Object-Oriented Design Heuristics

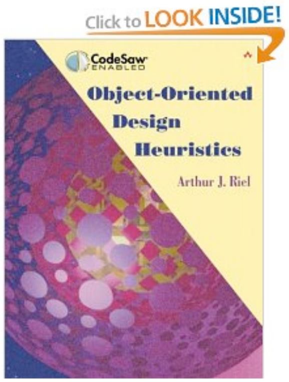
Alexandre Bergel
abergel@dcc.uchile.cl

24/05/2011

Source

Book from Arthur J. Riel

- Addison-Wesley Professional (May 10, 1996)
- 978-0201633856



Java 1.6

Pharo 1.0

Goal of this lecture

Provide useful and simple programming "rules"

Insight into object-oriented design improvement

Intended to

- Increase the readability and the quality of your code
- Facilitate software maintenance

Goal of this lecture

To make you better programmers and responsible engineers

This lecture provides good hints to not make people throw stones at you when they will look at your code

Classes and Objects

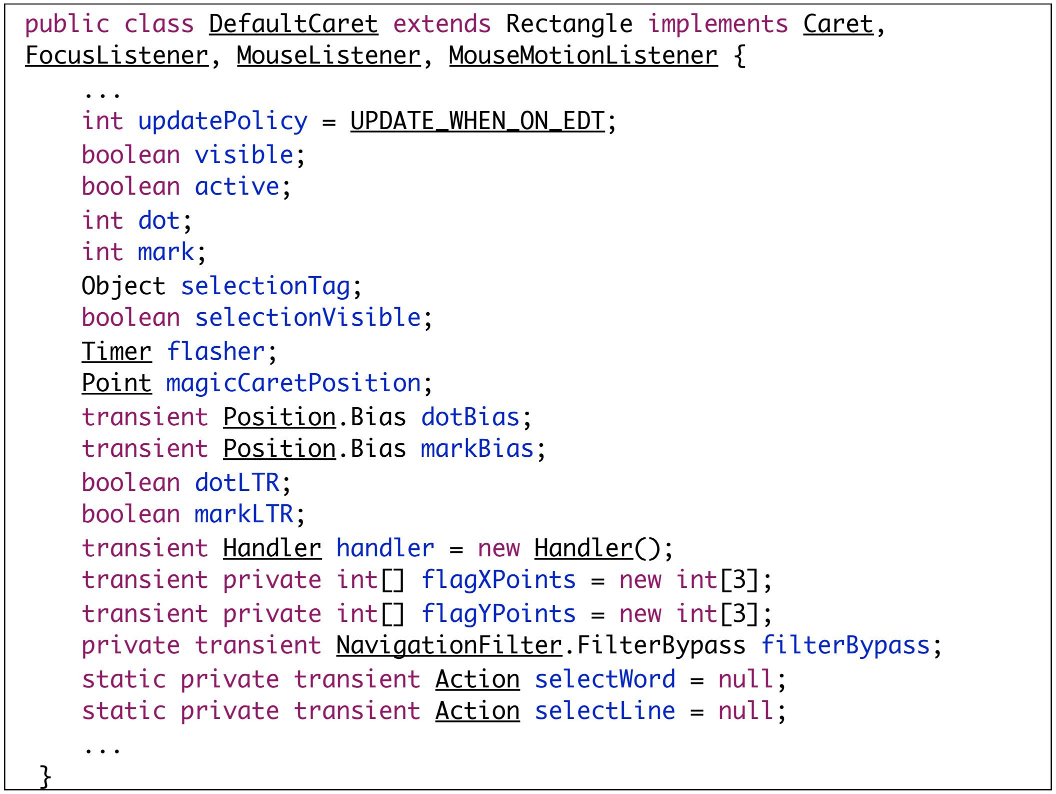
Classes and Objects

The building blocks of the object-oriented paradigm

An object will always have four important facets

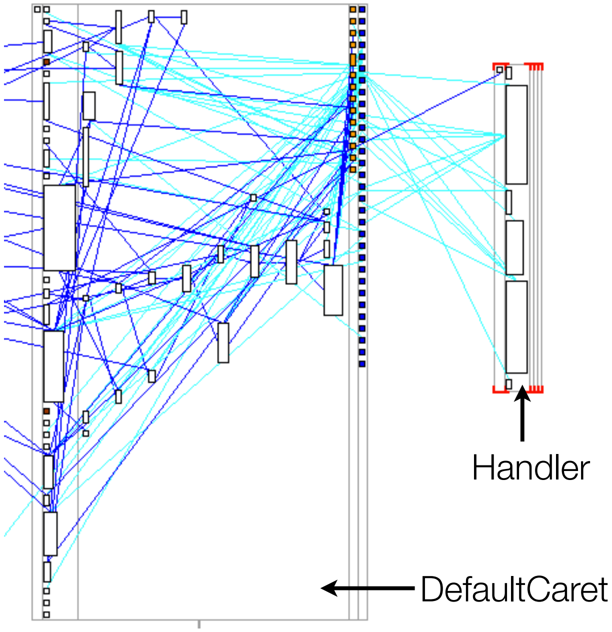
- its own identify (e.g., its address in memory)
- the attributes of its class (usually static) and values for those attributes (usually dynamic)
- the behavior of its class (the implementor's view)
- the published interface (the user's view)

All data should be hidden within its class



```java
public class DefaultCaret extends Rectangle implements Caret,
FocusListener, MouseListener, MouseMotionListener {
    ...
    int updatePolicy = UPDATE_WHEN_ON_EDT;
    boolean visible;
    boolean active;
    int dot;
    int mark;
    Object selectionTag;
    boolean selectionVisible;
    Timer flasher;
    Point magicCaretPosition;
    transient Position.Bias dotBias;
    transient Position.Bias markBias;
    boolean dotLTR;
    boolean markLTR;
    transient Handler handler = new Handler();
    transient private int[] flagXPoints = new int[3];
    transient private int[] flagYPoints = new int[3];
    private transient NavigationFilter.FilterBypass filterBypass;
    static private transient Action selectWord = null;
    static private transient Action selectLine = null;
    ...
 }
```

Piece of code extracted from the JDK 1.6. The class DefaultCaret belongs to the package javax.swing.text. It contains 15 public attributes

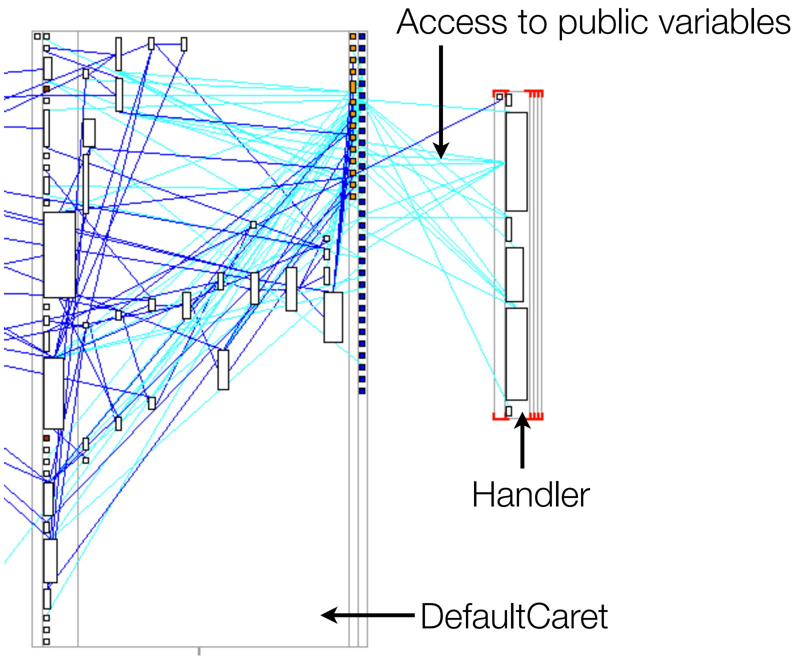


Class blueprint [Lanza 2003]

We will make heavy use of visualization along the semester. Visualizing software is a very handy and intuitive mechanism for getting a quantitative and qualitative impression about a system.

Class blueprint is a visualization that shows class internal. A class is represented as a box. Each box is composed of 5 part. Each part correspond to some particular elements that composes the class. From left to right: constructors, public methods, private methods, variable accessors and mutators (get and set methods), attributes.

Blue edges represent methods invocations. Cyan edges represent variable accesses.



Class blueprint [Lanza 2003]

The class DefaultCaret contains 15 public attributes. These attributes are accessed by other classes. Handler for example. This shows a poor programming style. Never make field public or package visible.

