Testing and Debugging
(See JFS chapters 20 and 21)

- Here we are concerned with *run-time errors*
  - *Testing* means attempting to find ways in which a program fails to work correctly
  - *Debugging* means identifying the causes of malfunctions and correcting them
- *All* real programs initially contain errors
  - The process of software development therefore *must* incorporate (as a matter of routine) testing and debugging phases
- First thought as regards testing:
  - Run it and see if it works!
- This is **inadequate**, except in the simplest of cases

Problems with testing:
"Run it and see if it works"

- Point 1:
  - "Run it" - what does this mean?
  - Maybe it will work differently with different inputs.
  - How do we select the inputs?
- Point 2:
  - "See if it works" - what does this mean?
  - We can't say whether it works unless we know what it is *supposed* to do
- If the program crashes, we can see that there is a problem, but if it simply does the wrong thing, then
  - The problem may be hard to spot,
  - It may not be obvious that it is wrong

An important fact

- "*Program testing can be used to show the presence of bugs, but never to show their absence!*" (E W Dijkstra)
- A useful idea: A successful test is **one that detects an error**!
  - Professionally, testing may be carried out by special testing teams
  - Their aim is to "break" the program!

Specifications

- For serious testing, it is necessary to have a *specification* to refer to
  - This means a description of what the program is supposed to do
- For simple programs, it may exist only in our heads, but for large-scale commercial programs it is essential to have a comprehensive written specification
- The question therefore is not "Does it work?", but "Does it do what the specification says it should?"
  - This is called "verification"
- A slightly different concept is "validation"
  - Making sure that the specification (and so the program that we build) is really what the client wants!
- We will deal with *verification*

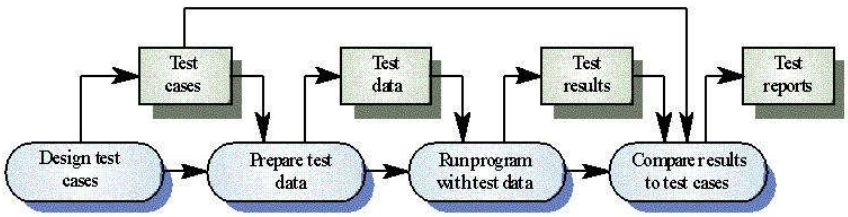
Exhaustive Testing

- Point 1 is relevant: How do we choose the inputs for a test run?
- You may say: Well let's try out all possible inputs
  - This is called *exhaustive testing*
- For all but the simplest of programs, this is **not feasible**
  - There are just too many possibilities
  - Consider a program that does something based on two input integers…
    `int`s are roughly $-2*10^9$ to $+2*10^9$
    So number of tests is $4*10^9 * 4*10^9 = 16*10^{18}$
  - At one test per microsecond: $16*10^{12}$ seconds
  - This is roughly: ?? years
- So in practice test data must be selected - *designed*

Designing Test Data

- When we design test data, we may make choices based on
  - The functionality of the program (what it should do), and/or
  - The internal structure of the program (how it does it)
- The first of these is called *black-box testing*
- The second is called *white-box testing*
- Important point: For any test data that we design:
  - We must also state the *expected* result
  - From reference to the specification
  - The result of carrying out tests should be compared carefully with the *predicted result*

- So we have this useful picture of the testing process:



(from Software Engineering, Ian Sommerville, publ Addison Wesley)

Designing tests: Black-Box Testing

- We should identify different kinds of typical inputs, corresponding to different possible outcomes
  - E.g. if the task is to enter one's age, and be told whether one can vote, then (say) 10 is one typical input, and 30 is another
- For some programs there may also be *special cases* - particular inputs which should have particular consequences.
- There is also the idea of *boundary value*
  - In the voting example, it would be a good idea to try the inputs 17, 18 and 19
- Testing should also cover inputs which are *outside* the expected ranges
  - Well-designed programs should be able to cope

