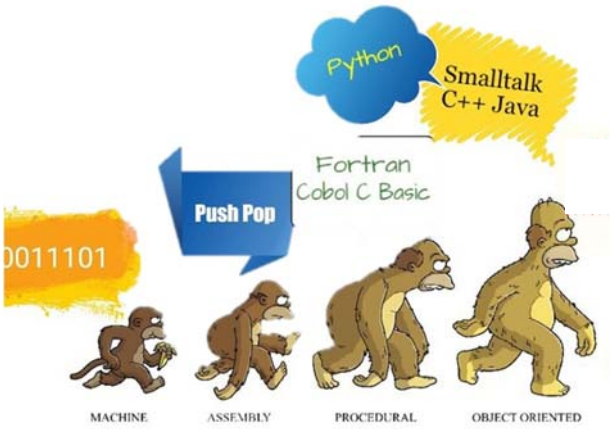
Object-Oriented Analysis and Design
using UML and Patterns



May 24-26, 2021

0

Goal of the Course

To provide a thorough understanding of OO analysis and design with UML and patterns

To follow the process of OO analysis and design from requirements capture through to design using an iterative process as the framework

Course Topics

Learn How to Think in Objects!

Fundamental Concepts of OO
- Objects, Classes, Inheritance, Polymorphism etc.

UML and (Agile) Unified Process (UP)

Object-Oriented Analysis (OOA)
- Use Cases, Domain Model

Object-Oriented Design (OOD)
- Responsibility-Driven Design, Heuristics, Design Patterns

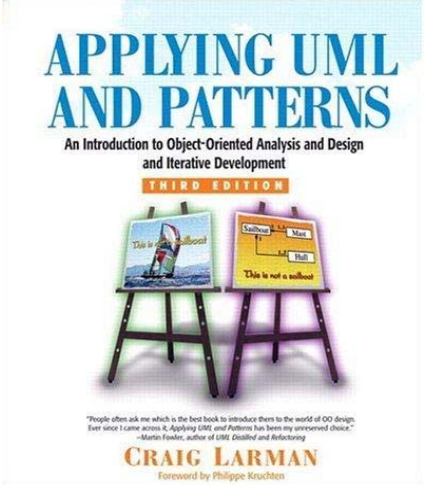
Advanced OO Principles
- SOLID Principles

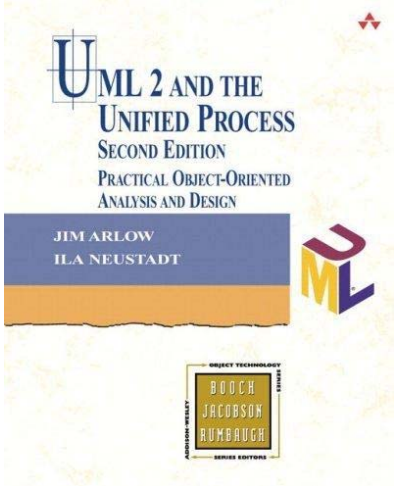
Architecture-level Refactoring

References

"*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*", Craig Larman, Prentice Hall, 3rd ed., 2005

"*UML 2 and The Unified Process: Practical Object-Oriented Analysis and Design*", Jim Arlow & Ila Neustadt, Addison-Wesley, 2nd ed., 2005



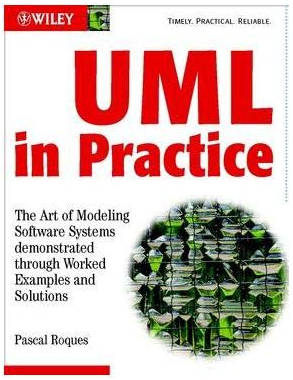


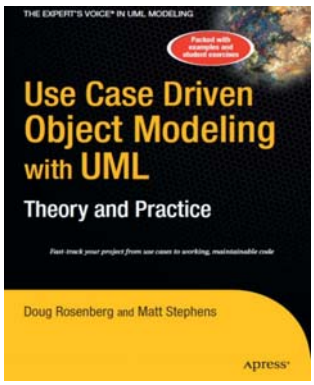
References

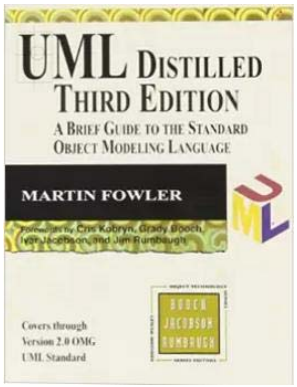
"*UML in Practice: The Art of Modeling Software Systems Demonstrated through Worked Examples and Solutions*", Pascal Roques, John Wiley & Sons Ltd., 2004

"*Use Case Driven Object Modeling with UML: Theory and Practice*", Doug Rosenberg & Matt Stephens, APress, 2008

"*UML Distilled*", 3rd ed., Martin Fowler, Addison-Wesley, 2003



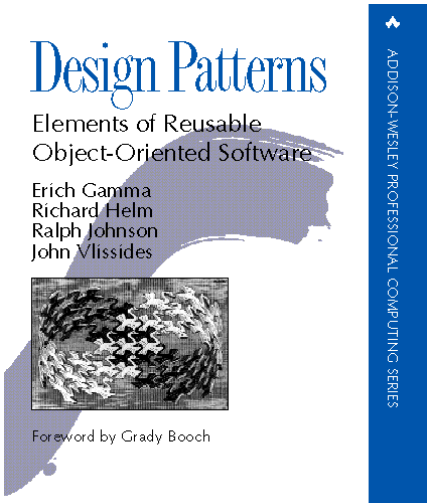




References

"Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

Gang of Four (GoF)



Major issues in Programming (Software Development)

What matters with software is …

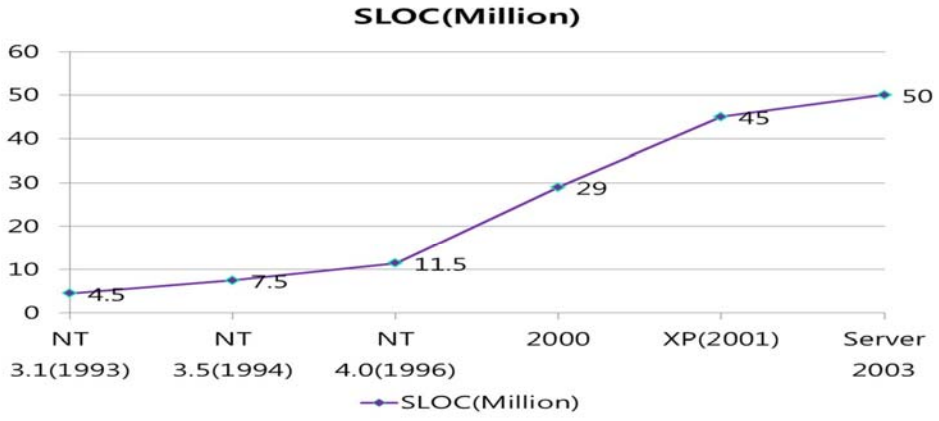
- Does the software do what is supposed to do?
- Is it of high quality?
- Can we rely on it?
- Can problems be fixed along the way?
- Can requirements change over time?

"All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version."

-- Ivar Jacobson

Growing Complexity: Windows

Windows XP: 40 million
Windows Vista : 50 million
Windows 7: 40 million (reduced from Vista).



Software Size: Automotive SW

- 90% of all innovations are driven by electronics and software
- Up to 40% of a vehicle's development costs are determined by electronics and software
- 50-70% of the development costs for an ECU are related to software

Source: AUTOSAR and Model-Based Design, MathWorks Automotive Conference 2012

BMW 7 series
- 270 user functions
- 2500 software functions
- 565MB binary code
- Over 67 processors

Source: Software Engineering for Automotive Systems: A Roadmap, FOSE 2007

10

Lehman's Laws

A classic study by Lehman and Belady [Lehm85a] identified several "laws" of system change.

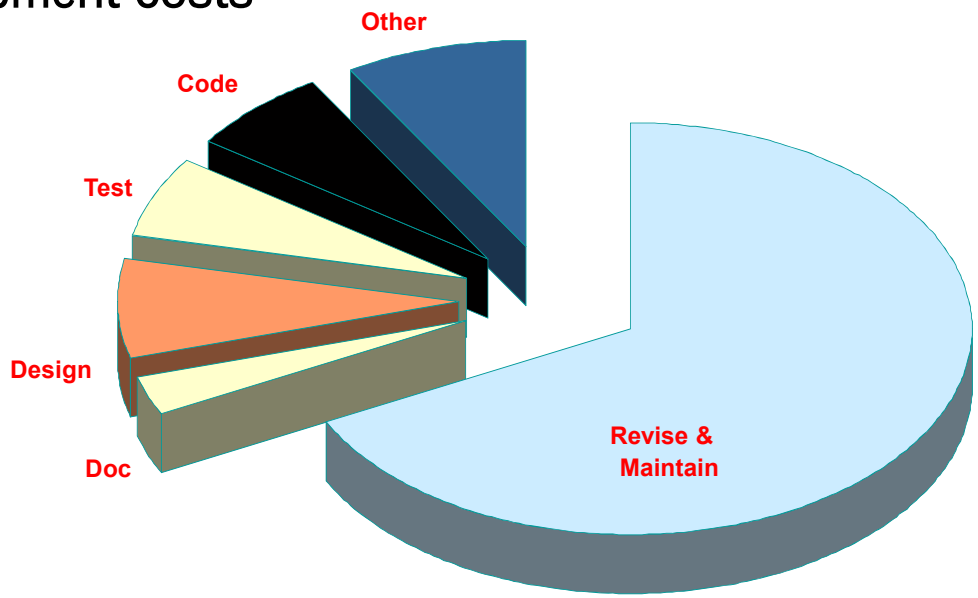
Continuing change

- A program that is used in a real-world environment *must change*, or become progressively less useful in that environment.

Increasing complexity

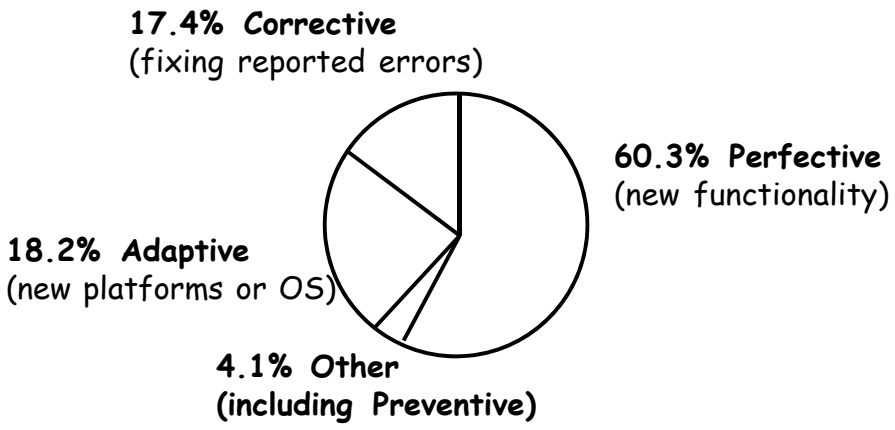
- As a program evolves, it becomes *more complex*, and extra resources are needed to preserve and simplify its structure.

Strategic rational system development plans are based on the complete cost of a system, not solely on development costs



Source: DP Budget, Vol. 7, No. 12, Dec. 1988

Maintenance Cost Due to Change Request



The bulk of the maintenance cost is due to *new functionality*

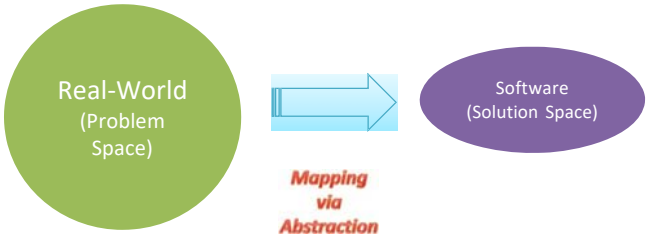
⇒ even with better requirements, it is hard to predict new functions

How to embrace change, reduce cost, increase productivity?

- Paradigm shift
  - Object-oriented paradigm, (*maybe* functional paradigm)
    - ➢Enabling technology to cope with complexity.
- Innovation of development Process
  - Iterative and incremental, architecture-centric, use case-driven process
  - Component-based development (CBD)
  - Software product line engineering (SPLE)
  - Reuse-based software engineering

What is Programming?

