Programming Foundations
FIT9131

Testing & Debugging

Week 7

Lecture outline

- **Software development and maintenance**
- **Program errors – syntax, execution, logic**
- **Testing**
  - **Unit testing (within BlueJ)**
  - **Designing and implementing a *Test Strategy***
    - ***needed for Assignment 2***
  - **Regression testing**
- **Debugging**

Stages in software development

(1) Design stage:

- **designing the solution**

(2) Implementation stage:

- **translating the design into code**

(3) Maintenance stage:

- **adding new features, fixing bugs, etc**

Design Stage

Design stage:

1. Understand the problem
2. Design a solution
3. Design test data for that solution

Implementation Stage

Implementation stage:

1. Translate these designs into code
2. Compile code
3. Run tests
4. Modify code as necessary until tests are satisfactory
5. File all documentation

Maintenance stage

Maintenance stage:

1. Read all documentation. Understand how the current system works (or why it doesn't work).
2. Figure out the required modification. Assess the impact of the modification upon the rest of the system.
3. Do all design and implementation steps for the change to be made.
4. Test the modified system, re-running all relevant previous tests. This is known as *regression testing*.
5. File all documentation.

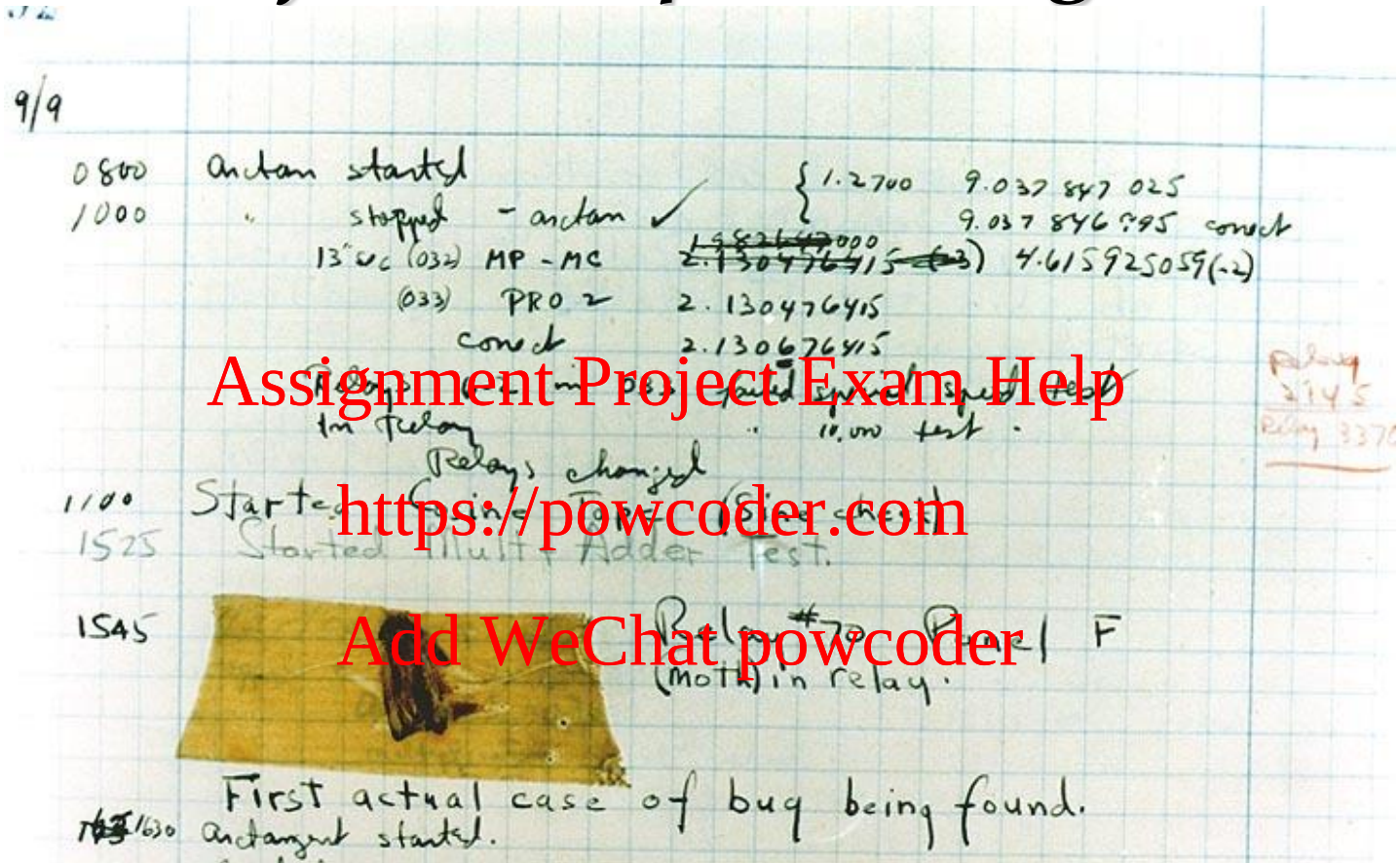
What errors can occur in programs?

