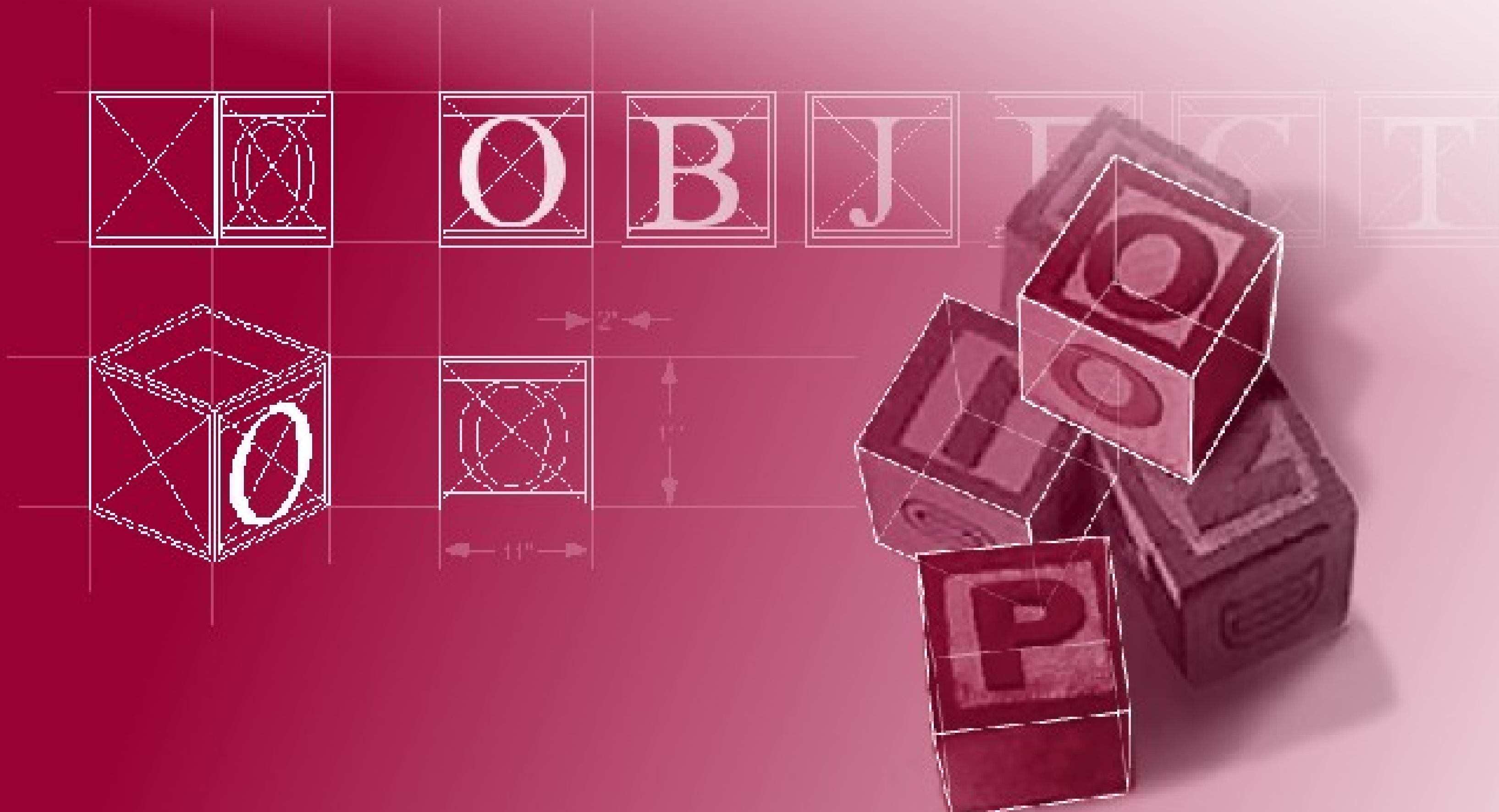
Object-Oriented Programming:
Introduction to Analysis and Design



Zeegee Software Inc.

RELEASE

INTRODUCTION

Object-oriented design (OOD) techniques allow developers to…

– **Identify what classes** a given application will need (abstraction).

– **Determine how to organize the classes** in a class hierarchy (also abstraction).

– **Identify the instance variables and methods** that should be defined for the classes, and how instances of those class interact (association).

– **Evaluate the "goodness" of a given design** by its use of inheritance, encapsulation, etc. (coupling and cohesion).

IDENTIFYING CLASSES AND OBJECTS

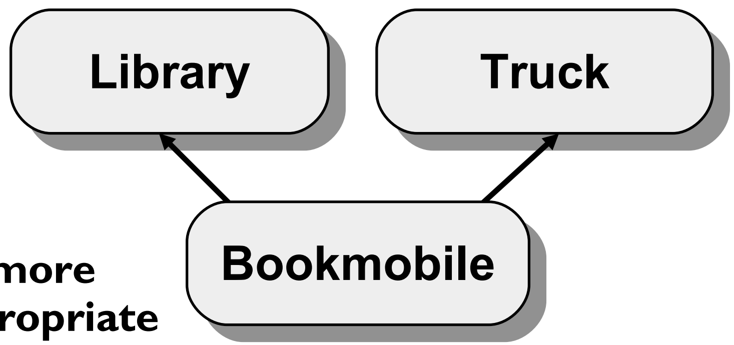
- Association
- Abstraction

What is association?

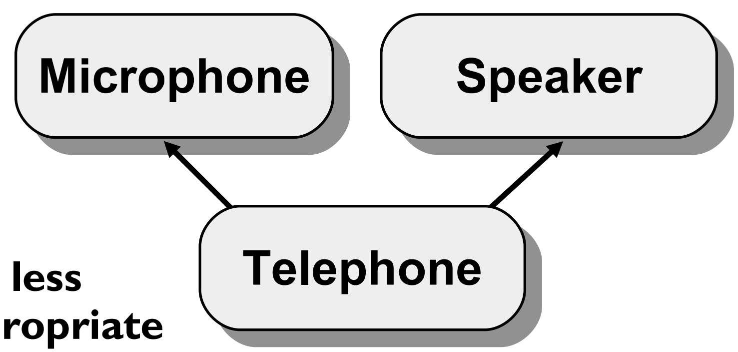
- Just as inheritance links classes together, ***association*** links **instances** of classes together:
  - An object of class A [is] an object of class B: *inheritance*.
  - An object of class A [has-part, talks-to, contains] an object of class B: *association*.
- Commonly, a binary relationship between two classes.
- Association determines:
  - The instance variables of classes.
  - The "contents" of collection classes.

Choosing association vs. inheritance

Inheritance

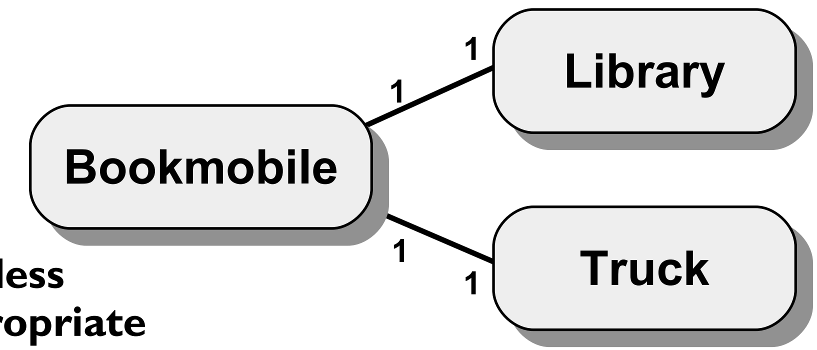


more appropriate

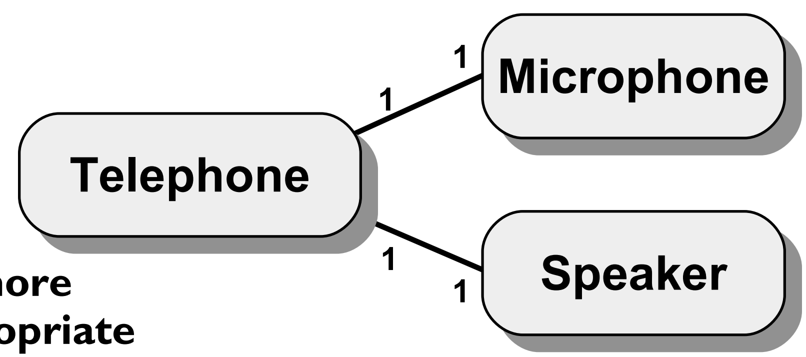


less appropriate

Association



less appropriate

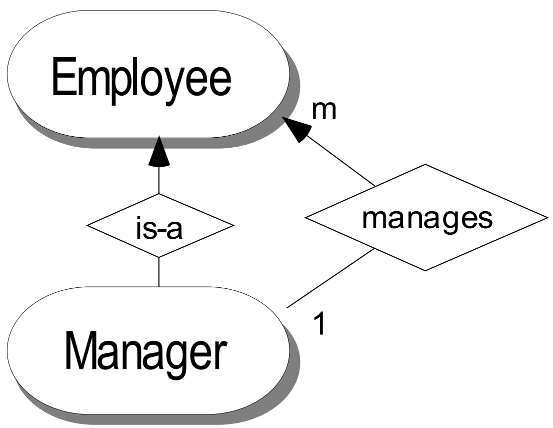


more appropriate

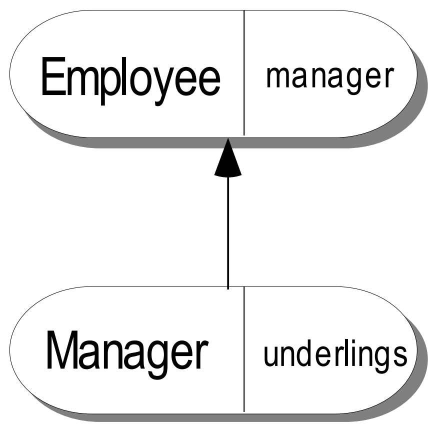
Who implements the association?

