristics of a class or method (etc.) at run time.)

Pass in the class (!) containing your tests

JUnit decides which methods to run based on their annotations

(An annotation is syntactically like a type.)

If a method is annotated with “@Test”, TestRunner will execute it

Example, from the command line:

```
java org.junit.runner.JUnitCore TestAdd
```

# Setup and teardown

There are three steps in running a test: setup, run, and teardown.

The setup phase is in a (single) method annotated with `@Before`.

If no such method, setup phase does nothing.

The teardown phase is in a single method annotated with `@After`.

In setup and teardown, you put test fixture creation and clean-up that is common to a number of tests

[Caution: be careful with statics.]

# Testing for exceptions

How do you test a case that is supposed to fail by throwing an exception?

@Test annotation takes an optional parameter.

```
@Test(expected=NumberFormatException.class)
public void testFromString() {
    int x = Integer.parseInt("m") + 3;
}
```

# Choosing test cases

## Test for success

- general cases

- well-formatted input

- boundary cases

## Test for failure

- invalid input

- will it throw the exceptions it is supposed to?

## Test for sanity

- if there is redundant information make sure it is maintained

- data structure consistency

# Design for testability

When you are writing code, think about what you need to test and how you can test it

- always write toString() methods

- write methods that do a single thing

- separate input, computation, and output when possible

- modularity, modularity, modularity

Choose Java member visibilities carefully

- public and protected are effectively public

- private is untestable

- "package private" can be very helpful

# Test-driven development

Write your tests first!

In the “real world” this is the ideal

Tests based on requirements rather than code

Tests determine the code you need to write

There is no code unless there is a test that requires it in order to pass

Promotes a “think before you act” approach

Aids in definition of requirements

Tangible evidence of progress (green light)