There are three kinds of errors:

- *Syntax* errors
- *Execution* (or *run-time*) errors
- *Logic* errors

- An error that occurs during the execution of the program is also known as a *bug*.

The first computer "bug"



Testing of the Mark II Aiken Relay Calculator, September 1947

Syntax (compilation) errors

The compiler cannot translate the program, because the code does not follow the rules of the language used.

Some examples:

These errors are generally the easiest to fix.

- **Misspelling, eg:**

  `Retrun` for `return`

- **Incorrect phrasing of instruction, eg:**

  `if mark < 50`

- **Structural error, eg:**
  - **no semicolon at end of a statement, method defined inside another one, etc**
- **Incorrect number of arguments passed to a method.**

Execution (run-time) errors

The compiler has successfully translated the program, but the program is run, it fails due to some unexpected errors.

These errors can generally be avoided by careful programming .

***Some examples*:**

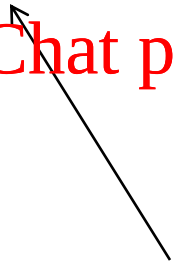
- **Divide-by-zero error.**
- **Attempting to access an array outside its boundaries.**

- **Request to write to a file when there is insufficient space on the disk.**

Logic errors

The program has been translated and run without any apparent problem, but its output/behaviour is not what was expected, due to some error in its coding logic.


```
private boolean validMark(int examMark)
{
    if (examMark < 0 && examMark > 100)
        return false;
    return true;
}
```

These errors are generally the hardest to fix.

Is there something strange here?

(Hint : try & read the boolean condition in English…)

Importance of testing

Some logical errors have no immediately obvious manifestation.

Commercial software is rarely (never?) error free, due to the size of the code.

We have to deal with errors …

Testing Failure Example : Mariner 1 disaster

Mariner 1

July 28, 1962 -- Mariner I space probe. A bug in the flight software for the Mariner 1 caused the rocket to divert from its intended path on launch. Mission control destroyed the rocket over the Atlantic Ocean.

The investigation into the accident discovered that a formula written on paper in pencil was improperly transcribed into computer code, causing the computer to miscalculate the rocket's trajectory.

Some other well-known software disasters ...

- Therac-25 (1985-1987)
- AT&T Phone System (1990)
- Y2K bug (2000)

Some interesting readings : http://en.wikipedia.org/wiki/List_of_software_bugs

Prevention vs. Detection
(Developer vs. Maintainer)

It is almost impossible to completely eliminate software bugs, especially for large-scale projects. However, we can :

- lessen the likelihood of errors.
  - Use software engineering techniques, like *encapsulation* to reduce the scope of variables – we will discuss this in a later lecture.

- improve the chances of detection.
  - Use software engineering practices, like *modularisation* and *documentation*.

Testing and debugging

These are crucial skills in programming.

- *Testing* searches for the presence of errors.
  - this allows programmers to remove/fix the errors.
- *Debugging* searches for the source of errors.
  - Note : the manifestation of an error may well occur some 'distance' from its source, making it harder to locate.

Unit testing

Each *unit* of an application may be tested. A "unit" could be a:

- method, class, module (package in Java).

Unit testing can (and should) be done during development. The benefits of early testing are:

- Finding and fixing errors early lowers development costs (e.g. programmer time).

- Enables a test suite to be built up. This can be used to test the unit repeatedly throughout development.

- Eg. How would you test your `Product` class for the assignment?

Testing fundamentals

Some points to consider:

- Understand what the unit should do – its *contract*.
- Look for *violations*.
- Use *positive* tests and *negative* tests.
- Consider *test pairwise*, e.g.
  - reading from an empty file.
  - adding to a data collection which is full.