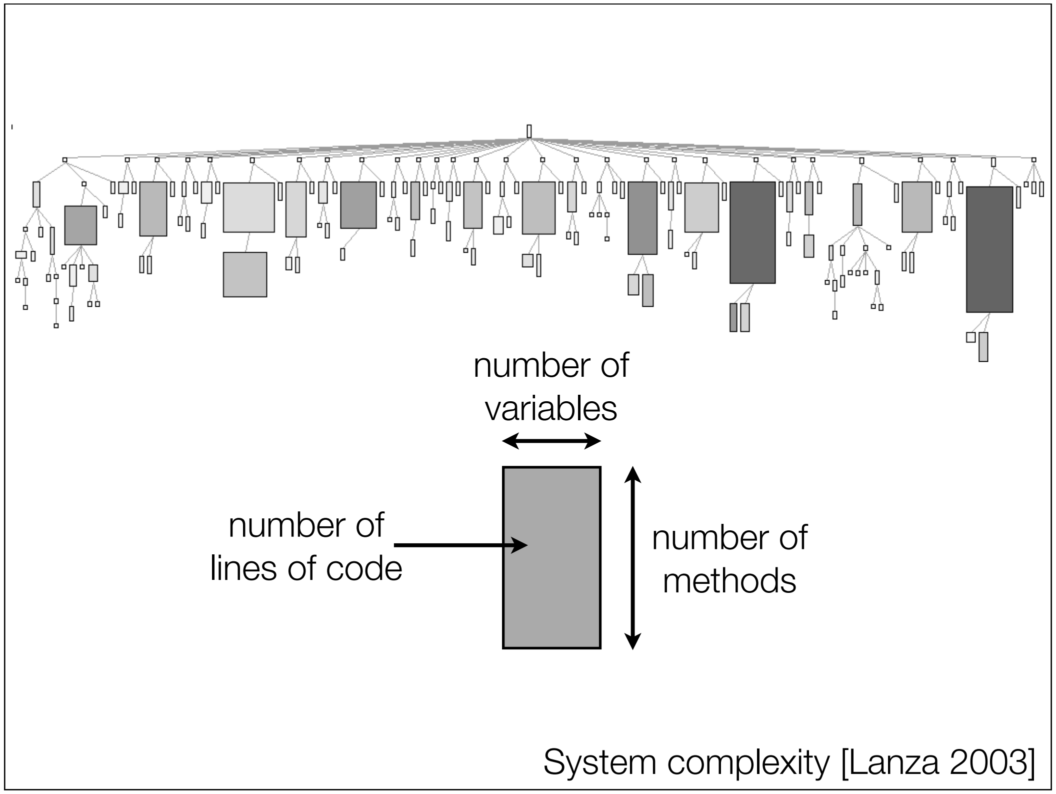
One may argue that for optimization reason, it may be preferable to have public variables instead of accessors. It was true some time ago when virtual machines and compiler were not that sophisticated. Today, making variable public is hardly considered as an efficient way to optimize your program.

How to hide data?

Visibility of variables should be set to private or protected

Define accessors and mutators when necessary

Minimize the number of messages in the protocol of a class



number of variables

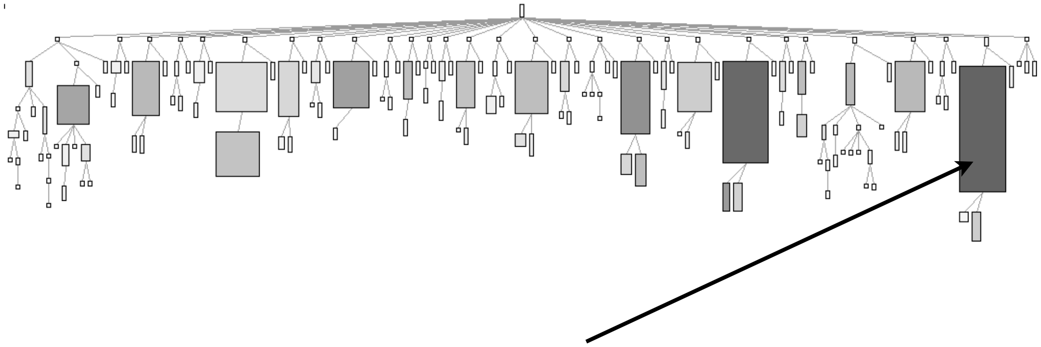
number of lines of code

number of methods

System complexity [Lanza 2003]

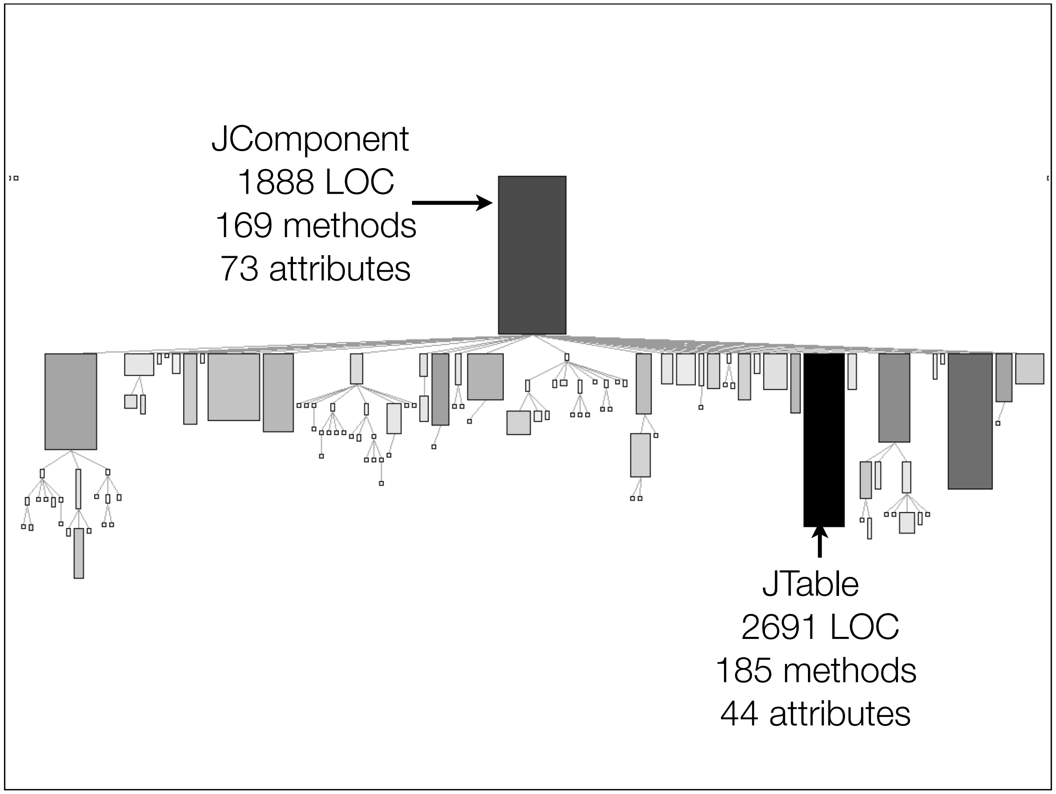
We can see another visualization, called System complexity. This visualization is about class hierarchies. Each class is represented as a box, shaped with three metrics: number of variables, number of methods and number of lines of code.

The hierarchy represented here is PLAF, the pluggable look and feel of Java. You can notice the irregularity of the hierarchies, which probably hide some missing functionalities.

