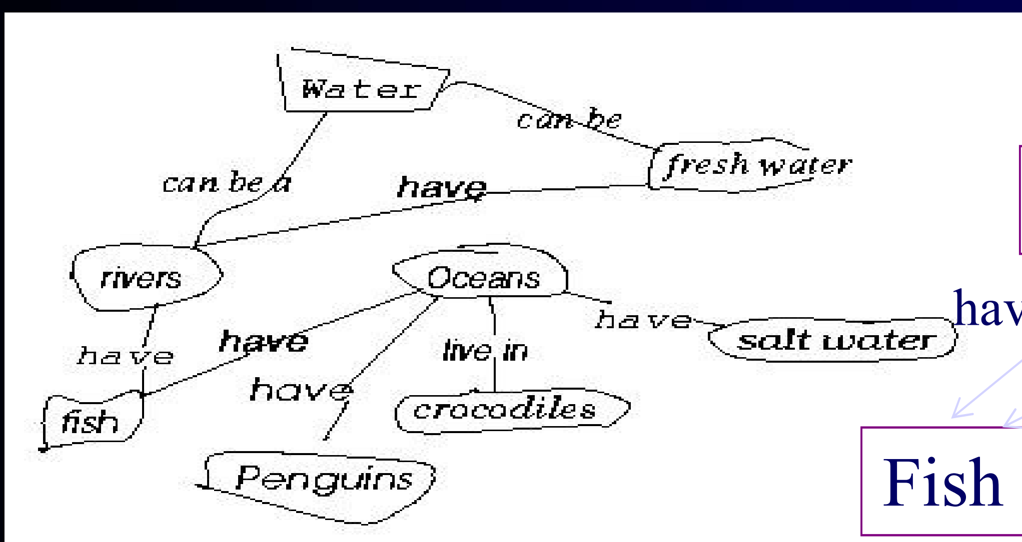
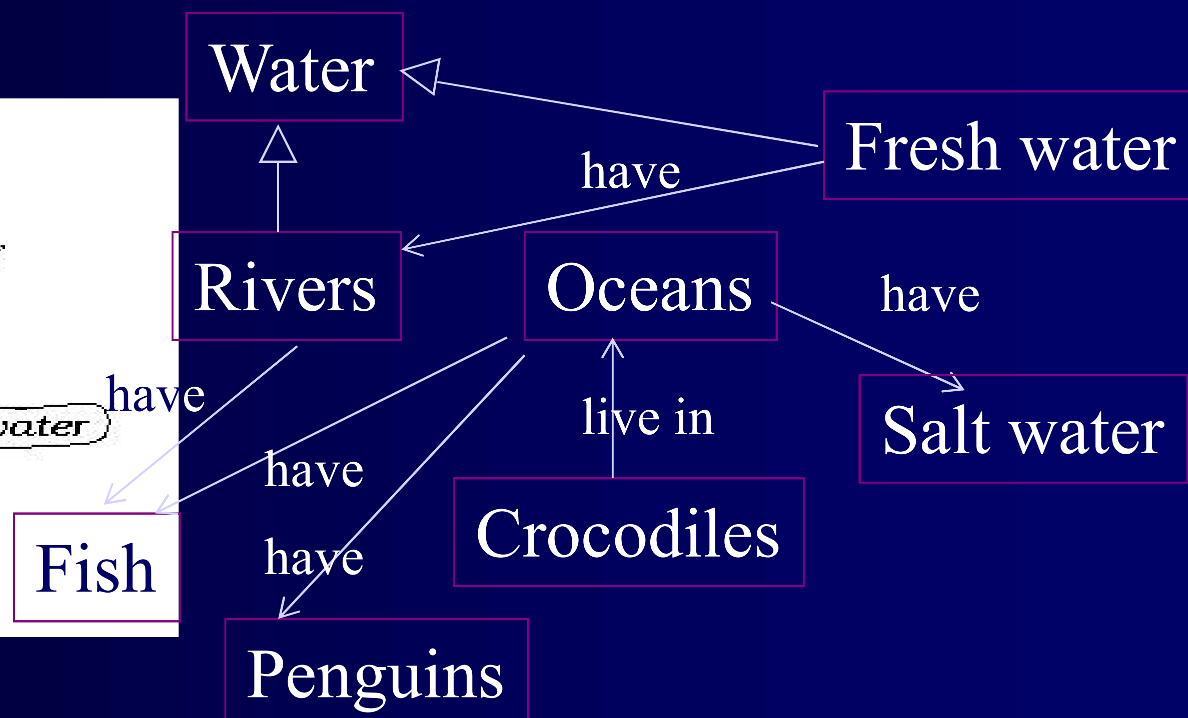
Basic Object-Oriented Concepts

What is a Model?

- A model is a simplification of reality

- A model is an *abstraction* of something for the purpose of *understanding*, be it the problem or a solution



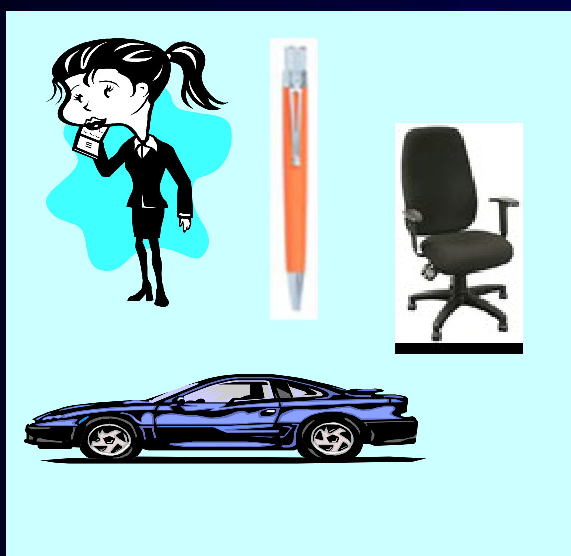


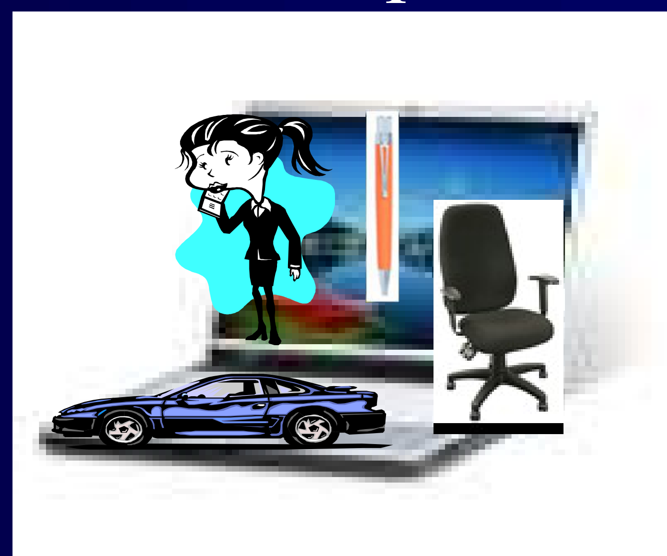
Why Do We Need Models?