There are typically many tests that need to be conducted to thoroughly test a program.

Testing strategically

We will first consider how to test a simple class. This technique can then be extended to test programs with multiple interacting classes.

When testing a class, every constructor and method in the class must be systematically invoked.

Testing strategically

BlueJ makes it easy to test a class:

- Objects of individual classes can be created
- Individual methods can be invoked
- The state of objects can be inspected

... but you need to document how you do the testing in a more formal manner, eg :

- which values did you use to perform the tests?
- what were the expected results?
- what were the actual results?

Test Strategy

A *Test Strategy* is a documentation which lists how you tested (and in some cases, how you will test) your class or program. *This document should be developed before the program's code is written*, and then updated throughout the coding process.

- *you will be expected to do this in Assignment 2*

Test Strategy

A *Test Strategy* typically contains 2 components :

1) A *Test Plan*
   - this is a *summary* of ALL the tests which will be eventually performed for the program.
2) The *Actual Tests*
   - These are the specifications/details of all the tests listed in the Test Plan.
   - Each test will contain the following 3 components :
     - *Test Data*
     - *Expected Results*
     - *Actual Results*

using ***actual values*** (eg. 5), not just general descriptions (eg. "**a number**")

√

χ

Example : Testing an Employee *class*

We will use a small class, `Employee`, to demonstrate how to develop and implement a test strategy.

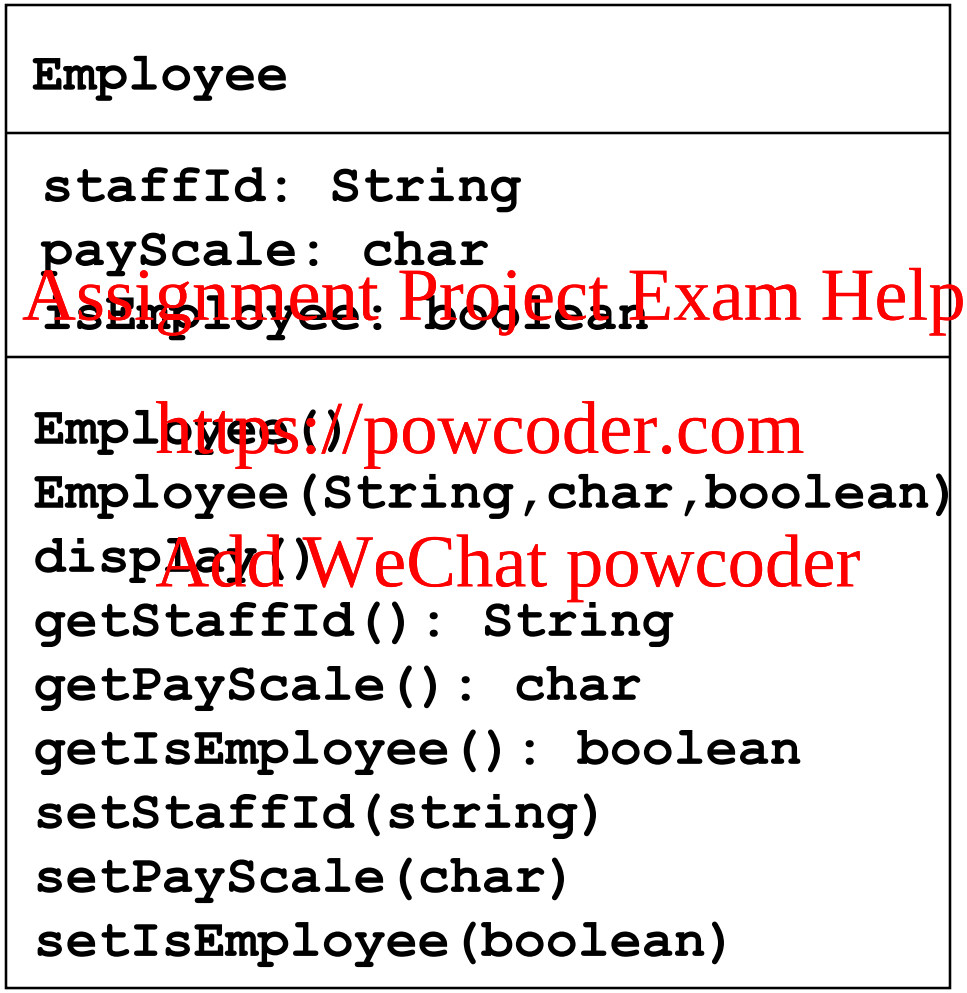
The `Employee` class has three attributes: *Staff ID*, *Pay Sccale* and *Employee Status*.