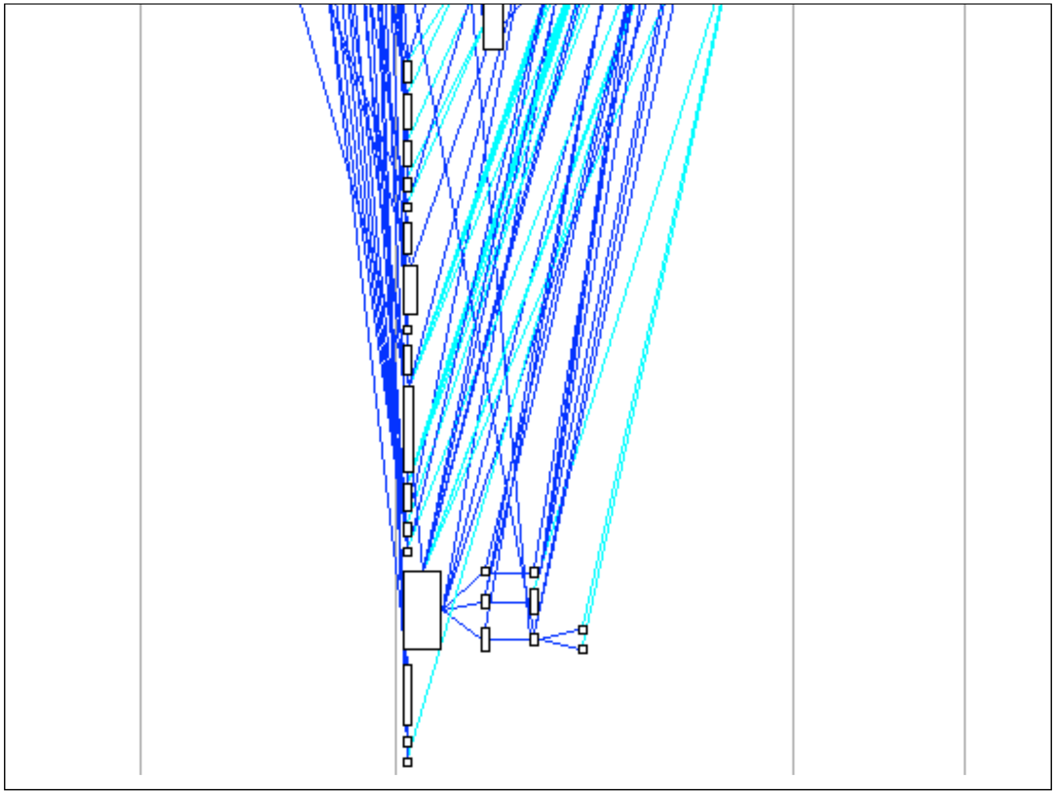


javax.swing.plaf.basic.BasicTreeUI
1636 lines of code
49 variables
131 methods

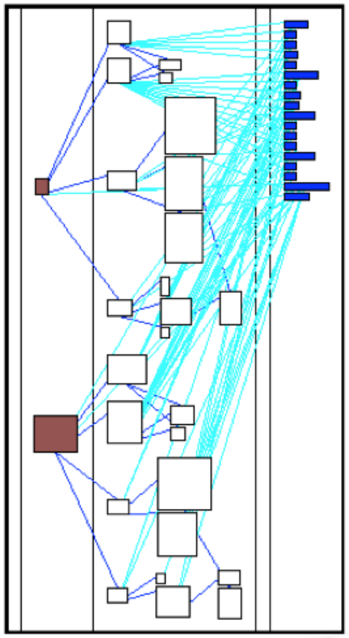
System complexity [Lanza 2003]



We can merely observe the two biggest classes of Swing: JComponent and JTable. However, we should not blame their developers. The root of a graphical user interface framework is inherently complex and difficult to implement. To convince yourself, have a look at the root class of any serious GUI framework.



A class should capture one and only one key abstraction



Example in ArgoUML

We can observe a class which has 2 public methods and many private methods. This class is quite particular in the sense that its private methods are divided into two distinct groups. Each group of private method is used by one public method.

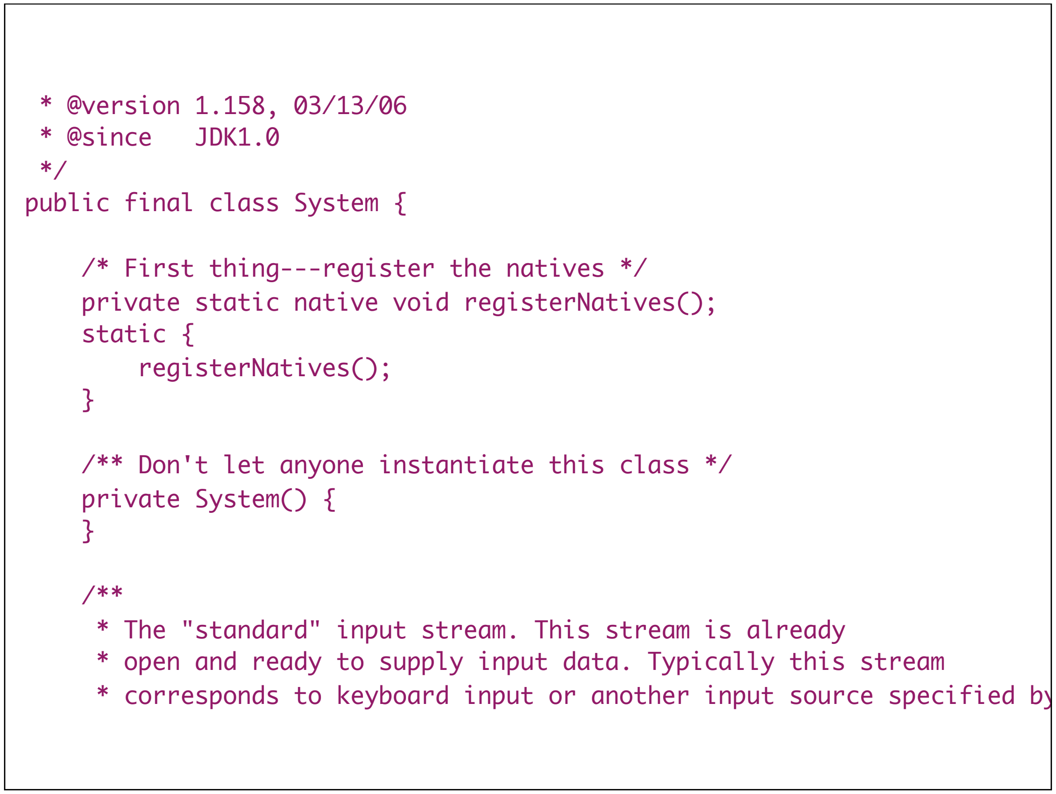
This is an example of a class that offers two distinct functionalities.

Action-Oriented vs Object-Oriented

The god class problem

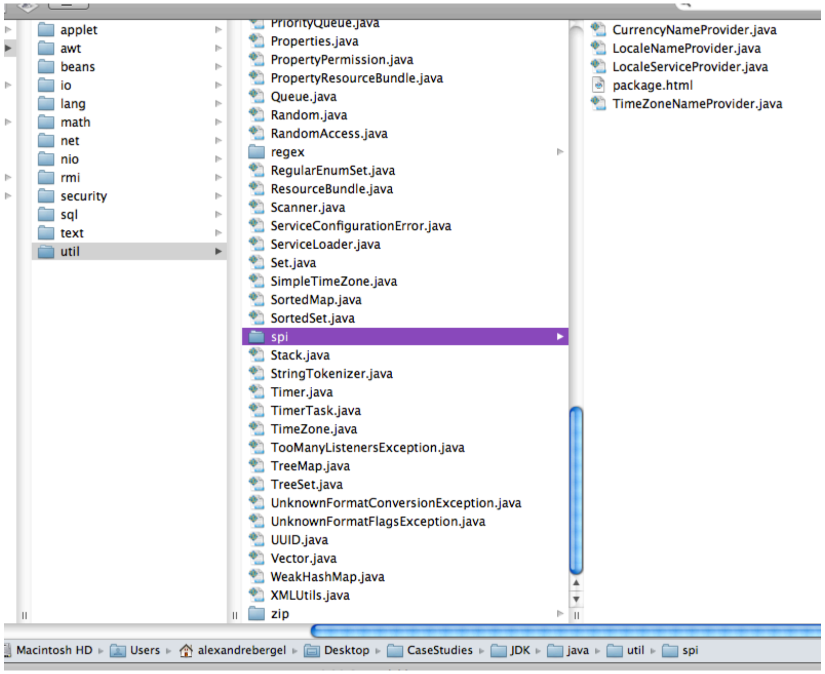
A “god” class performs most of the work, leaving minor details to a collection of trivial classes

Do not create god classes/objects in your system.
Be very suspicious of a class whose name contains
Driver, Manager, System, Subsystem, Utility

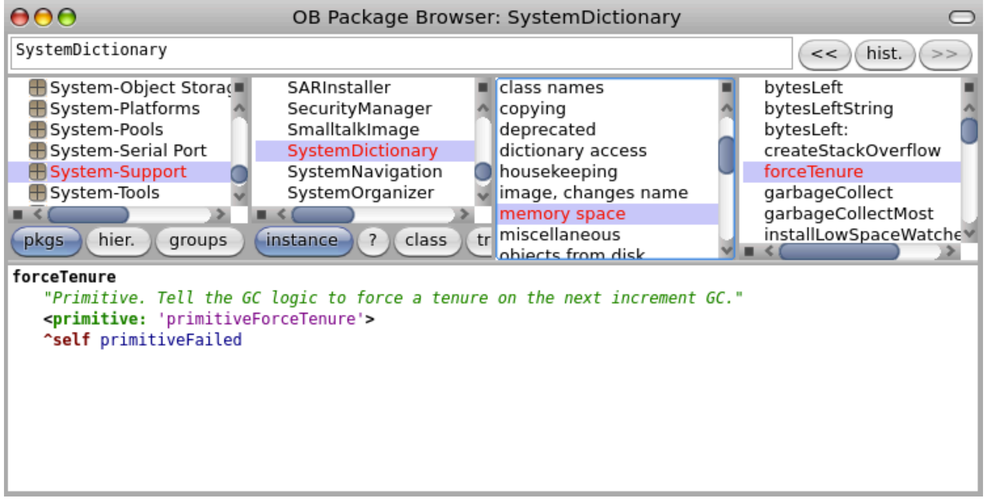


```
 * @version 1.158, 03/13/06
 * @since   JDK1.0
 */
public final class System {

    /* First thing---register the natives */
    private static native void registerNatives();
    static {
        registerNatives();
    }

    /** Don't let anyone instantiate this class */
    private System() {
    }

    /**
     * The "standard" input stream. This stream is already
     * open and ready to supply input data. Typically this stream
     * corresponds to keyboard input or another input source specified by
```

java.lang.System is the perfect example. It offers methods ranging from writing on the standard streams to copying arrays and managing security.



