Units and their tests

- Looking at stories
- Formulating a test strategy
- How to write unit tests
- How to run unit tests
- How to document test results

Analysing stories

- Stories come in many sizes –
  - some are small pieces of functionality
  - some are more complex
  - these can sometimes be broken down into a sequence of tasks
  - don’t forget any necessary data or process *validation*

Two test strategies – **Test first** and **Test last**

- **Test first –** write the unit tests before starting coding
- Write the code
- Apply the tests
- Believed to produce better code
- Harder to do – rather different approach to what is usually taught
- What happens if the unit has to be changed?

Test last

- More traditional approach
- Write the code for the unit
- Design the tests
- Apply the tests
- In both strategies the next step is debugging
- Record all the results in a database/spreadsheet

The key thing is the test set design

For each unit:

a) what are the ways in which the method will be accessed and what, if any, are the preconditions on the data that is supplied to it?

b) what are the ranges of values that need to be provided for the methods?

Identify test inputs for each input type

- a value below the lower boundary;
- a value equal to or at the lower boundary;
- a mid range value;
- a value equal to or at the higher boundary;
- a value above the higher boundary;
- a value in an incorrect format;
- a null value or no input.

Example: Type of literals

- *<return>*
- *a*
- *abcdef*
- *abcdefghijklmnopqrstuvwxyz1234*
- *fkdiorufberk5486jfkjfkdlk*
- *309475bfbldflkjslkj*
- *abcdefghijklmnopqrstuvwxyz12346*
- *4onkfkdpkfmlk8e3;im65687^^7E@@cmei;pd*
- *%`¬*&*
- *null_input*

Record the tests

test number	input name (string)	boolean flag	expected result	comments
1	*<return>*	*true*	error "no proper input"	to GUI
2	*<return>*	*false*	error "no proper input"	to GUI
3	*a*	*true*	screen message "already present"	to GUI
4	*a*	*false*	"submit to database" message	needs to connect with database
5	*abcdef*	*true*	screen message "already present"	to GUI
6	*abcdef*	*false*	"submit to database" message	needs to connect with database
7	*abcdefghijklmnopqrstuvwxyz1234*	*true*	screen message "already present"	to GUI
8	*abcdefghijklmnopqrstuvwxyz1234*	*false*	"submit to database" message	needs to connect with database

More complex units

This Example illustrates how to use the X Machine to express the active sequences for a class. The CFile class is a basic file I/O class which implements the basic non buffer file *read* and *write* operation. The class members are list below:

