List of Slides

17 Input test cases
18 Continued: input test cases
19 Output test cases
20 Test data
21 Combinations of test cases
22 Structural testing
23 Which approach is best?
24 Static verification
25 **Section 2:** Debugging
26 Debugging
27 Error location
28 Examining test results
29 Example: a change calculator
30 Examining the code
31 Some examples of common errors
32 Example of variable scoping problem
33 Examining the value of variables
34 Error elimination

Topic 10

Testing and Debugging

Objectives

- To explore the important topics of testing and debugging.
- Test data is systematically designed.
- Debugging is a lateral thinking process.

Section 1

Testing

Testing

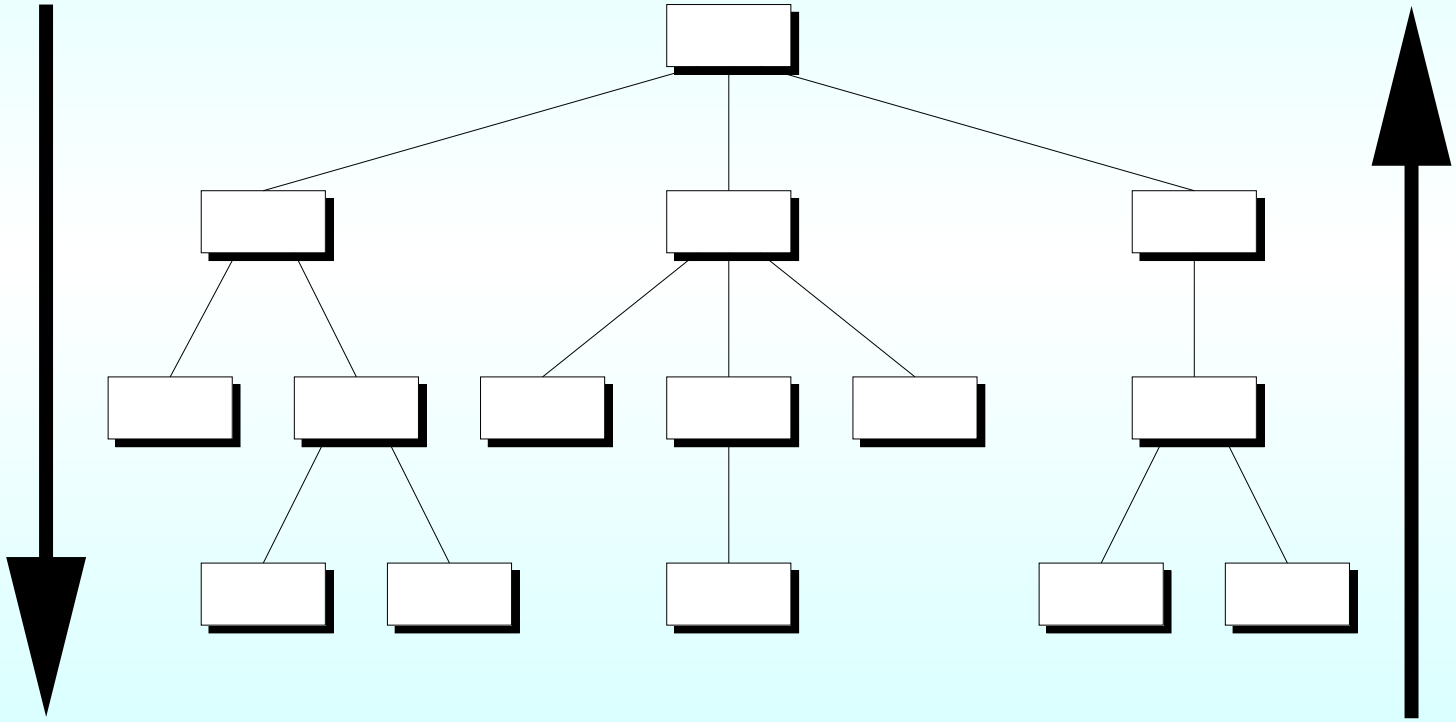
- Testing is executing an implementation with ‘dummy’ data.
- Testing can *never* show that a program is correct.
- A **successful test** is one which shows an error.
- Testing is destructive by nature: be *Devilish*.
- It has a tendancy to fail because nobody wants to destroy their creation.
- Perhaps programs are best tested by somebody other than the programmer?
- On the the other hand, detailed knowledge of the implementation might help.