Both objects, if each object must send many messages to the other:

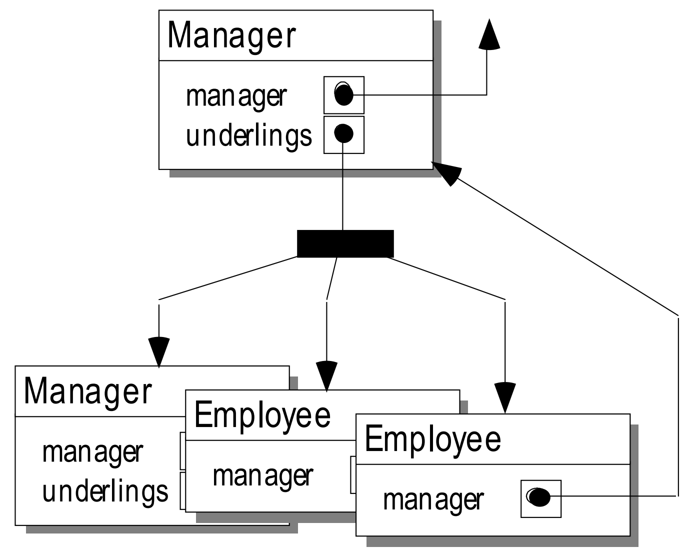
E-R diagram



Class design



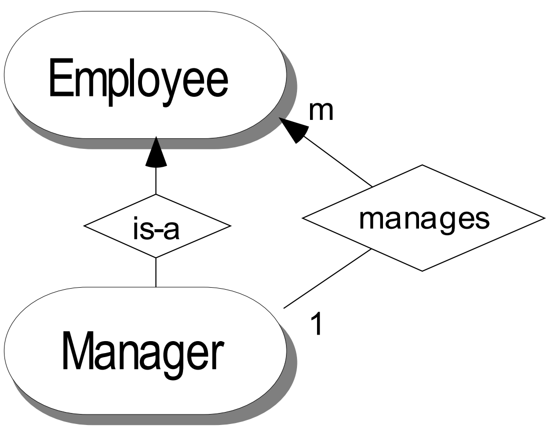
Sample objects



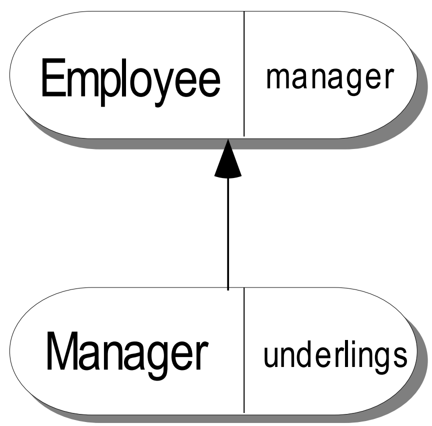
Who implements the association?

Only one object, especially in the case of "part-of" or "contains" relationships, when one object is much simpler than the other:

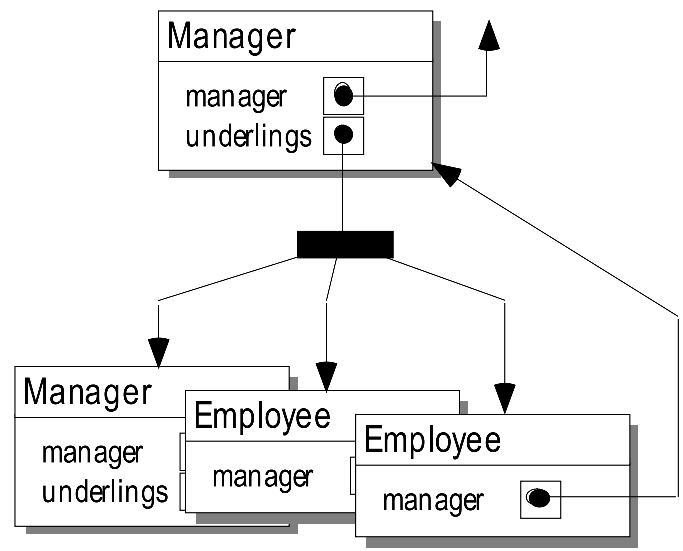
E-R diagram



Class design



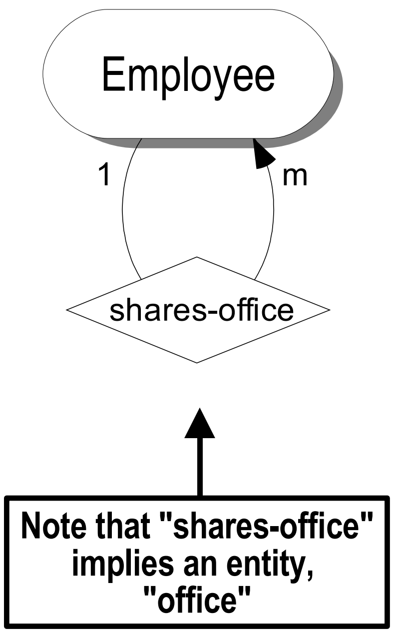
Sample objects



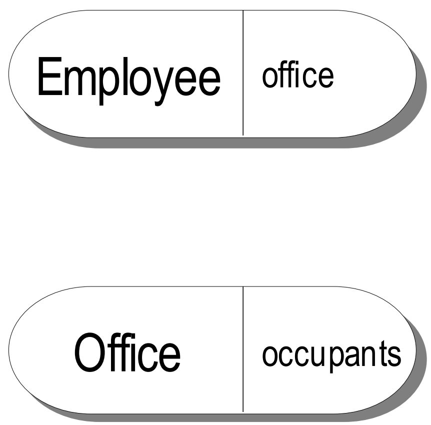
Who implements the association?

Neither object, if the association is overly complex; e.g., if it implies the existence of other entities which would *really* be managing the association:

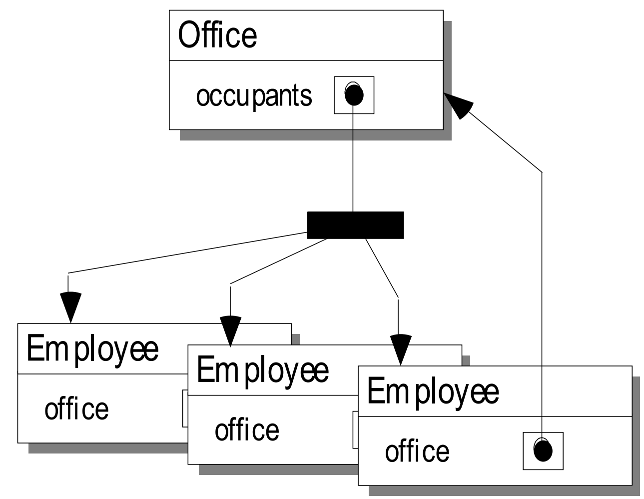
E-R diagram



Class design



Sample objects

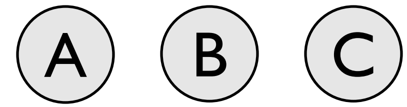


What is abstraction?

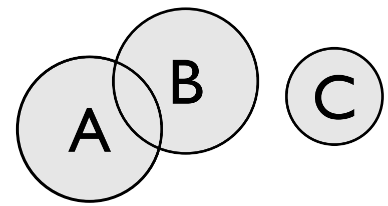
- One job of an OO developer is to take a problem domain, and from it deduce which classes will be needed, and what instance/variables go in each class.
- Generally easy to identify the lowest-level classes, but we often want to make use of inheritance!
- ***Abstraction*** is the technique of deciding…
  - What classes do we need?
  - How do we organize our class hierarchy?
  - Which variables/methods do we put in the superclasses, and which do we put in the subclasses?

Basic approach to abstraction

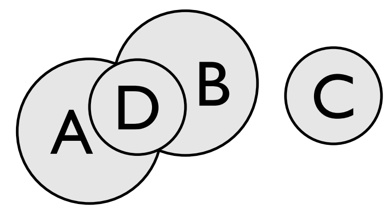
1. Take a collection of proposed classes.



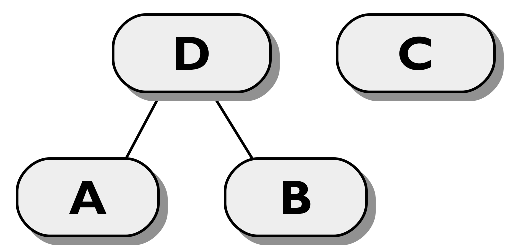
2. Identify common attributes (instance variables) and behaviors (methods).



