Inf2C tutorial SE3: Testing and Sequence Diagrams: NOTES

1 Testing

This will work best if they have prepared and done this at the computer, but the code is simple enough that it can be done on the board. You may want to skip quickly through the first parts and write the test class gradually on the board, discussing as you go.

Basic way to proceed with JUnit 4 using annotations Please reinforce, that for purposes of this course the students are expected to know how to write basic JUnit 4 tests, but we provide a JUnit 3 solution as well:

1. Decide which functionality is being tested
2. Create the test class.
3. Do any common setup, e.g. in our case creating a Fibonacci class to test
4. Define tests with the @Test annotation. They will use methods like `assertEquals` to check the results.

Everything should have very descriptive names, even if that makes them long: the aim is to make the tests self-documenting.

Test-driven development is where not only are tests written before the code, but also, the tests serve as the (most formal) specification. Someone coming new to the project will expect to read the tests in order to understand the system. This is a comparatively cheap way to document, since you have to write tests anyway; it's easy to keep the documentation up to date (because the tests will start failing if they get out of date!); etc. Test-first is useful (whether or not the tests are used as important documentation) because it ensures tests are actually written, because it helps to ensure effort is not wasted writing code whose specification is not really understood, because it helps to avoid bending the spec to what's most easily written, etc. etc.

Here's a JUnit 4 version, using the @Test annotation:

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

public class FibonacciTest {
        Fibonacci fib = new Fibonacci();

        @Test
        public void test3rdFibonacciIs2() {
                assertEquals(2,fib.calcFibonacci(3));
        }

        @Test
        public void test6thFibonacciIs8() {
                 assertEquals(8,fib.calcFibonacci(6));
        }

}
```

And here's a JUnit 3 version in case you want to show it. Here the test class must extend the junit-provided TestCase class:

```
import junit.framework.TestCase;

public class FibonacciTest extends TestCase {

        public Fibonacci fib = new Fibonacci();

        public static void main(String[] args) {
                junit.textui.TestRunner.run(FibonacciTest.class);
        }

        public FibonacciTest(String arg0) {
                super(arg0);
        }

        public void test3rdFibonacciIs2(){
                assertEquals(2,fib.calcFibonacci(3));
        }

        public void test6thFibonacciIs8(){
                assertEquals(8,fib.calcFibonacci(6));
        }
}
```

To write this in Eclipse you choose File, New, JUnit test, and follow your nose. To run it, look under the Run menu and follow your nose. Eclipse will put up a green bar if all is well. If any test fails, Eclipse will show a *Failure Trace* which can be used to make appropriate changes in the faulty implementation (or test!).

You might want to discuss:

- the fact that we test calcFibonacci not printFibonacci: the former has an easier interface to test since it's much easier to check that a returned result is correct than that a printed answer is. Similar problems occur in spades when you try to test GUIs, though specialist tools exist. In this case, little harm done since all printFibonacci will (presumably) do is call calcFibonacci and print the result.

- the fact that there are several very similar tests; that's OK, because tests inherently involve specific values and it's more readable to hard code them in than to write a testNthFibonacciIsM function or similar, at least I think so. They are very simple anyway!

- the choice of values - what's wrong/incomplete with my solution? It doesn't test 0,1,2 or negative integers... For more on this they should attend the 3rd year testing course, choice of test data is outside the scope of Inf2C-SE really but it's interesting. Never test boundary/weird cases at the expense of testing mainstream ones, though, you look very silly if the first time someone tries your software it fails on perfectly ordinary input!

- You could discuss whether calcFibonacci should be private. That would almost certainly have been a design mistake (it would be very odd indeed to write a Fibonacci class which prints out the result as its only interface) but it also raises an issue interesting for testing. How do you test private methods? The orthodox answer is "you can't, very easily, and that is good, because if it doesn't suffice to test the things in the accessible interface this is a sign that your interface is wrong". Of course you can do anything via Java reflection... If you or students are interested there are various articles on this on the web, e.g.

  `http://www.artima.com/suiterunner/private.html`

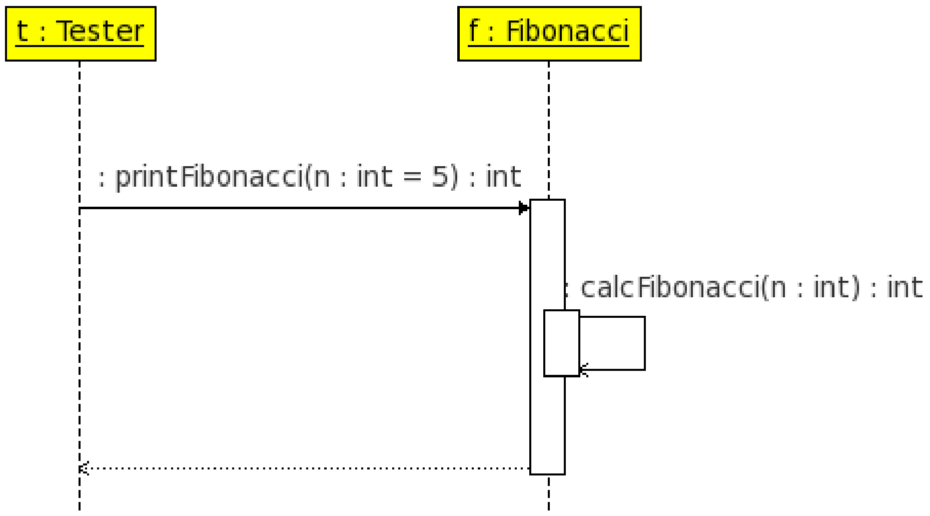
  – see also this discussion at StackOverflow (a very useful site btw):

  `http://stackoverflow.com/questions/34571/whats-the-best-way-of-unit-testing-private-met`

- For completeness here's my Fibonacci implementation. (It's the simplest thing that could possibly work, at least, that's my intention! If you discuss, you could discuss caching, and more generally, the tradeoff that sometimes arises between readable code and efficient execution.)