- To understand *why* a software system is needed, *what* it should do, and *how* it should do it
- To communicate our understanding of why, what and how
- To detect commonalities and differences in your perception, my perception, his perception and her perception of reality
- To detect misunderstandings and miscommunications

What is an Object?

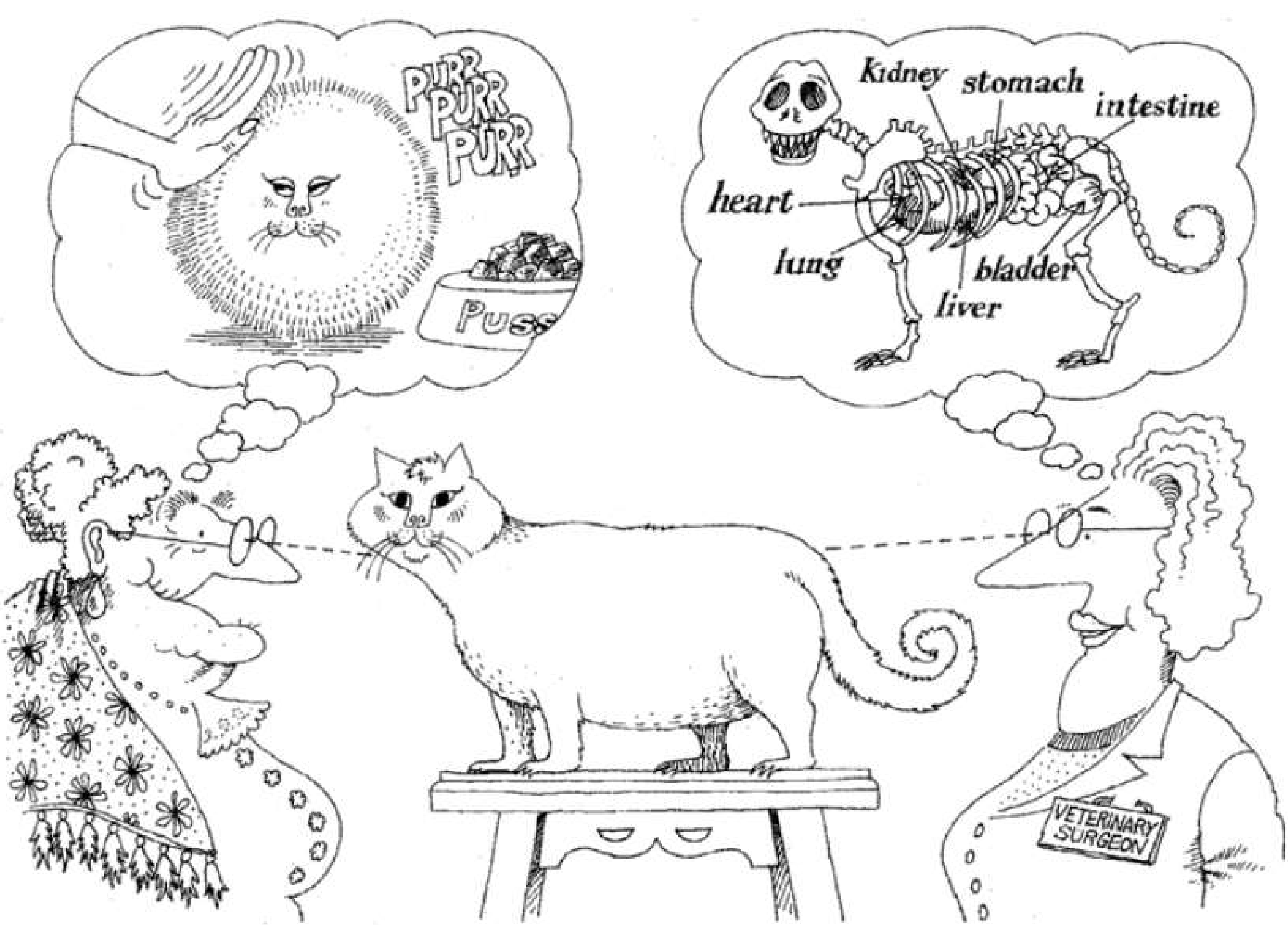
- An object is anything to which a concept applies, ***in our awareness***
- Things drawn from the problem domain or solution space
- A structure that has identity and properties and behavior
- It is an instance of a collective concept, i.e., a class





Abstraction & Encapsulation

- Abstraction
  - Focuses on the essential
  - Omits tremendous amount of details
  - Focuses on what an object "is and does"
- Encapsulation
  - Also known as Information Hiding
  - Objects encapsulate/hide
    - properties
    - behavior as a collection of methods invoked by messages
    - state as a collection of instance variables



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

■ Entity abstraction	An object that represents a useful model of a problem domain or solution domain entity
■ Action abstraction	An object that provides a generalized set of operations, all of which perform the same kind of function
■ Virtual machine abstraction	An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
■ Coincidental abstraction	An object that packages a set of operations that have no relation to each other

Concept: An object has behaviors

- In old style programming, you had:
  - data, which was completely passive
  - functions, which could manipulate any data
- An object contains both data and methods that manipulate that data
  - An object is *active,* not passive; it *does* things
  - An object is *responsible* for its own data
    - But: it can *expose* that data to other objects

Concept: An object has state

- An object contains both data and methods that manipulate that data
  - The data represent the state of the object
  - Data can also describe the relationships between this object and other objects
- Example: A CheckingAccount might have
  - A balance (the internal state of the account)
  - An owner (some object representing a person)

Example: A “Rabbit” object

- You could (in a game, for example) create an object representing a rabbit
- It would have data:
  – How hungry it is
  – How frightened it is
  – Where it is
- And methods:
  – eat, hide, run, dig

Concept: Classes describe objects

- A collection of objects that share common properties, attributes, behavior and semantics, in general
- A collection of objects with the same data structure (attributes, state variables) and behavior (function/code/operations) in the solution space

Concept: Classes describe objects

- Every object belongs to (is an instance of) a class
- The act of creating an instance is called instantiation.
- An object may have fields, or variables
  - The class describes those fields
- An object may have methods
  - The class describes those methods
- A class is like a template, or cookie cutter

Concept: Classes are like Abstract Data Types

- An Abstract Data Type (ADT) bundles together:
  - some data, representing an object or "thing"
  - the operations on that data
- Example: a `CheckingAccount`, with operations `deposit`, `withdraw`, `getBalance`, etc.
- Classes enforce this bundling together

Abstract vs. Concrete Classes

- Abstract Class.
  - An incomplete superclass that defines common parts.
  - Not instantiated.
- Concrete class.
  - Is a complete class.
  - Describes a concept completely.
  - Is intended to be instantiated.