- Mapping problem domain to solution domain.
- Deliver programmer's **intent** to the computer.



Structured Programming

Nicklaus Wirth:

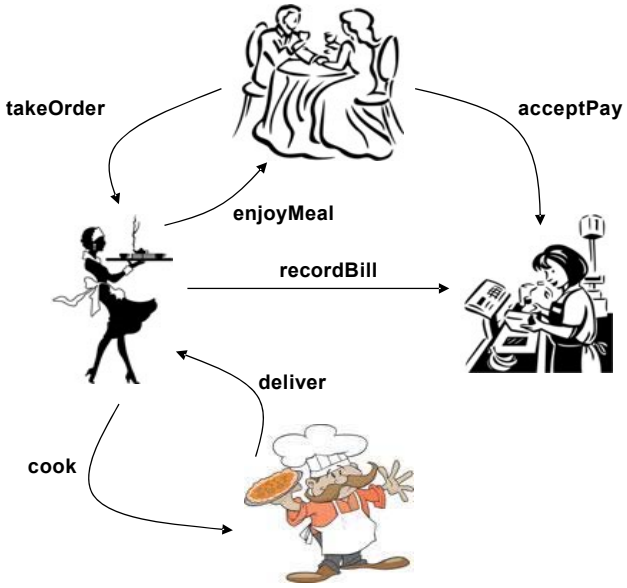
(Creator of PASCAL Language)

Programs = Data Structures + Algorithms

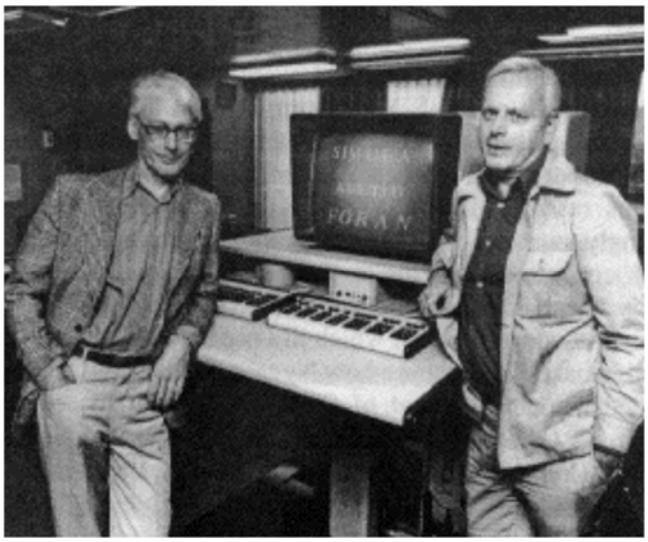
Object-oriented Programming

A program is a collection of interacting objects

Objects communicate by sending 'messages' to each other



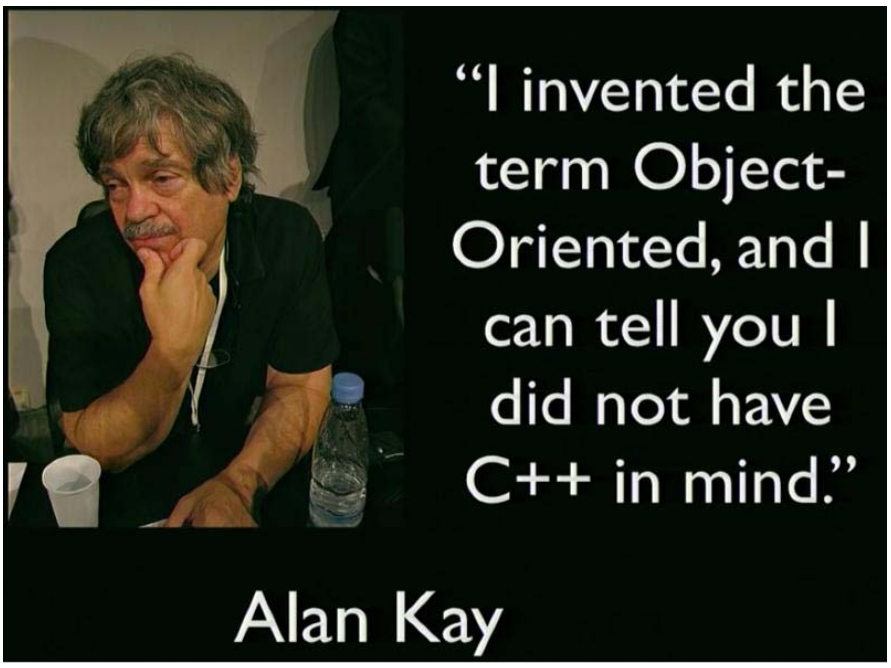
Birth of Object-Orientation: Simula67



Ole-Johan Dahl (left) and Kristen Nygaard, 1982

Mainstream OO

- C++
- Java



Reveal Intentions

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."

Clear Intent does not mean … Familiar!

Find the square of the second even number which is greater than 7.

```
Integer find(List<Integer> ints) {
  int count = 0; int ans = 0;
  for (Integer num : ints) {
    if (num % 2 == 0) {
      if (num > 7) {
        count++;
        if (count == 2) {
          ans = num * num; break;
        }
      }
    }
  }
  return ans;
}
```

It means Clear and Simple!

Find the square of the second even number which is greater than 7.

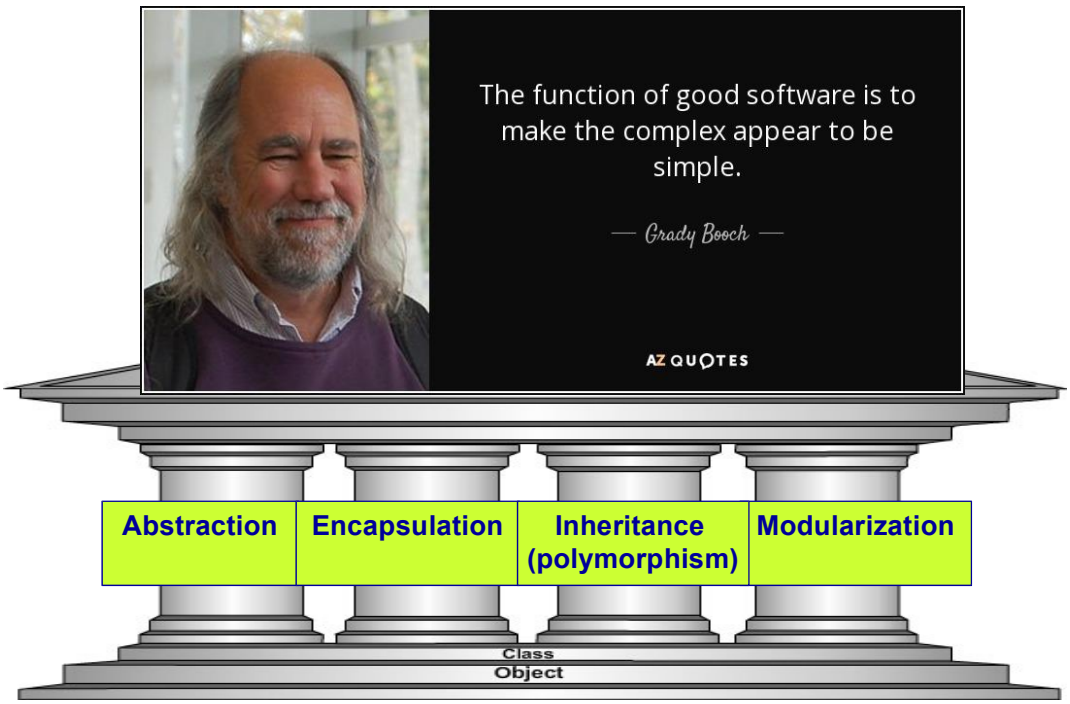
```
Optional<Integer> find(List<Integer> ints) {
  return(
    ints.stream()
        .filter(n -> n % 2 == 0)
        .filter(n -> n > 7)
        .skip(1)
        .map(n -> n * n)
        .findFirst()
);
```


Object-Oriented Analysis and Design using UML and Patterns

Fundamentals of

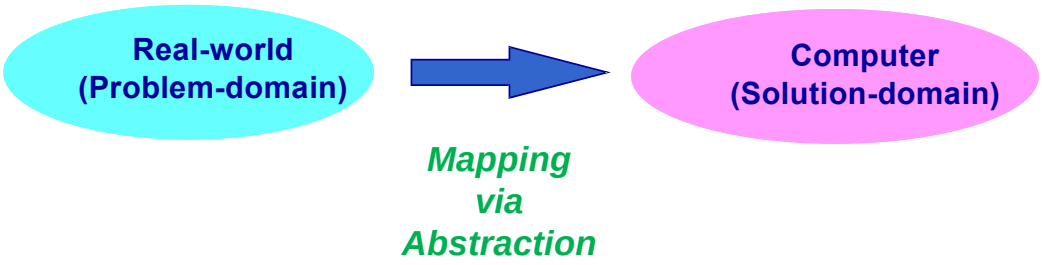
Object-Oriented Concepts

Four Pillars of Object-Orientation



Is Object-Oriented Programming… ?

If we see OO from the *evolutionary* *perspective, it is an* approach to mastering the software complexity



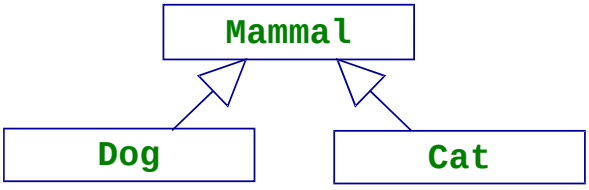
Assembly Language (Mnemonics)
Human-oriented Languages (High Level Languages)
Subprograms (Procedures or Functions)
Modules
Abstract Data Types
Objects

OO paradigm builds on three important concepts of objects, classes and inheritance

Objects

Classes

Inheritance



OO also requires 'polymorphism'
(i.e., dynamic binding)

Question 1

What is an Object?

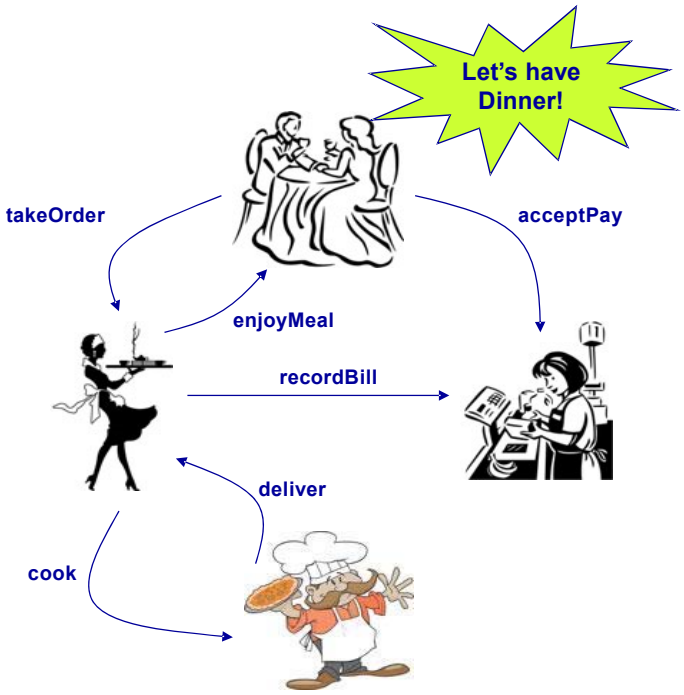
① Instance of a class
② Encapsulated entity with state and behavior
③ Entity with unique identity
④ Things or concepts with crisp boundary

Objects are building blocks of software systems

A program is a collection of interacting objects

Objects communicate by sending 'messages' to each other

In this way, they cooperate to complete a task



Objects are things or concepts with crisp boundaries and meaning *(for the problem at hand)*

Tangible Things

Customer
Store
Parking Lot
Car
Tire
…

Concepts & Processes

Strategy
Contract
Class Meeting
Finding a path through a maze
Sorting a card deck
…

Objects may have attributes that describe them

Are these also objects?