CFile Class Members

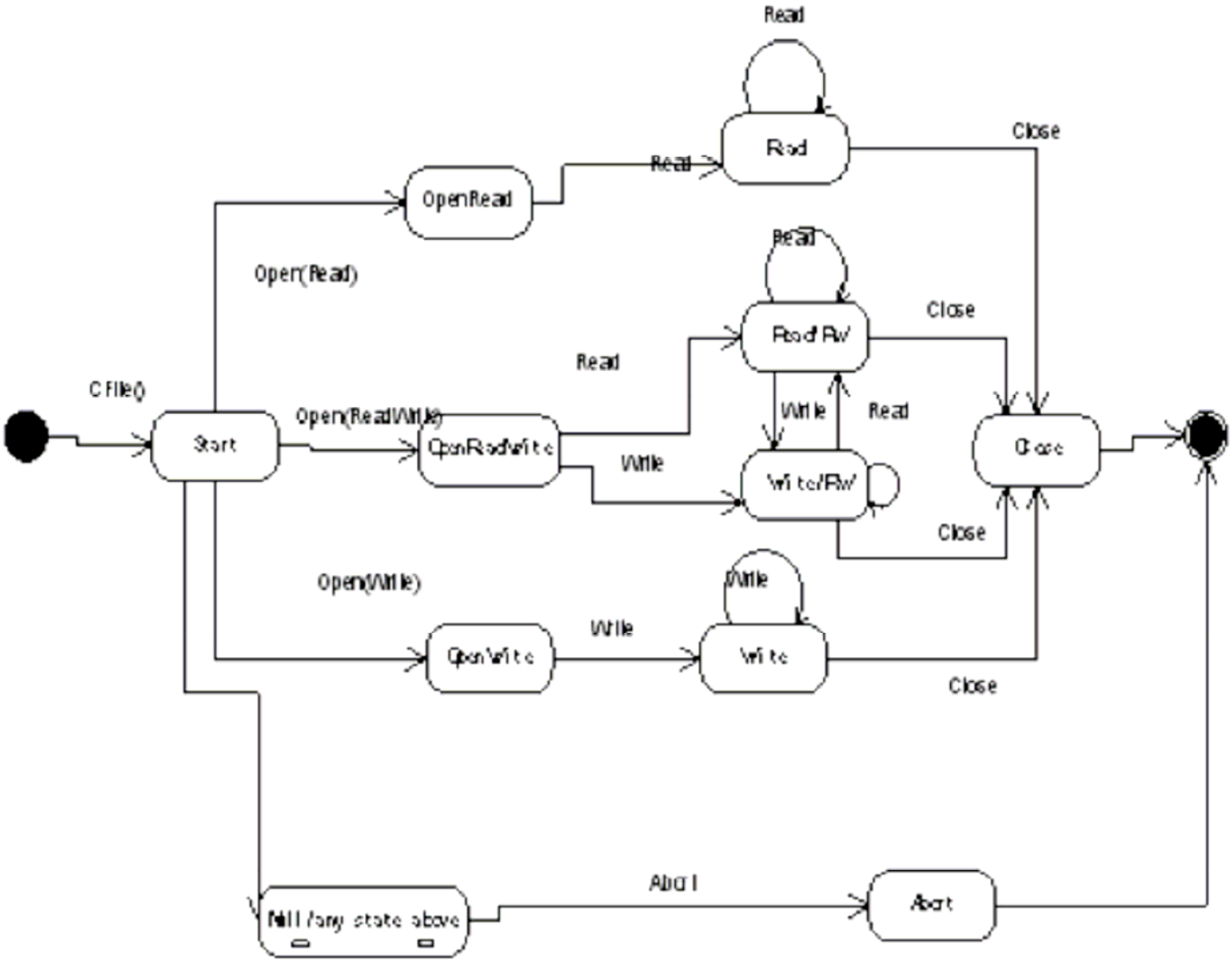
Data Members

m_hFile	Usually contains the operating-system file handle.

CFile	CFile()	Constructs a **CFile** object
Abort	Void Abort()	Closes a file ignoring all warnings and errors.
Open	virtual BOOL Open(LPCTSTR *lpszFileName*, UINT *nOpenFlags*, CFileException* *pError* = NULL);	Safely opens a file with an error-testing option.
Close	virtual void Close( ); throw(CFileException);	Closes a file and deletes the object.

Input/Output

Read	virtual UINT Read(void* lpBuf, UINT nCount); throw(CFileException);	Reads (unbuffered) data from a file at the current file position.
Write	virtual void Write(const void* lpBuf, UINT nCount); throw(CFileException);	Writes (unbuffered) data in a file to the current file position.


Read
Read
Close
OpenRead
Read
Open(Read)
Read
Close
ReadRW
CFile()
Read
Open(Read/Write)
Start
OpenReadWrite
Write
Write
Read
WriteRW
Close
Close
Open(Write)
Write
Write
OpenWrite
Write
Close
All/any state above
Abort
Abort

Some sample test sequences

- CFile();Open(Read);Open(Read)
- CFile();Open(Read);Open(Read/Write)
- CFile();Open(Read);Open(Write)
- CFile();Open(Read);Read()
- CFile();Open(Read);Write()
- CFile();Open(Read);Close()
- CFile();Open(Read);Abort()
- CFile();Open(Read/Write);Open(Read)
- etc.

