CS 535 Object-Oriented Programming & Design
Fall Semester, 2003
Doc 12 Some OO Terms

Some OO Terms ................................................................. 2
Abstraction ................................................................. 3
Encapsulation ............................................................. 5
Information Hiding ....................................................... 6
Coupling ................................................................. 16
Cohesion ................................................................. 16
Ralph Johnson’s Suggestions for Finding Abstractions........ 17
Polymorphism ............................................................. 25
Avoid Case Statements ................................................... 27
Simplistic Example ......................................................... 28
Linked List Example ........................................................ 32

References


Object-Oriented Design Heuristics, Riel, Chapter 2

Copyright ©, All rights reserved. 2003 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
Some OO Terms

- Abstraction
- Encapsulation
- Information Hiding
- Coupling
- Cohesion
- Polymorphism
Abstraction

“Extracting the essential details about an item or group of items, while ignoring the unessential details.”

Edward Berard

“The process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use.”

Richard Gabriel

Example

Pattern: Priority queue

Essential Details: length
  items in queue
  operations to add/remove/find item

Variation: link list vs. array implementation
  stack, queue
Heuristic 2.8

A class should capture one and only one key abstraction

Look at nouns in requirements specification or system description

Look at these phrases. Some will be obvious classes, some will be obvious nonsense, and some will fall between obvious and nonsense. Skip the nonsense, keep the rest. The goal is a list of candidate objects. Some items in the list will be eliminated, others will be added later. Finding good objects is a skill, like finding a good functional decomposition.

A refrigerator has a motor, a temperature sensor, a light and a door. The motor turns on and off primarily as prescribed by the temperature sensor. However, the motor stops when the door is opened. The motor restarts when the door is closed if the temperature is too high. The light is turned on when the door opens and is turned off when the door is closed.
Encapsulation

Enclosing all parts of an abstraction within a container

Class contains

• Variables

• All the code that accesses the variables

Heuristic 2.9

Keep related data and behavior in one place

Code that uses a lot of accessing methods of an object should be used to that object

\[(aPoint \times \text{squared} + aPoint \times y \text{ squared}) \sqrt \text{verses} \]

\[aPoint \text{ r} \]
**Information Hiding**

An object should hide design decisions from its users

Hide

- What is stored & what is computed
- Classes used

How does Point story its data?

How does OrderedCollection hold elements?

We use the classes without knowing
**Heuristic 2.1**

All data should be hidden within it class

Smalltalk instance variables in can be accessed in:

- Instance methods of Class where they are defined
- Instance methods of subclasses of the Class where they are defined

Most languages have a construct for global access to data

- Smalltalk has shared variables
- Use sparingly
- Use for constants
- What is a constant?
Hiding Instance Variables

Some argue that only two methods should access an instance variable

Class BankAccount
Instance variable: balance

balance ^balance

balance: aNumber
  balance := anumber

deposit: aNumber
  self balance: (self balance + aNumber)

This protects the class from changes in instance variables
• Change variable to computed result

Makes easy to enforce constraints

balance: aNumber
  aNumber < 0 ifTrue: [ NegativeBalanceError raiseSignal].
  balance := aNumber
Hiding Instance Variables & Refactoring Browser

Refactoring browser
• Lists all methods accessing an instance variable
• Change all accesses to be through access methods
• Removes all access through access methods

So don’t worry about hiding instance variables

If later you need to hide them it is easy to do
Smalltalk and Private Methods

Private method

• Used for some internal computation
• Not to be called from outside of the class

All instance methods in Smalltalk are publicly accessible

Put private methods in “private” protocol

Smalltalk programmers know not to use such methods
Engineering Heuristics, Absolutes & Beginners

All design decisions involve tradeoffs

Heuristics are design decisions that are nearly always true

No heuristic is correct all the time

Beginners violate heuristics because

• They don’t understand the tradeoffs involved

• Don’t know about alternatives

• Habit
Two View of a Class: Inside & Outside

Users of a class care about

• Public methods

• English description

• Examples

• Tests

Users don’t need to know implementation details

To a user a class is a black box
Inheritance (White Box) Verses Composition (Black Box)