Note that the heuristic given before is also valid for packages. Consider the package java.util. This package contains 229 classes, most of them are collections. But it also contains the classes Data, JapaneseImperialCalendar, Locale, Random, XMLUtils and many more unrelated classes.



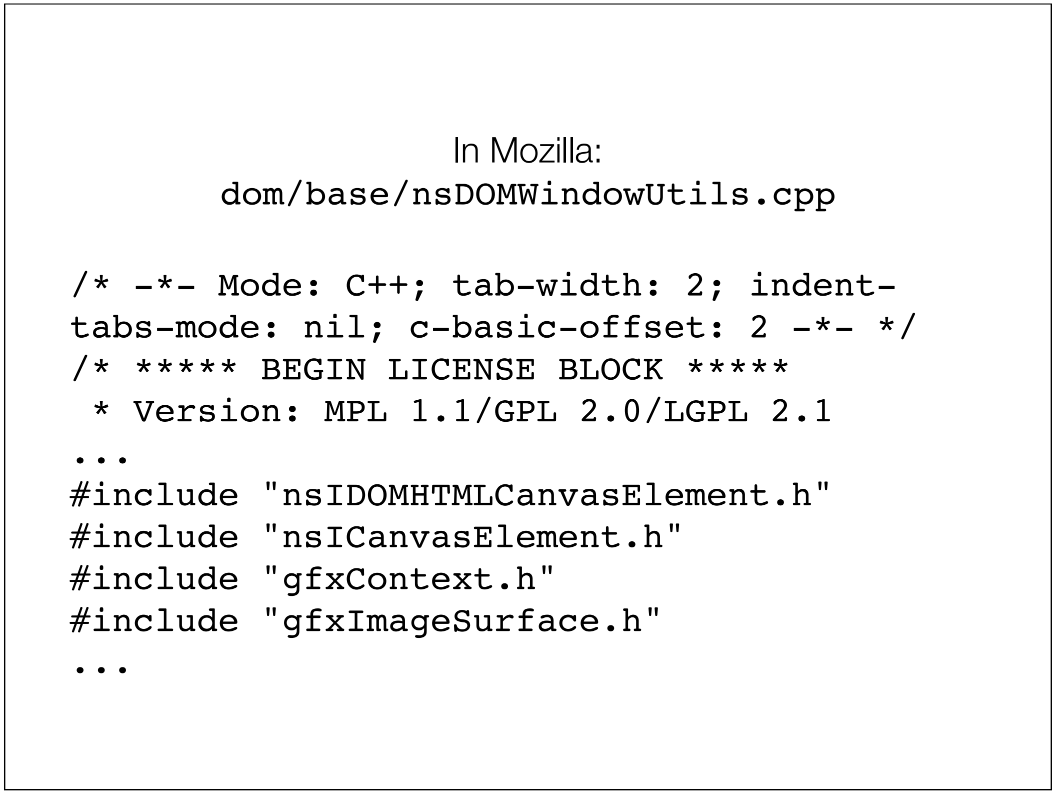
```
forceTenure
   "Primitive. Tell the GC logic to force a tenure on the next increment GC."
   <primitive: 'primitiveForceTenure'>
   ^self primitiveFailed
```

In the Pharo and Squeak Smalltalk languages, the class SystemDictionary is another example of a god class.

SystemDictionary enables one to control the garbage collectors, streaming objects, accessing classes, querying the systems. It has little to do with the notion of dictionary!

In application that consist of an object-oriented model interaction with a user interface, the model should never be dependent on the interface.

The interface should be dependent on the model



In Mozilla:

`dom/base/nsDOMWindowUtils.cpp`

```
/* -*- Mode: C++; tab-width: 2; indent-
tabs-mode: nil; c-basic-offset: 2 -*- */
/* ***** BEGIN LICENSE BLOCK *****
 * Version: MPL 1.1/GPL 2.0/LGPL 2.1
...
#include "nsIDOMHTMLCanvasElement.h"
#include "nsICanvasElement.h"
#include "gfxContext.h"
#include "gfxImageSurface.h"
...
```

Have a look at the definition of the class nsDOMWindowUtils, which is central to the DOM component of Mozilla.
This class has references to some graphical packages, which goes against the idea of having a clean and modular DOM.

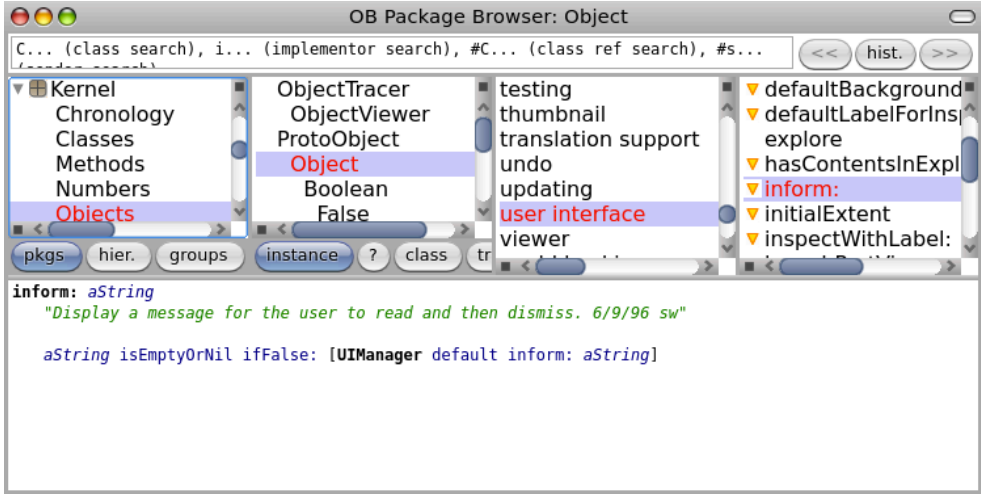
belong to the core

In Mozilla:

`dom/base/nsDOMWindowUtils.cpp`

```
/* -*- Mode: C++; tab-width: 2; indent-
tabs-mode: nil; c-basic-offset: 2 -*- */
/* ***** BEGIN LICENSE BLOCK *****
 * Version: MPL 1.1/GPL 2.0/LGPL 2.1
...
#include "nsIDOMHTMLCanvasElement.h"
#include "nsICanvasElement.h"
#include "gfxContext.h"
#include "gfxImageSurface.h"
...
```

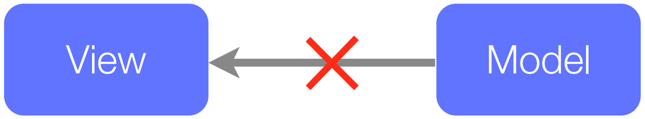
belong to gfx package



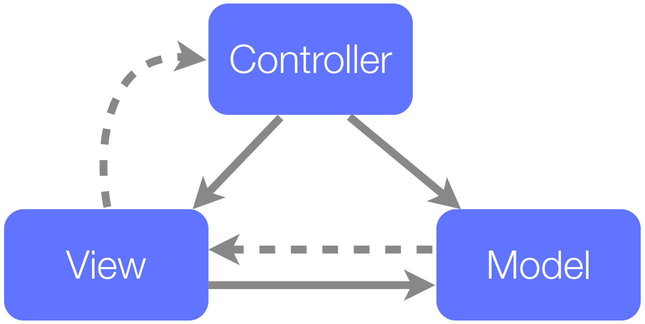
```
inform: aString
	"Display a message for the user to read and then dismiss. 6/9/96 sw"

	aString isEmptyOrNil ifFalse: [UIManager default inform: aString]
```

Another example of the kernel of Pharo. The class Object contains a reference to the UIManager, which belongs to the package ToolBuilder. The method #inform: is clearly wrongly packaged

Wrong dependency



Model-view-controller



Model-view-controller (MVC) is a software architecture, considered as an architectural pattern