Example of a class

```
class Employee {
   // fields
   String name;
   double salary;

   // a method
   void pay () {
      System.out.println("Pay to the order of " +
                                        name + " $" + salary);
   }
}
```

Approximate Terminology

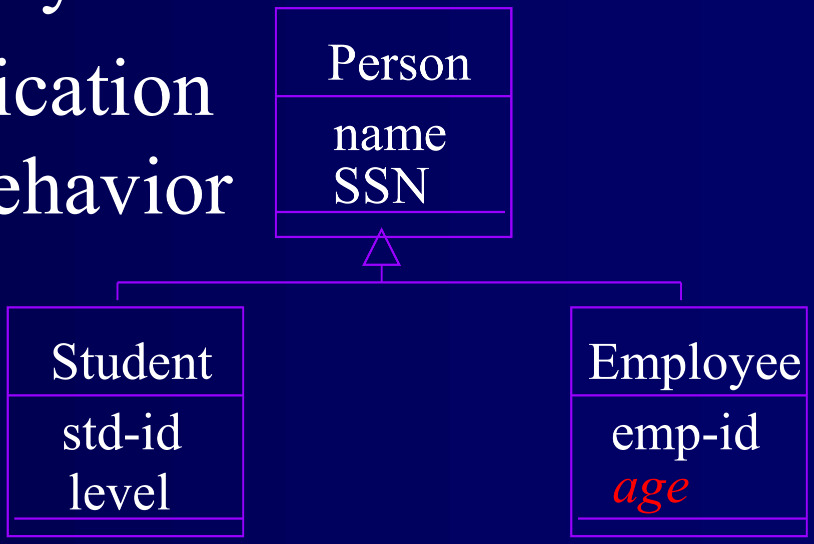
- instance = object
- field = instance variable
- method = function
- sending a message to an object = calling a function
- These are all *approximately* true

Concept: Classes form a hierarchy

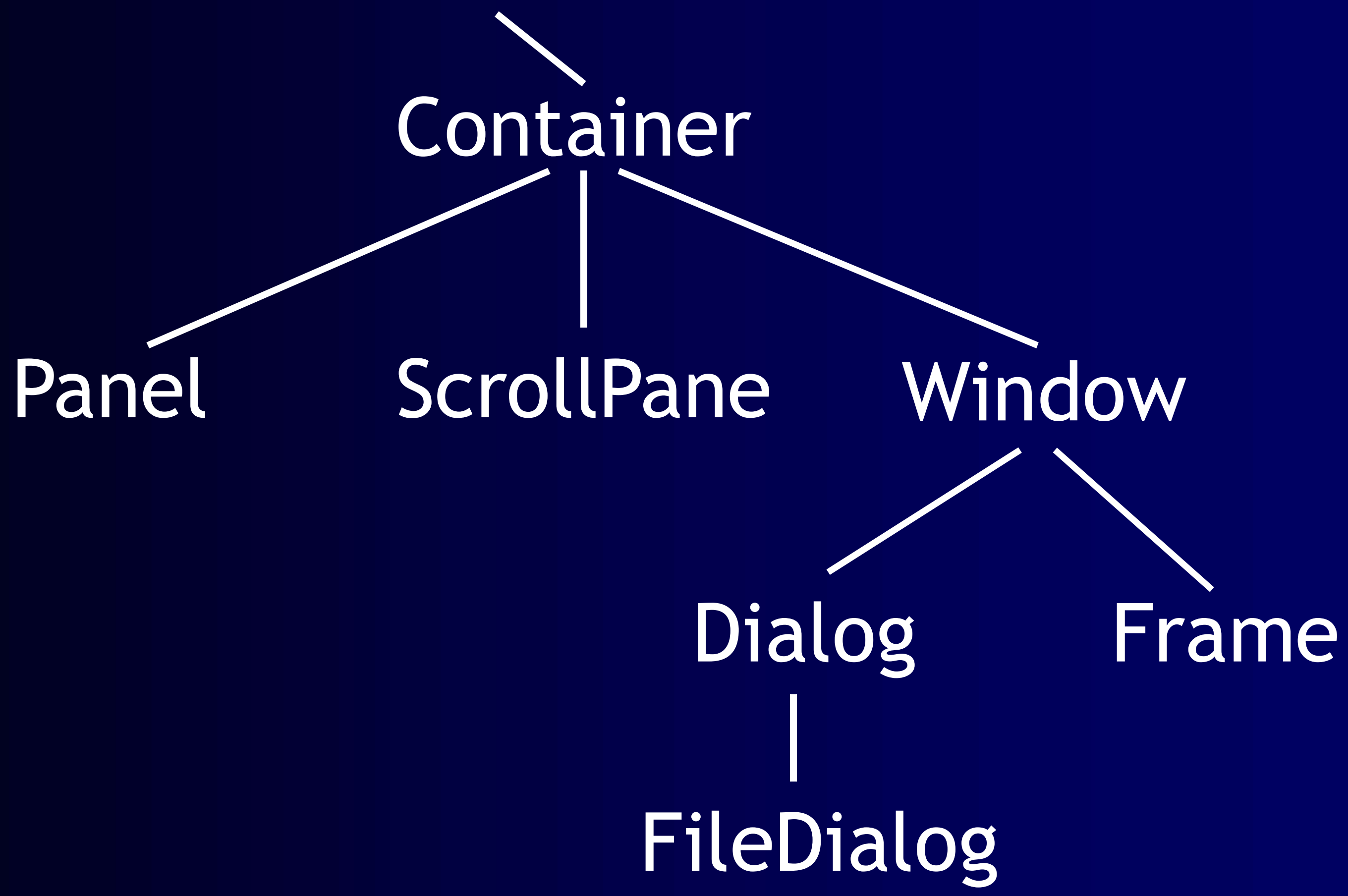
- Classes are arranged in a treelike structure called a hierarchy
- The class at the root is named `Object`
- Every class, except `Object`, has a superclass
- A class may have several ancestors, up to `Object`
- When you define a class, you specify its superclass
  - If you don't specify a superclass, `Object` is assumed
- Every class may have one or more subclasses

Terminology

- Specialization: The act of defining one class as a refinement of another
- Subclass: A class defined in terms of a specialization of a superclass using inheritance
- Superclass: A class serving as a base for inheritance in a class hierarchy
- Inheritance: Automatic duplication of superclass attribute and behavior definitions in subclass



Example of (part of) a hierarchy



A FileDialog is a Dialog is a Window is a Container

C++ is different

- In C++ there may be more than one root
  - but not in Java!
- In C++ an object may have more than one parent (immediate superclass)
  - but not in Java!
- Java has a single, strict hierarchy

Concept: Objects inherit from their superclasses

- A class describes fields and methods
- Objects of that class have those fields and methods
- But an object *also* inherits:
  - the fields described in the class's superclasses
  - the methods described in the class's superclasses
- A class is *not* a complete description of its objects!