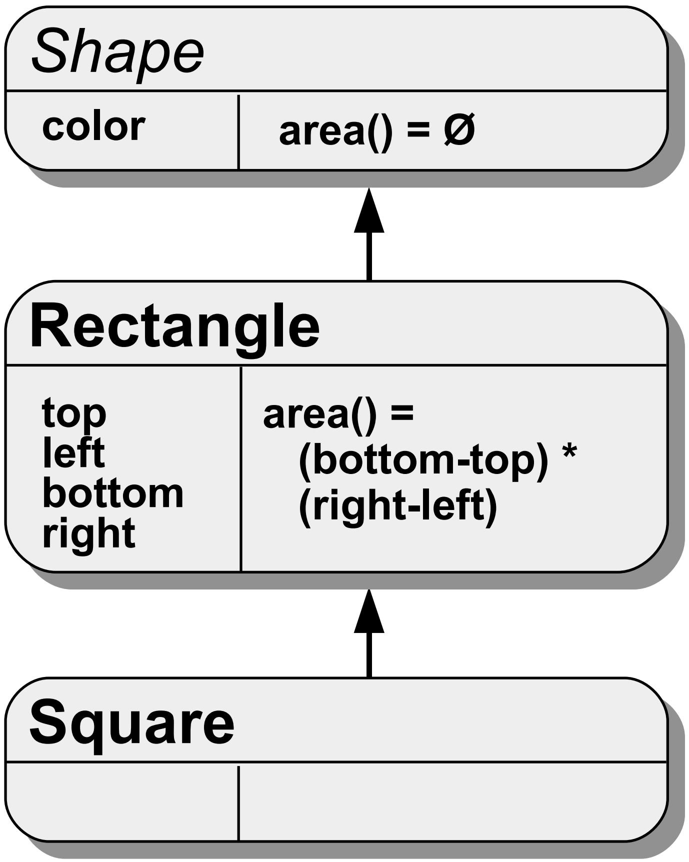
3. Remove the common attributes and behaviors, and put them in a new class.



4. New class is a base class which contains the common elements; derived classes contain what's left.



By functionality...

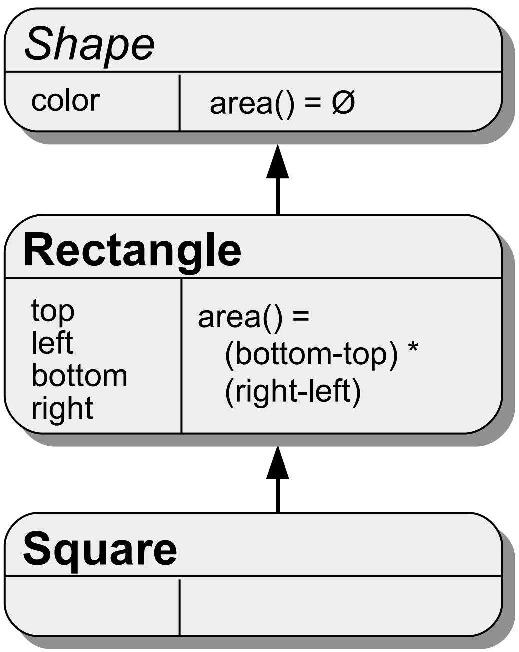


Strategy:

We look at the messages we want objects of each class to respond to, and organize our class hierarchy accordingly.

Emphasis is on the **role the object plays** in the problem domain.

By functionality...



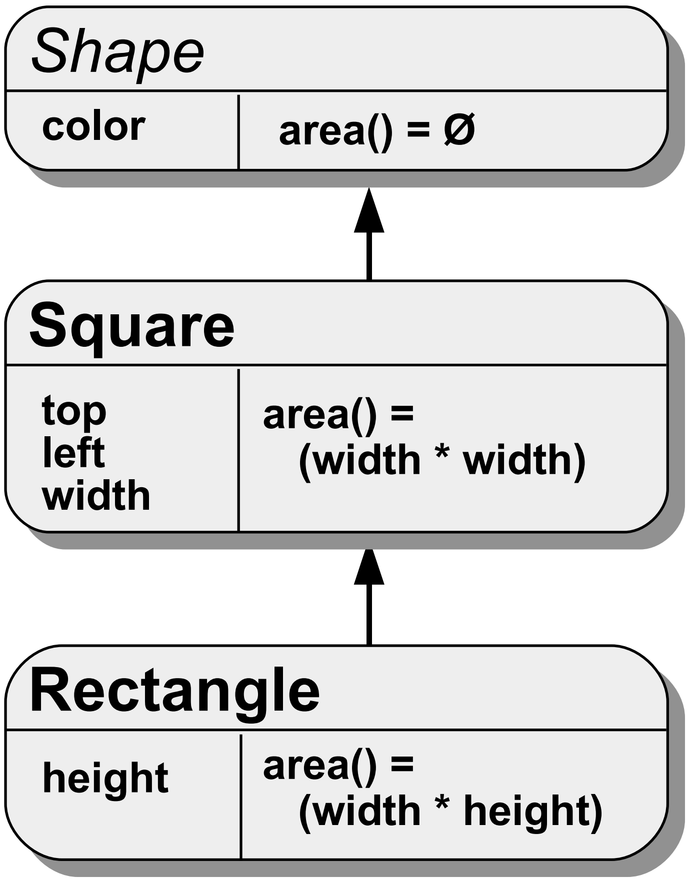
Advantages

- Good selection of methods for each class (strong functional cohesion).
- Intuitive class hierarchy, tending to reflect "reality."
- Facilitates addition of new subclasses after initial design and implementation.

Disadvantages

- Structure of derived classes may be less than optimal.
- Derived classes may contain superfluous attributes.

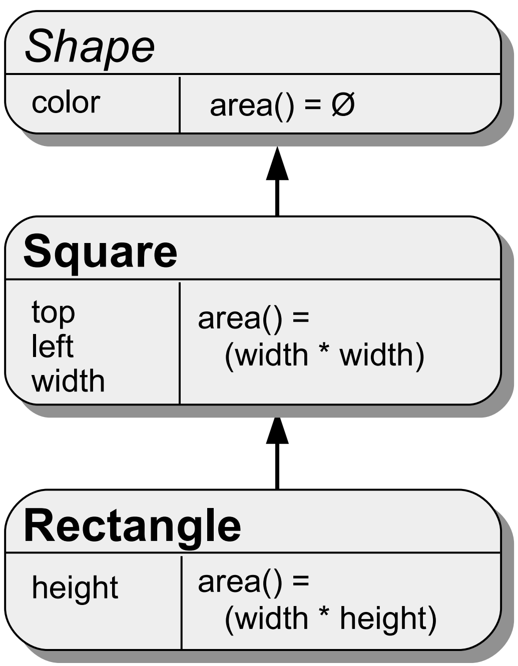
By structure...



Strategy:

We look at the attributes we want objects of each class have, and organize our class hierarchy for efficiency.

By structure...



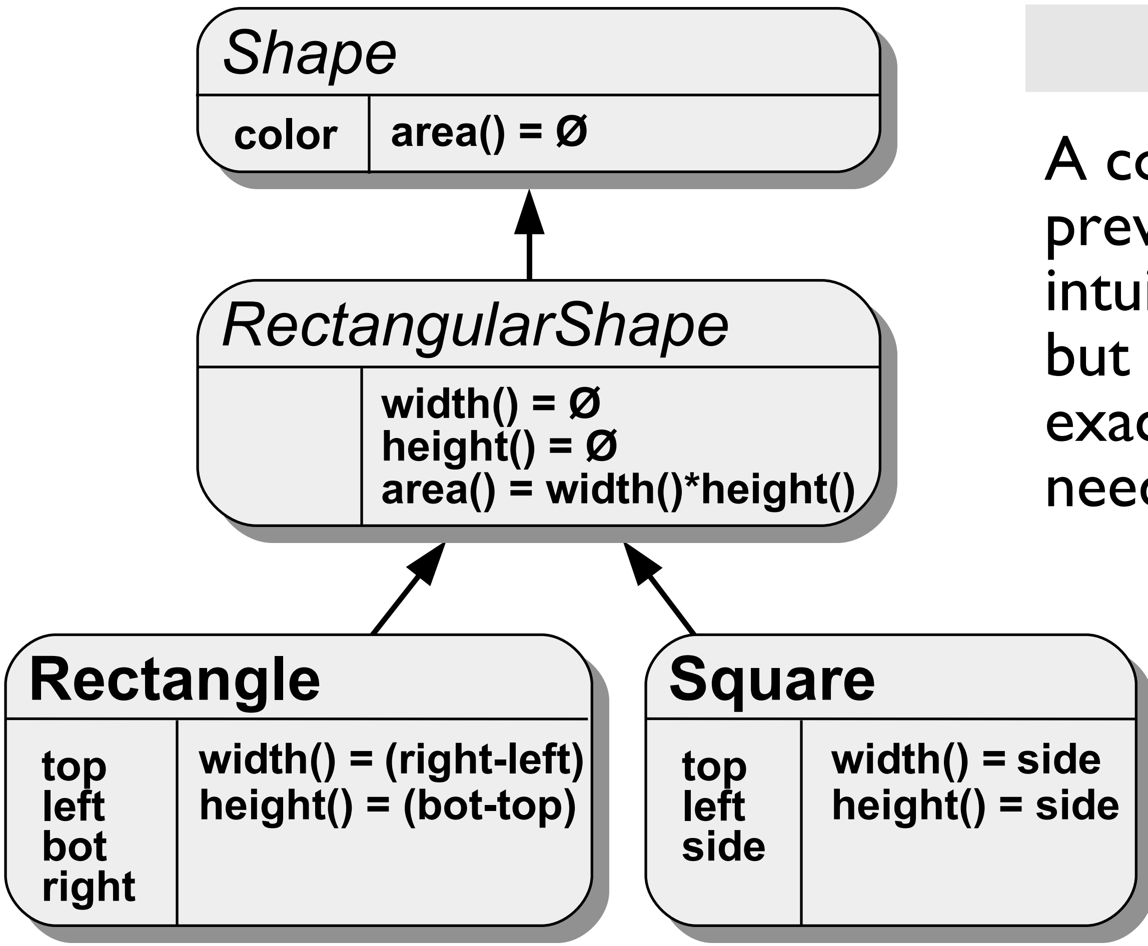
Advantages

- Efficient structure thoroughout class hierarchy
- Useful for applications with a large number of objects and tight space.