Color
Beauty
Hungry
Anger
Love
Time
Age
Address
…

} Attributes

Objects must have externally observable behaviors as a result of performing operations

Every object has a service that an object offers to its clients (or responsibility that it performs for them)

The services/responsibilities are represented as a set of operations

The set of operations constitute an interface (or type) of an object

Objects may have states which could affect how objects act or react

Can you get cokes or snacks at any time from the machine? If not, why?

The behavior of the vending machine may depend on time or the order in which one operates on the machine.

This time-dependent behavior implies the existence of state within the vending machine.

The state of an object is determined by the current (usually dynamic) values of each of the attributes

Attributes can be:

Coins deposited,

Change,

Height,

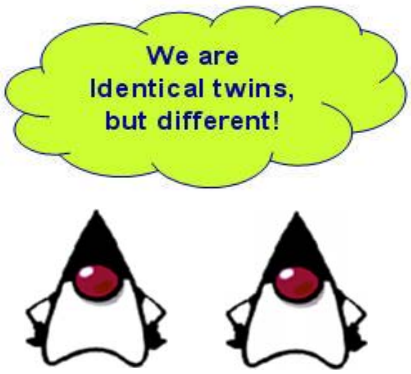
Current number for each items

etc.

It is good engineering practice to encapsulate the state of an object rather than expose it

Objects have unique identities

Identity is that property of an object that distinguishes itself from all other objects



How many button objects exit in the code?

```
class Button { ... };

Button  b1 = new Button();
Button  b2 = b1;
```

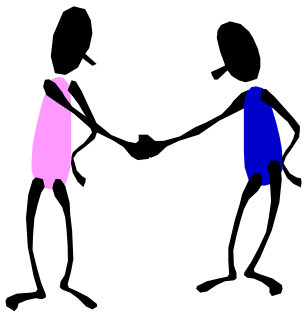
Do not confuse between the name (or handle) of an object and the object itself!

Objects perform actions by making requests of each other through a mechanism of messages

No object exists in isolation. Objects must collaborate with other objects by exchanging messages

A message is a request for action

An object may accept a message, and in return will perform an action and may return a result



Message passing is equivalent to invoking an operation (method in Java or member function in C++) on another object

In a message, there is a designated *receiver* (or *target*) that accepts the message

When we send a message to an object obj, the obj is a receiver object

```
// In C++, if obj is a pointer var
obj->msg();

// In C++, if obj is a ref. var
obj.msg();

// In Java, obj must be ref. var.
obj.msg();
```

The actual behavior performed by the receiver may be different, depending upon the type of the receiver.

Polymorphism!

Ask not what you can do to your objects, but ask what your objects can do for you

Anthropomorphism
(i.e, live objects)

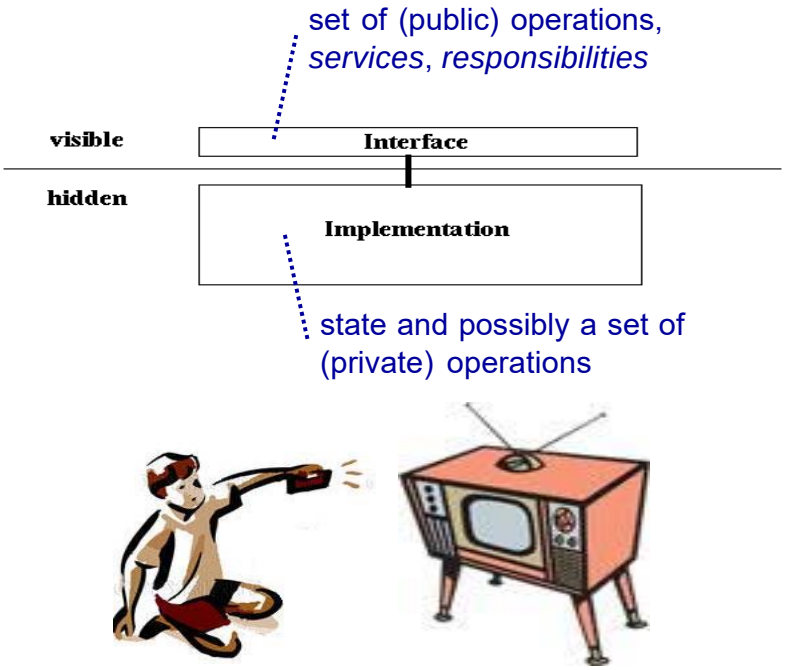
"Instead of a bit-grinding processor ... plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires"
-- Dan Ingalls

It is good engineering practice to encapsulate implementation details rather than expose it

Encapsulation:
separation of interface from implementation

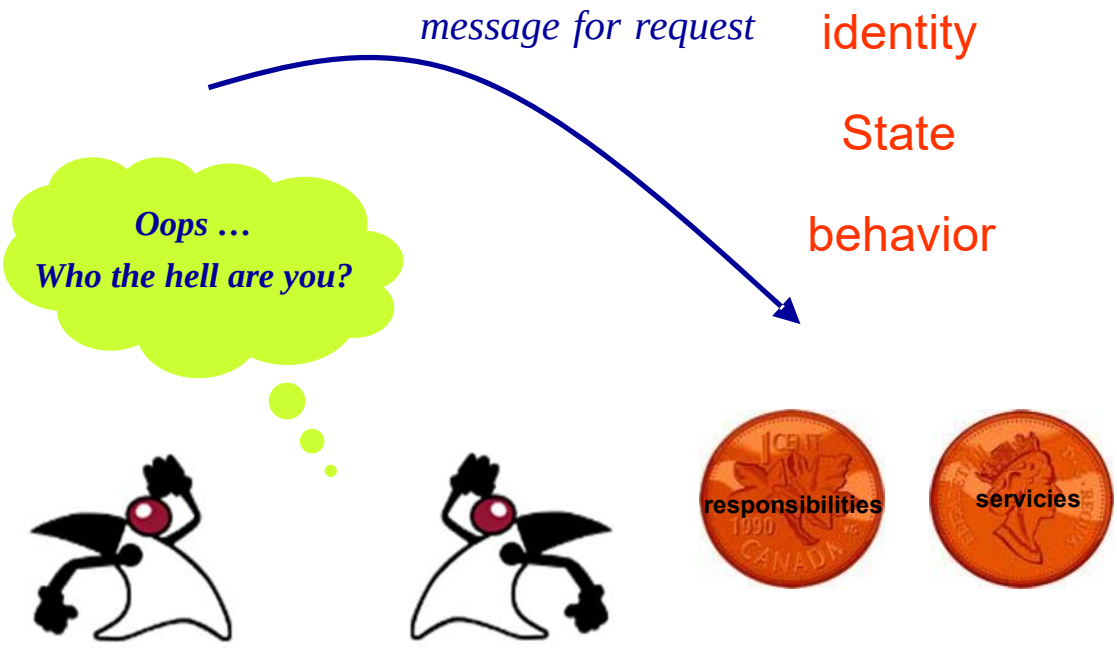
Object is a black box. Its internal workings and parts are hidden.

Helps code reuse and reliability
- strong cohesion
- loose coupling



Do not need to know how an object will perform actions, but need to know what messages it will understand

An object is an encapsulated entity with behavior, state, and unique identity



Question 2

What is a Class?

Why do we need classes in addition to objects?

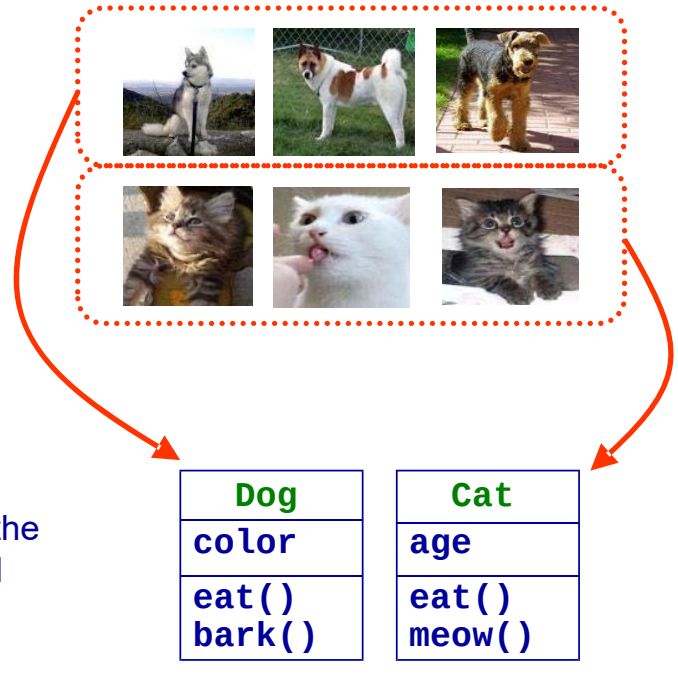
① **Type**
② **Category**
③ **Repository for behavior and representation**
④ **Object template**

A class is a group of objects with the same attributes, behavior, relationships, and semantics

There are too many objects in the real world, which is impractical to handle

Classifying objects factors out commonality among sets of similar objects
- describe what is common just once
- create any # of copies later

A class is a repository for behavior and the internal representation of the associated objects

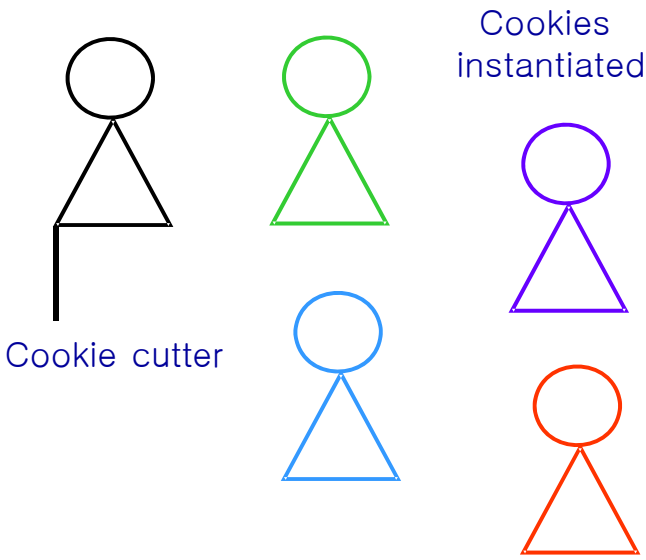


Objects are created from classes through the process of instantiation

Objects are actual instances of a class

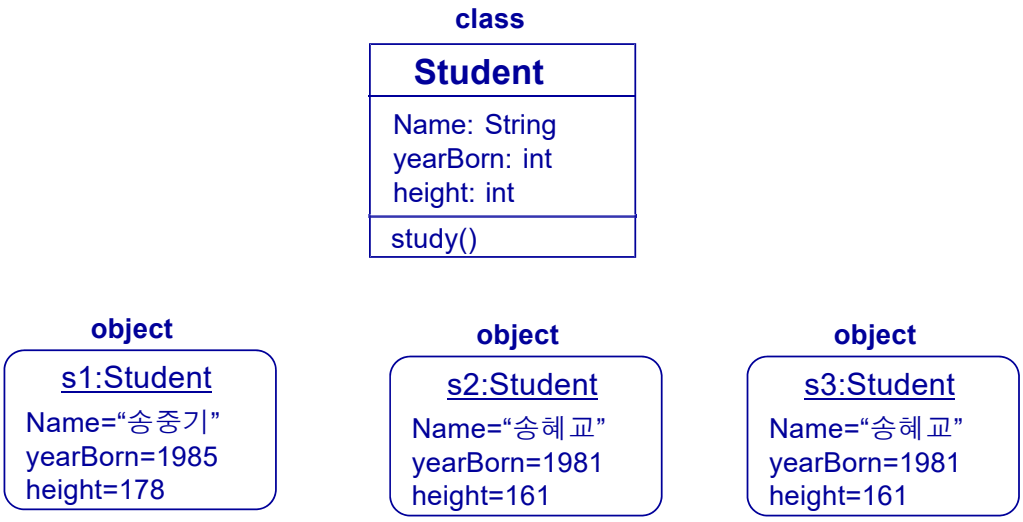
Classes are cookie cutters that make cookies
- As cookie cutter is not a cookie by itself, a class is not an object by itself.

Classes are factories and objects are products from the factory.



The terms instance and object are interchangeable.