Example of inheritance

```
class Person {
   String name;
   String age;
   void birthday () {
      age = age + 1;
   }
}
```

```
class Employee
         extends Person {
   double salary;
   void pay () { ...}
}
```

Every Employee has a name, age, and birthday method *as well as* a salary and a pay method.

Concept: Objects must be created

- int n; does two things:
  - it declares that n is an integer variable
  - it allocates space to hold a value for n
- Employee secretary; does *one* thing
  - it declares that secretary is type Employee
- secretary = new Employee ( ); allocates the space

Notation: How to declare and create objects

Employee secretary; // declares secretary

secretary = new Employee (); // allocates space

Employee secretary = new Employee(); // both

- But the secretary is still "blank"

secretary.name = "Adele"; // dot notation ←

secretary.setName () = “Sauron"; //setter

secretary.getName (); //getter

secretary.birthday (); // sends a message/calls function

Notation: How to reference a field or method

- Inside a class, no dots are necessary

```
class Person { … age = age + 1; …}
```

- Outside a class, you need to say which object you are talking to

```
if (john.getAge () < 75) john.birthday ();
```

- If you don't have an object, you cannot use its fields or methods!

Concept: this object

- Inside a class, no dots are necessary, because
  - you are working on this object
- If you wish, you can make it explicit:

```
class Person { … this.age = this.age + 1; …}
```

- this is like an extra parameter to the method
- You usually don't need to use this

Concept: A variable can hold subclass objects

- Suppose B is a subclass of A
  - A objects can be assigned to A variables
  - B objects can be assigned to B variables
  - B objects can be assigned to A variables, but
  - A objects can *not* be assigned to B variables
    - Every B is also an A *but* not every A is a B
- You can cast: `bVariable = (B) aObject;`
  - In this case, Java does a runtime check

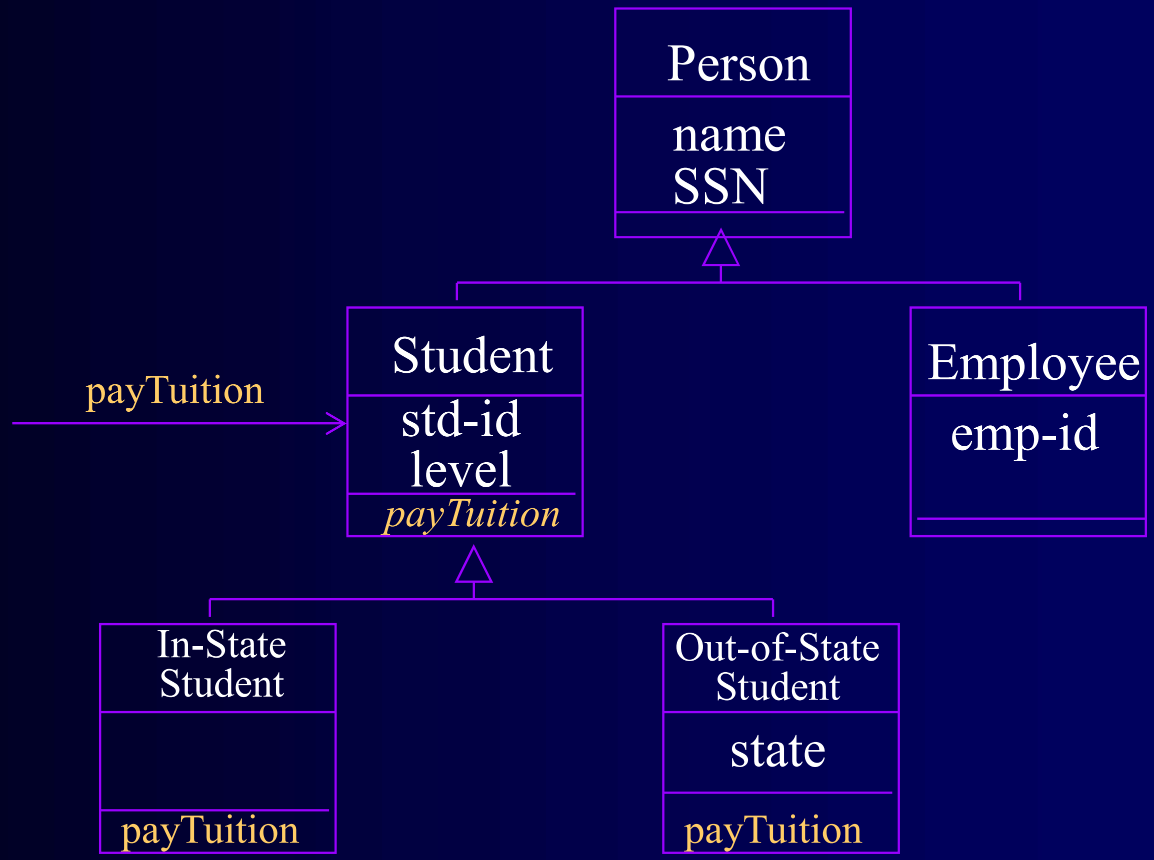
Example: Assignment of subclasses

```
class Dog { ... }
class Poodle extends Dog { ... }
Dog myDog;
Dog rover = new Dog ();
Poodle yourPoodle;
Poodle fifi = new Poodle ();

myDog = rover;                    // ok
yourPoodle = fifi;                // ok
myDog = fifi;                     //ok
yourPoodle = rover;               // illegal
yourPoodle = (Poodle) rover;      //runtime check
```

Polymorphism

- Objects of different classes respond to the same message differently.



Concept: Methods can be overridden

```
class Bird extends Animal {
  void fly (String destination) {
    location = destination;
  }
}
class Penguin extends Bird {
  void fly (String whatever) { }
}
```

- So birds can fly. Except penguins. And ostriches.

