CS 635 Object-Oriented Design & Programming
Spring Semester, 2004
Doc 1 Introduction

References

Testing for Programmers, Brian Marick, OOPSLA 2000 Tutorial, OOPSAL Oct 2000, Minneapolis, Minnesota,

Used with permission

A PDF version of the course can be downloaded from
http://www.testing.com/writings.html

Refactoring: Improving the Design of Existing Code, Fowler, 1999, pp. 110-116, 237-270

The Pragmatic Programmer, Hunt & Thomas, Addison Wesley Longman, 2000

Quality Software Management Vol. 4 Anticipating Change, Gerald Weinberg, Dorset House Publishing, 1997

Reading Assignment

Abstraction, Encapsulation, and Information Hiding available at:
http://www.toa.com/shnn?searticles

What is this Course About?

Writing quality OO code

Some basic tools:

- Abstraction
- Information Hiding
- Encapsulation
- Unit Testing
- Coupling & Cohesion
- Design Patterns
- Refactoring

Unit Testing

Testing

Johnson's Law

If it is not tested it does not work

Types of tests

- Unit Tests

  Tests individual code segments

- Functional Tests

  Test functionality of an application

Why Unit Testing

If it is not tested it does not work

The more time between coding and testing

- More effort is needed to write tests
- More effort is needed to find bugs
- Fewer bugs are found
- Time is wasted working with buggy code
- Development time increases
- Quality decreases

Without unit tests

- Code integration is a nightmare
- Changing code is a nightmare

When to Write Unit Tests

First write the tests

Then write the code to be tested

Writing tests first saves time

- Forces you to understand the interface & functionality of the code
- Removes temptation to skip tests
- Helps you avoid bugs

What to Test

Everything that could possibly break

Test values

- Inside valid range
- Outside valid range
- On the boundary between valid/invalid

GUIs are very hard to test

- Keep GUI layer very thin
- Unit test program behind the GUI, not the GUI

Testing Catalog

Common problem areas - what to focus tests on.

Numbers

Test the boundaries of the range

- Smallest number
- Just below the smallest number
- Largest number
- Just above the largest number

Test zero - special number that often is not handled correctly

```
for (int k = 12; k < 23; k++)
   {
   put your code here
   }
```

Strings

Test your code using the empty string

Collections

Test

- Empty collection
- Collection that contains exactly one element
- Maximum possible size (at least more than one)

Does your linked list work in these cases?

Searching

Test a search that has:

- No matches
- Exactly one match
- More than one match

These are the places were searches make mistakes

When you find a bug in your code

Write a test for it

Then fix the bug

Keep track of the bugs you make

Those that do not remember history are condemned to repeat the mistakes of the past

Programmers tend to repeat the same mistakes

Keep a list of the types of mistakes you make

Write tests for them before you write code

Coupling & Cohesion
Coupling

Strength of interaction between objects in system

Cohesion

Degree to which the tasks performed by a single module are functionally related

Reading Smalltalk
OOPS Rosette Stone

Java	Smalltalk
this	self
super	super
Field	Instance variable
Method	Method, message
"A String"	'A String'
/* a comment */	" a comment"
x = 5;	x := 5.
x == y	x == y
x.equals(y)	x = y

if (y > 3)
  x = 12;
	y > 3
	  ifTrue: [x := 12].

if (y > 3)
  x = 12;
else
  x = 9;
	y > 3
	  ifTrue: [x := 12]
	  ifFalse: [x := 3].

z = Point(2, 3);	z := 2 @ 3.
Circle x = new Circle(); Circle y = new Circle(0, 0 3);	| x y | x := Circle new. Y := Circle origin 0 @ 0 radius: 3
a.method()	a method
a.foo(x)	a foo: x
a.substring(4,7)	a copyFrom: 4 to: 7
return 5;	^5.

Java	Smalltalk

class Circle {
  public float area() {
  return this.radius().squared() * pi();
  }
}
	Circle>>area
	  ^self radius squared * self pi

Note Class>>method is not Smalltalk syntax. It is just a convention to show which class contains the method

The Weird Stuff
Methods - No Argument

C/C++/Java	Smalltalk
method()	method

Java

```
public class LinkedListExample
   {
   public static void main( String[] args )
      {
      LinkedList list = new LinkedList();
      list.print();
      }
   }
```

Smalltalk

```
| list |
list := LinkedList new.
list print.
```

Methods - One Argument

C/C++/Java	Smalltalk
method(argument)	method: argument

Java

```
public class OneArgExample
  {
  public static void main( String[] args )
    {
    System.out.println( "Hi mom");
    }
  }
```

Smalltalk

```
Transcript show: 'Hi Mom'.
```

Methods - Multiple Arguments

C/C++/Java	Smalltalk
method(arg1, arg2, arg3)	method: arg1 second: arg2 third: arg3

Java

```
public class MultipleArgsExample
   {
   public static void main( String[] args )
      {
      String list = "This is a sample String";
      list.substring(2, 8);
      }
   }
```

Smalltalk

```
| list |
list := 'This is a sample String'.
list
   copyFrom: 2
   to: 8
```

Cascading Messages

```
Transcript
   show: 'Name: ';
   show:  _name;
   cr;
   show: 'Amount: ';
   show: outstanding;
   cr.
```

