CS 535 Object-Oriented Programming & Design
Fall Semester, 2013
Doc 11 Some OO Review
Oct 3 2013

Copyright ©, All rights reserved. 2013 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/openpub/) license defines the copyright on this document.

References

Object-Oriented Design Heuristics, Riel

Principles of OO Design, or Everything I Know About Programming, I Learned from Dilbert, http://alanknightsblog.blogspot.com/2011/10/principles-of-oo-design-or-everything-i.html

Smalltalk Best Practice Patterns, Beck, Prentice Hall,1997

Some Review

Names

Structure

ClassNames

methodName

variableName

Use full words in names

Semantics

Select meaningful names

What a class is

What a method does

Role a variable plays

```
insert: n and:r
|temp|
r isNil
ifTrue:
[
temp:= Node new:n.

^temp.
]

ifFalse:
[
(n < (r info))
ifTrue:
[
r llink:( self insert:n and:(r llink)).
]
ifFalse:
[
r rlink: (self insert:n and:(r rlink)).
].
^r.
]
```

Control o

```
insert: n and:r

    | temp |
    r isNil
      ifTrue:
          [temp := Node new: n.
          ^temp]
      ifFalse:
          [n < r info
              ifTrue: [r llink: (self insert: n and: r llink)]
              ifFalse: [r rlink: (self insert: n and: r rlink)].
          ^r]
```

Using Full Names

```
insert: aValue into: aNode

    aNode isNil
      ifTrue:
          [^Node new: aValue]
      ifFalse:
          [aValue < aNode info
              ifTrue: [aNode left: (self insert: aValue into: aNode left)]
              ifFalse: [aNode right: (self insert: aValue into: aNode right)].
          ^aNode]
```

```
variance
| n ans sum resultArray|
n := self size.
sum := 0.
self isEmpty ifTrue: [^0].
self do: [ :i |sum := sum + i].
ans := (sum / n) asFloat.
sum :=0 asFloat.
resultArray := self collect: [ :j | (j - ans) * (j-ans)].
resultArray do: [ :i |sum := sum + i].
^(sum / (n-1)) asFloat
```

variance

```
| Average num Variance Result size |
Variance := 0.
size:= self size.
Average := self averages.
num:= self calculate: Average.
^(num / (size - 1)) asFloat
```

```
squares
" This method returns a collection that contains the squares of the values in the receiver collection "
| arr1 |
arr1:= self collect: [ :each | each * each ].
^ arr1
```

```
arrayAverage

    | sumOfElements averageOfArray |
    sumOfElements := 0.
    self do: [:each | sumOfElements := each + sumOfElements].
    averageOfArray := (sumOfElements / self size) asFloat.
    ^averageOfArray
```

Rant and Rave about names & programmer status

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

Encapsulation

Enclosing all parts of an abstraction within a container

Information Hiding

Hiding of design decisions in a computer program

Hide decisions are most likely to change,
To protect other parts of the program

Class

Represents an abstraction

Encapsulates data and operations of the abstraction

Hide design decisions/details

Not so much a definition of a class as a goal how we should use a class.

Heuristics

2.1 All data should be hidden within it class

2.8 A class should capture one and only one key abstraction

2.9 Keep related data and behavior in one place

Numbers of the heuristics are from the text Object-Oriented Design Heuristics by Riel

Non-OO items

Helper methods

Data classes

Helper method

Method in class that

- Does not access any field (data member, instance variables)
- Just uses parameters

Helper Method

```
printTree:aStream node:traverseNode

    traverseNode = nil
        ifFalse:  [self printTree:aStream node: traverseNode left.
            aStream print: traverseNode data.
        aStream nextPutAll:','.
            self printTree:aStream node: traverseNode right].
```

Data Class

Node Instance Variables

left
right
value

Node instance methods

left
left:
right
right:
value
value:

Principles of OO Design, or Everything I Know About Programming, I Learned from Dilbert

Alan Knight

http://alanknightsblog.blogspot.com/2011/10/principles-of-oo-design-or-everything-i.html

1. Never do any work that you can get someone else to do for you

Excuse me Smithers. I need to know the total bills that have been paid so far this quarter. No, don't trouble yourself. If you'll just lend me the key to your filing cabinet I'll go through the records myself. I'm not that familiar with your filing system, but how complicated can it be? I'll try not to make too much of a mess.