**Composition**

Smalltalk defineClass: #ComposedWordStream
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'input '
  classInstanceVariableNames: "
  imports: "
  category: 'CS535'

**Class Methods**

on: aCollection
  ^self input: (ReadStream on: aCollection)

input: aStream
  ^super new setInput: aStream

**Instance Methods**

atEnd
  ^input atEnd

next
  "some code to get the next word"
  blah

peek
  ^input peek

setInput: aStream
  input := aStream
Inheritance

Smalltalk.Core defineClass: #WordStream
   superclass: #\{Core.ReadStream\}
   indexedType: #none
   private: false
   instanceVariableNames: "
   classInstanceVariableNames:"
   imports: "
   category: 'CS535'

next
   "some code to get the next word"
   blah

Changing next changes how peek works
But what if we don't want peek changed?
Inheritance (White Box) Verses Composition (Black Box)

**Inheritance**

Need to know how super class works

Hard to change super class

May inherit methods you don’t want

Can replace uses of super class (ReadStream) with new class (WordStream)

**Composition**

Don’t need to know how composed object (ReadStream) works

Can change type of composed object

Controls which methods of composed object to expose

Forward messages to composed object

In languages that declare types may not be able to replace composed object with new class (WordStream)
**Coupling**

Strength of interaction between objects in system

How tangled together the classes are

**Cohesion**

Degree to which the tasks performed by a single module are functionally related
Ralph Johnson’s Suggestions for Finding Abstractions

• Do one thing
• Eliminate duplication
• Keep rate of change similar
• Decrease coupling, increase cohesion
• Minimize interfaces
• Minimize size of abstractions
• Minimize number of abstractions
Do One Thing

Method should do on thing

• Method name should tell what it does

    findString:startingAt:
    asNumber
    asUppercase
    dropFinalVowels

Class should be what its name says

    String
    OrderedCollection
    Array
   ReadStream

Break complex classes/methods into simpler ones
Eliminate Duplication

\[(\text{self asInteger} - \$a \text{ asInteger} + \text{anInteger}) \mod 26 - (\text{self asInteger} - \$a \text{ asInteger})\]

\[(\text{self alphabetValue} + \text{anInteger}) \mod 26 - \text{self alphabetValue}.\]
Keep rate of change similar

• Separate initial conditions from algorithm’s temporary variables

• Separate tax tables from employee data from time cards
Minimize interfaces

Use the smallest interface you can

   Use Number instead of Float

Avoid embedding classes in names

   add: instead of addNumber:

Don’t check the class of an object
Minimize size of abstractions

Methods should be small

• Median size is 3 lines
• 10 lines is starting to smell

Classes should be small

• 7 variables is starting to smell
• 40 methods is starting to smell

VW 7.0 Base System

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables / class</td>
<td>2.1</td>
<td>1</td>
</tr>
<tr>
<td>Methods / class</td>
<td>16.7</td>
<td>9</td>
</tr>
<tr>
<td>Carriage returns/method</td>
<td>7.6</td>
<td>5.0</td>
</tr>
</tbody>
</table>
Code used to generate Numbers

Variables Per Class

classes := Smalltalk allClasses reject: [:each | each isMeta]
variablesInClass := classes collect: [:each | each instVarNames size].
average := ((variablesInClass fold: [:sum :each | sum + each] )/
variablesInClass size) asFloat.
mean := variablesInClass asSortedCollection at: variablesInClass size // 2.
max := variablesInClass fold: [:partialMax :each | partialMax max: each]

Methods Per Class

classes := Smalltalk allClasses reject: [:each | each isMeta]
methodsInClass := classes collect: [:each | each selectors size].
average := ((methodsInClass fold: [:sum :each | sum + each] )/
methodsInClass size) asFloat.
mean := methodsInClass asSortedCollection at: methodsInClass size // 2.
max := methodsInClass fold: [:partialMax :each | partialMax max: each]
Minimize number of abstractions

