Testing and Debugging
An overview

- The need for testing, and the limits of testing
- Test case design
- General debugging techniques
- Interactive debugging

Testing and Debugging

- Here we are concerned with *run-time errors*
  - *Testing* means attempting to find whether there are any ways in which a program fails to work correctly
  - *Debugging* means identifying the causes of detected 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

Point 1: Exhaustive Testing

- 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…

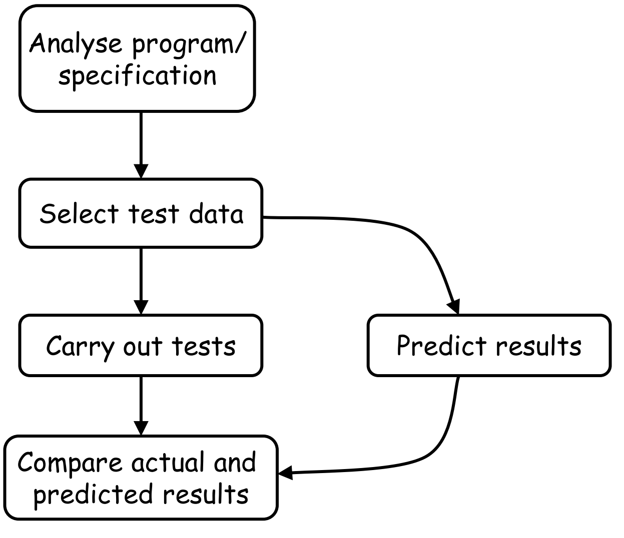
    `ints` are roughly $-2*10^9$ to $+2*10^9$

    So the 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*
- "*Program testing can be used to show the presence of bugs, but never to show their absence!*" (E W Dijkstra)

Point 2: Specifications

- For serious testing, it is necessary to have a *specification* to refer to
  - This means a precise 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
  - possibly mathematical, diagrammatic, ...
- 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"/"acceptance testing"*
  - Making sure that the specification (and so the program that we build) is really what the client wants!
- We will deal with *verification*

The testing process

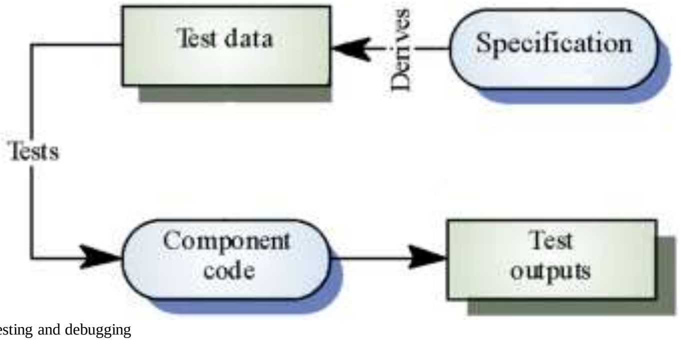


Designing Test Data

- 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*
- A useful idea: **A successful test is one which detects an error**!
  - Professionally, testing may be carried out by special testing teams
  - Their job is to "break" the program!
  - The idea of testing is to "put the program through its paces"

Black-box testing

- An approach to testing where the program, or a component, is considered as a 'black-box'
- The program test cases are based on the *system specification*
- Test planning can begin early in the software process
- The code is run with the test data – and actual outputs are compare with predicted outputs

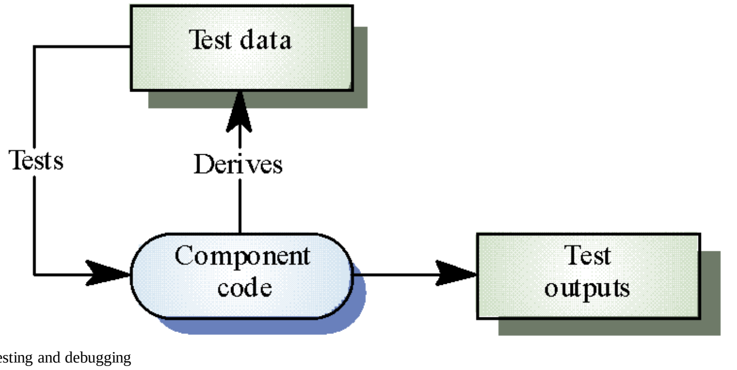


Black-Box testing (cont)

- We should identify *different kinds of typical inputs*, corresponding to *different possible outcomes*
  - *"Equivalence partitions"*
  - E.g. if the task is to enter one's age, and be told whether one can vote (from 18), then (say) 10 is one typical input, and 30 is another
- 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
- For some programs there may also be *special cases* - particular inputs which should have particular consequences
- Testing should also cover inputs which are *outside* the expected ranges
  - Well-designed programs should be able to cope

White box testing

- Sometimes called *white-box or glass-box testing*
- Test cases are derived according to *program structure*
- The objective is to *exercise all program statements*
  - Predicted outputs are determined for that test data
- The code is run with the test data – and actual outputs are compare with predicted outputs



White-Box testing (cont)

- 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
  - And probably when **a < b** is "only just" true and false

Debugging

- The procedures above may result in bugs being *detected*
  - But *finding* and *fixing* the bugs is another matter
- If the cause is *not* "obvious", then we usually need to investigate and monitor the internal workings of the code in question. Typically:
  - What path does execution take?
  - What values are computed and held in variables?
- The simplest thing to do is to insert *diagnostic statements:*

```
System.out.println("paintScreen entered");
System.out.println("x coord = " + x);
```