Designing tests: White-Box Testing

- The principle here is:
  - Try to design inputs so that every statement in the program gets executed during at least one test run
- This is quite hard to describe in general, because of the variety of program structures. Simple example:
- Suppose that `a` and `b` are numbers which are input (or are computed using values which are input). The program may contain

```
if (a < b)
  { section 1 }
else
  { section 2 }
```

- In testing, we should make sure to run tests in which `a < b` is true, and tests in which `a < b` is false

Unit Testing and Integration Testing

- Programs have structure:
  - We have already seen that classes are subdivided into methods
  - Soon we will see that object-oriented programs are subdivided into classes
- It is inadvisable to wait until the whole program is written before engaging in any testing
  - For your assignments we have advised the approach of incremental development
  - A partially complete program may still be run, to test its partial functionality, but …
- It would be useful to have a way of testing individual methods or classes *in isolation*. This is *unit testing*
- By contrast, *integration testing* is the systematic process of putting the methods and classes together, in groups, and testing, until eventually the program is an integrated whole

Unit Testing (not in JFS)

- Here is part of a class, which might be part of a larger program:

```
public class SumUp extends ... {
   ...
   public int sum( int start, int end ) {
      int total = 0;
      for (int count = start; count <= end;
                                    count++) {
         total = total + count;
      }
      return total;
   }
   ...
}
```

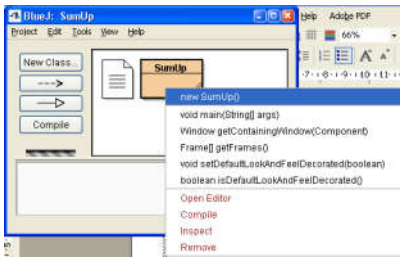
- Method `sum` adds up a range of numbers given by its parameters
- It is also `public` – could be made `private` later

- For testing purposes, we could adapt class `SumUp`'s main method:
  - Removing its GUI aspects
  - Adding method calls to test `sum`, for example:

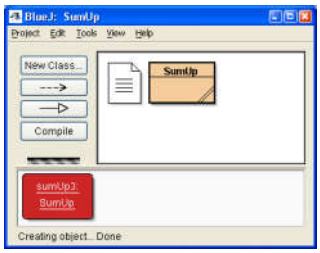
```
public static void main(String[] args) {
  SumUp test = new SumUp();
  // Typical: 2-5: result should be 14
  System.out.println("From 2 to 5: "
                          + test.sum(2, 5));
  // Special: 5-5: result should be 5
  System.out.println("From 5 to 5: "
                          + test.sum(5, 5));
  // Abnormal: 5-2: result should be 0
  System.out.println("From 5 to 2: "
                          + test.sum(5, 2));
  // etc
}
```

A test "*driver*"

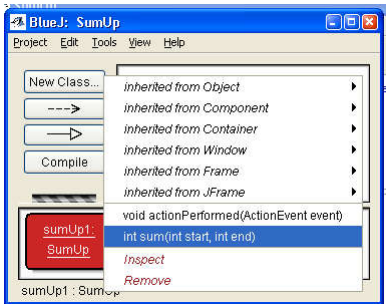
- BlueJ offers an elegant alternative:
  - We can carry out precisely the same process but without writing any code or modifying the program!
  - First: Right-click on **SumUp** in the main window and choose **new SumUp()**



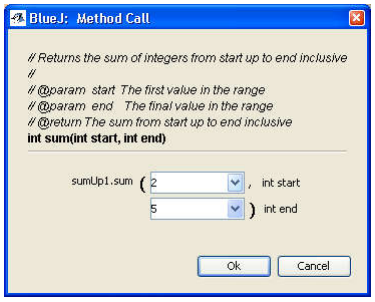
  - We get an instance on the "object workbench"



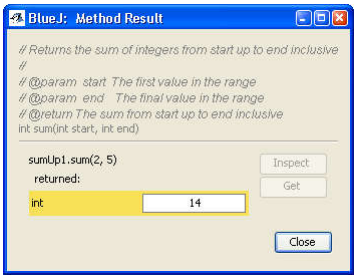
  - Second: Right-click on the object on the workbench and select the method **sum**