Concept: Don't call functions, send messages

```
Bird someBird = pingu;
someBird.fly ("South America");
```

- Did `pingu` actually go anywhere?
  - You sent the message `fly(…)` to `pingu`
  - If `pingu` is a penguin, he ignored it
  - otherwise he used the method defined in `Bird`
- You did *not* directly call any method

Sneaky trick: You can still use overridden methods

```
class FamilyMember extends Person {
  void birthday () {
    super.birthday ();  // call overridden method
    givePresent ();     // and add your new stuff
  }
}
```

Concept: Constructors make objects

- Every class has a constructor to make its objects
- Use the keyword `new` to call a constructor

```
secretary = new Employee ( );
```

- You can write your own constructors; but if you don't,
- Java provides a default constructor with no arguments
  - It sets all the fields of the new object to null
  - If this is good enough, you don't need to write your own
- The syntax for writing constructors is almost like that for writing methods

Syntax for constructors

- Instead of a return type and a name, just use the class name
- You can supply arguments

```
Employee (String theName, double theSalary) {
  name = theName;
  salary = theSalary;
}
```

Trick: Use the same name for a parameter as for a field

- A parameter overrides a field with the same name
- But you can use `this.`name to refer to the field

```
Person (String name, int age) {
  this.name = name;
  this.age = age;
}
```

- This is a very common convention

Internal workings: Constructor chaining

- If an Employee is a Person, and a Person is an Object, then when you say new Employee ()
  - The Employee constructor calls the Person constructor
  - The Person constructor calls the Object constructor
  - The Object constructor creates a new Object
  - The Person constructor adds its own stuff to the Object
  - The Employee constructor adds its own stuff to the Person

The case of the vanishing constructor

- If you don't write a constructor for a class, Java provides one (the *default constructor*)
- The one Java provides has no arguments
- If you write *any* constructor for a class, Java does *not* provide a default constructor
- Adding a perfectly good constructor can break a constructor chain
- You may need to fix the chain

Example: Broken constructor chain

```
class Person {
   String name;
   Person (String name) { this.name = name; }
}
class Employee extends Person {
   double salary;
   Employee ( ) {
      // here Java tries to call new Person() but cannot find it;
      salary = 12.50;
   }
 }
```

Fixing a broken constructor chain

- Special syntax: `super(...)` calls the superclass constructor
- When one constructor calls another, that call *must be first*

```
class Employee {
 double salary;
   Employee (String name) {
      super(name);  // must be first
      salary = 12.50;
   }
}
```

- Now you can only create Employees with names
- This is fair, because you can only create Persons with names

Trick: one constructor calling another

- `this(...)` calls another constructor for this same class

```
class Something {
   Something (int x, int y, int z) {
      // do a lot of work here
   }
   Something ( ) { this (0, 0, 0);  }
}
```

- It is poor style to have the same code more than once
- If you call `this(...)`, that call *must be the first thing* in your constructor

Concept: You can control access

```
class Person {
    public String name;
    private String age;
    protected double salary;
    public void birthday { age++; }
}
```

- Each object is responsible for its own data
- Access control lets an object protect its data
- We will discuss access control shortly

Concept: Classes themselves can have fields and methods

- Usually a class describes fields (variables) and methods for its objects (instances)
  - These are called instance variables and instance methods
- A class can have its own fields and methods
  - These are called class variables and class methods
- There is exactly *one* copy of a class variable, not one per object
- Use the special keyword `static` to say that a field or method belongs to the class instead of to objects

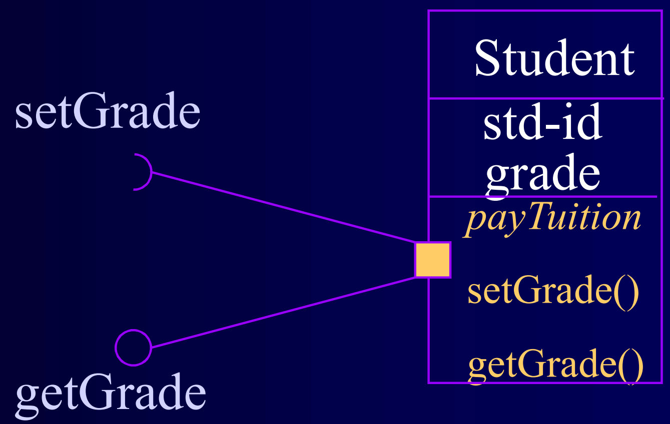
Example of a class variable

```
class Person {
    String name;
    int age;
    static int population;
    Person (String name) {
        this.name = name;
        this.age = 0;
        population++;
    }
}
```

Interfaces

- Information hiding - all data should be hidden within a class, at least in principle
- make all data attributes private
- provide public methods to get and set the data values

- e.g. Grade information is usually confidential, hence it should be kept private to the student. Access to the grade information should be done through interfaces, such as setGrade and getGrade



Advice: Restrict access

- Always, *always* strive for a narrow interface
- Follow the principle of information hiding:
  - the caller should know as little as possible about how the method does its job
  - the method should know little or nothing about where or why it is being called
- Make as much as possible private

Advice: Use setters and getters

```
class Employee extends Person {
  private double salary;
  public void setSalary (double newSalary) {
    salary = newSalary;
  }
  public double getSalary () { return salary; }
}
```

- This way the object maintains control
- Setters and getters have conventional names

Kinds of access

- Java provides four levels of access:
  - public: available everywhere
  - protected: available within the package (in the same subdirectory) and to all subclasses in other packages via inheritance
  - [default]: available within the package
  - private: only available within the class itself
- The default is called package visibility

What is OOAD?

- Analysis — understanding, finding and describing concepts in the problem domain.
- Design — understanding and defining software solution/objects that *represent* the analysis concepts and will eventually be implemented in code.
- OOAD — Analysis is object-oriented and design is object-oriented. A software development approach that emphasizes a logical solution based on objects.