Complex Programs and Systems

- Real world programs, and systems, are large.
- It is not possible to test them in one go.
- A **system** has many packages.
- A **package** has many classes.
- A **class** has many methods.
- How do we test them?
  - We test the *separate* parts.
  - We test the **integration** of parts.

Bottom up versus top down

- Components (e.g. methods, classes, packages) of a system depend on others, which depend on yet more, etc..
- These often, although not always, form a tree shape.



- Should we do **top down testing**, or **bottom up testing**?

Top down testing

- In top down testing we test each component before we test those it relies on.
- We write **stub**s for the lower level components.
  - Maybe they are interactive with the tester, and the tester supplies the answers.
  - Maybe they return arbitrary results.
- Top down testing allows testing to be simultaneous with top down development.
- Design errors might be found early on.

Bottom up testing

- In bottom up testing we test each component before we test any that rely on it.
- This means we do not have to write stubbs.
- But we cannot test incrementally with top-down development.

Tests are serious and organised

- Tests are not aribitrary!
- They are not based on ad-hoc input data conjured up as soon as the program first compiles.
- Ideally much of our test data is *designed* before, or during, program design.
- We distinguish two aspects of testing.
  - **test case**s are the features we want to test.
  - **test data** is used to test them.

Black box versus white box

- **black box testing** is when we look *only* at the specification of the behaviour for the component in order to construct test cases and test data.
  - Does this suit testing by someone other than the implementor?
- **white box testing** (or **glass box testing**) is when we *also* look at the implementation of the component when we construct test data.
  - Does this suit testing by the implementor?

An example: searching a list

- Suppose we want to test a method that searches for a given integer in a randomly ordered array of integers.
- It is supposed to return the position of the number in the list, if it is present, or -1 if it is not present.
- How many tests should we have?

Reasonable test cases

- Here is a reasonable set of test cases.

No.	List structure	Search item found
1	Empty list	Not found
2	One element in list	Found
3	One element in list	Not found
4	Many elements in list	Not found
5	Many elements in list	First item
6	Many elements in list	Last item
7	Many elements in list	‘In middle’

Test data

- Here is a corresponding set of test data.

No.	List	Search	Expect
1	{}	0	-1
2	{17}	17	0
3	{17}	10	-1
4	{17,23,28}	14	-1
5	{17,23,28}	17	0
6	{17,23,28}	28	2
7	{17,23,28}	23	1

Equivalence partitioning

- **equivalence partitioning** is a technique for constructing test data from the specification (only) of the component.
- We partition the set of all possible input data into classifications with common properties.
- We do the same for the output data.
- We identify these classifications as test cases.
- We choose representative values from the test cases to be used as test data.
  - We pay particular attention to edge values as these are often overlooked and so are most likely to lead to successful tests.

An example: parsing an integer

- Suppose we want to test a method that claims to convert an input string of characters into an integer which it returns.
- The input is guaranteed to be in the following form.

  ```
  { ' ' } [ '-' ] digit { digit }
  ```

  That is, zero or more leading spaces, a possible minus sign, a digit then zero or more digits.
- There is no need to test other formats of data: the implementor of the method assumed the data format fits the above. (It is a **precondition**.)
- Suppose the method will return a number which is within plus and minus 32767, inclusive. If the representation is for a number less than -32767 then -32767 will be returned. If the representation is for a number greather than 32767 then 32767 will be returned.

How many tests?

