CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005
Doc 1 Introduction
Contents

Reading Assignment 3
What is this Course About? 4
Reading Smalltalk 5
  The Weird Stuff 6
Coupling & Cohesion 10
  Coupling 10
  Cohesion 10
Design Patterns Intro 11
  Examples of Patterns 12
    A Place To Wait 12
    Chicken And Egg 14
  Benefits of Software Patterns 15
  Common Forms For Writing Design Patterns 16
  Design Principle 1 17
  Design Principle 2 19
Refactoring 22
  The Broken Window 23
  The Perfect Lawn 24
  Familiarity verse Comfort 25
  Refactoring 26
  Sample Refactoring: Extract Method 27
    Motivation 27
    Mechanics 28
    Example 30

References

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

Patterns for Classroom Education, Dana Anthony, pp. 391-406, *Pattern Languages of Program Design 2*, Addison Wesley, 1996

A Pattern Language, Christopher Alexander, 1977

Software Patterns, James Coplien, 1996, 2000, http://www1.bell-labs.com/user/cope/Patterns/WhitePaper/

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995

Reading Assignment

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

Design Patterns chapter 1.

What is this Course About?

Writing quality OO code

Some basic tools:

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

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.
```

Coupling & Cohesion
Coupling

Strength of interaction between objects in system

Cohesion

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

Design Patterns Intro

What is a Pattern?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"

Christopher Alexander on architecture patterns

"Patterns are not a complete design method; they capture important practices of existing methods and practices uncodified by conventional methods"

James Coplien

Examples of Patterns
A Place To Wait[1]

The process of waiting has inherent conflicts in it.

Waiting for doctor, airplane etc. requires spending time hanging around doing nothing

Cannot enjoy the time since you do not know when you must leave

Classic "waiting room"
- Dreary little room
- People staring at each other
- Reading a few old magazines
- Offers no solution

Fundamental problem
- How to spend time "wholeheartedly" and
- Still be on hand when doctor, airplane etc arrive

Fuse the waiting with other activity that keeps them in earshot
- Playground beside Pediatrics Clinic
- Horseshoe pit next to terrace where people waited

Allow the person to become still meditative
- A window seat that looks down on a street
- A protected seat in a garden
- A dark place and a glass of beer
- A private seat by a fish tank

[1] Alexander 1977, pp. 707-711

A Place To Wait

Therefore:

"In places where people end up waiting create a situation which makes the waiting positive. Fuse the waiting with some other activity - newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simple waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence"

Chicken And Egg[2]

Problem

Two concepts are each a prerequisite of the other

To understand A one must understand B

To understand B one must understand A

A "chicken and egg" situation

Constraints and Forces

First explain A then B

- Everyone would be confused by the end

Simplify each concept to the point of incorrectness to explain the other one

- People don't like being lied to

Solution

Explain A & B correctly by superficially

Iterate your explanations with more detail in each iteration

[2] Anthony 1996

Benefits of Software Patterns

By providing domain expertise patterns

- Reduce time to find solutions
- Avoid problems from inexpert design decisions

Patterns reduce time to design applications

- Patterns are design chunks larger than objects

Patterns reduce the time needed to understand a design

Common Forms For Writing Design Patterns

Alexander - Originated pattern literature

GOF (Gang of Four) - Style used in Design Patterns text

Portland Form -Form used in on-line Portland Pattern Repository

http://c2.com/cgi/wiki?PortlandPatternRepository

Coplien

Design Principle 1

Program to an interface, not an implementation

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

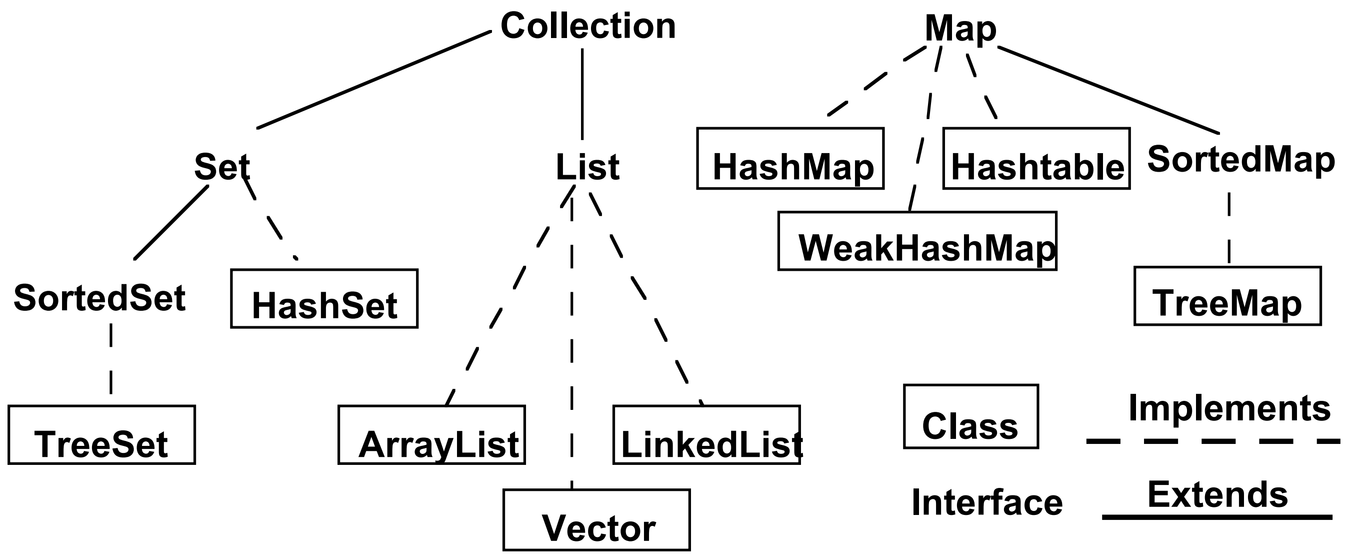
Declare variables to be instances of the abstract class not instances of particular classes

Benefits of programming to an interface

Client classes/objects remain unaware of the classes of objects they use, as long as the objects adhere to the interface the client expects

Client classes/objects remain unaware of the classes that implement these objects. Clients only know about the abstract classes (or interfaces) that define the interface.

Programming to an Interface
Java Collections



Collection students = new XXX;
students.add(aStudent);

students can be any collection type

We can change our mind on what type to use

Design Principle 2

Favor object composition over class inheritance

Composition

- Allows behavior changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance

```
class A {
  Foo x
  public int complexOperation() { blah }
}

class B extends A {
  public void bar() { blah}
}
```

Composition

```
class B {
  A myA;
  public int complexOperation() {
    return myA.complexOperation()
  }

  public void bar() { blah}
}
```

Parameterized Types

Generics in Ada, Eiffel, Java (jdk 1.5)
Templates in C++

Allows you to make a type as a parameter to a method or class

```
template <class TypeX>
TypeX min(  TypeX a, Type b )
   {
   return a < b ? a  :  b;
   }
```

Parameterized types give a third way to compose behavior in an object-oriented system

Designing for Change

Some common design problems that GoF patterns that address

• Creating an object by specifying a class explicitly

Abstract factory, Factory Method, Prototype

• Dependence on specific operations

Chain of Responsibility, Command

• Dependence on hardware and software platforms

Abstract factory, Bridge

• Dependence on object representations or implementations

Abstract factory, Bridge, Memento, Proxy

• Algorithmic dependencies

Builder, Iterator, Strategy, Template Method, Visitor

• Tight Coupling

Abstract factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

• Extending functionality by subclassing

Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

• Inability to alter classes conveniently

Adapter, Decorator, Visitor

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[3]

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

[3] 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[4]

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

[4] 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[5]
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.
```

[5] 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