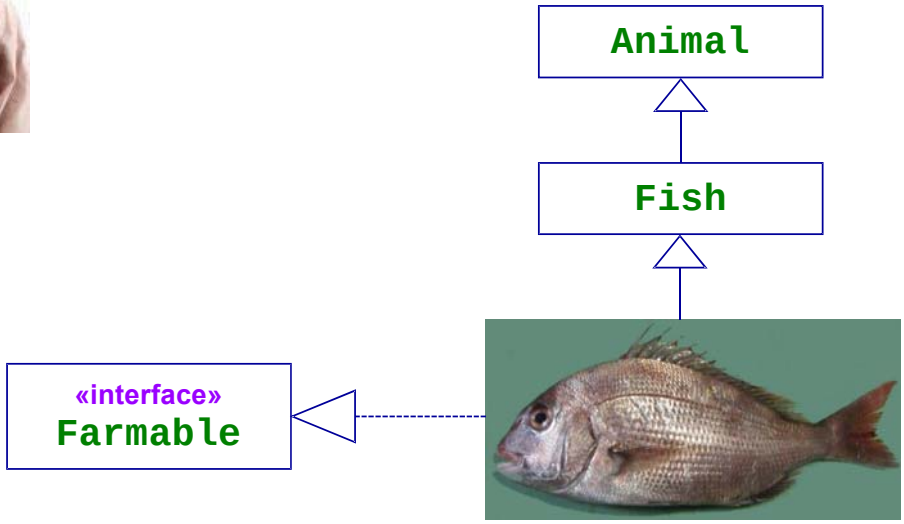
Instances created from the same class may or may not have the identical attribute values, but they are unique at all times



Student s1 knows how to do everything student s2 does, but s1 is not s2 or vice versa. Even the s2 and s3 are different.

Question 3

It is possible for a given object to have more than one types other than the type defined by its class?



An object's *interface* is a set of all signatures that can be sent to the object

A operation's signature is
type of its return value, ⟹ `void`
the operation's name, ⟹ `paintIcon(`
types of its parameters ⟹ `Component c,Graphics g,int x,int y )`

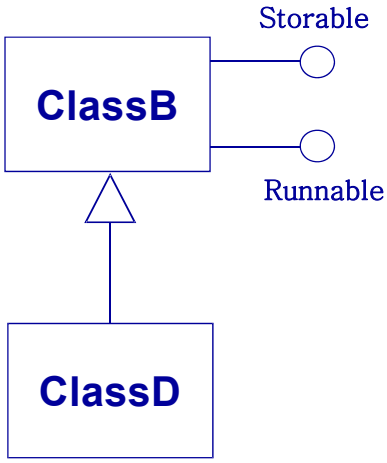
Icon Interface

```
int getIconHeight()
int getIconWidth()
void paintIcon(Component c,Graphics g,int x,int y)
```

An object's type is a name denoting only a particular interface

Since a class defines the operations an object can perform, it also defines the object's type

An object can have many types and objects of different classes can have the same type



Types of a:ClassD

ClassD, ClassB, Storable, Runnable

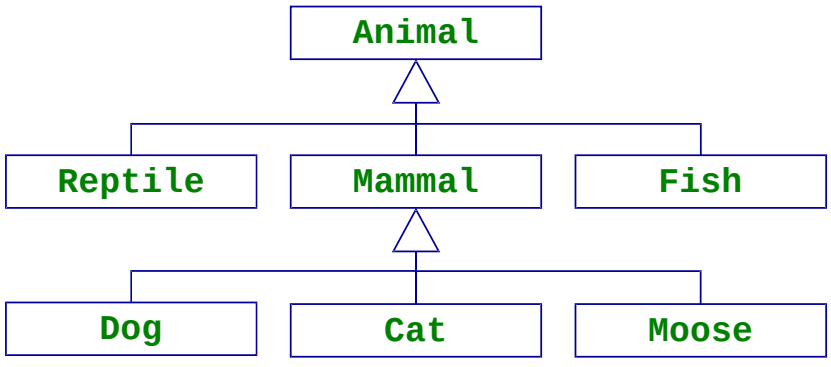
Both a:ClassD and b:ClassX are of type

Runnable

Classes might be organized into a tree called an inheritance hierarchy

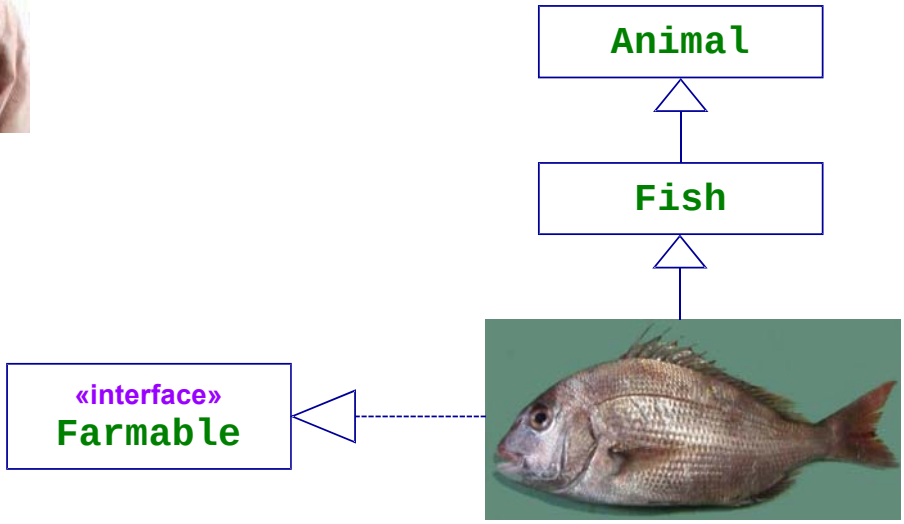
Inheritance allows us to arrange class definitions that reflect natural category hierarchies

Information (data and/or behavior) that can be found at a level in a class hierarchy is automatically applicable to lower levels of the hierarchy.



Question 4

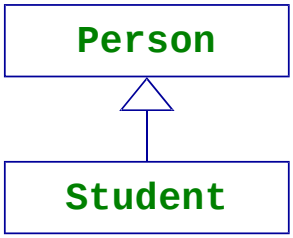
What is polymorphism
and why is it important?



Question 5

If "Student" is defined to be a subclass of "Person", what does this hierarchy imply?

In other words, what kind of relationship does these two classes have and what is the intention of the designer for this hierarchy.



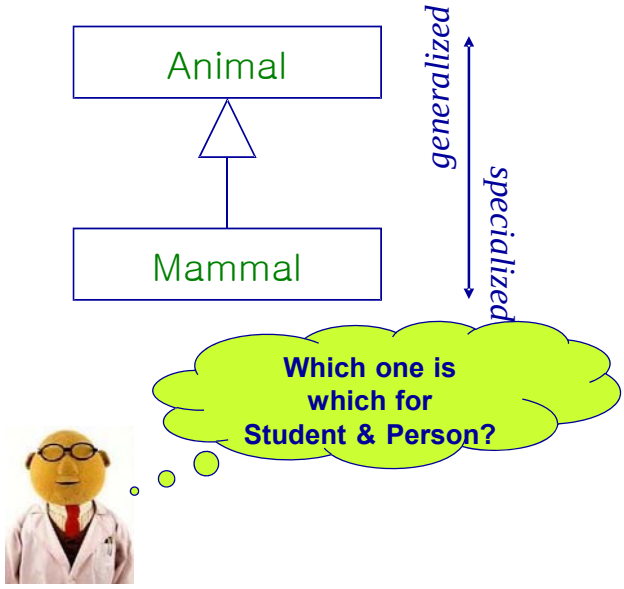
The single most important rule in OO is this: (Public) inheritance means "*is-a*" relationship (aka, *generalization/specialization* relationship)

A Mammal *is an* Animal

A Student *is a* Person

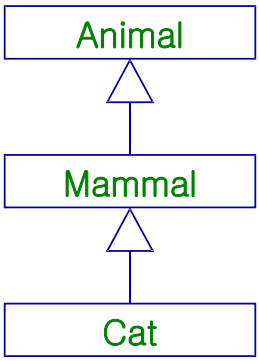
Animal is called a superclass or base class

Mammal is called a subclass or derived class



A class can be a superclass as well as a subclass at the same time and the "is_a" relationship is transitive

A Mammal *is an* Animal

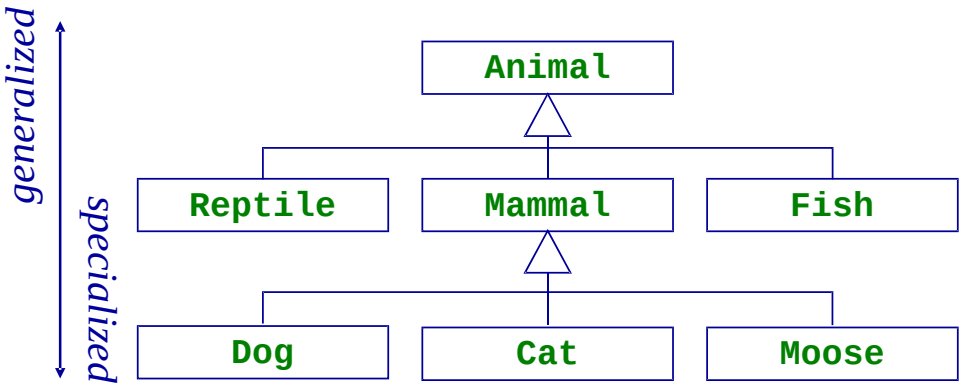


Mammal is a superclass of Cat and a subclass of Animal

A Cat *is a* Mammal

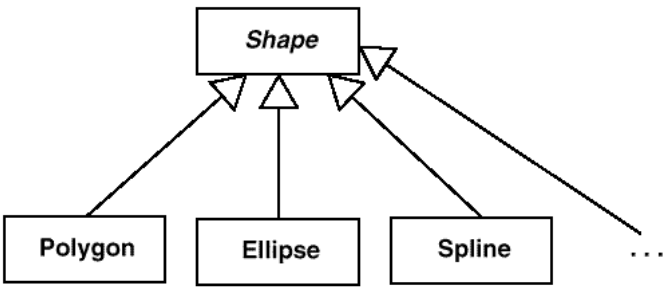
Transitivity: A Cat is an Animal, too

More than one classes can inherit from a given class

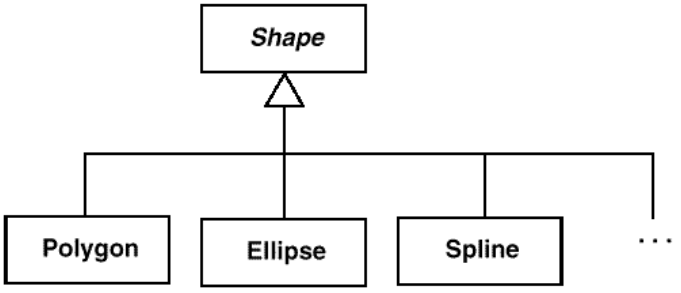


The root provides more general properties shared by all its descendents while the descendents typically add specializing properties which make them distinct among their siblings and their sibling's descendents