Is short hand notation for:

```
Transcript show: 'Name: '.
Transcript show:  _name.
Transcript cr.
Transcript show: 'Amount: '.
Transcript show: outstanding.
Transcript cr.
```

Refactoring

We have code that looks like:

```
at: anInteger put: anObject
   (smallKey ~= largeKey)
      ifTrue:
         [(anInteger < smallKey)
            ifTrue: [self atLeftTree: anInteger put: anObject]
            ifFalse: [(smallKey = anInteger)
               ifTrue: [smallValue := anObject]
               ifFalse: [(anInteger < largeKey)
                  ifTrue: [self atMiddleTree: anInteger put: anObject]
                  ifFalse: [(largeKey = anInteger)
                     ifTrue: [largeValue := anObject]
                     ifFalse: [(largeKey < anInteger)
                        ifTrue: [self atRightTree: anInteger put: anObject]]]]]]
      ifFalse:
                     [self addNewKey: anInteger with: anObject].
```

Now what?

The Broken Window[1]

In inner cities some buildings are:

- Beautiful and clean
- Graffiti filled, broken rotting hulks

Clean inhabited buildings can quickly become abandoned derelicts

The trigger mechanism is:

- A broken window

If one broken window is left unrepaired for a length of time

- Inhabitants get a sense of abandonment
- More windows break
- Graffiti appears
- Pipes break
- The damage goes beyond the owner's desire to fix

Don't live with Broken Widows in your code

[1] Pragmatic Programmer, pp. 4-5

The Perfect Lawn

A visitor to an Irish castle asked the groundskeeper the secret of the beautiful lawn at the castle

The answer was:

- Just mow the lawn every third day for a hundred years

Spending a little time frequently

- Is much less work that big concentrated efforts
- Produces better results in the long run

So frequently spend time cleaning your code

Familiarity verse Comfort

Why don't more programmers/companies continually:

- Write unit tests
- Refactor
- Work on improving programming skills

Familiarity is always more powerful than comfort.

-- Virginia Satir

Refactoring

Refactoring is the modifying existing code without adding functionality

Changing existing code is dangerous

- Changes can break existing code

To avoid breaking code while refactoring:

- Need tests for the code
- Proceed in small steps

Sample Refactoring: Extract Method[2]

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method

Motivation

Short methods:

- Increase possible reuse
- Makes high level methods easier to read
- Makes easier to override methods

[2] Refactoring Text, pp. 110-116

Mechanics

- Create a new method - the target method

  Name the target method after the intention of the method

  With short code only extract if the new method name is better than the code at revealing the code's intention

- Copy the extracted code from the source method into the target method

- Scan extracted code for references to local variables (temporary variables or parameters) of the source method

- If a temporary variable is used only in the extracted code declare it local in the target method

- If a parameter of the source method is used in the extracted code, pass the parameter to the target method

Mechanics - Continued

- See if the extracted code modifies any of the local variables of the source method

  If only one variable is modified, then try to return the modified value

  If more than one variable is modified, then the extracted code must be modified before it can be extracted

  Split Temporary Variables or Replace Temp with Query may help

- Compile when you have dealt with all the local variables

- Replace the extracted code in source code with a call to the target method

- Compile and test

Example[3]
No Local Variables

Note I will use Fowler's convention of starting instance variables with "_".

```
printOwing
  | outstanding |

  outstanding := 0.0.
  Transcript
    show: '**********';
    cr;
    show: '***Customer Owes***';
    cr;
    show: '**********';
    cr.

  outstanding := _orders inject: 0 into: [:sum :each | sum + each].

  Transcript
    show: 'Name: ';
    show:  _name;
    cr;
    show: 'Amount: ';
    show: outstanding;
    cr.
```

[3] Example code is Squeak version of Fowler's Java example

Extracting the banner code we get:

```
printOwing
   | outstanding |

   outstanding := 0.0.
   self printBanner.

   outstanding := _orders inject: 0 into: [:sum :each | sum + each].

   Transcript
      show: 'Name: ';
      show:  _name;
      cr;
      show: 'Amount: ';
      show: outstanding;
      cr.

printBanner
   Transcript
      show: '**********************';
      cr;
      show: '***Customer Owes***';
      cr;
      show: '**********************';
      cr
```

Examples: Using Local Variables

We can extract printDetails: to get

```
printOwing
   | outstanding |
   self printBanner.
   outstanding := _orders inject: 0 into: [:sum :each | sum + each].
   self printDetails: outstanding

printDetails: aNumber
   Transcript
      show: 'Name: ';
      show:  _name;
      cr;
      show: 'Amount: ';
      show: aNumber;
      cr.
```

Then we can extract outstanding to get:

```
printOwing
   self
      printBanner;
      printDetails: (self outstanding)

outstanding
   ^_orders inject: 0 into: [:sum :each | sum + each]
```

The text stops here, but the code could use more work

Using Add Parameter (275)

```
printBanner
   Transcript
      show: '**********';
      cr;
      show: '***Customer Owes***';
      cr;
      show: '**********';
      cr
```

becomes:

```
printBannerOn: aStream
   aStream
      show: '**********';
      cr;
      show: '***Customer Owes***';
      cr;
      show: '**********';
      cr
```

Similarly we do printDetails and printOwing

```
printOwingOn: aStream
   self printBannerOn: aStream.
   self
      printDetails: (self outstanding)
      on: aStream
```

Perhaps this should be called Replace Constant with Parameter