```
public class Fibonacci {
        public void printFibonacci (int n) {
                System.out.println("The "+n+"th Fibonacci is"+(calcFibonacci(n)));
        }
        public int calcFibonacci(int n) {
                if (n==0) return 0;
                if (n==1) return 1;
                return (calcFibonacci(n-1)+calcFibonacci(n-2));
        }
}
```

2 Sequence diagrams


t : Tester
f : Fibonacci
: printFibonacci(n : int = 5) : int
: calcFibonacci(n : int) : int

1. The sequence diagrams should be straightforward but some students seem to have a tendency to get the message arrows the wrong way round. Please check for

   - use of actor (*instance* of an actor here, e.g. labelled t:Tester)
   - they should not be underlining the lifeline labels – some may, as old versions of UML used to and some books still show this
   - correct arrow heads on the message arrows (filled in black triangles, for synchronous messages)
   - activation bars with correct extent: a common error is to have the activation bar begin above ("before") the arrival of the message that causes it. The message arrow head should point at the very top of the activation bar.
   - `calcFibonacci` is a self-message, so starts off a nested activation: check they understand this

2. This object works by recursion, and showing more than one level of recursion on a sequence diagram gets painful (and pointless). We often elide self-messages altogether, but a situation in which this can cause angst is if, say, an object is responding to message1 and sends message2 to itself, and then part of its reaction to message2 is to send message3 to a different object. If we elide message2, we really must be happy to elide message3 with it - showing message3 emanating from the activity bar caused by the invocation of message1 would be really misleading. The bottom line is that, typically for UML, there is more than one possible choice and which is best depends on the circumstances.

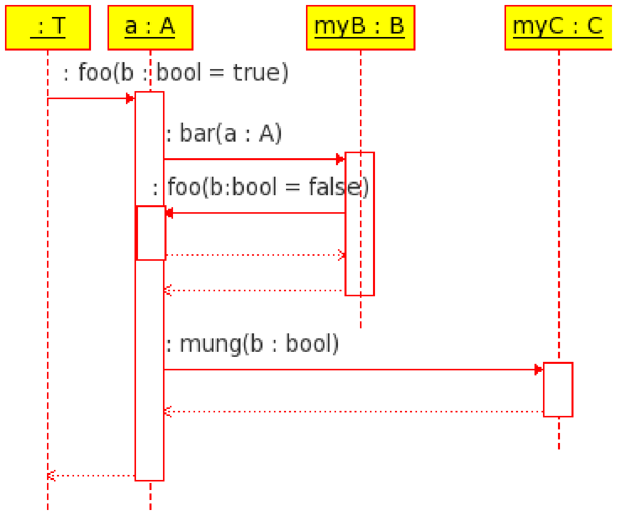
3. It's arguable that for example print(n) instead of printFibonacci(n) would have been better as an interface for printing the nth Fibonacci number. Assuming the client names their object of class Fibonacci appropriately - e.g. fibonacci! - then code that says `fibonacci.printFibonacci(n)` is redundant, compared with cleaner `fibonacci.print(n)`. I think this is mostly a matter of taste, but it's worth drawing attention to the general phenomenon - naming things "clearly" doesn't always mean giving them names as

long as possible, since repetition of information in typical pieces of code can also get annoying.

3 Exam question

Here are the marking guidlines:

1. Bookwork. 2 marks for the definition – a way to represent intER-object behaviour, i.e., the pattern of message-passing between objects. 1 mark for each of two reasonable circumstances, e.g. "to help with discussion of design of inter-object behaviour", "to document how a system realises a use case", "to record the correct protocol for interacting with an object".

2. 

   Application of knowledge. 1 mark for labelled actor, 2 for the other objects and lifelines. 1 for initial message foo(true). 1 each for messages bar, foo, mung, mung, with correct source and target and in the correct order. 1 for activation bars, 1 for return arrows.

3. Bookwork. 2 for definition. The relevant practices are collective ownership, coding standards and pair programming: in each case, 1 mark for the name, 1 for the explanation. If a less relevant XP practice is named, 1 mark for both the name and the explanation.

4. Problem-solving (the course discussed XP in enough detail to make this straightforward, but did not discuss this precise point). 1 mark each for any three reasonable points, such as, it's more in keeping with XP communication to draw a SD on a whiteboard where everyone can see it; simple design should need fewer anyway; SDs are normally developed before code, but XP doesn't have a protracted design phase where this might happen; tests serve a similar purpose.