- the **Staff ID** is a string of 3 numeric characters.
- the **Pay Scale** is a character : 'A', 'B', 'C' or 'X'.

- the **Employee Status** is either *true* to indicate employed, or *false* to indicate not currently employed

There are two constructors, plus the usual accessors & mutators.

Class diagram for `Employee` *class*



(1) Test Plan for `Employee` class

Test Plan

1. Create an `Employee` object with the default constructor.

2. Create `Employee` objects with all non-default constructors.

3. Test all the `get` methods.
4. Test all the `set` methods.
5. Test the `display` method.

a summary of what are to be tested

(2) Actual Tests : Test 1

Test 1

Create an `Employee` object with the default constructor.

description of the test to be carried out

Test Data:

- **No input**

description of the test data to be used

description of the actual results/outputs

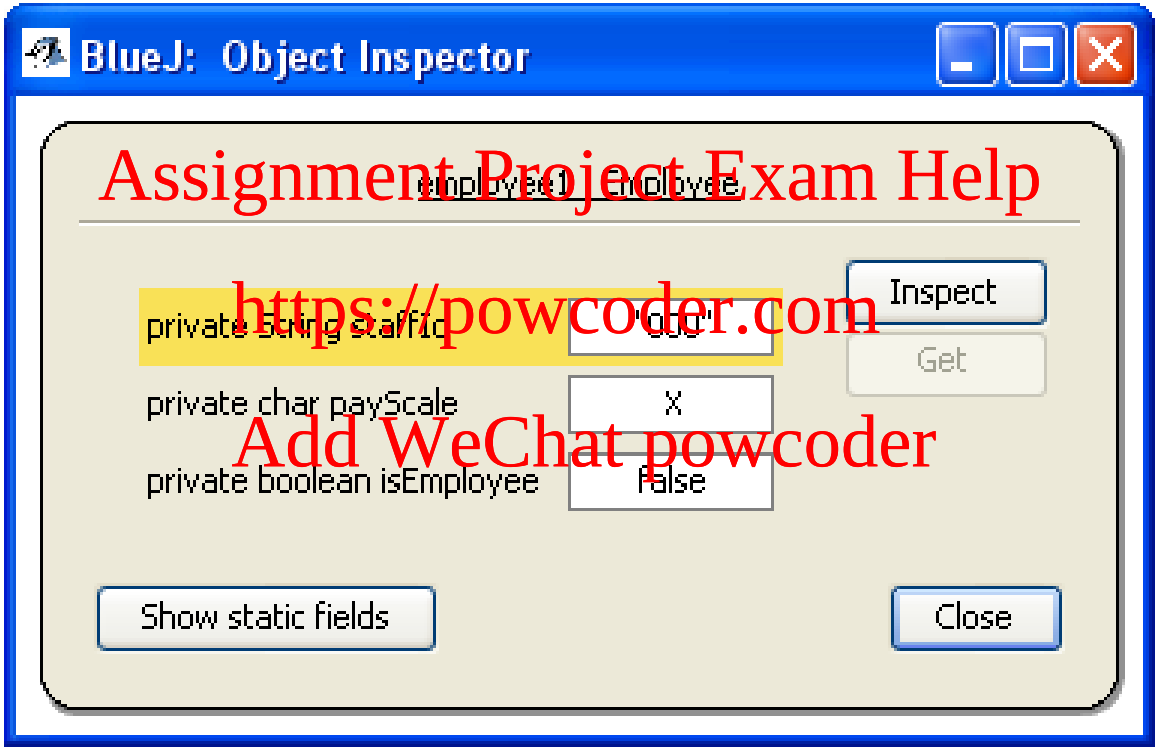
Expected Results:

- **staffId:** **"000"**
- **payScale:** **'X'**
- **isEmployee:** **false**

NOTE : using ***actual values***, not just general descriptions

Test 1 continued : Actual Results

Actual Results:



Screen capture of Test 1's results

Example : `Employee` *class default constructor*

```
public Employee()
{
    staffId = "000";
    payScale = 'X';
    isEmployee = false;
}
```

if these were wrong, the errors will show up in Test 1 (the Expected Results will not match the Actual Results)

Test 2

description of the test to be carried out

Test 2

Create an Employee object with the non-default constructor.