UML Notation

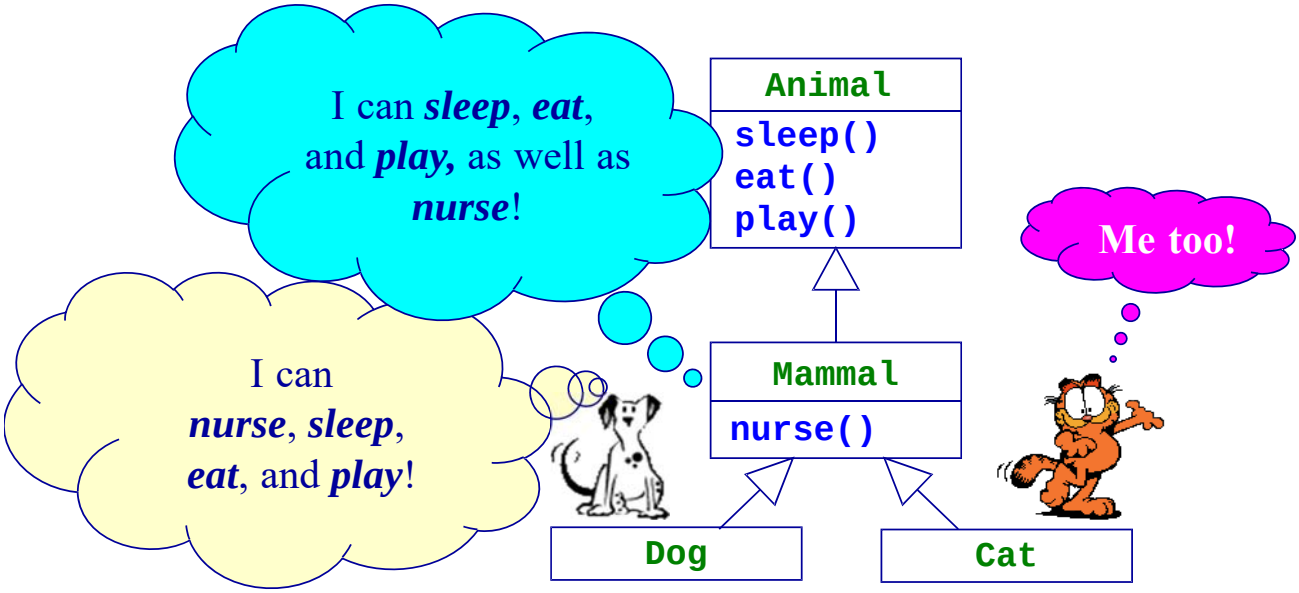


(a) Separate Target Styl



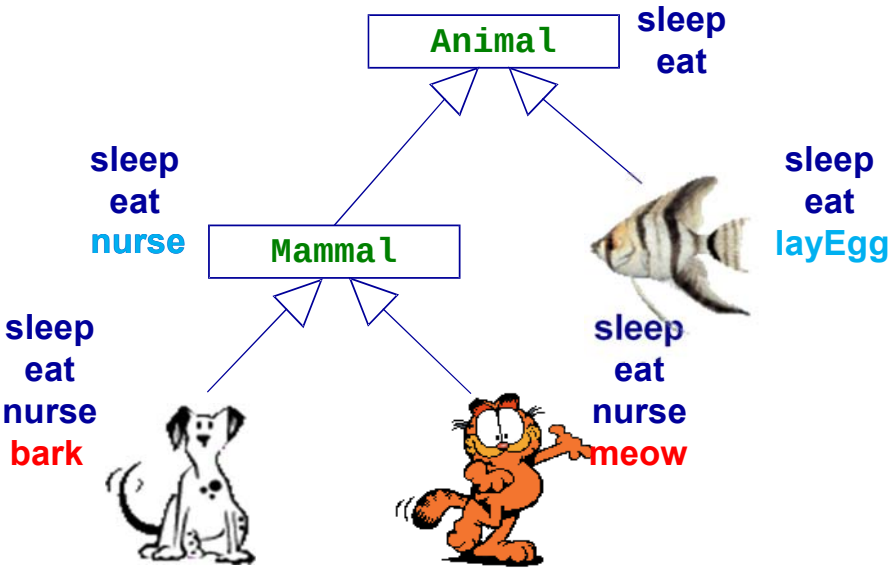
(b) Shared Target Style

Subclasses inherit all attributes and operations from its superclass and its ancestors



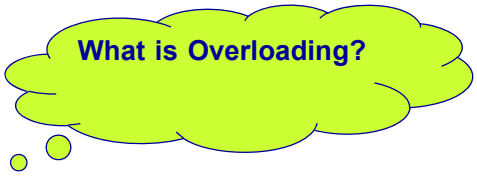
Information (data and/or behavior) that can be found at a level in a class hierarchy is automatically applicable to lower levels of the hierarchy

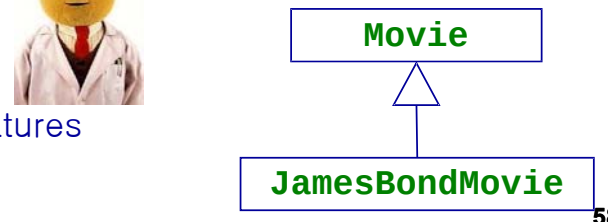
Superclass factors out members common to its subclasses



Subclasses can redefine inherited operations (Overriding)

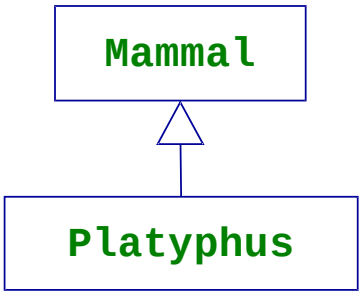
```
public class Movie extends Attraction {
  public int rating() {
    return scripting + acting + directing;
  }
}
public class JamesBondMovie extends Movie {
  public int rating() { // overriding
    return 10 + acting + directing;
  }
}
```





Operations must have the same signatures

One Big Mistake (?) in One of the Famous OO Books!



Subclasses can alter or override information inherited from parent classes:

- ➔ All mammals give birth to live young
- ➔ A platypus is an egg-laying mammal

The practical meaning of the *"is a"* relationship

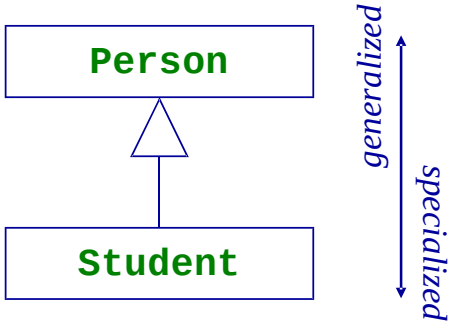
Every object of Student type is also an object of Person type, *but not vice-versa*

Person represents a more general concept than Student, and Student represents more specialized concept than Person

Anything that is true of an object of Person type is also true of an object of Student type, *but not vice-versa*

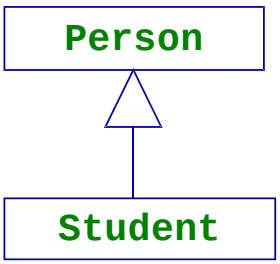
Student *is a* Person

```
class Person { ... };
class Student :
      public Person { ... }
```



Anything a superclass can do, a subclass can do as well

Anywhere an object of Person type can be used, an object of Student type can also be used just as well, *but not vice versa*
→ Liskov Substitution Principle (LSP)




```
class Person { ... };
class Student : public Person { ... }

void sleep(const Person& p);      // anyone can sleep
void study(const Student& s);     // only students study (?)

Person p;     // p is a Person
Student s;    // s is a Student

sleep(p);     // OK, p is a Person
sleep(s);     // OK, s is a Student & a Student is-a Person

study(s);     // OK
study(p);     // Error! - p is not a Student
```

A pointer (or reference) to a subclass can be implicitly converted to a pointer (or reference) to a superclass, but *not vice versa*

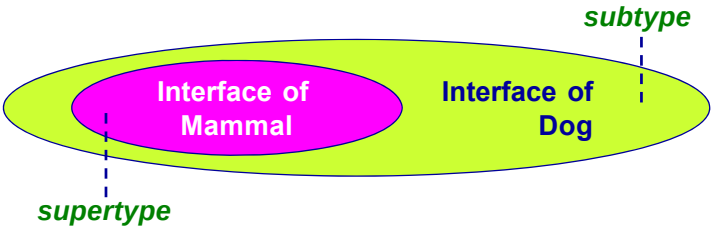
```
Student s;
Person* pPtr = &s;   // OK

Person p;
Student* sPtr = &p;
```

Error! - needs explicit casting
sPtr = dynamic_cast<Student*>(&p)

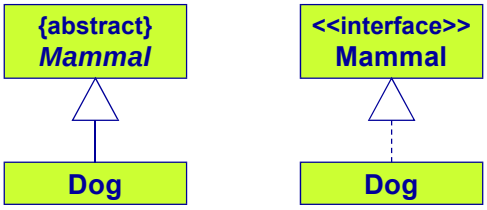
Subclass designers decide what to share: Interface or Implementation?

A type is a subtype of another if its interface contains the interface of its supertype



Subtyping uses inheritance as a mechanism for interface sharing

- enforces the "is_a" relationship
- also called interface inheritance

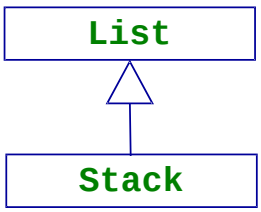


Most common mechanism for interface sharing

Subclass designers decide what to share: Interface or Implementation? (Cont'd)

Subclassing uses inheritance as a mechanism for implementation sharing (i.e., code and representation sharing)

- enforces "implemented in terms of" relationship
- also called class inheritance or implementation inheritance




The interface of List should not be visible to the clients of Stack!

Superclass designers must decide what to pass down for each operation when gets inherited.

```
class Shape {
public:
   virtual void draw() const = 0;          // pure virtual
   virtual void error(const string& msg);  // virtual
   int objectID() const();                 // nonvirtual
}

class Circle : public Shape { ... };
class Rectangle : public Shape { ... };
```

Inheritance comes in two flavors at function-level:

- inheritance of function interface only and
- inheritance of function interface & implementation

As a class designer, we need to explicitly specify our intention using appropriate types of member functions (or methods)

Pure virtual function: `virtual void draw() const = 0;`

To have derived class inherit a function interface only

Virtual function: `virtual void error(const string& msg);`

To have derived classes inherit a function interface as well as a default implementation

Nonvirtual function: `int objectID() const();`

To have derived classes inherit **a function interface as well as a mandatory implementation**

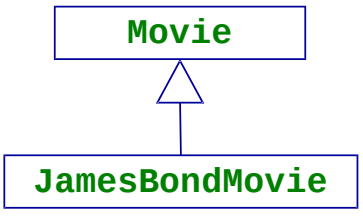
A reference (or pointer) variable has two types associated with it

Static type	Dynamic type
– the type declared at program	– the type of object it actually refers to
– fixed and never changed	– can be changed during lifetime

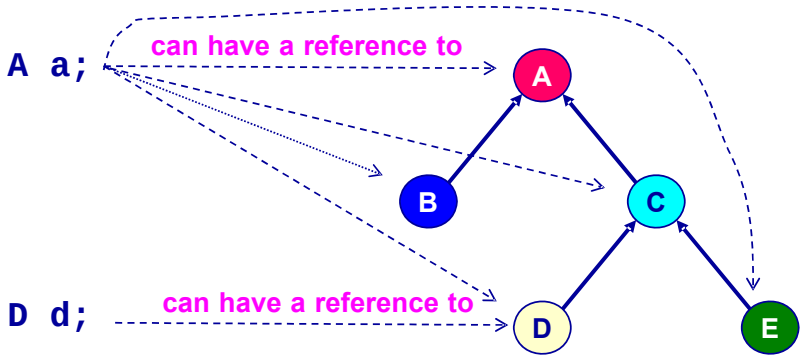
Example:

```
Movie m = new Movie();
m = new JamesBondMovie();

JamesBondMovie jm;
jm = new JamesBondMovie();
m = jm;
jm = m; // error!
```



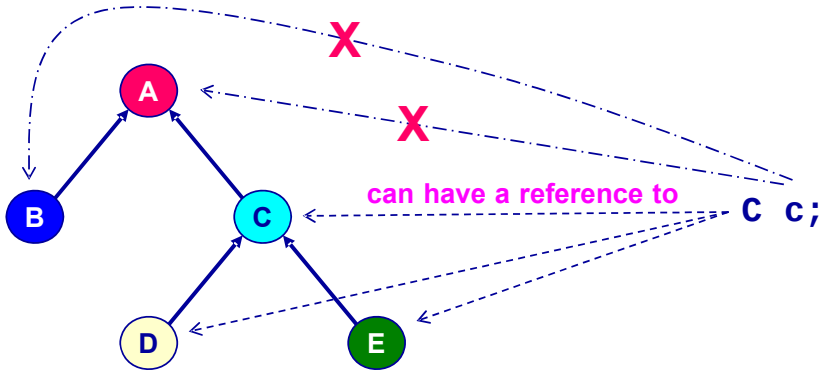
A subclasses can be assigned to its superclass, but not *vice versa*



Rule: A reference variable of type *A* (e.g., Movie) can have a reference to an instance of any subclass *extended* from *A* (e.g.,a JamesBondMovie)