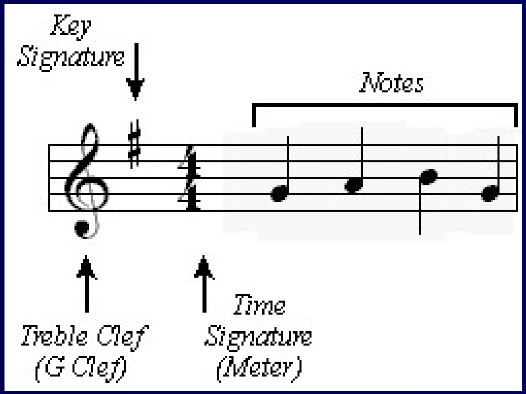
Involves both a notation and a process

Notation vs. Process

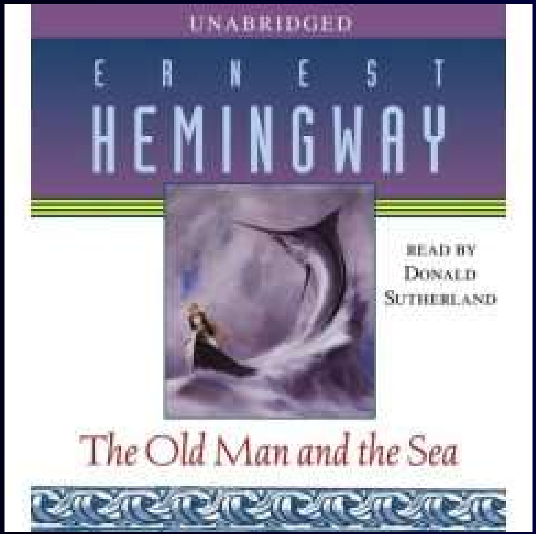
- UML is a notation

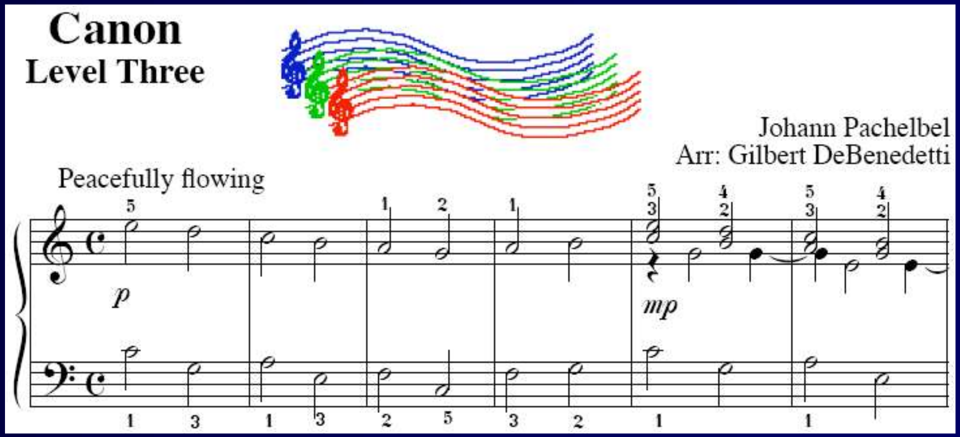
Other notations are

English, Elvish, or any medium that enables communication

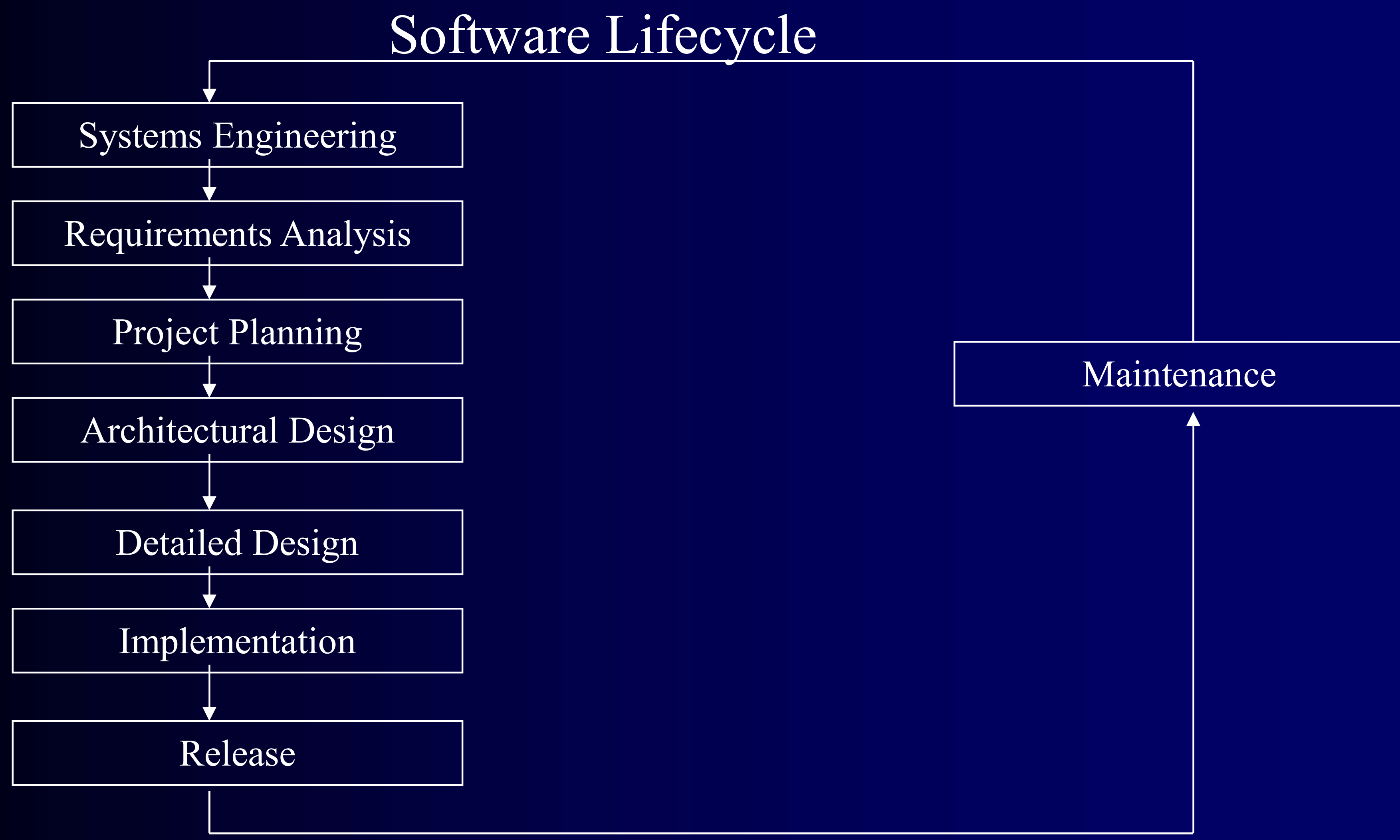


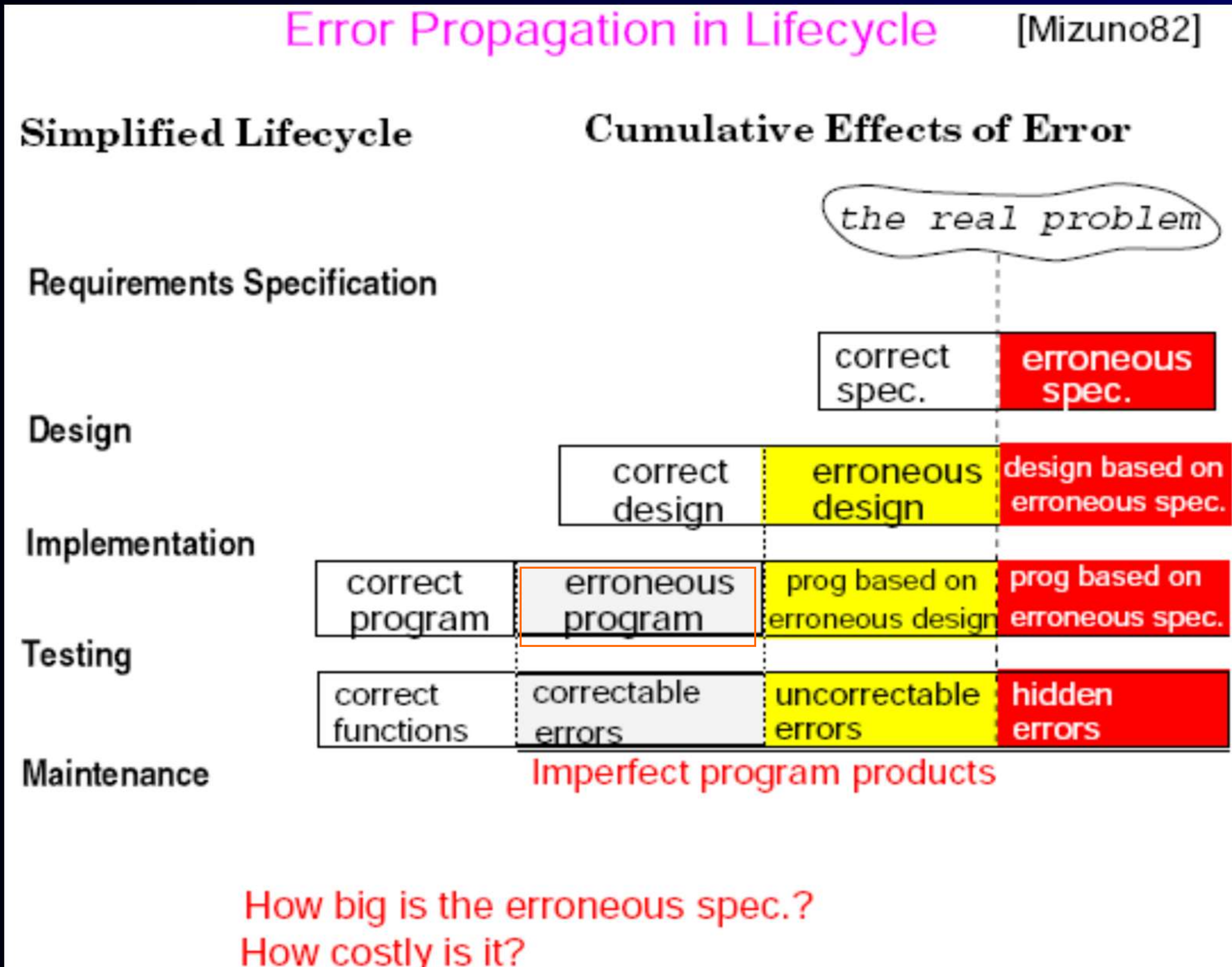
But notation doesn't mean much until





Where to Use OO?





Artificial problem

Accidental design

Traceability!

Why RE?

How big is the "erroneous specification"?

? **Bell Labs and IBM studies**

80% of all defects are inserted in the requirements phase.
Improving the requirements definition process reduces
the amount of testing and rework required.

And the above figures do not include the end user losses
who have to live with poor software on a daily basis[Testing Techniques Newsletter]

? **U.S. Air Force projects**

36% of all defects were due to faulty requirements translation.
Only 9% of these errors were resolved (in the requirements phase)[Sheldon92]

? **Voyager and Galileo spacecraft**

Of the 197 significant software faults found during integration & sy
only 3 of those errors were programming errors;
the vast majority of the faults were requirements problems. [Lutz93

? **Application Specific Integrated Circuits [ASICs)**

>1/2 are faulty on first fabrication. A majority of these faults are related to reqs.