Test Data:

- **staffId:** **"517"**
- **payScale:** **'C'**
- **isEmployee:** **true**

NOTE : using ***actual values***, not just general descriptions

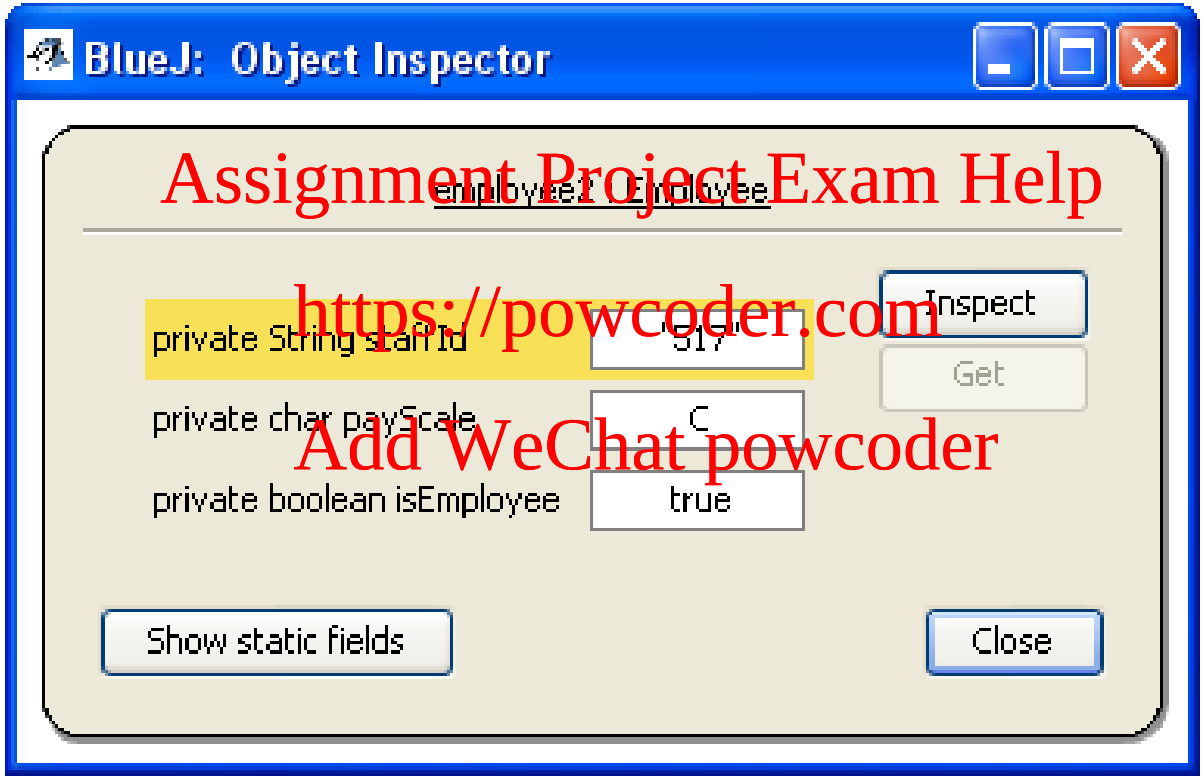
Expected Results:

- **staffId:** **"517"**
- **payScale:** **'C'**
- **isEmployee:** **true**

NOTE : using ***actual values***, not just general descriptions

Test 2 continued : Actual Results

Actual Results:



Screen capture of Test 2's results

Positive and Negative tests

Goal of testing is to identify, isolate and correct as many errors as possible.

A *positive test* demonstrates that the program works correctly when the input data is as expected.

(all the tests shown so far have been positive ones. It means that we feed in the kind of data that is supposed to work.)

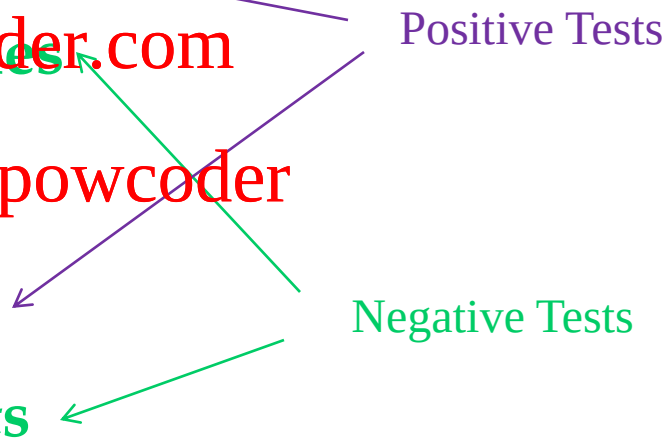
A *negative test* demonstrates that the program does not do anything incorrect if the input data is not as expected.

eg. what would you expect to happen if you entered a string of length 2 characters for the staffId?