Model-view-controller

The MVC pattern isolates the *domain logic* from *the user interface*

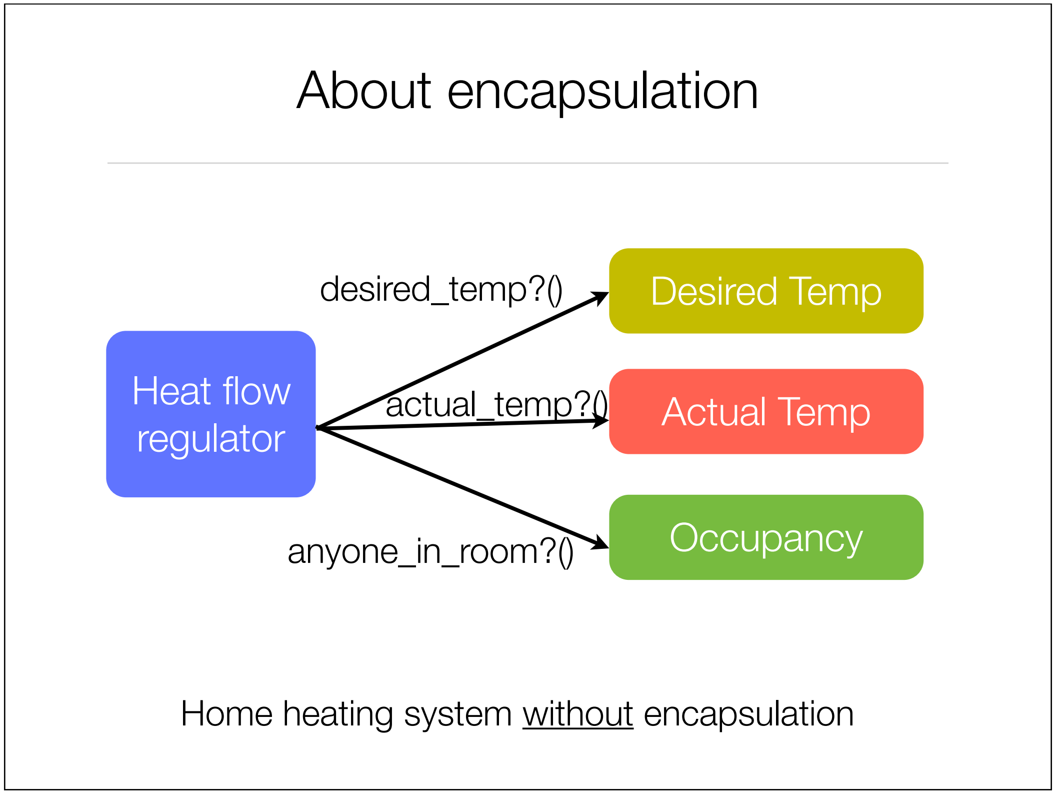
MVC permits

- independent development
- testing
- maintenance (separation of concerns)

Model-view-controller

MVC is typically associated with *frameworks*

Update of the view by the model and/or controller is commonly realized with the *observer/observable design pattern*



About encapsulation

Heat flow regulator

desired_temp?()

Desired Temp

actual_temp?()

Actual Temp

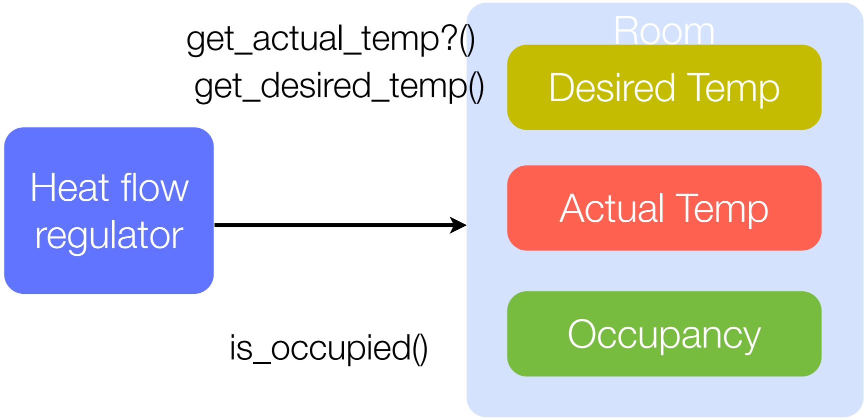
anyone_in_room?()

Occupancy

Home heating system without encapsulation

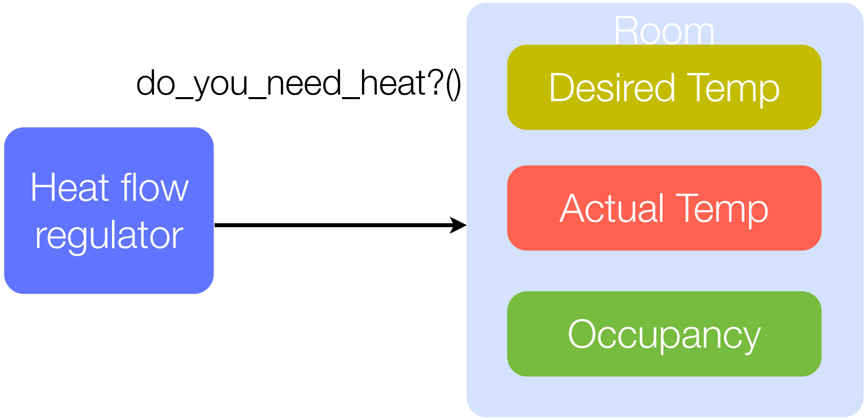
Example of poor system intelligence distribution

About encapsulation



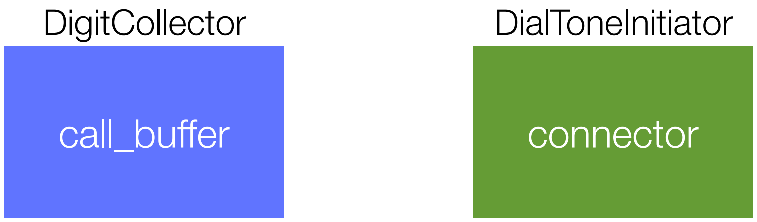
Home heating system with encapsulation

About encapsulation



Home heating system with distributed intelligence

Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb.