  - A method call dialogue appears into which we enter the test parameters and click OK (Note: BlueJ shows us the doc comments attached to the method)



- The method is called, and a method result box is displayed showing the returned result



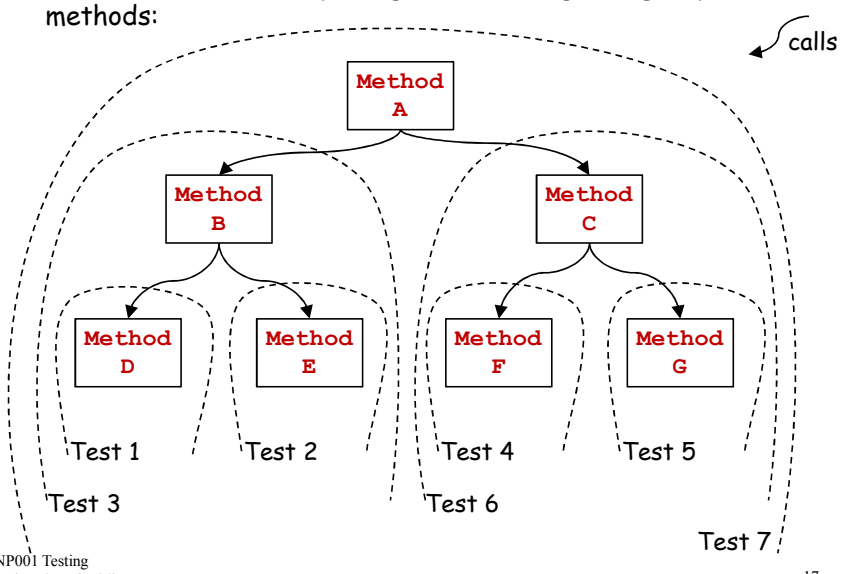
- We then check that the result is as expected

- This procedure can also be used for `void` methods
  - But we then need some way to "see" or otherwise check the results – so a bit more awkward
- We could carry out this process as many times as we need to test the methods in a class

Integration Testing

- An object-oriented program will consist of a number of methods (or classes). There will be dependencies among them:
  - In one method (or class) A we may call methods (or use objects of other classes), say B and C
- In testing, therefore, it is sensible to test methods (classes) B and C *before testing method (or class) A*
  - Only once methods (classes) B and C are trusted will it be sensible to test method (class) A
- First we unit test B and C, and then we apply the same technique to A, which will automatically use B and C
- Thus the whole program can be tested *incrementally*, in a *"bottom-up"* fashion
  - From components to assemblies

- Incremental bottom-up integration testing of a group of methods:



Debugging

- The procedures above may result in bugs being *detected*
  - But *finding* and *fixing* the bugs is another matter
- The simplest thing to do is to insert *diagnostic output statements* into our program, like

```
System.out.println("xCoord = " + xCoord);
```

  at strategic places, e.g.
  - before and after method calls,
  - inside loops
  - inside `if` … `else` … statements
  - before & after calculations
  - Before output actions

- Some IDEs (including BlueJ) incorporate *debuggers*.
  - The Java Development Kit provides one, but it works with a command-line interface (DOS window)
- A debugger may provide facilities such as *breakpoints* and *single-stepping*
- We may set *breakpoints* in a program:
  - The debugger will launch then run the program until a breakpoint is reached…
  - The debugger takes control and the program is paused
  - We can inspect the values of relevant variables at that point
  - The debugger then allows the program to continue to the next breakpoint…

- Instead of continuing to the next breakpoint, we may *single-step:*
  - Single-stepping simply means running the program one statement at a time
  - After each step we can inspect variables
  - The debugger may allow the option of executing particular methods without showing every step: "stepping over" the method call rather than "stepping into"
- Usually breakpoints can be added and removed whenever the program is paused
- Breakpoints and single-stepping are used in conjunction
- Practicals will include practice with BlueJ's debugger
  - And it will be valuable during assignments!

End of section