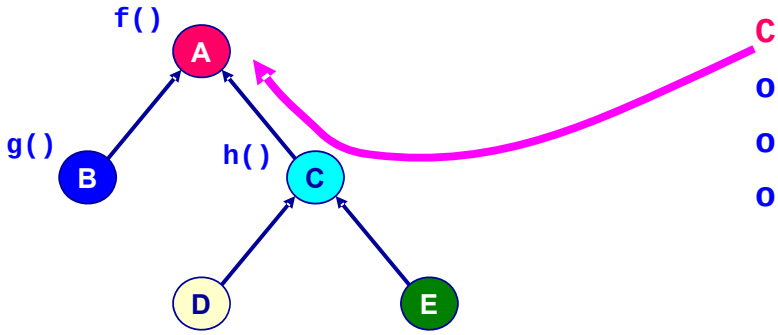
```
A a = new A(); // OK
a = new C();   // OK
B b = new B(); // OK
          …
a = b;         // OK
```


A superclass cannot be assigned to its subclass




```
A a = new A(); // OK
C c = new C(); // OK
a = c;   // OK
…
c = a;   // Error!
```

When a message is sent to a receiver object, the compiler checks its legality by performing an operation lookup using static type




```
C obj = new C();
obj.h();    // OK
obj.f();    // OK
obj.g();    // illegal!
```

The compiler searches up starting from the static type until it finds the invoked operation

A subclass can do anything a superclass can do, but not vice versa

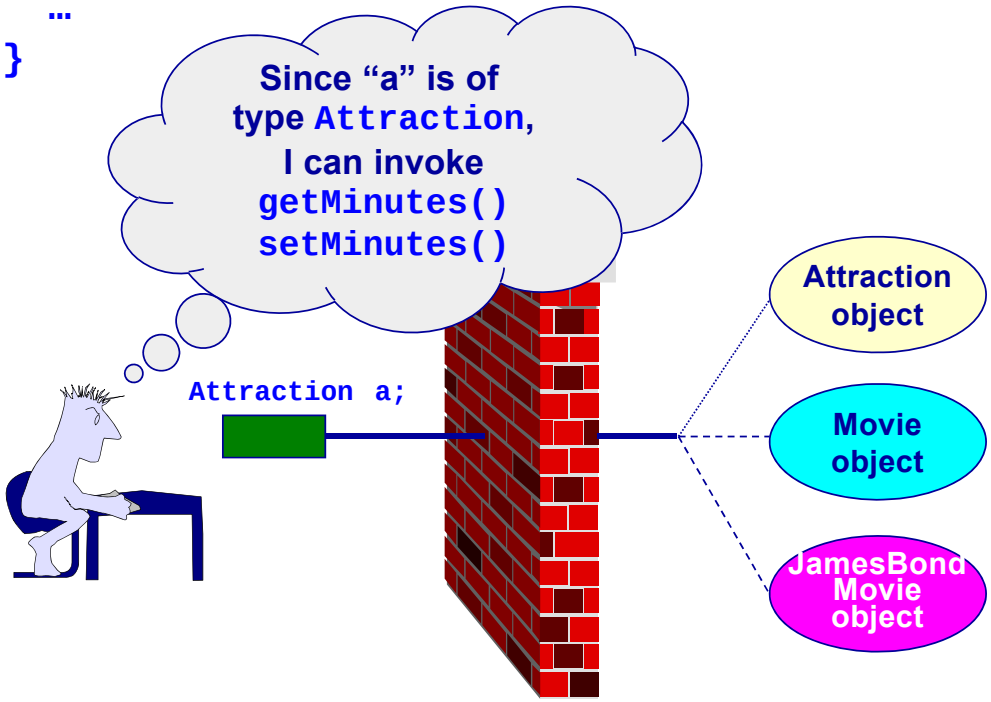
```
void someMethod(Attraction attraction) {
  int minutes = attraction.getMinutes();
  …
  attraction.setMinutes(minutes);
}
```

Since user of an object can access it's operations only through a reference variable, all she knows is the set of operations defined in the class of the variable's static type

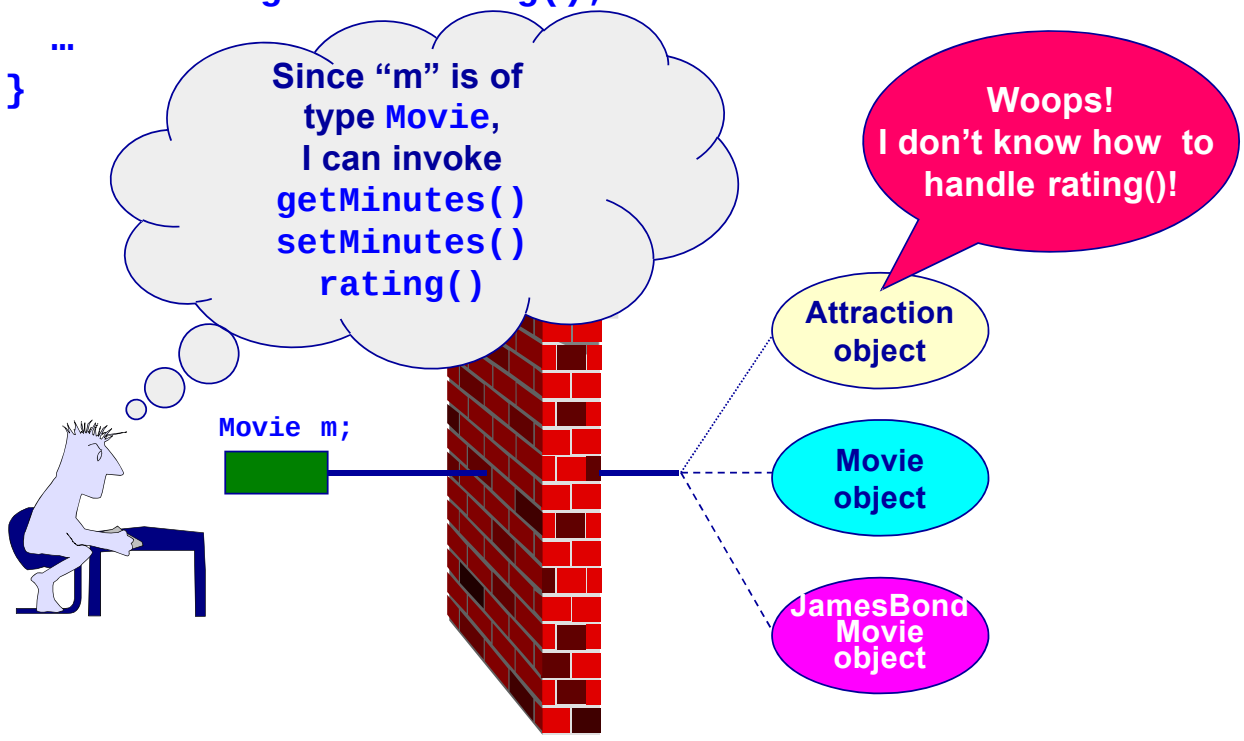
A subclass inherits operations from all of its ancestors

Therefore, it is safe to send a message for its superclass to an instance of a subclass, but *not vice versa*.

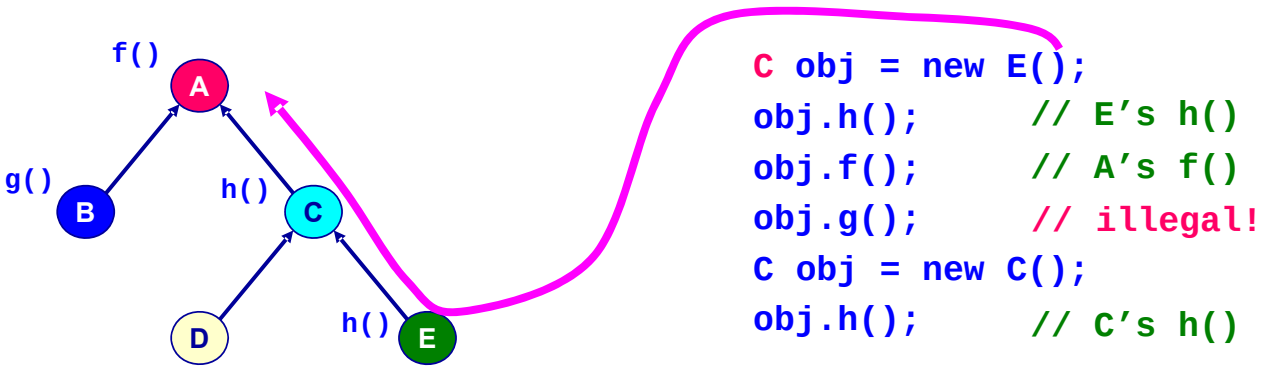
```
void someMethodA(Attraction a) {
  int minutes = a.getMinutes();
  …
}
```




```
void someMethodB(Movie m) {
  int rating = m.rating();
  …
}
```



When a message is sent to a receiver, the runtime selects an operation using dynamic type



The runtime searches up starting from the dynamic type until it finds the first invoked operation

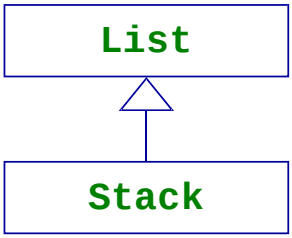
Question 7

Consider an inheritance hierarchy in which "Stack" is defined as a subclass of "List".

Item insertions and deletions can be performed only at the top of the Stack, whereas both operations are permitted at either front or rear of the List.

What do you think of this hierarchy design?

```
class Stack extends List {
        void push(Object o) { insert_front(o); }
        void pop() { delete_front(); }
        Object top() { return first(); }
}
```

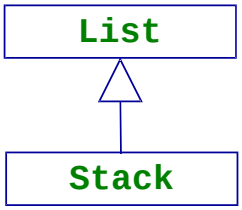


Subclass designers decide what to share: Interface or Implementation? (Cont'd)

Subclassing uses inheritance as a mechanism for *implementation sharing* (i.e., code and representation sharing)

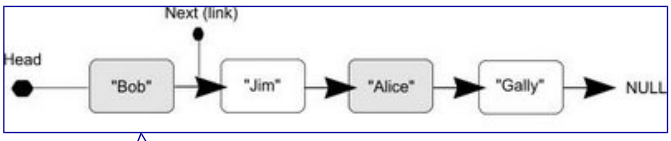
- **enforces *"implemented in terms of"* relationship**

- **also called *class inheritance* or *implementation inheritance***



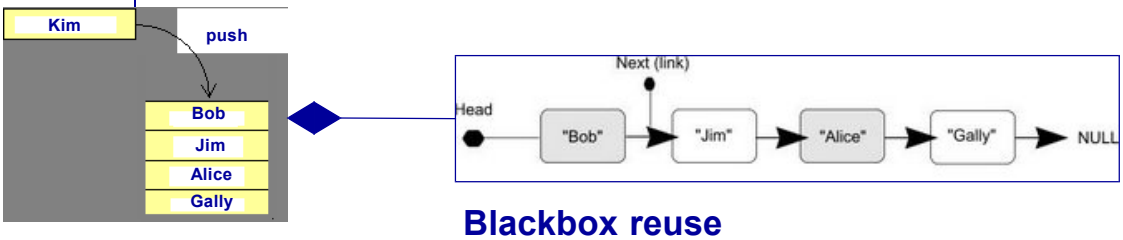
The interface of List should not be visible to the clients of Stack!

The two most common techniques for reusing functionality are *class inheritance* and *object composition*



Whitebox reuse

- **internals of parent classes are often visible to subclasses**



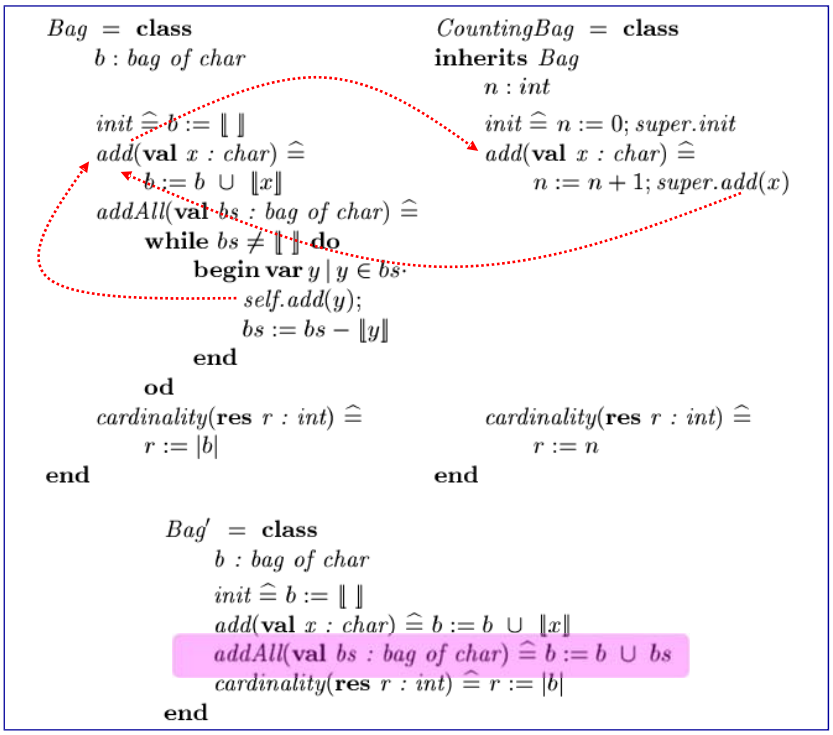
Blackbox reuse

- **no internal details of objects are visible**

Fragile Base Class (FBC) Problems: Example I

Seemingly unharmful modification to base class can cause a significant problem!