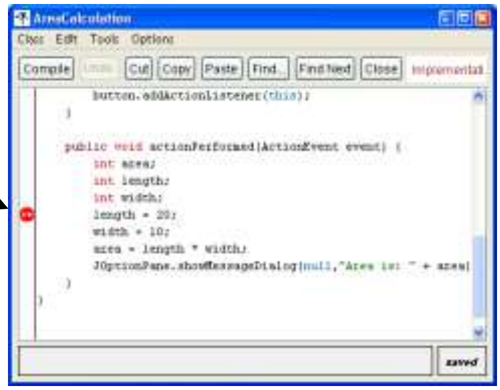
  at strategic places, e.g.
  - At the start and end of method bodies
  - Before and after method calls,
  - Inside loop bodies, **if** … **else** … statement branches

- The use of simple diagnostic statements to trace execution and to monitor variables is a universally applicable technique
  - All programming languages will have some equivalent facility
- But it has disadvantages too:
  - The diagnostic output can be copious and hard to analyze
  - The program code must be edited to add and remove the diagnostic statements – giving more opportunities for introducing bugs!
- An important alternative is that some IDEs incorporate *interactive debuggers*
  - The Sun Java Development Kit provides one, but it works with a command-line interface (DOS window)
  - BlueJ has a useful, straightforward, point-and-click interactive debugger

Interactive debugging – typical facilities

- Setting "breakpoints" to automatically pause the program
  - The debugger will run the program until a breakpoint is reached, then stop to allow us to see the values of relevant variables at that stage, then continue to the next breakpoint …
- Inspecting the values of variables
  - And possibly *modifying* the values of variables
- "Single stepping" from breakpoints
  - We can run the program *one statement at a time*
  - Inspecting detailed changes in variables' values
- "Watches" : automatic monitoring of selected variables
- Evaluating expressions using current variables
- A quick look at BlueJ... (a practical includes practice with BlueJ's debugger)

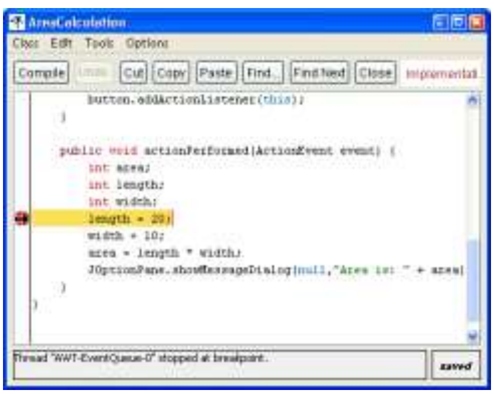
To set a breakpoint:
Before running the program, click in the left margin next to the point of interest

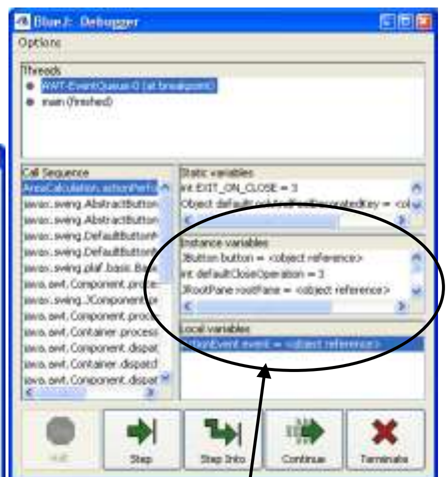


- More than one breakpoint can be set
- Click again to remove

Then run the program in the usual way…

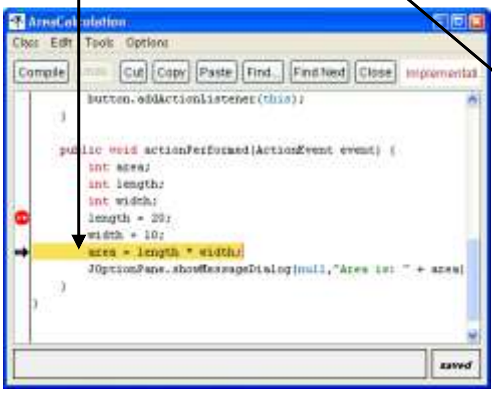
The program pauses when the breakpoint is reached – just before executing the statement, and the debugger window is displayed

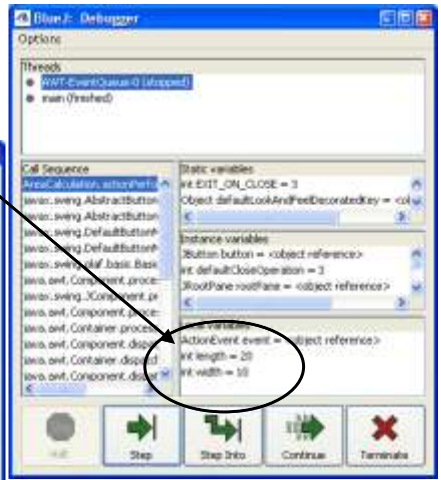




The most useful sections: variable values

After clicking Step twice, we see where execution has reached and the values assigned to variables





Click Step again, or Continue to run until the next breakpoint, or Terminate

End of section