# Testing

# Overview

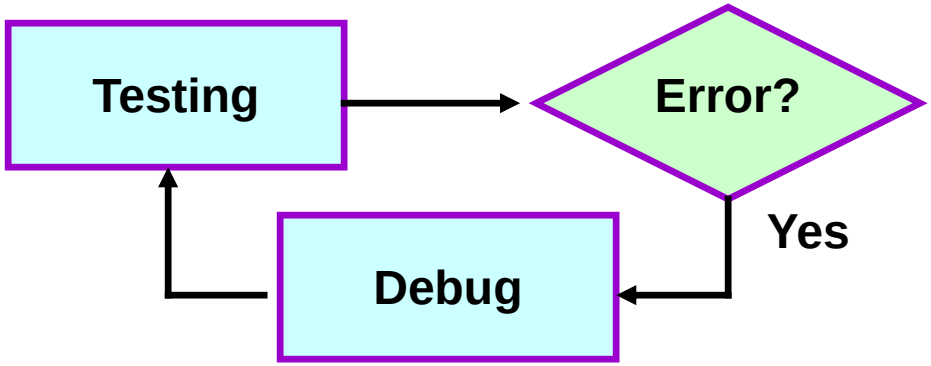
- Testing and debugging are important activities in software development.
- Techniques and tools are introduced.
- Material borrowed here heavily from Aaron Tan's presentation: http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0CCsQFjAC&url=http%3A%2F%2Fwww.comp.nus.edu.sg%2F~cs1101x%2Flect%2Ftesting_and_debugging.ppt&ei=qf0NVJG9KqGjigL8sIHoDw&usg=AFQjCNHOoLaP7m7-Q6tCgZJ23fyDLOVbyg&bvm=bv.74649129,d.cGE.

# Programming Errors

- Compilation / Syntax errors
  - Grammatically incorrect statement
  - Occur during the parsing of input code
    - Example: missing a semi-colon
  - Easiest type of errors to fix.
- Runtime errors
  - Occur at runtime.
    - Example: File not found
  - Java's exception mechanism can catch such errors.
- Logic / Semantic errors
  - Program runs but produces incorrect result.
    - Example: Division by zero
  - Hard to characterize, hence hardest to fix.
- Programming errors are also known as bugs
  - Origin: a moth in the Mark I computer.

# Testing and Debugging

- Testing
  - To determine if a code contains errors.
- Debugging
  - To locate the error and fix it.
- Documentation
  - To improve maintainability of the code.
  - Include sensible comments, good coding style and clear logic.



# Testing and Debugging

- Unit testing
  - Test of individual parts of an application – a single method, a single class, a group of classes, etc.
- Positive versus negative testing
  - Positive testing – testing of functionality that we expect to work.
  - Negative testing – testing cases we expect to fail, and handle these cases in some controlled way (example: catch handler for exception).
- Test automation
  - *Regression testing* – re-running tests that have previously been passed whenever a change is made to the code.
  - Write a *test rig* or a *test harness*.

# Testing and Debugging

- Modularization and interfaces
  - Problem is broken into sub-problems and each sub-problem is tackled separately – divide-and-conquer.
  - Such a process is called modularization.
  - The modules are possibly implemented by different programmers, hence the need for well-defined interfaces.
  - The signature of a method (its return type, name and parameter list) constitutes the interface. The body of the method (implementation) is hidden – abstraction.
  - Good documentation (example: comment to describe what the method does) aids in understanding.

| | |
|---|---|
| static double | **max(double a, double b)**<br>Returns the greater of two double values. |

# Testing and Debugging

- Manual walkthroughs
  - Pencil-and-paper.
  - Tracing the flow of control between classes and objects.
  - Verbal walkthroughs

# Testing and Debugging

- Print statements
  - Easy to add
  - Provide information:
    - Which methods have been called
    - The value of parameters
    - The order in which methods have been called
    - The values of local variables and fields at strategic points
  - Disadvantages
    - Not practical to add print statements in every method
    - Too many print statements lead to information overload
    - Removal of print statements tedious

# Testing and Debugging

- Debugger
  - Provides
    - Stepping (step and step-into)
    - Breakpoint
    - Tracking of every object's state

> Program testing can be used to show the presence of bugs, but never to show their absence. – Edgar Dijkstra

# Testing and Debugging

- Tips and techniques
  - Start off with a working algorithm
  - Incremental coding/test early
  - Simplify the problem
  - Explain the bug to someone else
  - Fix bugs as you find them
  - Recognize common bugs (such as using ‘=’ instead of ‘==’, using ‘==’ instead of equals( ), dereferencing null, etc.)
  - Recompile everything
  - Test boundaries
  - Test exceptional conditions
  - Take a break