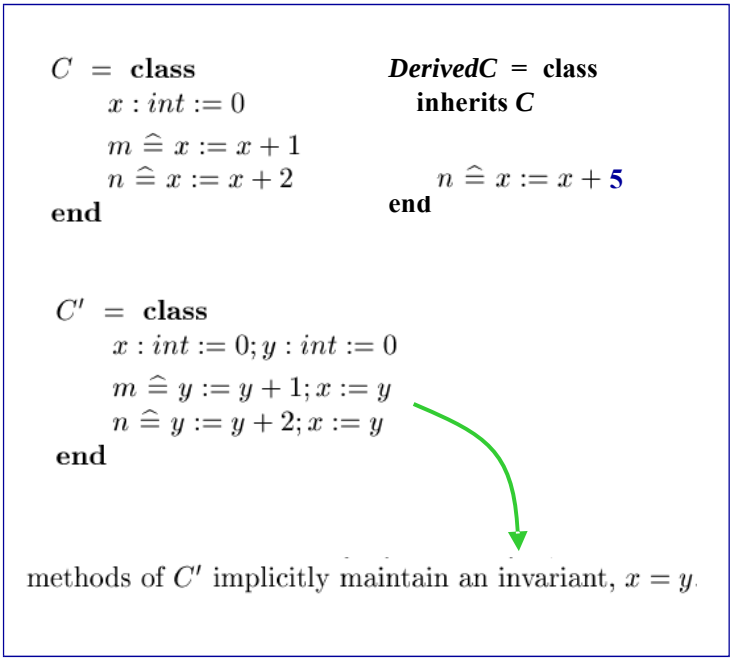
Beware of self-recursion!



Fragile Base Class (FBC) Problems: Example II

Consider a call sequence *obj.n ; obj.m*

Do not modify inherited attributes!



Question 7

Are these two codes equivalent in terms of their behavior?

```java
// In JAVA
class Foo {
    private String x = null;
    public Foo(String val) { x = val; }
    public int length() {
          System.out.println("Foo");
          return x.length();
    }
}

class Bar extends Foo {
    private int cache = 0;
    public Bar(String val) {
          super(val);
          cache = val.length();
    }
    public int length() {
          System.out.println("Bar");
          return cache;
    }
}
```

```cpp
// In C++
class Foo {
private:
    const char* x = 0;
public
    Foo(const char* val) { x = val; }
    int length() {
          cout << "Foo" << endl;
          return ::strlen(x);
    }
};

class Bar : public Foo {
private:
    int cache = 0;
public:
    Bar(const char* val) : Foo(val) {
          cache = ::strlen(val);    }
    int length() {
          cout << "Bar" << endl;
          return cache;  }
};
```


Actually, the determination of what behavior to perform may be made at compile-time or at run-time

At compile-time

- static binding or early binding
- static functions in Java/C++
- nonvirtual functions in C++

At run-time

- dynamic binding or late binding
- virtual functions in Java/C++

Even if we send the same message to objects, the behavior can be different depending on the receiver object -- *polymorphism*

Poly (multi) + *Morphism* (form), i.e., a fancy word for multi-forms

When a method must *accept* an instance of superclass as a parameter, it can accept the instance of any subclasses

Likewise, when a method must *return* an instance of superclass, it can return the instance of any subclasses

Question 8

What do you think is the real power and/or advantages of object technology?

① Code reuse
② Cost savings
③ Increased productivity
④ Elegantly tackle complexity & create easy adaptability

The real power and advantage of OT is *its capacity to tackle complex systems* and *to support easily adaptable systems*, lowering the cost and time of change

The Corporate Use of OT, Dec 1997, Cutter Group.
Prioritized reasons for adopting OT:

1. Ability to take advantage of new operating systems and tools
2. Elegantly tackle complexity & create easy adaptability
3. Cost savings
4. Development of revenue-producing applications
5. Encapsulation of existing applications
6. Improved interfaces
7. Increased productivity
8. Participation in "the future of computing"
9. Proof of ability to do OO development
10. Quick development of strategic applications
11. Software reuse

To obtain flexible and reusable systems, it is better to base the structure of software on the objects rather than on the actions

Rationale behind OO paradigm:

In general, systems evolve, functionality changes, but data objects, interfaces, and components relations tend to remain relatively stable over time.

Use it for large systems &
for systems that change often

Any other benefits?

It is essential to decompose the complex software system into smaller and smaller parts, each of which may then refine independently (i.e., *stepwise refinement*)

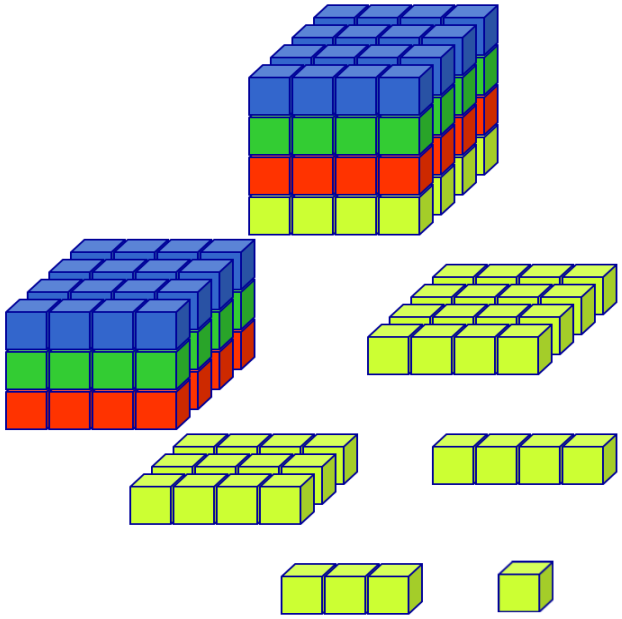
Maximum number of chunks of information an individual can simultaneous comprehend is the order of 7 ± 2

--- Miller (1956)

The technique of mastering complexity has been known since ancient times:

Divide and Conquer

-- Dijkstra (1979)



Structured vs. Object-Oriented Decompositions

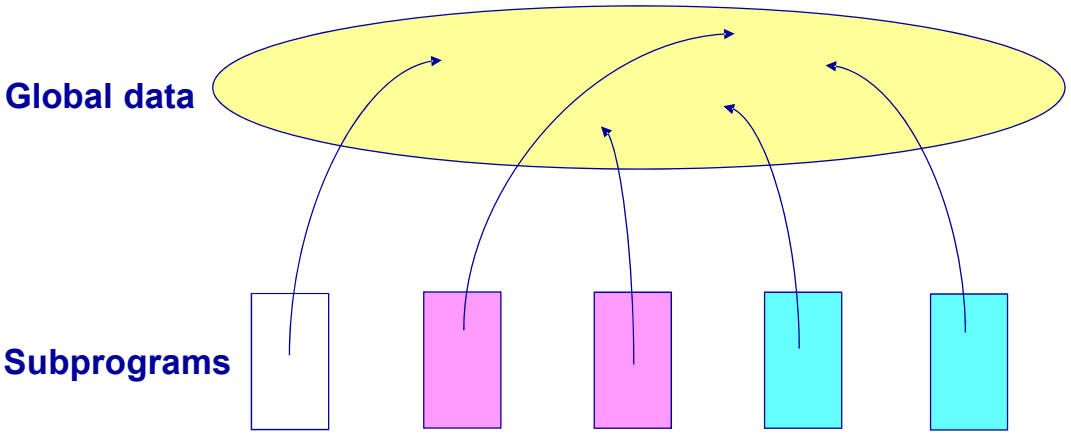
Structured Decomposition

- Organize a system around procedures/functions
- *Program = (Algorithms + Data Structures)*
- SA/SD/SP

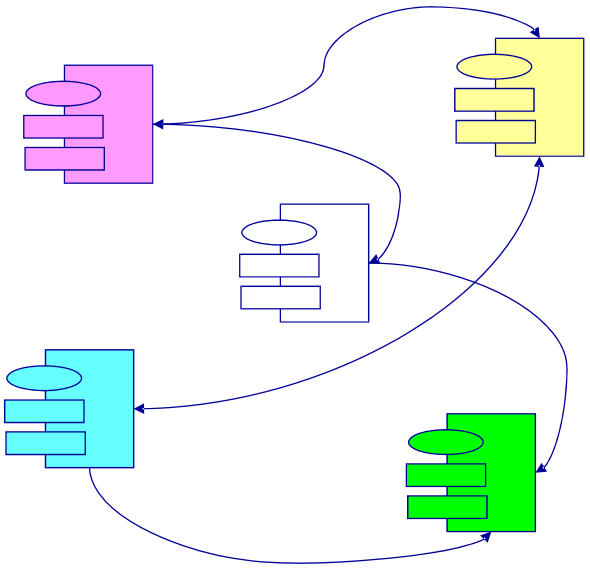
Object-Oriented Decomposition

- Organize a system around objects
- *Object = (Algorithm + Data Structures)*
- *Program = (Object + Object + …)*
- OOA/OOD/OOP

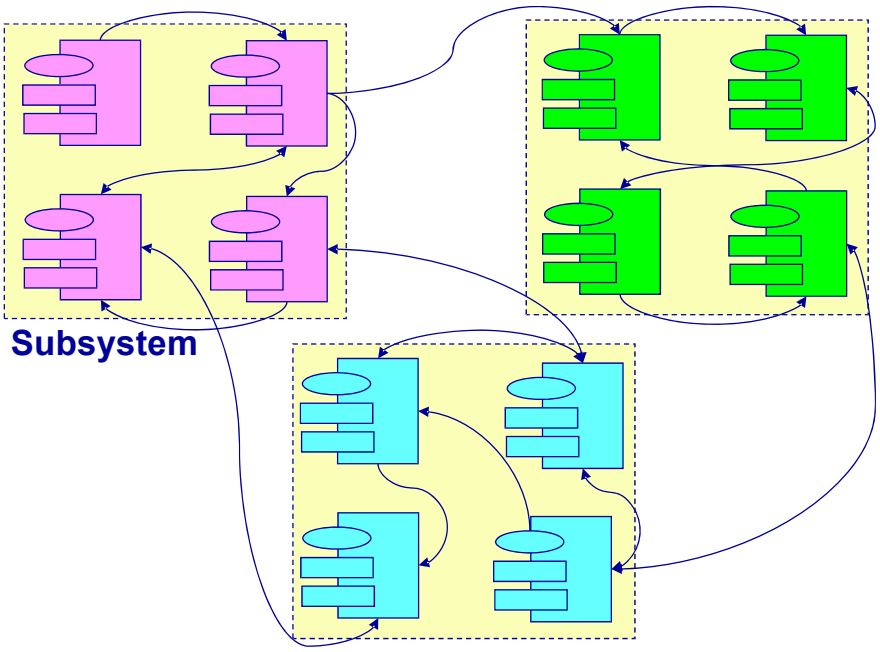
Design Structure Based on Subprograms



Design Structure Based on Objects
(Small Scale)



Design Structure Based on Objects
(Large Scale)