Disadvantages

- Class hierarchy strongly driven by implementation of classes at all levels.
- Requires most/all classes to be known a priori.
- Very unintuitive hierarchy organization.

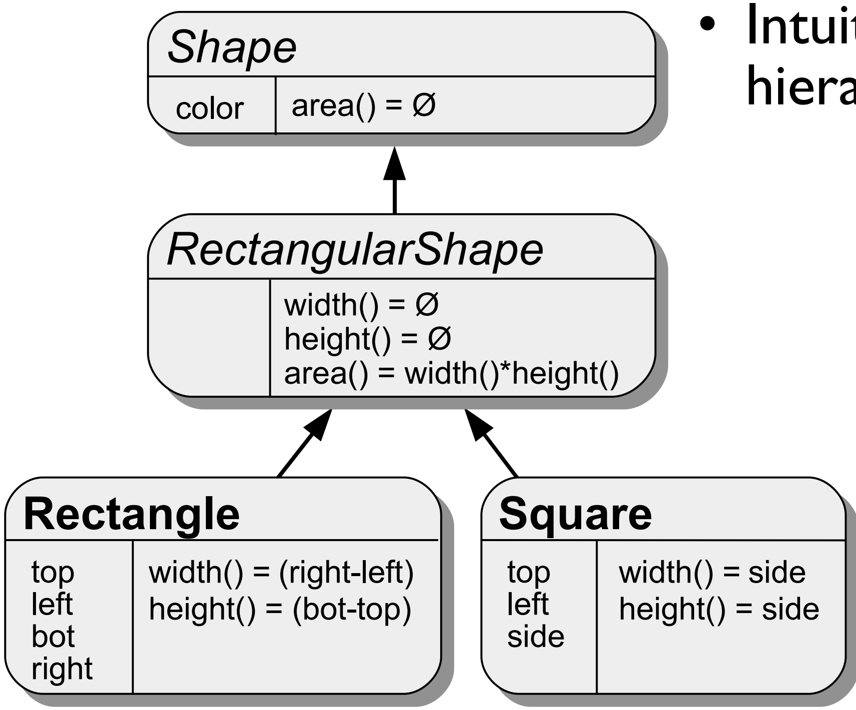
By functionality and structure...



Strategy:

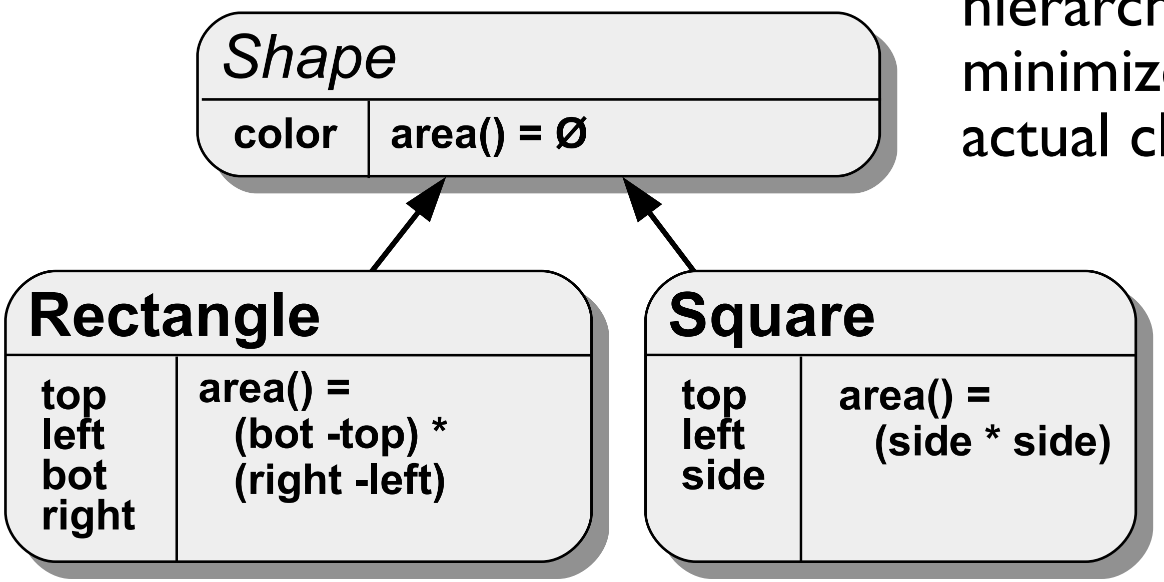
A combination of the previous two: we look for intuitive class organization, but also try to model exactly the attributes needed.

By functionality and structure...



Advantages	Disadvantages
• Efficient structure and functionality. • Intuitive class hierarchy.	• More difficult to design well. • Tends to add abstract classes to hierarchy. • Overuse of abstracted methods may sacrifice performance.

Minimal or none...



Strategy:

Create as flat a class hierarchy as possible, and minimize the number of actual classes.

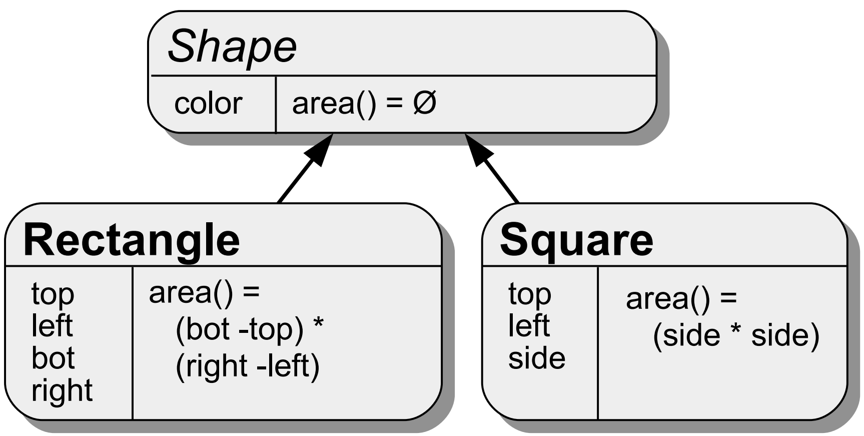
Minimal or none...

Advantages

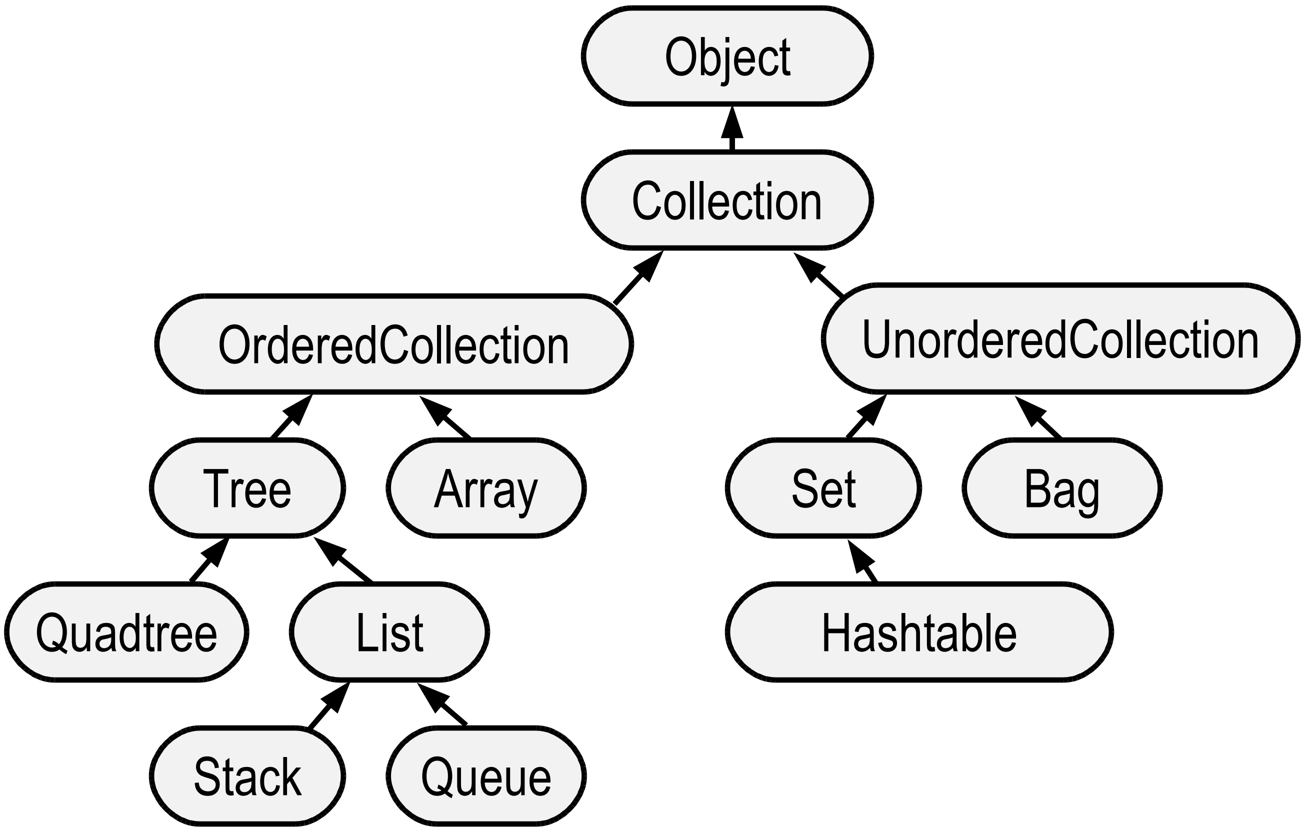
- Simplifies design.
- Eliminates inheritance of unwanted structure.
- Easier to debug.