White box testing

- In this approach we look at the structure of the code – only suitable for Test Last approach in the pure form
- Try to find test inputs that send the code through all paths
- Or to each decision point at least.
- Code becomes complex when we try to cope with unusual circumstances
- For example trying to deal with unusual combinations of inputs or events, exceptions etc.
- It is hard to think of all these so functional testing should provide us a key if the program crashes in situations we did not expect.

X-machines

- If you have a detailed X-machine then it should consider all possible operational situations
- It should enable us to deal with all the nasty circumstances.
- Test first teams should therefore try to build a really robust X-machine
- Then tests will be found by traversing the state machine graph rather than the code
- In full X-machine testing we try each function at all states – some *should* fail – this tells us that the program *does all it should and does not do anything it shouldn't*

Automating unit tests

- Use a tool such as JUnit
- (Beck, URL: www.XProgramming.com).
- This allows you to create a test class around the class under test and to submit tests to the code in a simple and effective way.
- It is highly recommend-ed that you look at this tool.

Other tools

- Similar tools are freely available for other programming languages,
- see Appendix D for
  - VBUnit and
  - PHPUnit

Writing Unit Tests in JUnit

- The easiest way to structure unit tests using JUnit is as follows:
- A unit test in JUnit can be taken to mean a test of a class.
- If there's a class called `Vector` in the package `mypackage.util`, there should be a class that tests it called `TestVector` in the package `mypackagetest.util`.
- For ease of reference, the directory structure of this test package should match that of the main package.

- The template structure for a test class is as follows:

```
package mypackagetest;


 import junit.framework.*;//the junit testing framework
 import mypackage.*;
public class VectorTest extends TestCase //Must extend TestCase
{
private Vector empty, full; //Just some vectors we can test on
```

```
 super(name);
 }
 public VectorTest(String name) {//Standard
    constructor, cut & paste
 public static Test suite() {//This is used later in
    collecting tests
return new TestSuite(VectorTest.class);//standard
    structure
 }
```

```
/**
* In here we can set up the variables we will be
  using in our tests.
* This method is run immediately before every
  individual test method.
*/
public void setUp() {
full = new Vector(2);
full.add(“element1”);
full.add(“element2”);
empty = new Vector();
}
```

```
//--- Now some actual tests ---
 public testAdd() {
 assert(empty.add("1").size() != 0);
 //...
 }
 public testRemove() { //if Remove removes the last element
 assert(full.remove().size() == 1);
 assert(empty.size() == 0); //setup() is called before each test
    method
 assert(empty.remove().size() == 0);
 //...
 }
 }
```

- Each test method is essentially just a list of assert statements, which will raise an exception if passed `false` as an argument.
- JUnit automatically collates all methods whose name starts with “test” to add to the set of tests, so if we were to add a method named `testInsert()` to the above code, JUnit would automatically detect it.
- Once we’ve got some of these test classes, we’ll want to be able to run them.

Create a test collection class in the following format:

```
 package mypackagetest;
import junit.framework.*;
public class AllTests {
public static Test suite() {
TestSuite suite = new TestSuite("All of the mypackage
   tests");
suite.addTest(VectorTest.suite());
suite.addTest(StringTest.suite());
suite.addTest(WhateverTest.suite());
return suite;
}
}
```

Running tests

- This is done in the following way: either

```
java junit.swingui.TestRunner
mypackagetest.AllTests
```

or

```
java junit.textui.TestRunner
mypackagetest.AllTests
```

Method name	Input	Prerequisites	Expected output	Actual Output/ Action	Status
PrintAction (constructor)	*name* of type **String** *printTitle* of type **String**	parameters given are initialised	sets up variables of the action	Constructor sets up variables correctly. Print button shows as required.	Tested Jane (12/03/ 02)
actionPer-formed	*event* of type **event**	**ActionEvent** occurs in the **JIn-ternalFrame** returned by clicking Print button	shows a *print dialog*		To Do
PrintTable-	*name* of type **string**	All input types valid	input parameters	The JButton displays	Tested and