Reuse is not usually achieved or worthwhile at the object-level

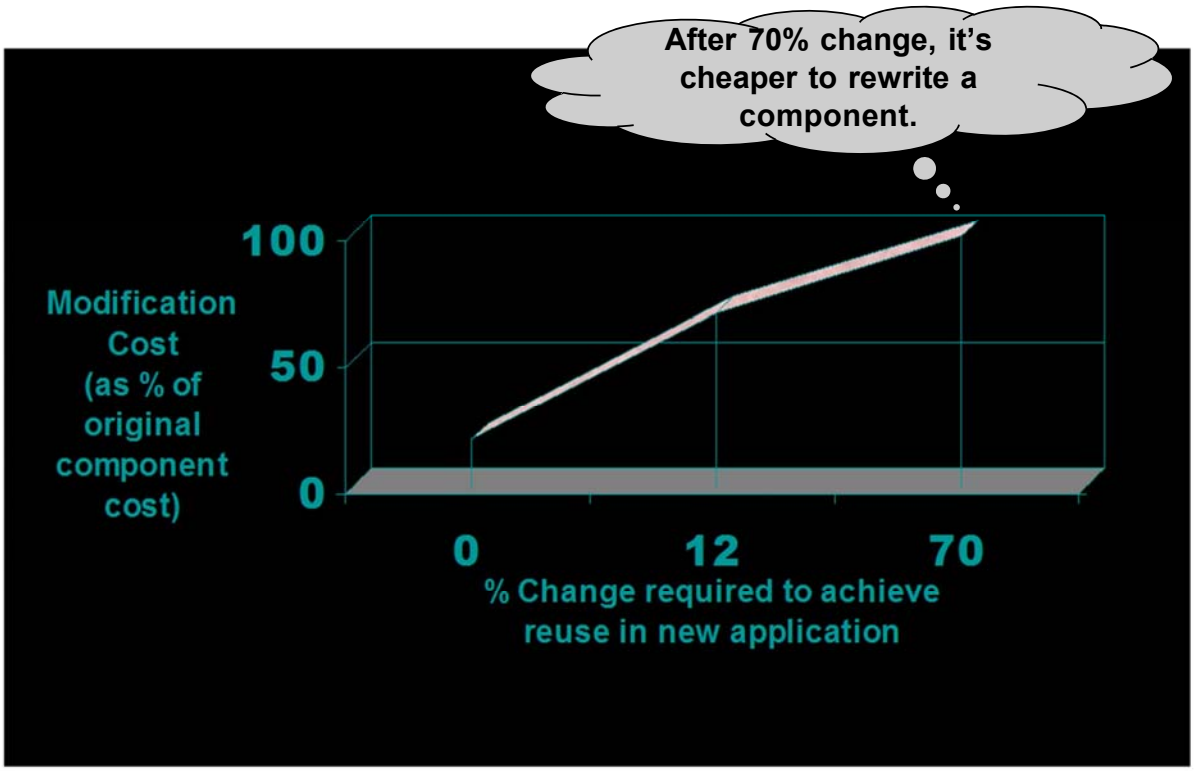
Research shows no relationship between increased reuse and collecting a library of reusable components from prior projects.

-- Communications of the ACM, pp 75-87 June, 1995

Focus on:

- **A culture of *framework* creation and use.**
- **Reuse of architecture, analysis and *design patterns***

Reuse at What Level?



Design pattern is a description of a *problem*/*solution* pair in a certain *context*

> "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 time twice."
>
> *-- Christopher Alexander*

Design Pattern Example: Adapter Pattern

Name: ***Adapter***

Also known as: Wrapper

Context: Client objects call methods of a Supplier object

Problem: Client objects expect another interface than the Supplier provides

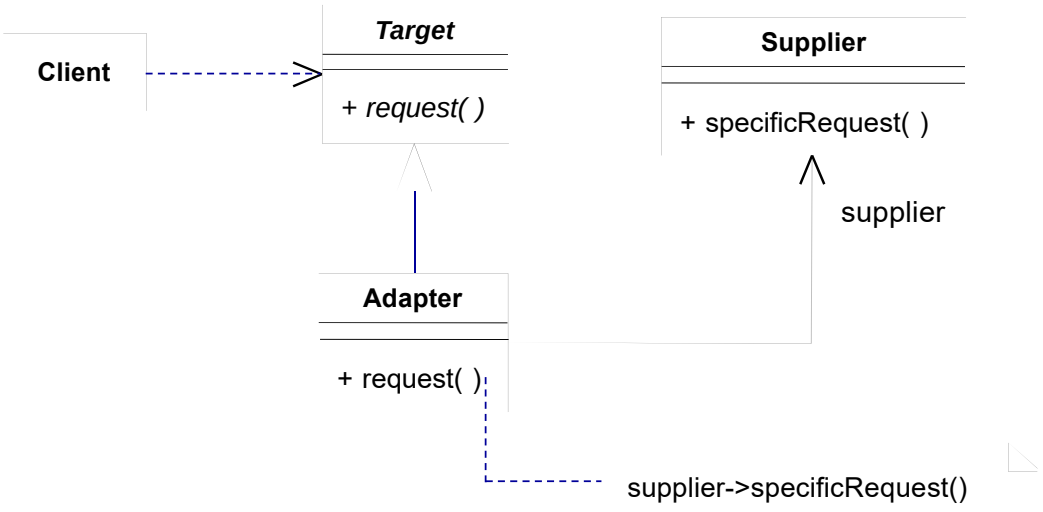
Solution: Use an Adapter object which adapts one interface to the other

Solution alternatives:

Class adapter
(relies on multiple inheritance like C++ or Eiffel, or interface inheritance like Objective C or Java)

Object adapter
(relies on single inheritance and delegation)

Object Adapter Solution



A framework is a class library, but more than an ordinary toolkit (such as math, file i/o, data structures ...)

An integrated set of cooperating classes

A *semi-complete application*: abstract framework classes are specialized in the application

Inversion of control

The "main event loop" is often in the framework, rather than in the application code

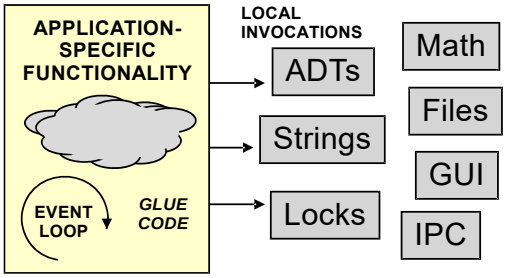
Code in the framework can invoke code in the application by dynamic binding

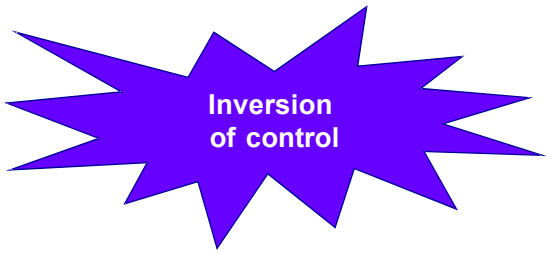
Domain-specific

business,
telecommunications,
windows system,
databases,
etc.

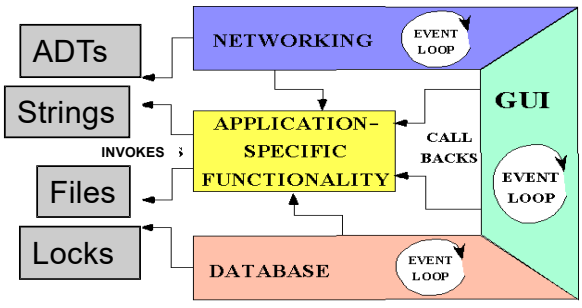
Hollywood Principle: Don't call me, we'll call you!

Class Library Architecture





Framework Architecture

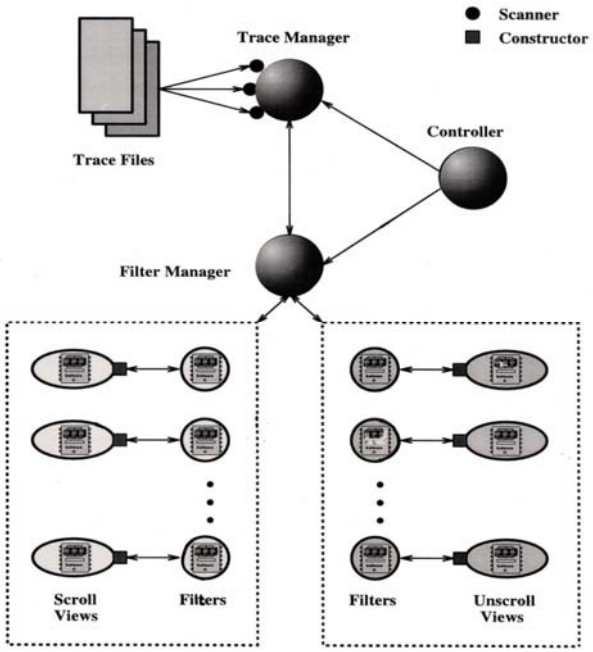


Frameworks allow us to reuse design and code

Framework: family of similar applications

Very fast application development

Powerful parameterization mechanism (subclassing and dynamic binding)



Claims

Today, (architectural & design) patterns make the most effective technique of conducting and training OOD!

Question 9

What are two most important underlying design principles behind GoF patterns?

OO Design Principle (I):
Program to an interface, not an implementation!

OO Design Principle (II)
Favor object composition over class inheritance

Designing for Change

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, Façade, Mediator, Observer

Extending functionality by subclassing
Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

Inability to alter class conveniently
Adapter, Decorator, Visitor

Exercises

1. Consider the following declaration:

class B extends A { ... } // Assume public inheritance used

What this declaration implies? Select all that apply.

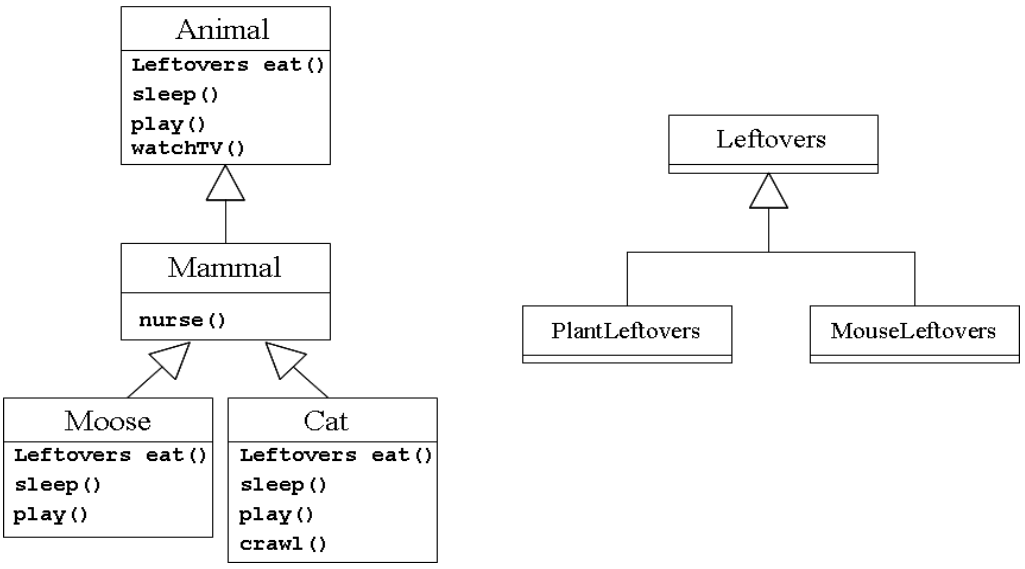
① Every object of type B is also an object of type A, but not vice-versa.

② Anything that is true of an object of type B is also true of an object of type A, but not vice-versa.

③ A represents a more general concept than B, and B represents a more specialized concept than A.

④ Anything where an object of type B can be used, an object of type A can be used as well, but not vice-versa.

Consider the following class diagrams. All the operations are public.



6. PlantLeftovers r = new Leftovers(); (a)_____

7.
```
Animal anim;
Moose myMoose = new Moose();
anim = myMoose;
anim.eat();
myMoose = anim;
```
(a)_____
(b)_____
(c)_____

8.
```
Cat myCat = new Cat();
MouseLeftover rem = myCat.eat();
```
(a)_____

※ Consider the following method ***foo*** and answer either "***Yes***" or "***No***"

```
void foo(Mammal m) { ... }
```

9. Is it legal to call *m.watchTV()* in *foo*? (a)_____
Is it legal to call *m.crawl()* in *foo*? (b)_____