Disadvantages

- Fails to capitalize on benefits of inheritance
- Introduces code redundancy.
- Does not preserve "real-world" relationships between classes.



Don't overdo it!



EXERCISE: Abstraction

Construct a hierarchy for the following classes:

PNTs are image files which have a 2-dimensional array of 1-byte pixels, and a color lookup table.

JPEGs are image files which have a 2-dimensional array of 3-byte (RGB) pixels.

GIFs are image files which have a 2-dimensional array of 1-byte pixels, and a color lookup table. One special color may be designated the *transparent* color.

HTMLs are files of ASCII text, where certain markup sequences denote headings, font, etc.

PICTs are image files which represent graphical objects as combinations of rectangles, ellipses, lines, etc.

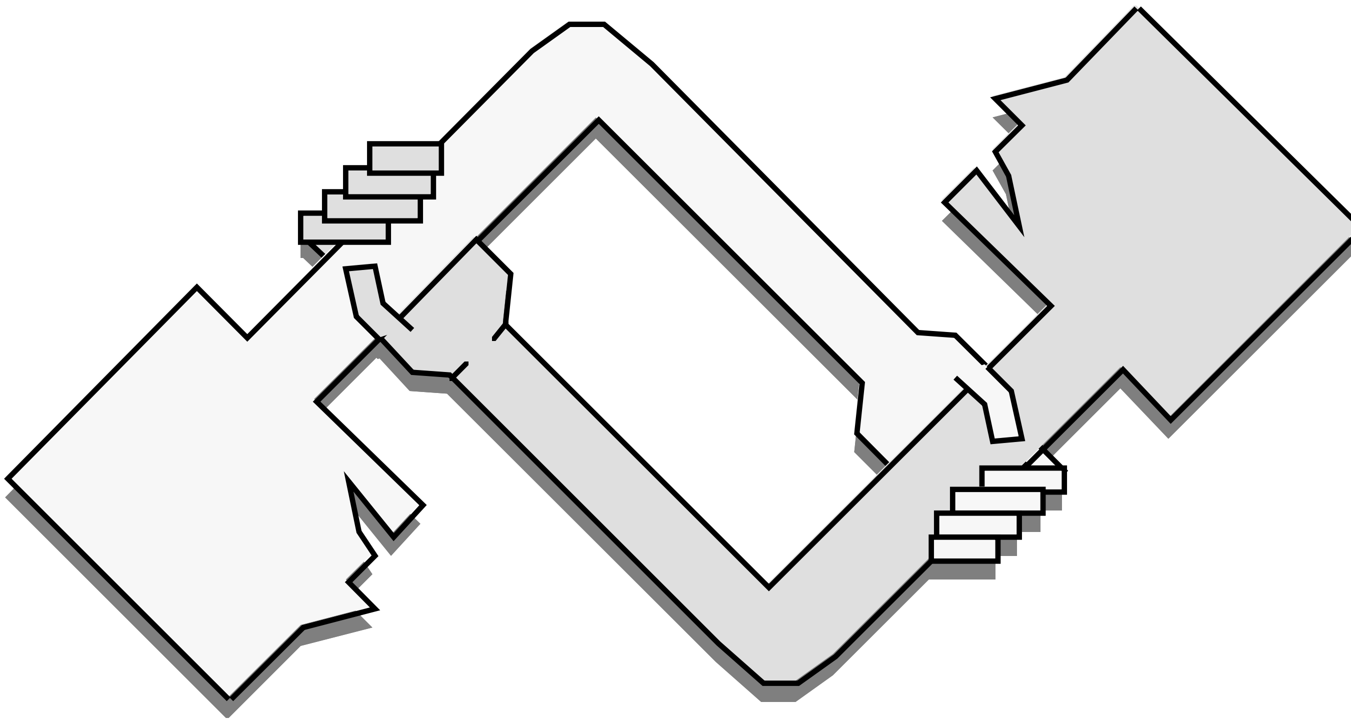
EVALUATING A DESIGN

- Coupling
- Cohesion

What is coupling?

- ***Coupling*** is a rough measure of how strongly two objects are related.
- A **high degree of coupling** (also called ***strong coupling***) between two classes is indicated if…
  – Both classes depend on each other's **internal structure**.
  – The instances of both classes **send many different messages** to each other.
  – The messages passed between the two classes are **complex**; e.g., certain messages must be sent in a sequence, the messages take many parameters, etc.

Strong coupling, in short



The Law of Demeter

For loose coupling and modifiability, follow this rule...

A method of a class should not depend in any way on the structure of any class, except its own.

In the weak form: direct access to the instance variables of a superclass is allowed.

In the strong form: direct access to the instance variables of a superclass is prohibited.

The Law of Demeter

The bottom line: inside a method, one may only access or send messages to the following objects:

– The "self" object (the object actually executing the method).

– Instance/class variables defined in the same class which has defined the method being executed (*not* those defined in its base classes!).

– Arguments of the method.

– Global variables.

– Local variables of the method.

Appropriate coupling in OO Design

Strive for ***low message coupling***:

– Reduce the number of messages passed between objects.

– Simplify messages to a few parameters.

– Avoid requiring multi-message sequences.

```
car.pushPedal(CLUTCH);
car.shiftToGear(PARK);
car.insertKey(carKey);
car.pushPedal(GAS);
car.turnKeyInIgnition();
```

```
car.start(carKey);
```

Appropriate coupling in OO Design

Strive for ***low association coupling***:

– Reduce extent to which objects depend on the internal structure of each other

– Reduce the use of superclass' instance variables by subclasses (strong form of the Law of Demeter).

```
class Person {
   String first, last;
   getName { first + last }
}
```

```
class Doctor : Person {
   getName {
      "Dr" + first + last;
   }
}
```

```
class Doctor : Person {
   getName {
     "Dr" + Person::getName();
   }
}
```

Appropriate coupling in OO Design

Strive for ***moderate inheritance coupling***:

- Abstract so that subclasses depend on the *methods* (but *not the structure*!) of their superclasses.
- Use or refine as many of superclass' operations as possible in the child classes.

```
class Person {
   hello {
      print "Hi!"
   }
}
```

```
class Doctor : Person {
   greetings {
      print "Hi!";
   }
}
```

```
class Doctor : Person {
   greetings {
      hello();
   }
}
```

What is cohesion?

- ***Cohesion*** is a measure of how effectively a group of elements (i.e., instance variables, methods) models a concept (i.e., a class).
- Basically... how well does your model "hold together?"
- To measure it, we must know a few terms...

Cohesion

Relevance

- An element is ***relevant*** to an object if is *directly associated* with the object, instead of being associated through some chain of invisible objects.

```
class Person {
   String name;
   String automobileTag;
   String automobileYear;
}
```

```
class Person {
   String name;
   Car automobile;
}
```

```
class Car {
   String tag;
   int year;
}
```

Not relevant: **people don't have automobile tags; automobiles do!.**

- Be on the lookout for compound nouns (like *automobileTag*) ... they can indicate invisible objects!

Cohesion

Necessity

- An element is ***necessary*** to an object if its presence is required for the accurate modelling of the object.

```
class TriagePatient {
   Boolean inShock?
   String  bloodType;
   String  favoriteTVshow;
}
```

```
class TriagePatient {
   Boolean inShock?
   String  bloodType;
}
```

```
class RecoveringPatient {
   String  favoriteTVshow;
}
```

***Not necessary*: do we really need to know their TV show?**

- Be on the lookout for attributes which are only used under certain rare conditions... maybe your objects should "evolve" from class to class, or maybe the attributes belong in a subclass.

Completeness

- A model of a concept is ***complete*** if no necessary elements have been omitted:

```
class Stack {
   int size;
   Object elems[];

   push(elem) {...}
}
```

```
class Stack {
   int size;
   Object elems[];

   push(elem) {...}
   pop() { }
}
```

Not complete: **I see push()...**
but where's pop()?

- IMHO it is okay to omit some *relevant* elements... the real world is often too complex to model in entirety; knowing what to leave out is part of abstraction.