Verses

SMITHERS! I need the total bills that have been paid since the beginning of the quarter. No, I'm not interested in the petty details of your filing system. I want that total, and I'll expect it on my desk within the next half millisecond.

1. Never do any work that you can get someone else to do for you

Example 1 Total of bills that have been paid this quarter for a factory

```
total := 0
aFactory billings do: [:each |
   (each status == #paid and: [each date > startDate])
      ifTrue: [total := total + each amount]].
```

versus

```
total := aPlant totalBillingsPaidSince: startDate.
```

1. Never do any work that you can get someone else to do for you

```
averages

	| sum average |
	sum := 0.
	self size = 0 ifTrue: [^0].
	self do:
			[:each |
			each respondsToArithmetic
				ifTrue: [sum := sum + each]
				ifFalse: [^'array contains more than numbers']].
	average := sum / self size.
	^average asFloat
```

1. Never do any work that you can get someone else to do for you

Collection>>average

```
self isEmpty ifTrue: [^0].
^self sum / self size
```

Collection>>sum

```
self isEmpty ifTrue: [^0].
^self fold: [:a :b | a +b]
```

Encapsulation & Responsibility

Encapsulation is about responsibility

Who does the work

Who should do the work

2. Avoid Responsibility

If you must accept a responsibility, keep it as vague as possible.

For any responsibility you accept, try to pass the real work off to somebody else.

```
BinarySearchTree>>do: aBlock
    root do: aBlock
```

Kent Beck's Properties of Good Style

Kent Beck's Properties of Good Code Stype

Once and only once

Lots of little pieces

Replacing objects

Moving Objects

Rates of change

Once and Only Once

"In a program written with good style, everything is said once and only once"

If have

several methods with same logic

several objects with same methods

then rule is not satisfied

Lots of little pieces

"Good code invariably has small methods and small objects"

Small pieces allow you to satisfy "once and only once"

Pieces?

```
variance
| n ans sum resultArray|
n := self size.
sum := 0.
self isEmpty ifTrue: [^0].
self do: [ :i |sum := sum + i].
ans := (sum / n) asFloat.
sum :=0 asFloat.
resultArray := self collect: [ :j | (j - ans) * (j-ans)].
resultArray do: [ :i |sum := sum + i].
^(sum / (n-1)) asFloat
```

Example

variance

```
| mean meanDifferences |
mean := self average.
meanDifferences := self collect: [:each | each - mean].
^meanDifferences squares sum/(self size -1)
```