A class hierarchy 6-7 levels deep is hard to learn

Break large system into subsystems, so people only have to learn part of the system at a time
Polymorphism

Objects with the same interface can be substituted for each other at run-time

Variables take on many classes of object

Objects will behave according to their type

Code can work with any object that has the right set of methods

In C++ polymorphism requires
- Inheritance
- Pointers
- Virtual functions

In Java polymorphism requires
- Inheritance or
- Interfaces

In Smalltalk polymorphism does not require inheritance
Example

Counter>>printOn: aStream
    aStream
        nextPutAll: ‘Counter(‘;
        nextPutAll: count printString;
        nextPutAll: ‘)’

aStream can be any object that implements nextPutAll:

Note we do not write:

Counter>>printOn: aStream
    aStream class = FileStream ifTrue: [ write to file ].
    aStream class = WriteStream ifTrue: [write to write stream]
    aStream class = TextCollector ifTrue: [write to Transcript]
Avoid Case Statements

Smalltalk has no case statement

OO programmers send a message to object instead

Each type of object handles the message according to its type

Case statements make it harder to add new cases
Simplistic Example

Bank offers various types of accounts:

- Checking
- Savings
- CD
- Junior savings accounts

Each type has different rules for processing a transaction
Banking Classes

Customer

Transaction

Currency

Account (Abstract)

Checking

Interest Account (Abstract)

Savings

CD

Junior
Processing a Transaction

Using Case Statement

customer := Bank createNewAccount: type.

Etc.

customer class = Checking ifTrue:[ …]
customer class = Savings ifTrue:[ …]
customer class = CD ifTrue:[ …]
customer class = Jonior ifTrue:[ …]
Polymorphism

newCustomer.processTransaction: amount

Which processTransaction is called?

Adding new types of accounts to program requires:

  Adding new subclasses

  Changing code that creates objects

Avoid checking the class of an object
Linked List Example

Note all the checking at in the methods

Smalltalk defineClass: #LinkedList
    superclass: #{Core.SequenceableCollection}
    indexedType: #none
    private: false
    instanceVariableNames: 'value next '
    classInstanceVariableNames: "
    imports: "
    category: 'CS535'

Class Methods

with: anObject
    ^super new setValue: anObject

Instance Methods

addLast: anObject
    next ifNotNil: [^next addLast: anObject].
    next := LinkedList with: anObject.

includes: anObject
    value = anObject ifTrue:[^true].
    next ifNotNil: [^next includes: anObject].
    ^false
printOn: aStream
  aStream
    print: value;
    nextPutAll: ' '.
    next ifNotNil: [next printOn: aStream]

setValue: anObject
  value := anObject.

size
  next ifNil: [^1].
  ^next size + 1
**Linked List with Polymorphism**

A node to represent the end of the list

Smalltalk defineClass: #NilNode
declaring class
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: ""
    classInstanceVariableNames: ""
    imports: ""
    category: 'CS535'

**Instance Methods**

addLast: anObject
    self become: (LinkedList with: anObject)

includes: anObject
    ^false

printOn: aStream

size
    ^0
LinkedList with NilNode

Smalltalk defineClass: #LinkedList
  superclass: #{Core.SequenceableCollection}
  indexedType: #none
  private: false
  instanceVariableNames: 'value next '
  classInstanceVariableNames: "
  imports: "
  category: 'CS535'

Class Methods

with: anObject
  ^super new setValue: anObject

Instance Methods

addLast: anObject
  next addLast: anObject

includes: anObject
  value = anObject ifTrue:[^true].
  ^next includes: anObject

printOn: aStream
  aStream
    print: value;
    nextPutAll: '
  next printOn: aStream
setValue: anObject
  value := anObject.
  next := NilNode new.

size
  ^next size + 1