Strong cohesion

- Consider a class intended to model some real-world concept. Its elements are instance variables (which model structure) and methods (which model function).
- A ***high degree of cohesion*** is indicated if…
  - All the elements are **relevant** to the concept.
  - All the elements are **necessary** to adequately model the concept.
  - No necessary elements are missing from the group; i.e., the group is **complete**.

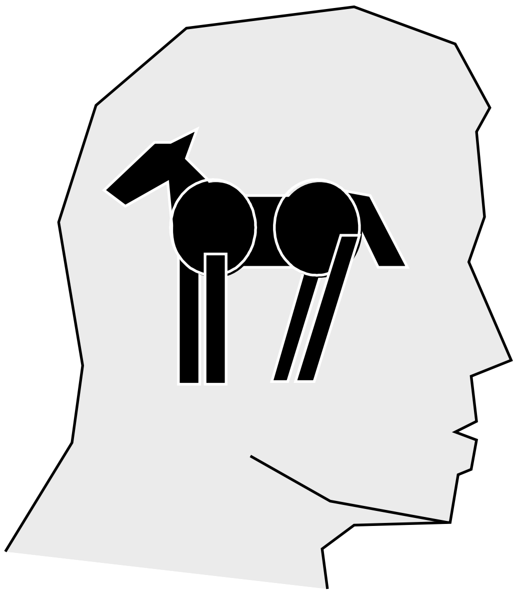
Appropriate cohesion in OO design

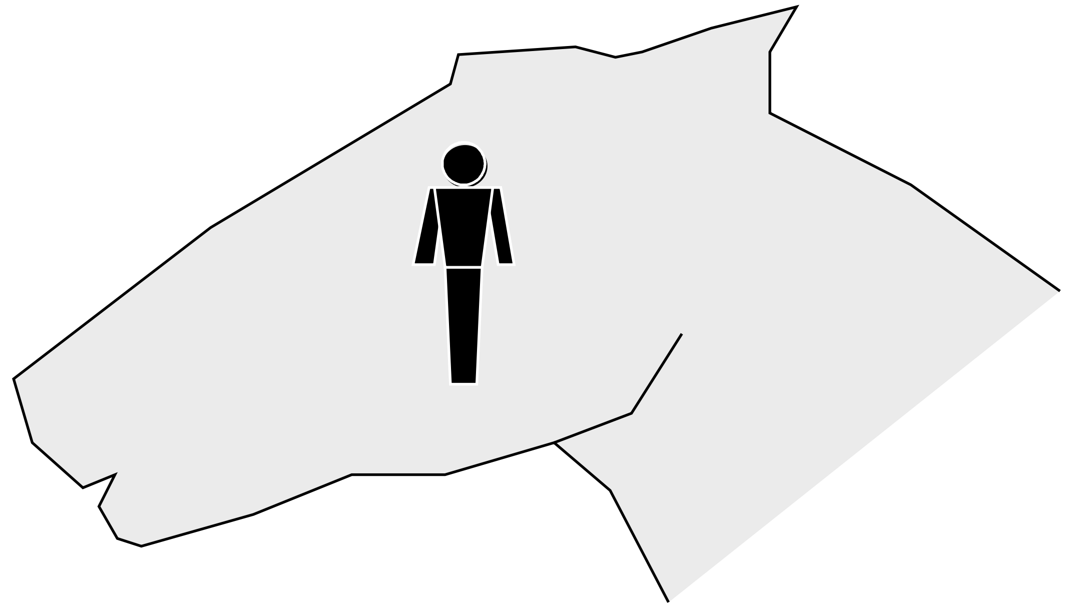
- **High cohesion is good**! Strive for...
- ***High structural cohesion***: Every **instance variable** should be *relevant* to the object, *necessary* for the object's intended use, and the object's essence should be adequately captured by all provided instance variables.
- ***High functional cohesion***: Every **method** should be *relevant* to the object, *necessary* for the object's intended use, and the the provided methods should adequately embrace the behavior of the object.

APPROACHES TO APPLICATION DESIGN

- Modelling external entities
- Interface classes
- Mix-ins
- Abstract interfaces

Modelling external entities





`hasSaddle?` `hasShoes?`

`climbOn()` `spur()` `feed()` `groom()`

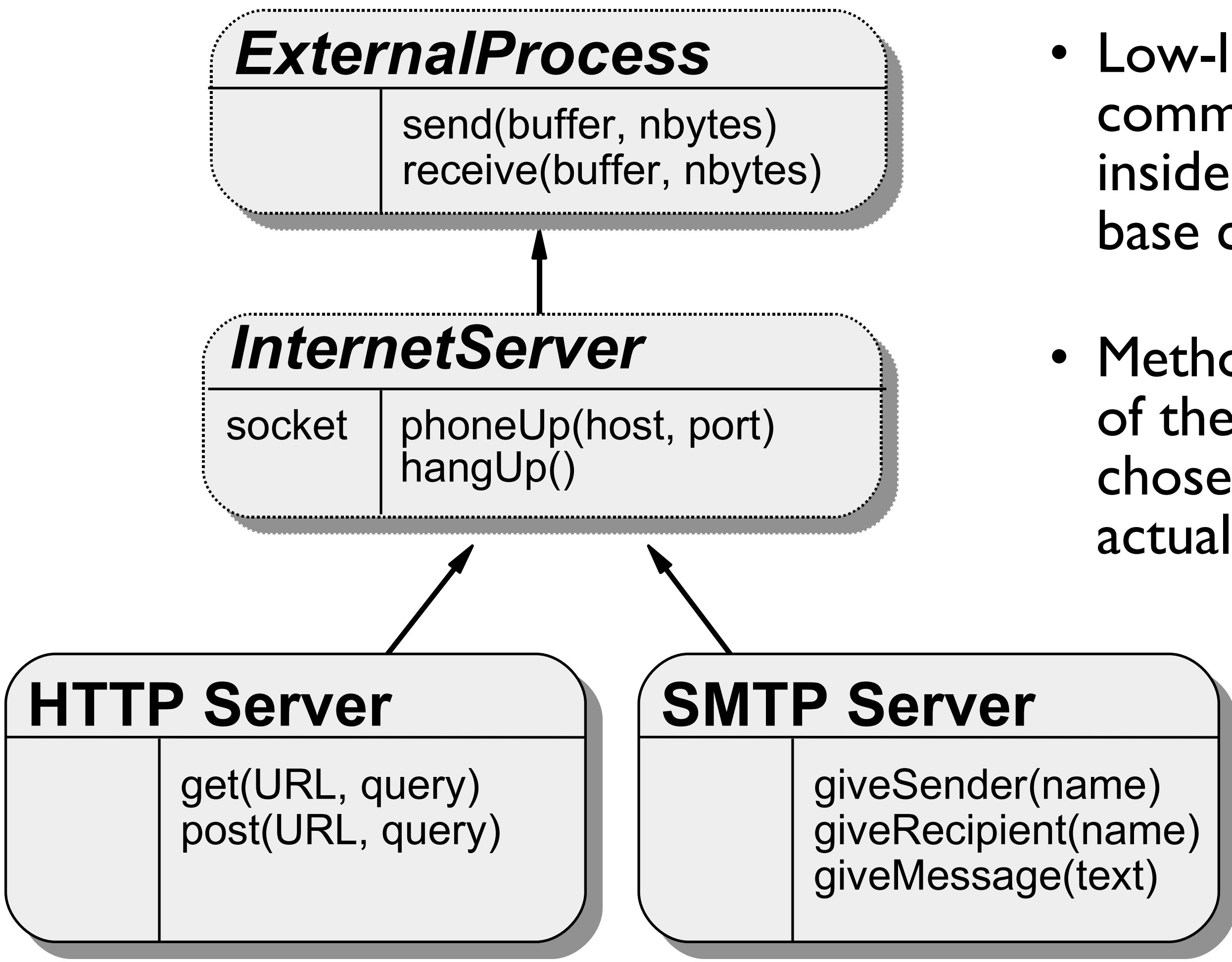
`hasSugar?` `weight` `likesToKick?`

`dumpInRiver()` `swatWithTail()`

What are interface classes?

- Just as **encapsulation** can help hide the "grotty" details of entities *inside* a program, so can it help simplify interaction with entities *outside* a program, when we provide intuitive "public interfaces" to…
  - Other processes (clients, servers, children)
  - Data files and databases
  - Hardware devices (printers, displays, etc.)
  - User interfaces
  - Foreign code
- Classes which represent external entities, and which exist for communicating with them, are called ***interface classes***.

Client-server inteface classes



- Low-level interprocess communicaton is done inside the methods of the base classes.
- Methods and return values of the derived classes are chosen to reflect server's actual "protocol."