? **[UK Health and Safety] Executive**

Specification 44.1%
Design and Implementation 14.7%
Installation and Commissioning 5.9%

Operation and Maintenanc
Changes after commission

[Her Majesty's Stationary Office 1995 ISBN 0 7176 0847 6]

Why RE?

How costly are requirements errors?

[Lindstrom93]

Get the requirements wrong, you'll destroy the project.

[Boehm87]

COST (correcting design/implementation errors)
= 100 X COST (correcting requirements errors)

[Humphrey, Managing the Software Process, Ch1, p11-12]

a useful rule of thumb: It takes about 1 to 4 working hours to find and fix a bug through inspections and about 15 to 20 working hours to find and fix a bug in function or system test.

[Curtis88]

Three most frequent problems plaguing large software systems:

communication and coordination
thin spread of domain application knowledge
changing and conflicting requirements

Defining the problem is The Problem

Why RE?

How costly are requirements errors?

[Lindstrom93]

Get the requirements wrong, you'll destroy the project.

[Boehm87]

COST (correcting design/implementation errors)
= 100 X COST (correcting requirements errors)

[Humphrey, Managing the Software Process, Ch1, p11-12]

a useful rule of thumb: It takes about 1 to 4 working hours to find and fix a bug through inspections and about 15 to 20 working hours to find and fix a bug in function or system test.

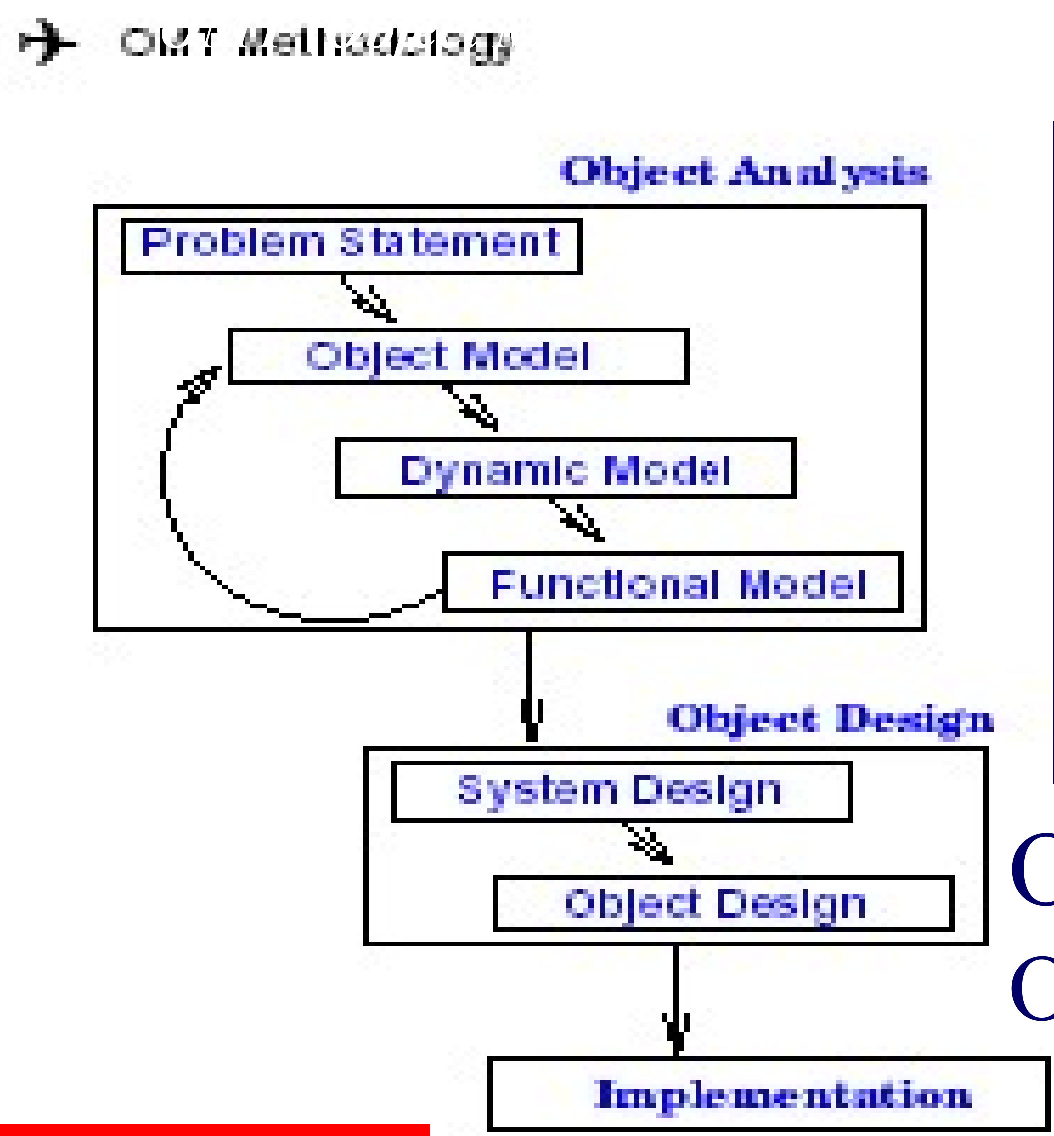
[Curtis88]