Pieces

- average
- squares
- sum

Replacing objects

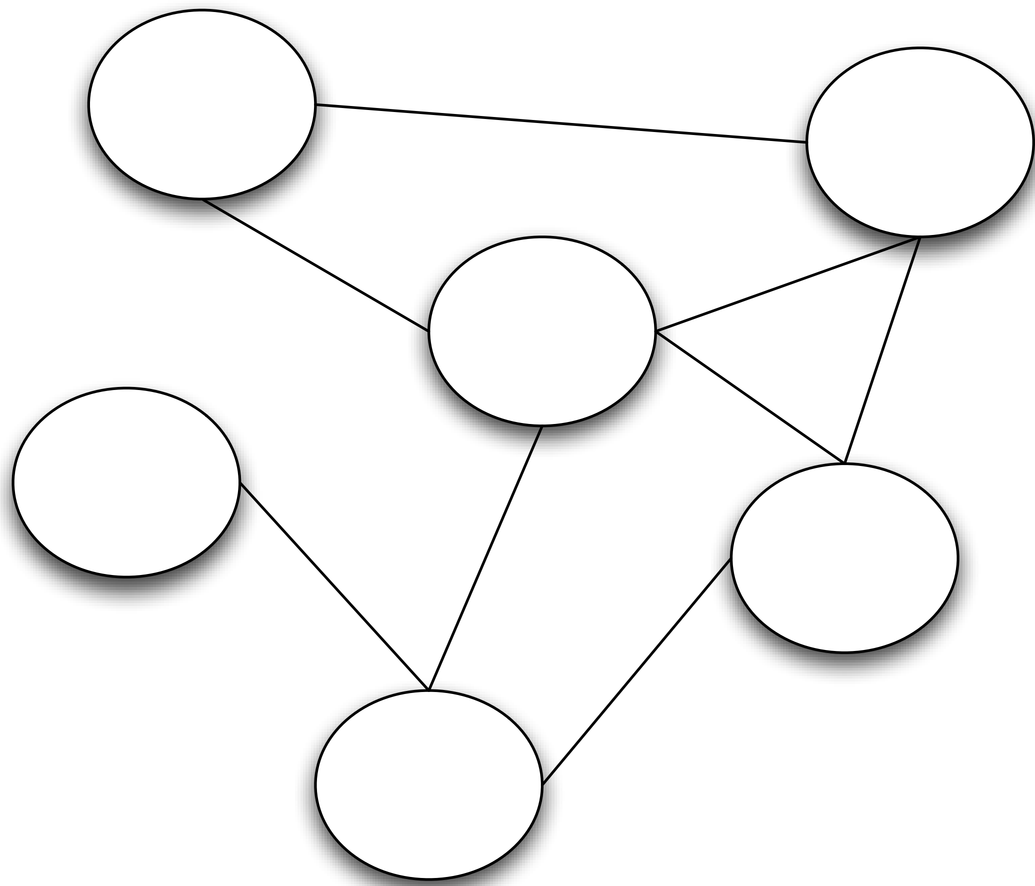
Good style leads to easily replaceable objects

When you can extend a system soley by adding new objects without modifying exisiting objects, then you have system that is fexible and cheap to maintain

Needs lots of little pieces

Some heuristics

OO Program



Building Blocks

OrderedCollection
String
Dictionary
Characters
Streams
Trolls
etc.

Class Builder verses Program Writer

"Main"

Adventure open

What does "main" in a program do? Think of a GUI application. Where is the main there?

Building Block = Class

2.8 A class should capture one and only one key abstraction

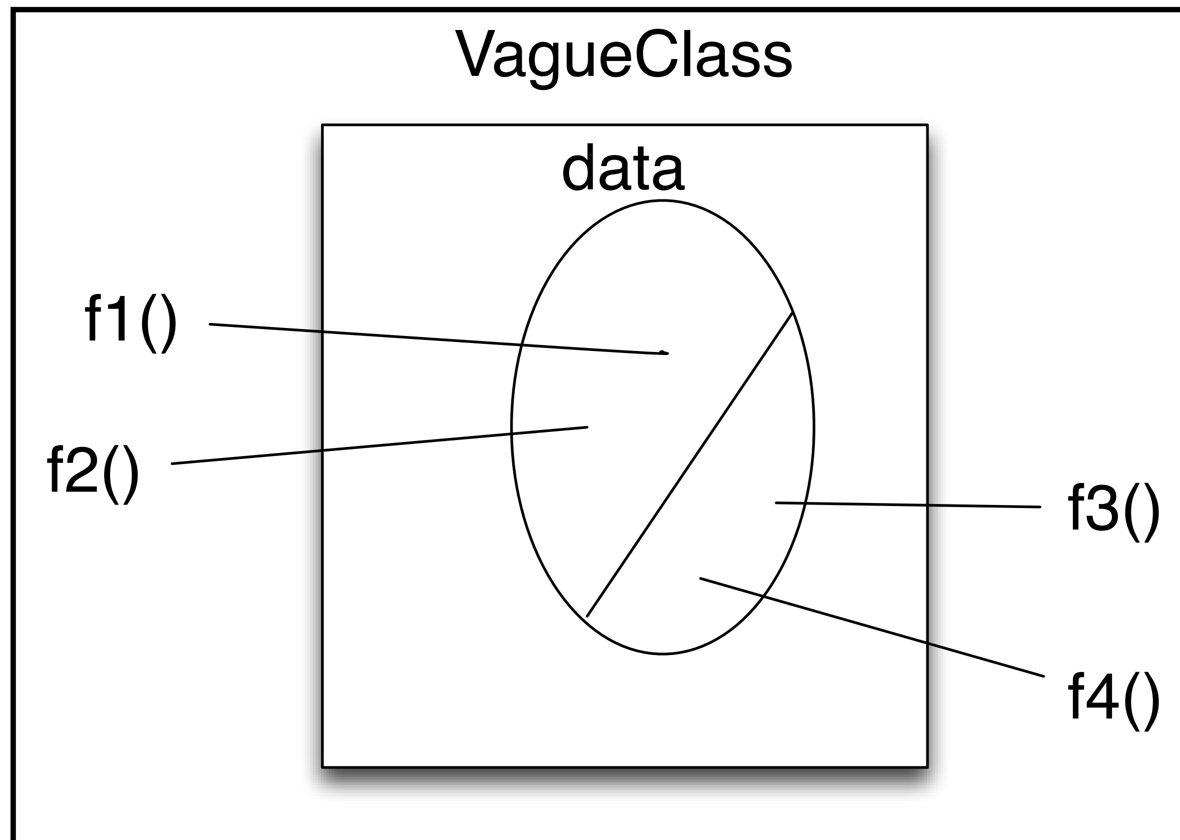
Keep related data and behavior in one place