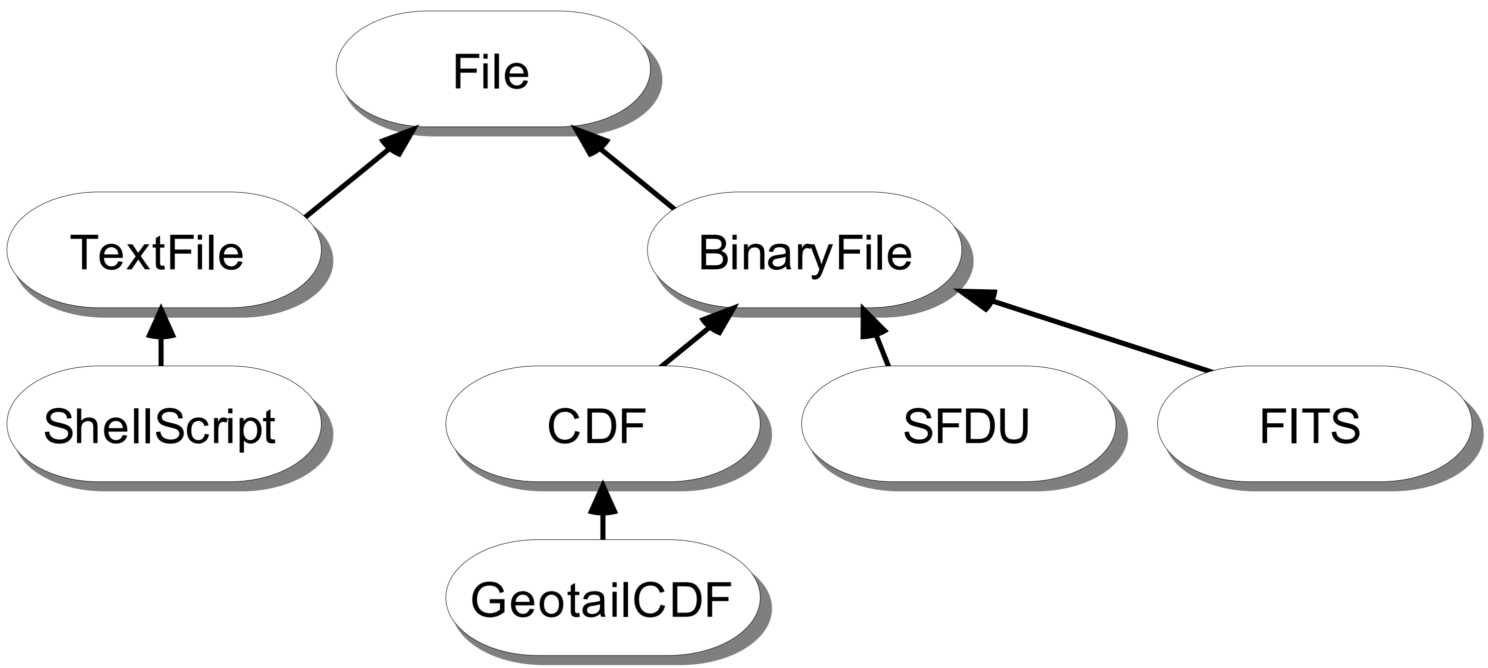
Sample SMTP interface class

Here's a possible dialog with an interface class designed for sending mail to another user. The class' methods reflect the commands that are actually being sent to the SMTP (Simple Mail Transfer Protocol) server:

```
MailServer s;


s.phoneUp("discovery.nasa.gov");
s.giveSender("davebowman");
s.giveRecipient("hal9000");
s.giveMessage("Open the pod bay door, HAL.");
s.hangUp();
```

Interface classes for data files

Most programs make extensive use of **data files**… encapsulating the internal structures of these files has the same benefits as encapsulating the implementation details of data structures:



Sample interfaces to data files

```
CDF f;

f.open("deathstar_blueprints.cdf");
  designer = f.getGlobalAttr("DESIGNER");
f.close();
```

using generic superclass

```
BlueprintFile bf;

bf.open("deathstar_blueprints.cdf");
  designer = bf.getDesigner();
bf.close();
```

using domain-specific subclass

Sample interfaces to databases

```
SybaseDatabase db;
SybaseTable table;

db.login("skywalker", "darthBites");
   db.newQuery();
   db.addToQuery("select cblk from prisoners p");
   db.addToQuery("where p.name = 'Leia'");
   table = db.doQuery();
   cellblock = table.row(1).field(1);
db.logout();
```

using generic superclass

```
PrisonerDatabase db;

db.login("skywalker", "darthBites");
   cellblock = db.getCellBlock("Leia");
db.logout();
```

using domain-specific subclass

Do we *care* how data is stored?

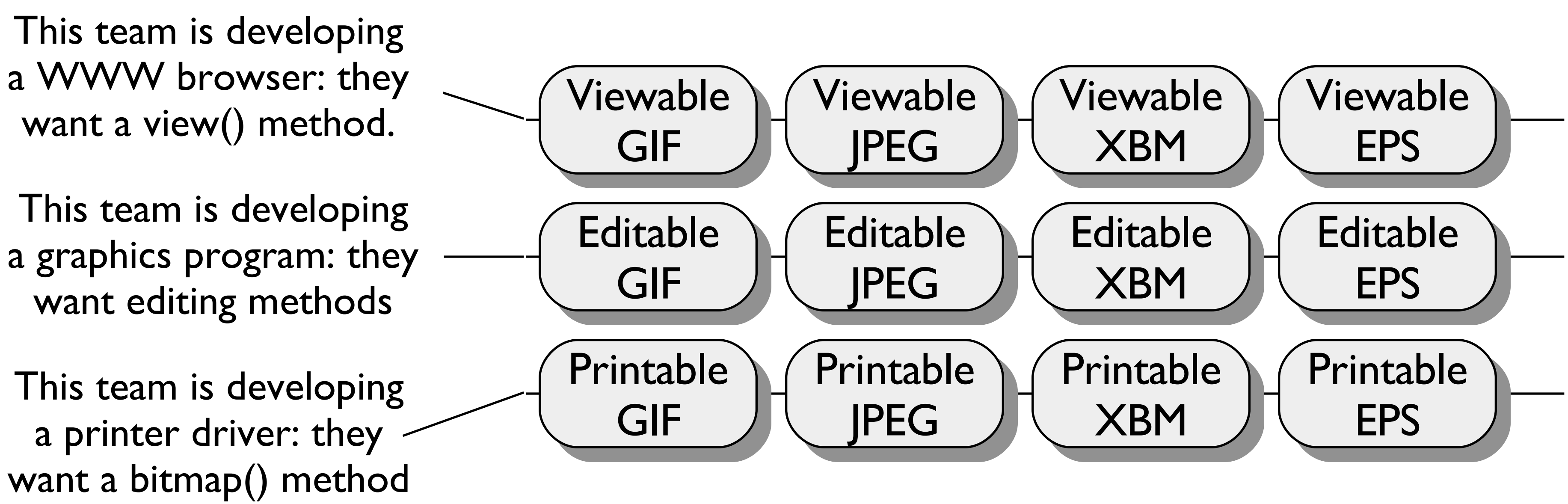
```
Database db;
Query    query;
Results  dbResults, totalResults;

/* Build the query from current user input: */
query = extractQuery(gUserInterface);

/* Send query to all DBs, and pool results: */
foreach db in gDatabases {
  dbResults = db.doQuery(query);
  totalResults.add(dbResults);
}
```


The high/low conflict...

- Two factors arise in creating an interface class to an external resource:
  - We want the **public interface** of the class to reflect the **high-level** attributes and behavior of the resource we're modelling.
  - We want to effectively **utilize tools** for **interacting** with the resource at a **low level**, if they exist.
- Now imagine a set of *n* different resources (e.g., image file formats), and a set of *m* problem domains, each of which wants to provide its own domain-specific interface to all *n* resources... ouch!

Combinatorial code explosion...

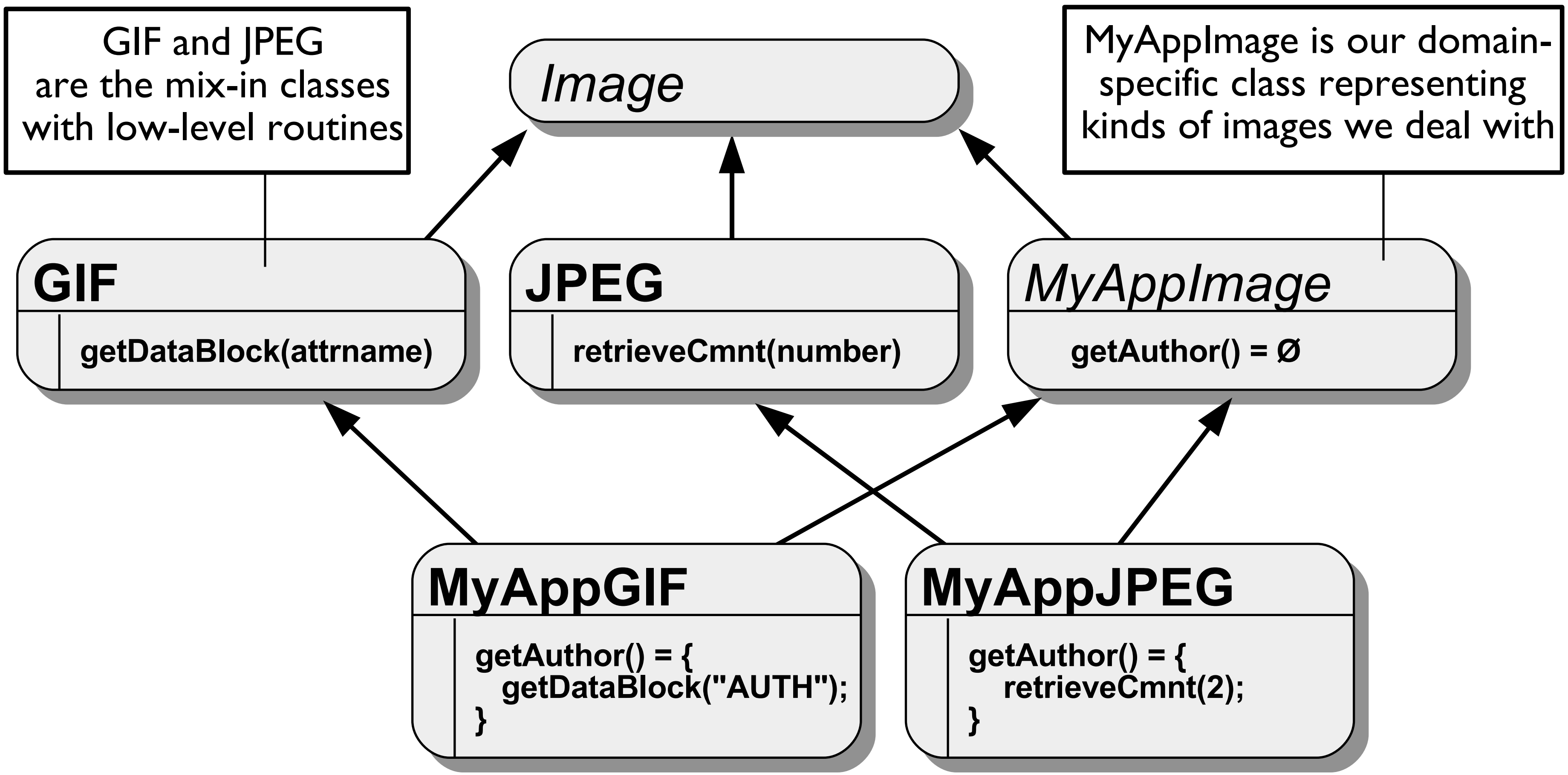
- Now imagine a set of *n* different resources (e.g., image file formats), and a set of *m* problem domains, each of which wants to provide its own domain-specific interface to all *n* resources... ouch!