- Make an estimate of how many tests there will be for this example and write it here. [ ]
- Later, write here the actual number of tests. [ ]

Input test cases

No.	Description
C1	No leading space
C2	One leading space
C3	Many leading spaces
C4	Minus sign
C5	No minus sign
C6	One digit
C7	Many digits
C8	No leading zeros
C9	One leading zero
C10	Many leading zeros

Continued: input test cases

No.	Description
C11	$Representation < -32767$
C12	$Representation > 32767$
C13	$-32767 <= Representation <= 32767$

Output test cases

No.	Description
C14	Result < 0
C15	Result $= 0$
C16	Result > 0

Test data

Input	Expected result	Cases tested
”-1”	-1	1,4,6,8,13,14
” -1”	-1	2,4,6,8,13,14
” -1”	-1	3,4,6,8,13,14
. . .		
” -00032768”	-32767	3,4,7,10,11,14
. . .		

- How many combinations of test cases are there, i.e. how manys tests will there be?

Combinations of test cases

- Multiplying the sizes of the ‘groups’ of input tests cases, gives us $3 \times 2 \times 2 \times 3 \times 3 = 108$
- However, many of these include contradictions, so there is actually ‘only’ 48 tests to be done!
- These 48 include 5 output tests for: $-32767, -1, 0, 1, 32767$

Structural testing

- **structural testing** is a technique for constructing test data by looking at the implemementation code.
- Ideally we choose data so that every combination of every path through the code is tried.
- In practise we ensure every part of the code is tried at least once.
- There is a danger of testing the *compiler* rather then the program, if we let the testing be driven totally by code rather than the specification as well.
- It is a good technique for checking against many simple run-time errors, since we make sure every part of the code is run at least once.

Which approach is best?

- A combination of both is best.
- Perhaps an ideal combination is:
  - Equivalence partitioning initially, followed by
  - Adding extra cases from structural testing to ensure every part of code is run at least once.

Static verification

- Formal, or at least semi-formal, **reasoning** gives evidence of *correctness*, whereas testing can at best give evidence of incorrectness.
- Symbolic (dry) execution can cover *all* cases.
  - A test is a real run that covers one set of data.
  - So an infinite number of tests would be needed to cover all data.
  - A symbolic execution can cover an infinite number of sets of data.
  - So a finite number (e.g. one) of symbolic executions is needed to cover all data.
- Safety critical systems will not be merely tested.

Section 2

Debugging

Debugging

- Debugging consists of two parts.
  - Error location.
    - This is the hard part.
  - Error elimination.
    - This is the easy part.
    - Or, we have to redesign our implementation!
      - Perhaps we also have to find a temporary **workaround**.

Error location

- Error location appeals directly to lateral thinking processes.
  - We have to examine the information available.
  - We have to make connections between the pieces of information.
  - We have to form a hypothesis and test it.
  - We have to ask questions to get new information.
- To get new information, questions are asked in three ways:
  - By running the program with test data and examining the results.
  - By examining the source code without running the program.
  - By running the program and examining the values of variables.

Examining test results

- In this approach we attempt to diagnose the fault by studying the symptoms.
- Examine the output data and look for patterns.
- Form a hypothesis about the symptoms.
- Attempt to verify the hypothesis by testing with new data.

Example: a change calculator

- A program is given the price of a purchase and an amount tendered, both in pence. It then prints a receipt showing this, together with the change, all in pounds and pence.
- When I enter a price of 123 pence, and an amount tendered of 200 pence, I get:

```
 Purchase Price:  UKP 1.0
Amount Tendered:  UKP 2.0

 Your Change is:  UKP 1.0
```

- What is the probable fault? Did you need to see the code?

Examining the code

- Ideally we look at the source code only after we have formed a hypothesis from the output behaviour.
- Search the source code for the cause of hypothesised fault – be cynical.
- Look for variables that are given values which are inconsistent with their meaning (c/f "How to write algorithms" lecture from CS1081).
- Consciously bear in mind common errors: our sub-conscious mind will often 'correct' them without us noticing.