DigitCollector

call_buffer

DialToneInitiator

connector

Classes which should be operations

TelephoneCall

call_buffer
connector

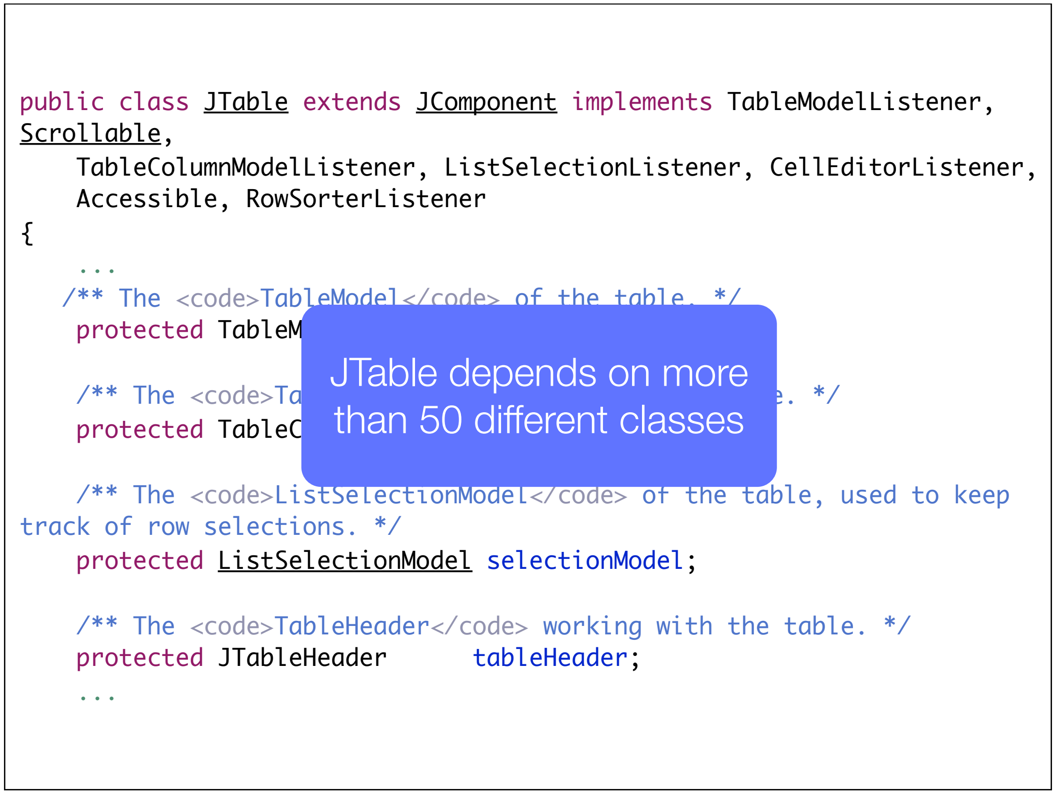
A better design for telephone services

Relat

The relationship between classes and objects

Minimize the number of classes with which another class collaborates

```
public class JTable extends JComponent implements TableModelListener, 
Scrollable,
    TableColumnModelListener, ListSelectionListener, CellEditorListener,
    Accessible, RowSorterListener
{
    ...
   /** The <code>TableModel</code> of the table. */
    protected TableModel        dataModel;

    /** The <code>TableColumnModel</code> of the table. */
    protected TableColumnModel  columnModel;

    /** The <code>ListSelectionModel</code> of the table, used to keep 
track of row selections. */
    protected ListSelectionModel selectionModel;

    /** The <code>TableHeader</code> working with the table. */
    protected JTableHeader      tableHeader;
    ...
```



```
public class JTable extends JComponent implements TableModelListener,
Scrollable,
    TableColumnModelListener, ListSelectionListener, CellEditorListener,
    Accessible, RowSorterListener
{
    ...
   /** The <code>TableModel</code> of the table. */
    protected TableM

    /** The <code>Ta                                     e. */
    protected TableC

    /** The <code>ListSelectionModel</code> of the table, used to keep
track of row selections. */
    protected ListSelectionModel selectionModel;

    /** The <code>TableHeader</code> working with the table. */
    protected JTableHeader      tableHeader;
    ...
```

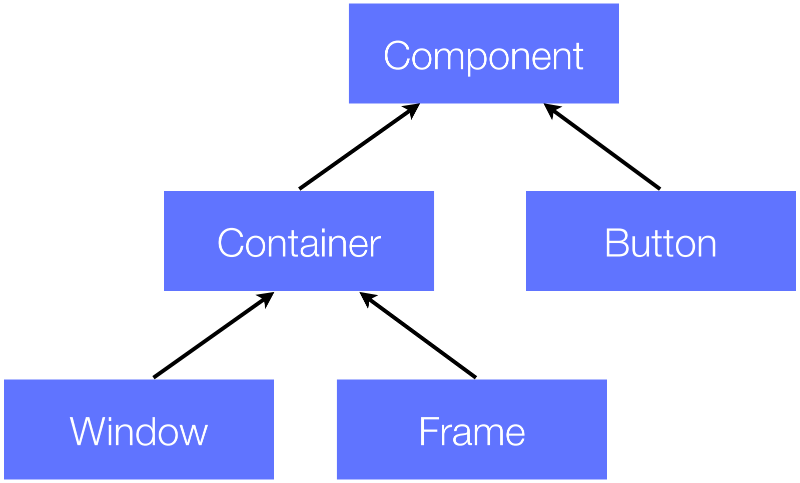
JTable depends on more than 50 different classes

The inheritance relationship

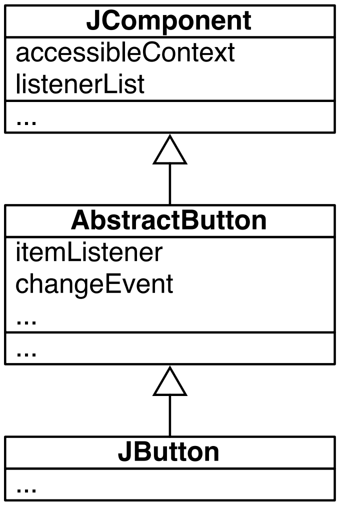
Inheritance

The Inheritance relationship is one of the most important relationships within object-orientation

It is best used to capture the *a-kind-of* relationship between classes



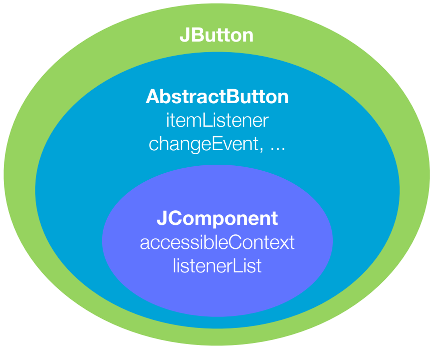
Example of the core of java.awt



JComponent
accessibleContext
listenerList
...

AbstractButton
itemListener
changeEvent
...
...

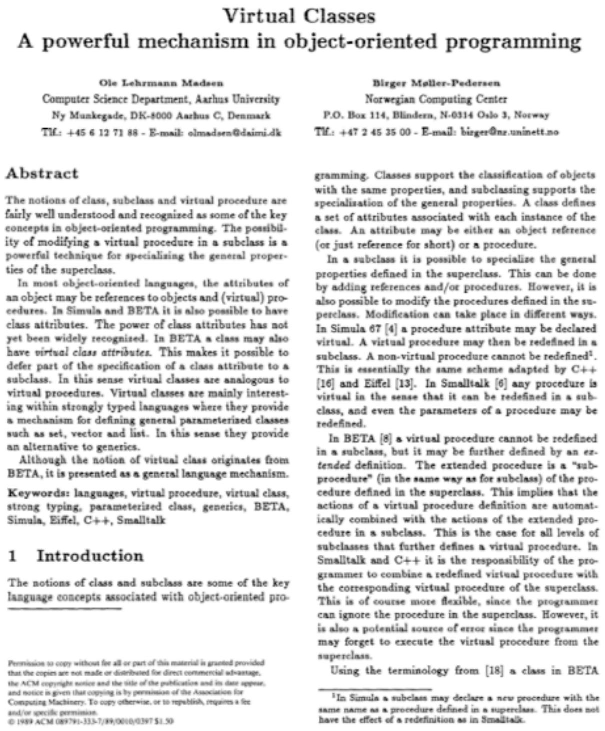
JButton
...



JButton

AbstractButton
itemListener
changeEvent, ...

JComponent
accessibleContext
listenerList



Virtual Classes
A powerful mechanism in object-oriented programming

Ole Lehrmann Madsen
Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
Tlf.: +45 6 12 71 88 - E-mail: olmadsen@daimi.dk

Birger Møller-Pedersen
Norwegian Computing Center
P.O. Box 114, Blindern, N-0314 Oslo 3, Norway
Tlf.: +47 2 45 35 00 - E-mail: birger@nr.uninett.no

Abstract

The notions of class, subclass and virtual procedure are fairly well understood and recognized as some of the key concepts in object-oriented programming. The possibility of modifying a virtual procedure in a subclass is a powerful technique for specializing the general properties of the superclass.

In most object-oriented languages, the attributes of an object may be references to objects and (virtual) procedures. In Simula and BETA it is also possible to have class attributes. The power of class attributes has not yet been widely recognized. In BETA a class may also have *virtual class attributes*. This makes it possible to defer part of the specification of a class attribute to a subclass. In this sense virtual classes are analogous to virtual procedures. Virtual classes are mainly interesting within strongly typed languages where they provide a mechanism for defining general parameterized classes such as set, vector and list. In this sense they provide an alternative to generics.

Although the notion of virtual class originates from BETA, it is presented as a general language mechanism.

Keywords: languages, virtual procedure, virtual class, strong typing, parameterized class, generics, BETA, Simula, Eiffel, C++, Smalltalk

1 Introduction

The notions of class and subclass are some of the key language concepts associated with object-oriented programming. Classes support the classification of objects with the same properties, and subclassing supports the specialisation of the general properties. A class defines a set of attributes associated with each instance of the class. An attribute may be either an object reference (or just reference for short) or a procedure.

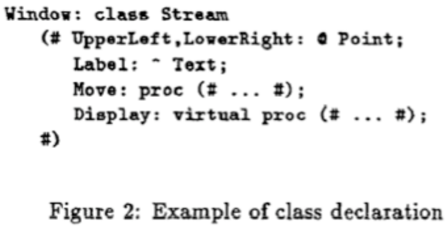
In a subclass it is possible to specialize the general properties defined in the superclass. This can be done by adding references and/or procedures. However, it is also possible to modify the procedures defined in the superclass. Modification can take place in different ways. In Simula 67 [4] a procedure attribute may be declared virtual. A virtual procedure may then be redefined in a subclass. A non-virtual procedure cannot be redefined[1]. This is essentially the same scheme adapted by C++ [16] and Eiffel [13]. In Smalltalk [6] any procedure is virtual in the sense that it can be redefined in a subclass, and even the parameters of a procedure may be redefined.

In BETA [8] a virtual procedure cannot be redefined in a subclass, but it may be further defined by an *extended* definition. The extended procedure is a "subprocedure" (in the same way as for subclass) of the procedure defined in the superclass. This implies that the actions of a virtual procedure definition are automatically combined with the actions of the extended procedure in a subclass. This is the case for all levels of subclasses that further defines a virtual procedure. In Smalltalk and C++ it is the responsibility of the programmer to combine a redefined virtual procedure with the corresponding virtual procedure of the superclass. This is of course more flexible, since the programmer can ignore the procedure in the superclass. However, it is also a potential source of error since the programmer may forget to execute the virtual procedure from the superclass.