Three most frequent problems plaguing large software systems:

communication and coordination
thin spread of domain application knowledge
changing and conflicting requirements

Defining the problem is The Problem

Rumbaugh's Object Modeling Technique



Object Model: describes the ***static*** structure of the objects in the system and their relationships -> Object Diagrams.

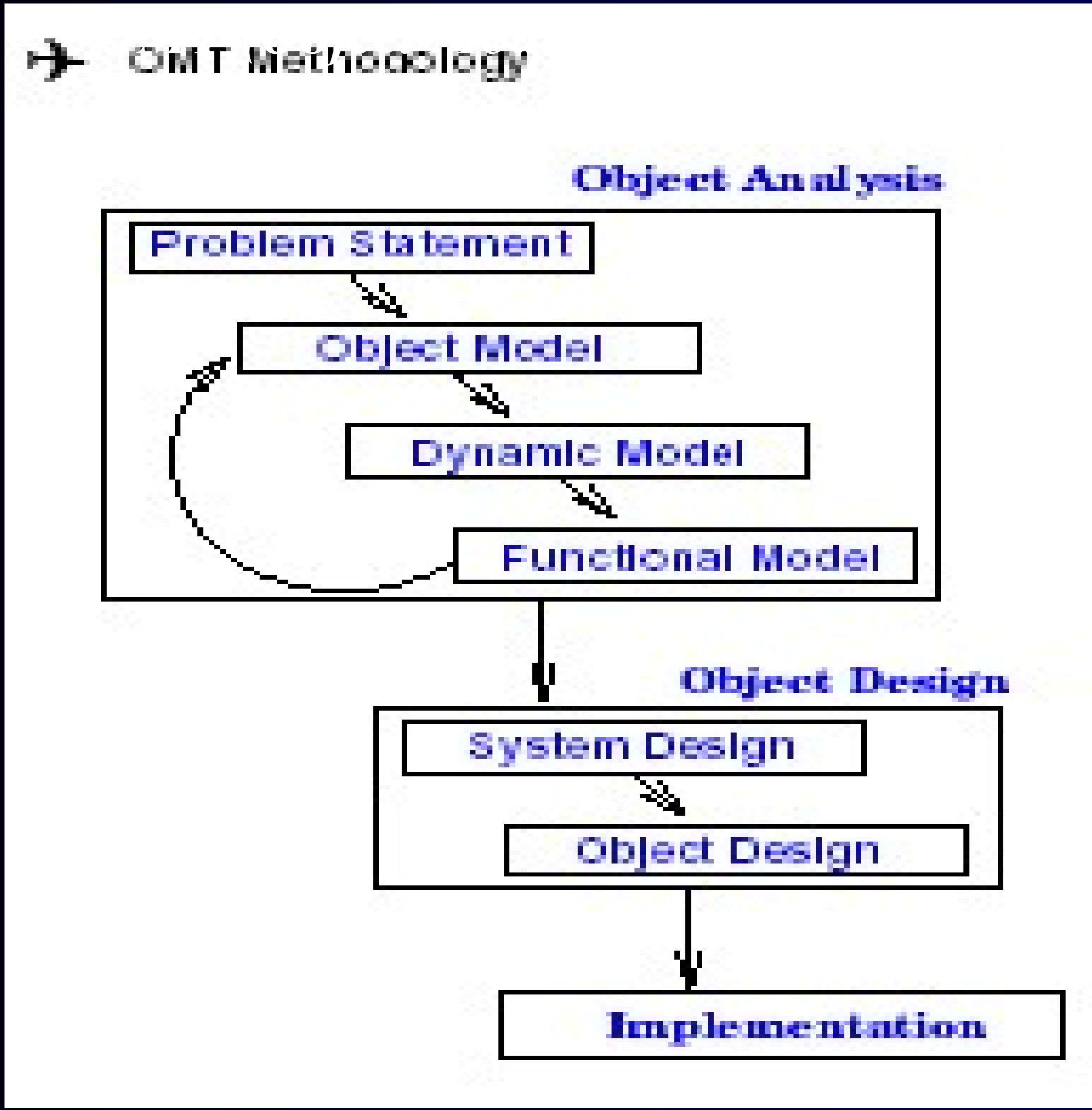
Dynamic Model: describes the ***interactions*** among objects in the system -> State Diagrams.

Functional Model: describes the data ***transformation*** of the system -> DataFlow Diagrams.

OMT as an OO Methodology

Traceability!

How to Do OOAD



Analysis:

i) Model the ***real world*** showing its important properties;

ii) Concise model of what the ***system*** will do

System Design:

Organize into subsystems based on analysis structure and propose ***architecture***

Object Design: Based on analysis model but with implementation details; Focus on ***data structures and algorithms*** to implement each class; Computer and domain objects

Implementation: Translate the object classes and relationships into a ***programming*** language

Traceability!

OOAD Historical Perspective

OO Technology	***Process Perspective***
OO Prog. Languages (Java, C++)	just program!
OO Design (Booch)	design then program
OO Analysis (Rumbaugh, Jacobson)	Analyze (use case) first, then design, then program

T