An improved Test Plan

Test Plan

1. Create an Employee object with the default constructor
2. Create an Employee object with the non-default constructor
   (a) with valid field values
   (b) with invalid field values
3. Test all the get methods
4. Test all the set methods
   (a) with valid arguments
   (b) with invalid arguments
5. Test the display method



Example : Test 2(b)

Test 2(b)

Create an Employee object with invalid attributes (where possible)

Test Data:

- **staffId:** **"99"**
- **payScale:** **'a'**
- **isEmployee:** **true**

invalid values are given for the attributes

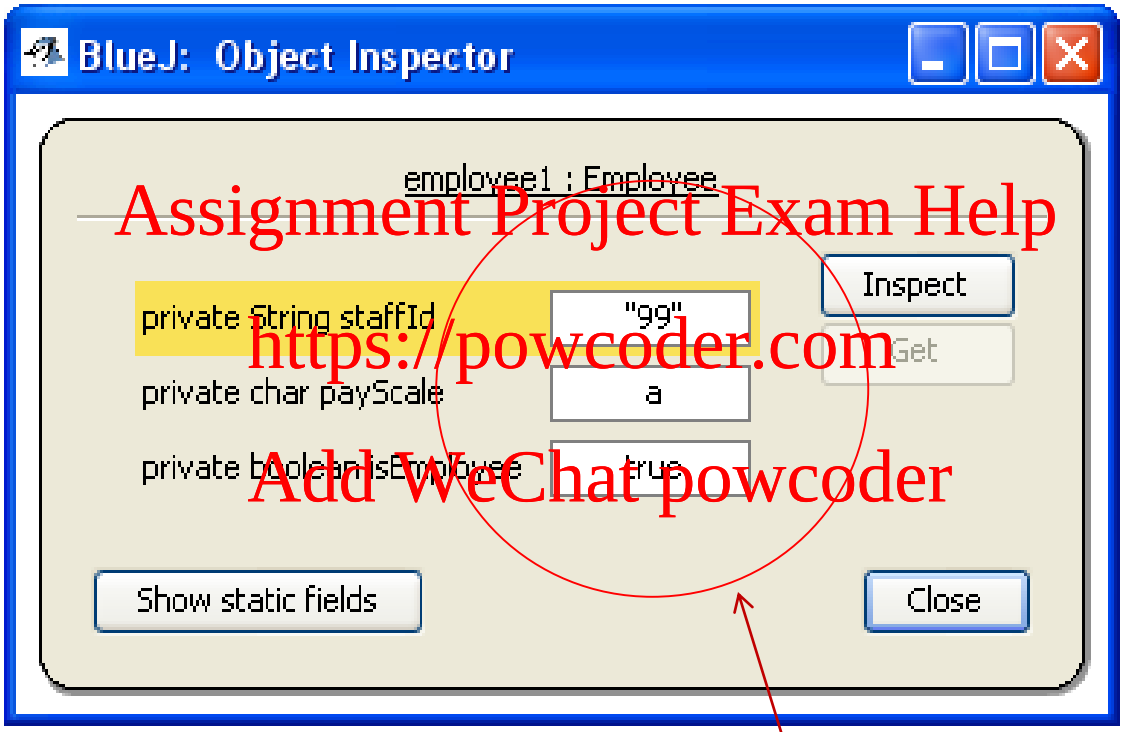
Expected Results:

- **staffId:** **"000"**
- **payScale:** **'X'**
- **isEmployee:** **false**

we are expecting some sensible default values here, since the given data is "wrong"

Actual results of Test 2(b)

Actual Results



Screen capture of Test 2(b).

these results would indicate errors in the code!!!

Elaborating on a test plan

Each test listed in the test plan has to be shown in much more detail, separately.