Using the terminology from [18] a class in BETA

[1] In Simula a subclass may declare a *new* procedure with the same name as a procedure defined in a superclass. This does not have the effect of a redefinition as in Smalltalk.

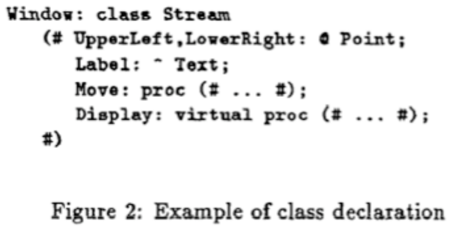
@OOPSLA'89



```
Window: class Stream
   (# UpperLeft,LowerRight: @ Point;
      Label: ^ Text;
      Move: proc (# ... #);
      Display: virtual proc (# ... #);
   #)
```

Figure 2: Example of class declaration

“In Figure 2 an example of a class is given. Class Window is described as a subclass of class Stream. ...”

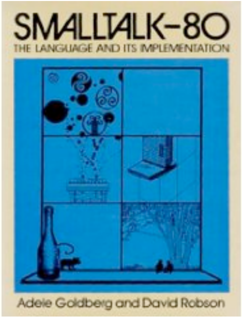


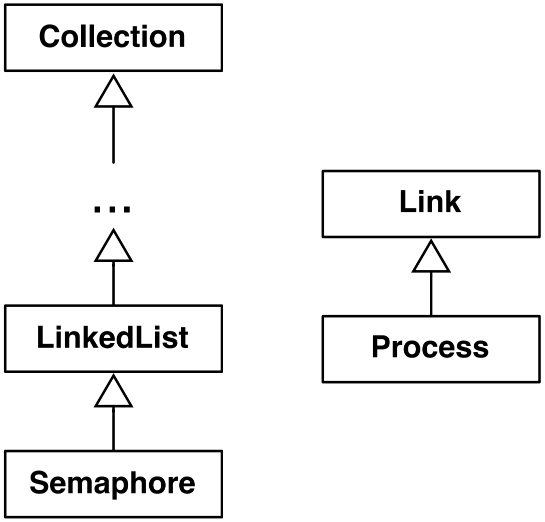
```
Window: class Stream
   (# UpperLeft,LowerRight: @ Point;
      Label: ^ Text;
      Move: proc (# ... #);
      Display: virtual proc (# ... #);
   #)
```

Figure 2: Example of class declaration

“In Figure 2 an example of a class is given. Class Window is described as a subclass of class Stream. ...”

Do you think a window can be considered as a stream?


SMALLTALK-80
THE LANGUAGE AND ITS IMPLEMENTATION
Adele Goldberg and David Robson



Collection

...

LinkedList

Semaphore

Link

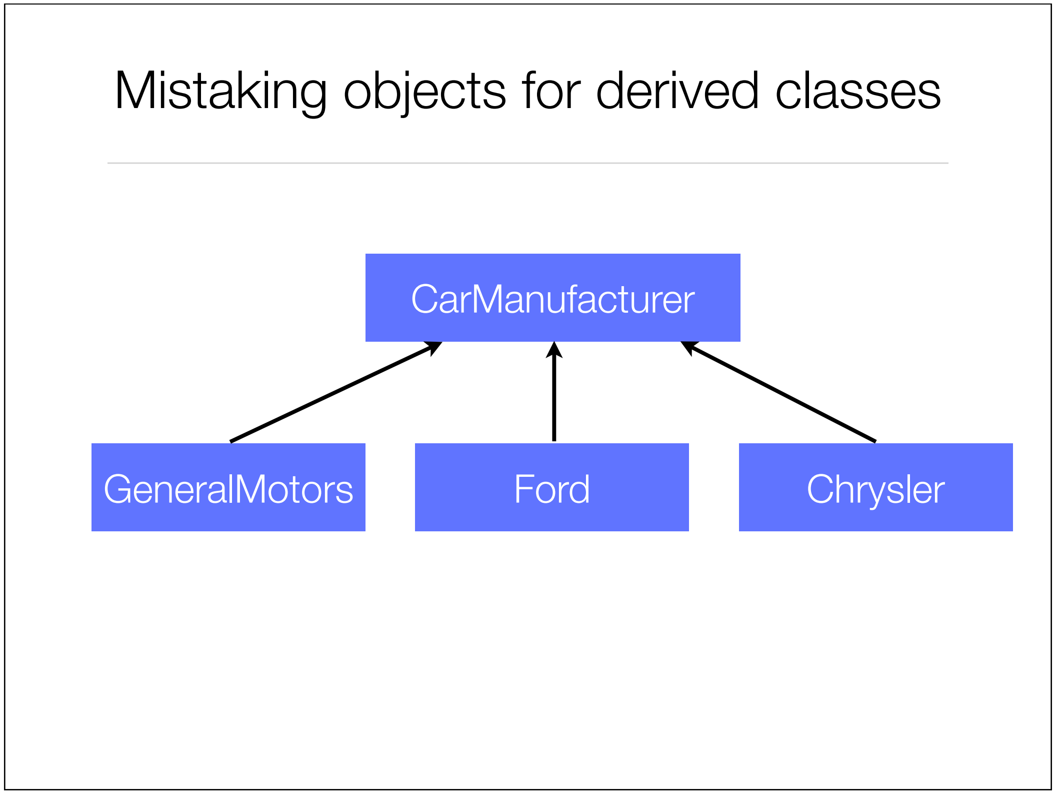
Process

Probably a semaphore can be seen as a collection, but is it worth subclassing LinkedList in that case?

All abstract classes must be base classes

All abstract classes must be base classes

Since an abstract class cannot be instantiated, does it make sense to have an abstract class leaf?



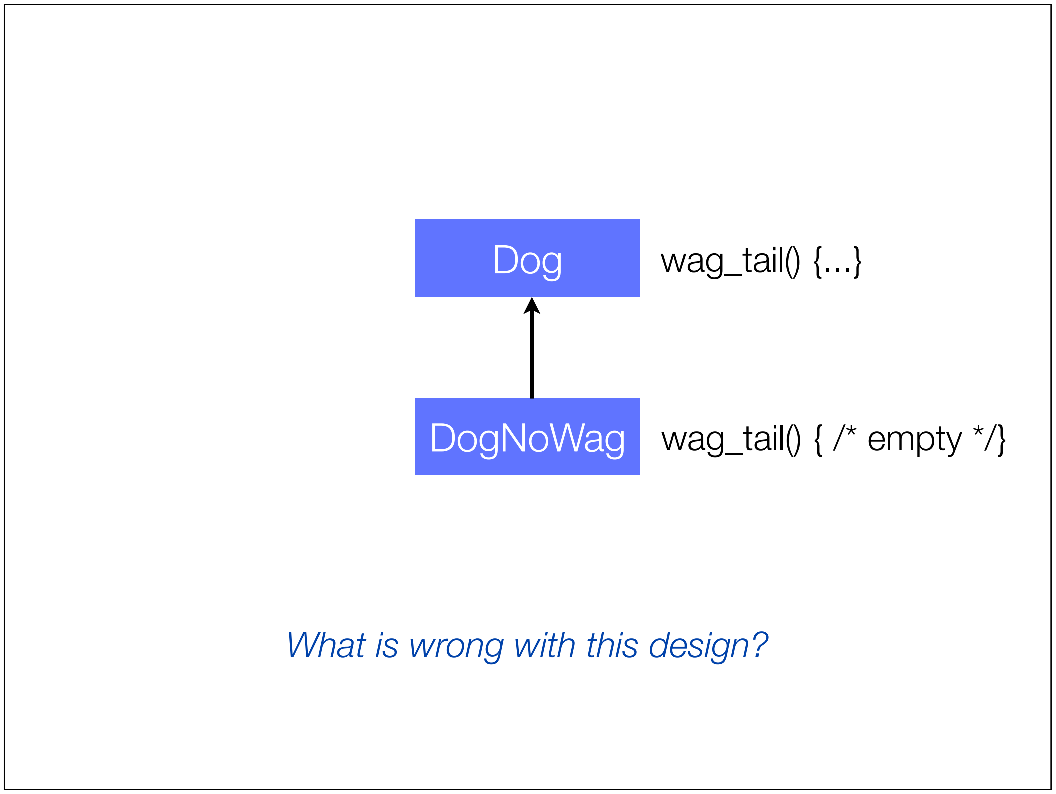
Mistaking objects for derived classes

Consider the inheritance hierarchy given on this slide. At first view the inheritance hierarchy looks correct. GeneralMotors, Ford and Chrysler are all special types of car manufacturers. On second thought, is GeneralMotors really a special type of car manufacturer? Or is it an example of a car manufacturer? This is a classic error and it causes proliferation of classes.

how many GeneralMotors objects are there? Ford objects? Chrysler objects? The answer for all three classes if one. In this case they should have been objects.