# Black-box and White-box Testing

- White-box testing indicates that we can “see” or examine the code as we develop test cases
- Black-box testing indicates that we cannot examine the code as we devise test cases
  - Seeing the code can bias the test cases we create
  - Forces testers to use specification rather than the code
- Complementary techniques

# Testing Thoroughly

- For example, Richard can't spot the error in his code.

```
// To find the maximum among 3 integer
// values in variables num1, num2, num3.
int max = 0;
if (num1 > num2 && num1 > num3)
  max = num1;
if (num2 > num1 && num2 > num3)
  max = num2;
if (num3 > num1 && num3 > num2)
  max = num3;
```

- He tested it on many sets of data: <3,5,9>, <12,1,6>, <2,7,4>, etc. and the program works for all these data.
- But he didn't test it with duplicate values! Eg: <3,3,3>, <7,2,7>, etc.

# Testing Thoroughly

- Richard wrote another program.

```
// To find the maximum among 3 integer
// values in variables num1, num2, num3.
int max = 0;
if (num1 > max)
  max = num1;
if (num2 > max)
  max = num2;
if (num3 > max)
  max = num3;
```

- He was told that the program doesn't work but again he couldn't figure out why. He has tested it on many data sets, including duplicate values!
- Can you tell him what he missed out in his testing?
- Don't forget the special cases!

# Testing Boundaries

- It is important to test the boundary conditions.

```
final int CALENDAR_START = 1583;
// validate input
if ((year < CALENDAR_START) || (month < 1) ||  (month > 12))
{
      System.output.println("Bad request: " + year + " " + month);
}
```

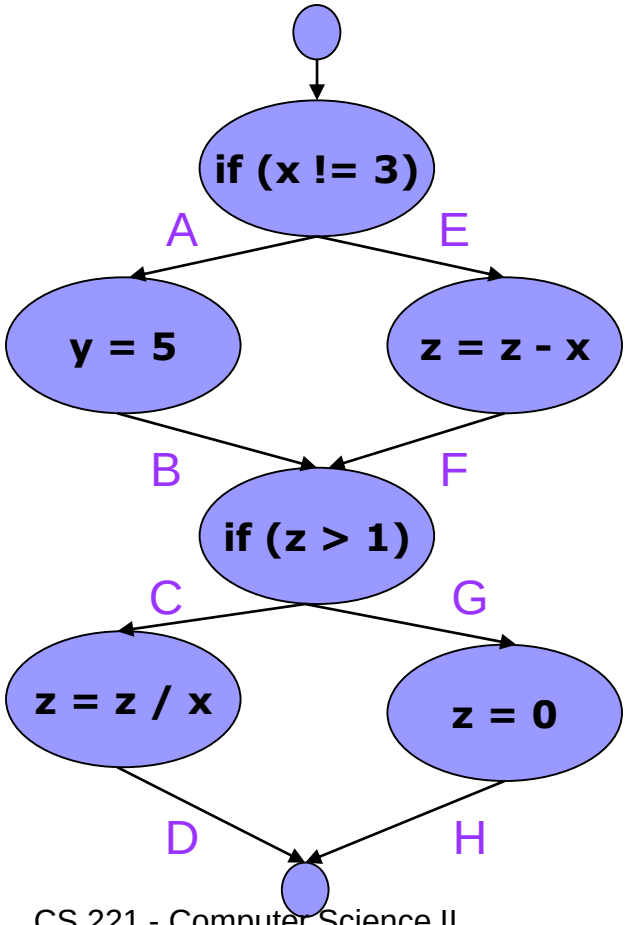
| Input Year | Input Month |
|---|---|
| 1582 | 2 |
| 1583 | 0 |
| 1583 | 13 |
| 1583 | 1 |
| 1583 | 12 |

# Path Testing

- Paths: different routes that your program can take
  - Design test data to check all paths
  - Example

```
if (x != 3) {
   y = 5;
}
else {
   z = z - x;
}

if (z > 1) {
   z = z / x;
}
else {
   z = 0;
}
```




<x=0, z=1>
Paths A, B, G, H.

<x=3, z=3>
Paths E, F, C, D.

# Integration and System Testing

- Integration testing is done as modules or components are assembled.
  - Attempts to ensure that pieces work together correctly
  - Test interfaces between modules
- System testing occurs when the whole system is put together

*This comes in when you start to write bigger programs.*

# Debugger

- Using the debugger
  - Stepping
  - Breakpoint
  - Inspecting variables
- Using Eclipse Debugger
  - http://www.vogella.com/tutorials/EclipseDebugging/article.html