Solution: mix-ins

- If **multiple-inheritance** is supported, one solution is to develop general-purpose classes called ***mix-ins***:
  - The **mix-in** provides a generic (low-level) **resource interface**.
  - The application defines one **abstract class** which defines the desired (high-level) **domain-specific interface**.
  - The application then defines one **concrete subclass** for each type of resource, **inheriting both from the abstract class and the mix-in.**
- So: to add the desired low-level functionality to any class... just "mix in" the needed code via inheritance!

A sample application



The scenario

Suppose you are developing a graphics library which lets you output any raster (pixellated) image in PNG format. You are all set to define classes GIF, JPEG, etc., each of which has an writePNG() method, when you think...

> *Why should I force people to use my classes? Why not allow them to provide their own? So long as they provide a few methods like* ***numRows()****,* ***numCols()****, and* ***getPixelColor(r,c)****, who cares how they've implemented it?*

So you turn it around, and provide an writePNG() routine which implements the real guts... **and all it demands is that the object you give it has a certain public interface.**

Yes, it's just encapsulation!

- **Just like encapsulation**, but it's the class *user* who draws up the "public interface contract"... and would-be class developers must obey it!
  - Contract might be drawn up before any real classes which *obey* that contract have been built...
  - Like inventing a phonograph, and telling people how to make records that will play on it.
- Good technique for providing reusable classes in a rich domain with many developers.
- Existing, incompatible classes can often be retrofitted through subclasses that obey the needed interface.

Abstract interfaces in practice

- Some OO languages (Java) let you **formally** define an interface: it's simply a set of method declarations.
- Some OO languages with run-time binding (Perl5) are **informal**: interface is just specified by documentation; up to class user to send in an appropriate object.
- Abstract interfaces form the basis of *tying* in Perl... objects with the right methods can become the "back end" of built-in data types like arrays and hashes!

WHEN WORLDS COLLIDE:
OO IN NON-OO DOMAINS

- Pseudoclasses
- Wrapper classes

Using OOD in non-OO domains

- Sometimes we don't have an OO programming language at our disposal, yet we recognize the strengths or relevance of an OO approach for a task.
- Remember some of the principle goals of OO languages:
  - **Modifiability**, for code designers.
  - **Understandability**, for code users.
  - **Reusability**, for code designers and users.

What are Pseudoclasses?

- Developed as a coding strategy for the C programming language, which has no classes, encapsulation, etc. Inspired by C++.
- Principle can be used in *nearly any non-OO language.*
- "Classes" are ordinary typedef'd structs.
- "Methods" are functions tied to a pseudoclass by consistent use of a conventions for:
  – Function naming.
  – Argument ordering.
  – Return value type and interpretation.

Pseudoclass coding standard

Classnames begin with capital letter...	`Stack`
Method names have class signature as prefix...	`StackPush()` `StackPop()`

Symbolic constants defined for return values, with class signature as prefix...	`StackOK`
	`StackOVERFLOW`
	`StackEMPTY`

First argument of method is always pointer to object in question, and parameter is always named "this"...	`StackPush(Stack *this, Elem elem);` `StackPop(Stack *this);`
Always a set of constructor functions for initiallizing a new object, and a destructor for destroying any object...	`Stack *StackNew();` `StackFree(Stack *this);`

A sample Pseudoclass header

```
#define StackOK          0
#define StackEMPTY      -1
#define StackOVERFLOW   -2

typedef struct Stack {
   int     numelems;
   Pointer elems[MAXELEMS];
} Stack;

Stack *StackNew  ();
void   StackFree (Stack *this);

int    StackPush (Stack *this, Elem elem);
Elem   StackPop  (Stack *this);
int    StackSize ();

int    StackDo   (Stack *this,
                   Func (*f)(Elem elem));
```

Stack.h

- Return values
- Class definition
- Constructors and destructors
- Principle methods
- Iterator

The transition to OO

- **Problem**: although you may have acquired an OO programming environment…
  - We **don't want to throw away** or rewrite mountains of existing, tested code in Fortran, C, assembly, etc.
  - We **may still need external non-OO libraries** for which source code is complex or unavailable (e.g., X Widgets).
  - Some **non-OO code may run much faster** than code in our OO environment (e.g., C vs. Smalltalk).

What are wrapper classes?

- **Solution:** Use ***wrapper classes***. A wrapper class is a class where...
  - The instance variables hold an externally defined data structure.
  - The methods are "pretty" front-ends to that data structure's supporting functions.
- The **idea** is that we provide developers with what *looks* like an object, but which *in fact* is just a thin front end onto non-OO routines.
- A wrapper class is a special kind **interface class**... providing an encapsulated interface to foreign code!

A C++ wrapper class

```
#include "Int_Stack.h"

class IntegerStack {

   Int_Stack S;

   int size() {
      return s_numelems(&S);
   }

   void push(int elem) {
      istack_push(elem, &S);
   }

   int pop() {
      return pop_int_stack(&S);
   }
}
```

Instance variable is the “wrapped” data structure

Methods are interfaces to functions operating on the data structure

Wrapping foreign code

```
void email_file(char *eaddress,
                char *sender,
                char *filename);
```

An external C routine which has been compiled and linked with a Smalltalk-like OOPL.

```
method: Person
email: filename from: sender

  System
    callFunc: "email_file"
    withArgs: [
       (self getEmailAddress)
       (sender getName)
       filename
    ].
%
```

A method, defined for class Person in the OOPL. Takes a file and another Person (the sender), and emails the file.

Real work done by sending a message to the System object, telling it to call the installed function.

Typically, only simple args (ints, floats, strings) can be passed.

Wrapping and tying in Perl

```
use GDBM_File;

# Open database (tie to object):
tie %DB, 'GDBM_File',
    "mydatafile.gdb", 1;

# Read and write:
print "Id = ", $DB{'Id'}, "\n";
$DB{'Name'} = 'Janeway';
$DB{'Rank'} = 'Captain';

# Close database:
untie %DB;
```

Here is code using the standard Perl5 wrapper class for accessing the GDBM libraries. And it allows you to use Perl's normal hash syntax!

In the Perl5 world, built-in datatypes (like hashtables) may be "tied" to your own custom classes, as long as your classes provide the needed interface methods.

In reality, each GDBM_File object contains a number that is really a struct pointer... this number is passed in opaquely to the libgdbm.a functions.

Wrapping and tying in Perl

```
package GDBM_File;

sub new { ... }
sub DESTROY { ... }

sub FETCH { ... }     # key
sub STORE { ... }     # key, val
sub FIRSTKEY { ... }
sub NEXTKEY { ... }
sub DELETE { ... }    # key
sub EXISTS { ... }    # key
```

```
# Really $x->STORE('Name', 'Kirk'):
$DB{'Name'} = 'Kirk';
```

The trick was that the GDBM_File class provided all the special interface methods that tie() demands for tying objects to hashes... FETCH, STORE, etc.

That allowed tie() to make an ordinary hash variable, %DB, into an interface to a GDBM file... without requiring the user to learn new access methods!

This is a wonderful example of the power of encapsulation, polymorphism, and wrapping!

PROJECT: MIME

The story you are about to read is true. The assignment is do-able... in fact, your instructor has done it. :-)

Your customer needs you to develop an email-handling system (a program that reads incoming mail, processes it, and maybe even replies to it).

This is complicated by the fact that nowadays, people can send MIME mail with attachments, and any attachment can be in 1 of 5 different encodings. So you need to be able to parse multipart MIME messages... and maybe even generate them.

Assume there is no software out there to do this in the language you're using.

Now, design it.