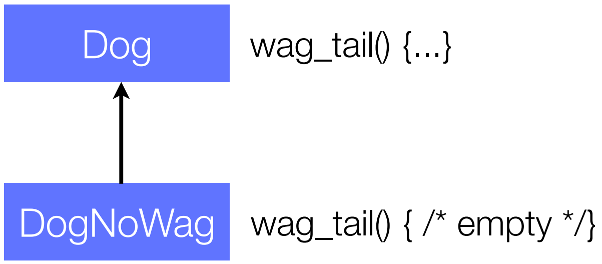
Keep in mind that not all derived classes that have only one instance in your system are manifestations of this error, but many will be.

It should be illegal for a derived class to override a base class method with a NOP method, that is, a method that does nothing



What is wrong with this design?

Consider a class Dog. The behaviors that all Dogs know how to carry out is bark, chase_cats and wag_tail. Consider that we want to have a dog that does not wag its tail, let’s say DogNoWag. This new class is exactly like a Dog except it doesn’t know how to wag its tail. A solution could be to have DogNoWag inherit from Dog and override the wag_tail method with an empty method (NOP).



Dog

wag_tail() {...}

DogNoWag

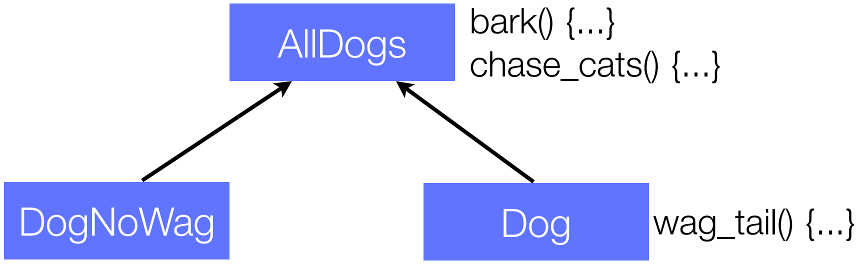
wag_tail() { /* empty */}

This design does not capture a logical relationship

It implies the following statements:

- All dogs know how to wag their tails
- DogNoWag is a special type of dog
- DogNoWag does not know how to wag its tail

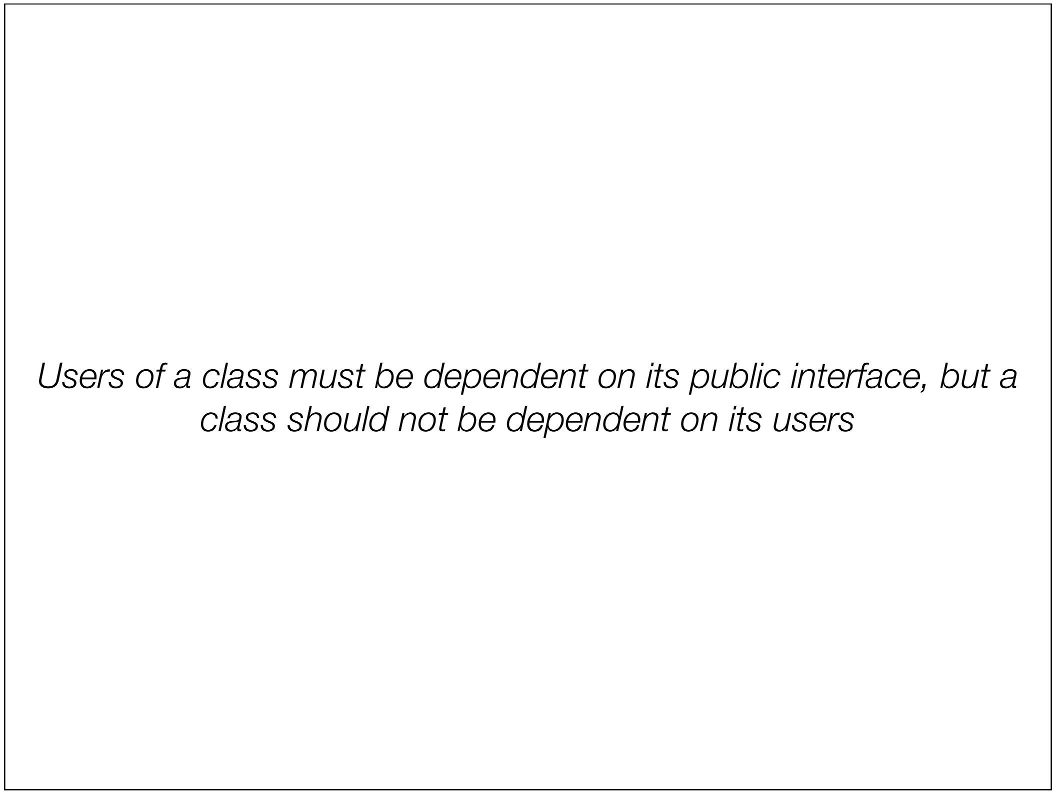
The rules of classic logic are not being obeyed



Dogs and their tails...

Other heuristics

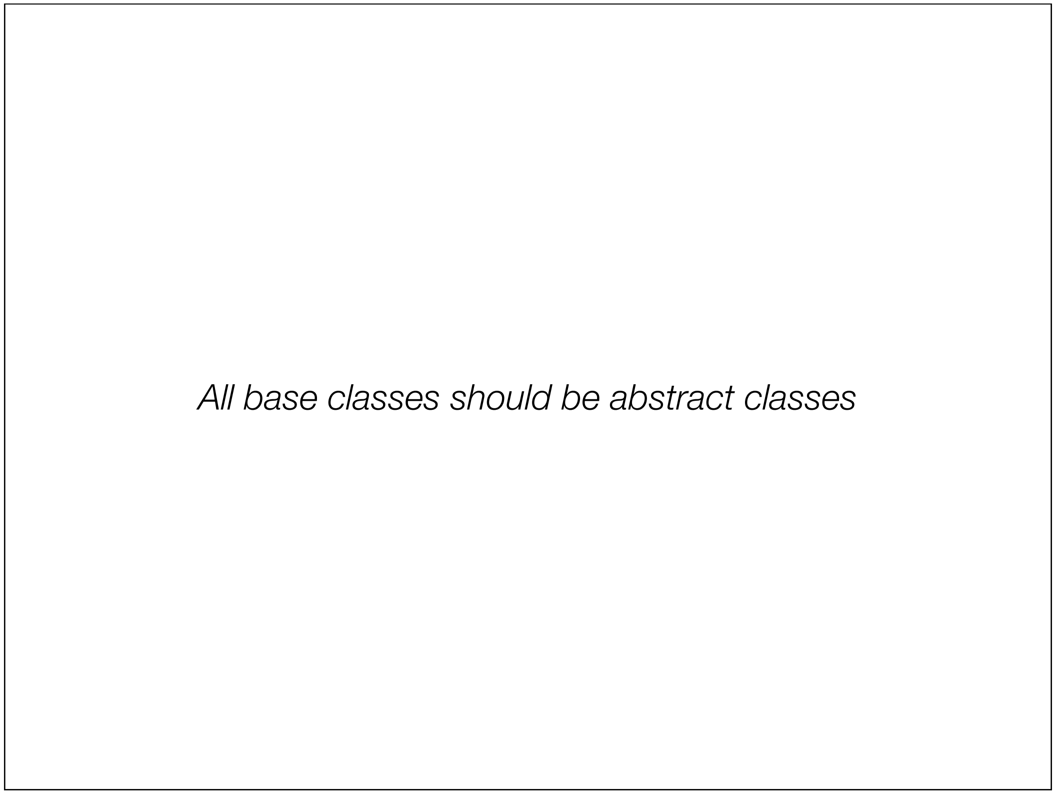
When building an inheritance hierarchy, try to construct reusable frameworks rather than reusable components



Users of a class must be dependent on its public interface, but a class should not be dependent on its users

Minimize the number of message sends between a class and its collaborator

A class must know what it contains, but it should not know who contains it



All base classes should be abstract classes

All base classes should be abstract classes

Not everybody will agree with this one (including me),
but this heuristic deserves some attention

What you should know!

What is the difference between encapsulation and information hiding?

What is the difference between an object and a class

Why is it important to have hidden data?

Why visualization is useful to understand large code?

What is a god class?

What is model-view-controller?

Why a leaf class cannot be abstract?

Can you answer these questions?

- Why information hiding and encapsulation favor maintainability?
- Why god classes are dangerous for an application health?
- Why MVC requires implementing the observer/observable design pattern?
- What are the criteria to assess the quality of a class hierarchy?

License

http://creativecommons.org/licenses/by-sa/2.5