Some examples of common errors

- Implicit type casting problems, especially with numbers and the division operator.
- Starting a counter from 1 instead of 0, or vice versa.
- Failing to update an array index.
- Updating an array index before inserting data instead of after, or vice versa.
- Failing to initialise a variable value (so something works first time, but not after).
- Variable scoping problems: a classic!

Example of variable scoping problem

```
public class ScopeProblem
{ public int value; // This holds an important value
  public void setValue()
  { int value;
    ...
    value = ...;
    ...
  } // setValue
  public void useValue() // WHY DOES THIS METHOD NOT WORK?
  { ...
    if (value ...)
      ...
    ...
  } // useValue
} // class ScopeProblem
```

Examining the value of variables

- Ideally the previous examinations will have revealed the bug.
- If not, we resort to collecting more information from the running program, and then treating it as extra test result information.
  - By inserting `System.out.println()` statements in the code to report carefully thought about interesting values.
  - Or by using an on-line debugging tool.
- We mark all debugging statements with an easy to find comment so they can be removed later.
- On-line debugging tools can encourage a hacking approach. They make it so easy to ask the questions that we can get too lazy to think:
  - What are good questions to ask?
  - What do the answers mean?

Error elimination

- Most software manufacturers admit that every 1000 bugs, once corrected, create another 100 bugs! These numbers are approximate, sometimes they are much worse than that.
- Do not use **anti-bug**s to kill existing bugs. *Never* correct a bug in an early stage of the program by making an 'adjustment' in a later part.
- For example, suppose we find out that some numeric answer is always 1 too low. It is tempting to just add 1 to the result before returning it.
- There is no such thing as anti-bugs, just bugs. A bug used to offset another might *appear* to work in the short term, but really we now have 2 bugs waiting to spring out when some obscure set of circumstances happens, or at some time in the future when the software is altered.
- Always fix the original bug. Redesign if necessary.
- Always test again after corrections are made.

However...

- Sometimes it is appropriate to make a quick fix known as a **workaround**.
- These typically fix the *symptom* of the bug, rather than the cause. In that sense they are an **anti-bug**.
- It is appropriate when we need the software to work better now, rather than have to wait for proper debugging.

Example: reversing a string

- We have a program, `Reverse` which takes a command line argument and is supposed to print it out in reverse.
- Suppose you know it works by copying the string into an array, then looping one index variable upwards from 0, and another downwards from the highest index. At each stage it swaps over the items at the two indices. The loop stops when the first index gets half way though the array.

Reverse string test results

Spot the pattern in the following test results, and suggest the bug.

```
jtl-) letters="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
      for length in ‘seq 1 10‘; do data=${letters:0:$length};
      echo "$data -> ‘java Reverse $data‘"; done
A -> A
AB -> AB
ABC -> CBA
ABCD -> DBCA
ABCDE -> EDCBA
ABCDEF -> FECDBA
ABCDEFG -> GFEDCBA
ABCDEFGH -> HGFDECBA
ABCDEFGHI -> IHGFEDCBA
ABCDEFGHIJ -> JIHGEFDCBA
```

Also, what test has been omitted?

Did you need to see the code?

- Hopefully, you spotted the pattern and guessed what the bug might be, without needing to see the code.
- A person with *really* good debugging skills, that is, someone who is really good at lateral thinking, would not even have had to know what the algorithm for the reverse is, let alone see the code!

The code

```
private static String reverse(String input)
{
  char [] charArray = input.toCharArray();
  int leftIndex = 0;
  int rightIndex = charArray.length - 1;
  while (leftIndex <= charArray.length / 2)
  {
    char thatWasAtLeftIndex = charArray[leftIndex];
    charArray[leftIndex] = charArray[rightIndex];
    charArray[rightIndex] = thatWasAtLeftIndex;
    leftIndex++;
    rightIndex--;
  } // while
  return new String(charArray);
} // reverse
```