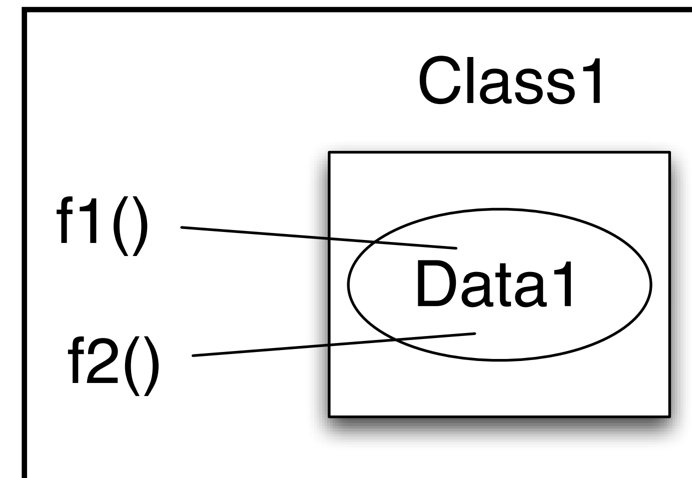
This is the most important idea in OO

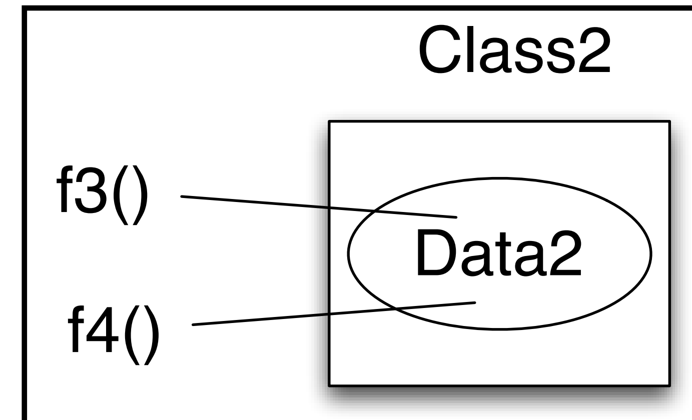
Corollary

To perform an operation send a message to the object that contains the data

Spin off nonrelated information into another class







God Class

God object is an object that knows too much or does too much

Behavioral Form

Replaces the main
Does too much

Definition is from Wikipedia

Heuristics

Distribute system intelligence horizontally as uniform as possible

Do not create god classes/objects

Be very suspicous of a class whose name contains Driver, Manager, System

Beware of classes that have many accessor methods defined in there public interface

Beware of classes that have too much noncommunicating behavoir

Using GUIs

Model should not depend on the interface
The interface should depend on the model

So interface needs to access data in the model