Some tests may need to be broken down more,

Eg :

Test 3 (test all the get methods) needs to be broken down into:

- Test 3(a) Test the `getTaxId` method
- Test 3(b) Test the `getPayScale` method
- Test 3(c) Test the `getEmployeeStatus` method

The `set` methods will be expanded in the same way.

Useful Testing Heuristic #1

1: Check any *boundaries* in the specification.

Example:

if each page of a report is to contain no more than 60 lines, then need to set up:

- **a set of test data that will print exactly 60 lines on the report (or as near as possible on the low side), *and***
- **another set of test data that will print 61 lines on the report (or as near as possible on the high side).**

boundaries

Useful Testing Heuristic #2

Eg :

```
public String calculateGrade()
{
    if (mark < 0)
       return "Invalid";
    if (mark < 50)
       return "N";
    if (mark < 60)
       return "P";
    if (mark < 70)
      return "C";
    if (mark < 80)
      return "D";
    if (mark <= 100)
      return "HD";
    return "Invalid";
}
```

2: Test *every path* that the flow of control can take through the logic of the program.

How many paths are there in this code?

Precision Issues

Sometimes we may need to consider possible precision issues, for instance :

- when double or floating point (real) numbers are involved, the user should be asked to specify the precision of the input data.
- if converting marks to grades, you should find out to what precision a mark can be recorded. Eg. If it is to one decimal place, your testing should include 49.9 and 50.0 If it is to 2 decimal places, the testing should include 49.99 and 50.00

Regression testing

Whenever you make a change to a program, you need to make sure that you haven't introduced any new errors.

- do not assume that errors will only occur in the area of the program where the changes were made.

Regression Testing is the process of re-running all previous tests, after some changes are made to the code

- sometimes new tests may need to be introduced, after errors are fixed, and the code then re-tested

Summary - testing a class

Each class should be tested separately. Each method in each class should be tested separately.

In order to test a method, create an instance of the class, then send it a message invoking the method to be tested, giving it various values as actual arguments.

The testing of the class should invoke all the methods in the public interface. If possible, all the private methods should also be invoked, by public methods in the interface.

For initial testing, make all methods public so that you can test them separately. When you are satisfied that everything works, you can make some methods private (if appropriate).

Debugging

It is important to develop code-reading skills.

- Debugging are often performed on code written by other people.

It is important to write code that can be easily read by others. Need to consider:

- Comments/documentations in code

- Layout of code – *Java coding standards*

There are techniques and tools designed to support the debugging process.

Debugging : Manual Walkthroughs

- This is a low-tech method where a program is "run by hand". The tester pretends to be the computer, and check through the code manually.
  - Conducted away from the computer using printouts of class diagrams or code.
  - Involves tracing the flow of control between classes and objects, while observing changes of state and other behaviour.
  - May use test data from test strategy
  - This is a relatively under-used, but powerful, debugging approach.

Debugging :Verbal Walkthroughs

- Explain to someone else what the code is doing.
  - They might spot the errors that you missed.
  - The process of explaining might help you to spot the errors for yourself.

- Group-based processes exist for conducting formal walkthroughs or *inspections*.

Debugging : Using Print Statements within the code

- If an error is not obvious, try printing out the values of relevant variables at critical points in the code, to see what is happening.
- This can be done by adding print statements temporarily to code.
  - Very common debugging technique.
  - No special tools are required.
  - Supported by all programming languages.
  - Output may be voluminous!
  - Turning printing off and on requires forethought.

Note that with BlueJ you may be able to see things more easily using the *Debugger* and the *Inspector*.

Dedicated "Debuggers"

- Debuggers are programs designed to debug code
  - They are typically both language- and environment-specific.
  - BlueJ has an integrated debugger :
    - Support breakpoints.
    - Step and Step-into controlled execution.
    - Call sequence (stack).
    - Inspection of object state.

Review - Testing

- Errors are a fact of life in programs – it is human to make mistakes.
- Good software engineering techniques can reduce the occurrence of errors.
- Testing is necessary, and testing and debugging skills are essential.
- Automate testing wherever possible.
  - Reduces tediousness.
  - Reduces human error.
  - Makes (re)testing easier.
- Practice a range of debugging skills.

Reminder

- The Unit Test for Assignment will take place during the tutorials in Week 8 (next week).
- Please remember to show up on time for your allocated tutorials.
- The Unit Test will be:
  - 1 hour long
  - 60 marks in total
  - Closed Book
  - Paper Based
- The second hour will be used for understanding how to define a test strategy.