CS314, Colorado State University
Software Engineering
Notes 2: Object-Oriented Design & Implementation Concepts

James M. Bieman

Focus: The Connection between OO Designs and OO Code

- Review object-oriented concepts.
- Introduce the UML design notation.
- Show the relationship between designs and program code.
- Demonstrate the process of implementing a design.

Object-Oriented Paradigm

- "Programming as simulation".
  Objects represent "real world" or virtual entities.
- Entities have state and and behavior.
  - State is hidden.
  - Behavior is accessed through public interfaces.

Interacting With an Object

- State viewed through interfaces:
  - Gauges.
  - Tachometer.
- Actions through access methods.
  - Gas pedal.
  - Steering wheel.

Objects & Encapsulation

- Public methods supply services.
- Private representation of object state.
- Private method implementations.
- Language enforcement of encapsulation.

Abstraction Mechanisms in Programming Languages

- Control abstraction:
  Loops, if-then-else constructs.
- Procedural abstraction:
  Procedures, functions, algorithms.
- Data abstractions:
  Abstract data type (ADT).
- Object abstraction: Class

...

Names identify *realizations* of abstractions.

Names in Software Engineering

- Juliet Capulet:
  "What's in a name? That which we call a rose
  By any other name would smell as sweet."
- Was Juliet right?
- Why do we talk about names in a software engineering course?
- Names should be descriptive of their use.

How not to name things

(from "How to write unmaintainable code")

- Take names from a baby naming book:
  Fred, Susan, Bob, Jane are great names.
- Single letter names.
- Creative misspelling.
- Be abstract: it, data, stuff.
- Use acronyms.
- Use alternate vocabulary to refer to the same action:
  display, show, present.
- Use names from other languages.
  …

Stack ADT

- Private state:
  Stores values of stack items.
- Public operations:
  push, pop, top, isEmpty, isFull.

Stack *object* --- instantiated stack ADT.

General Properties of Stack Objects (& other objects)

- May have several instantiated stack objects.
- Each stack object has its own representation.
  Each object may have different items stored.
- All stack items respond to the same messages.
  Operations encoded as methods in the class definition.

To Develop an OO System

- Identify object services.
- Identify system objects.
- Determine connection between objects.
- Implement by defining object classes.

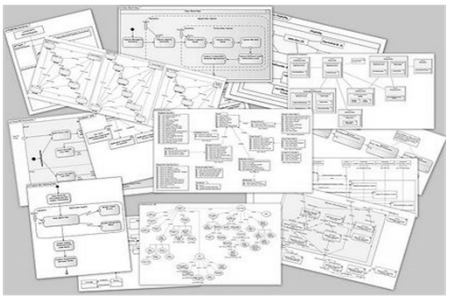
Modeling Classes & Objects

- Class: description of a set of similar objects.
- Class definition:
  - Class name.
  - State representation.
  - Public interface.
  - Private implementation of methods.

Unified Modeling Language (UML)

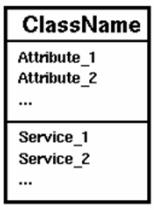
- Unifies several prior modeling notations:
  Booch, Rumbaugh (OMT), Jacobson, Shlaer-Mellor, Coad-Yourdan, Wirfs-Brock.
- UML diagram types:
  Class diagrams, object diagrams, use case diagrams, interaction diagrams, package diagrams, sequence diagrams, state diagrams, activity diagrams, deployment diagrams, component diagrams.
- Owned and managed by the Object Modeling Group.
  Members: IBM, Lockheed Martin, Eclipse, Raytheon, Hewlett-Packard, AT&T, Boeing, Red Hat, Sandia Nat. Lab., NASA, MITRE, Qualcomm, Samsung, Saab, and others.

UML is a family of model types

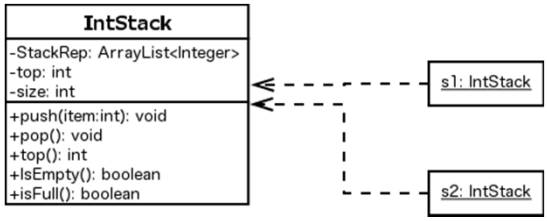


[Wikipedia image]

UML Class Model



Class Stack and Two Stack Objects in UML



Methods (Member Functions)

- Provide an object's services.
  Stack services: *push, pop, top, isEmpty*, and *isFull.*
- Public interface: method names & parameters.
- Private method bodies: implement the method.

Constructors

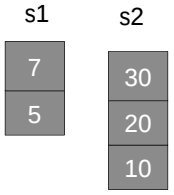
- Methods that define how an object is initialized.
- Run when an object is first created.
- Can be parameterized.
- One class can have several constructors.
  - Pattern selects a constructor.

Objects Are Dynamically Created at Run Time

- To create stack objects in Java:
  Stack s1 = new Stack();
  Stack s2 = new Stack();
- Stacks s1 and s2 reference different stacks.

Push Items Onto the Stacks

```
s1.push(5);
s1.push(7);
s2.push(10);
s2.push(20);
s2.push(30);
```

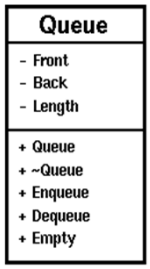


Queue Example

- Attributes:
  - Queue representation, length, max size, location of front & back of Queue.
- Operations or functions:
  - Queue(), ~Queue() (in c++)
  - Enqueue(DataItem data), Dequeue()
  - Empty(), Full()

UML Class Diagram of the Queue Class

- Private attributes & operations indicated with a -
- Public attributes & operations indicated with a +



Information Hiding

- Module implementation details are inaccessible (hidden) from other modules.
  Ex: front, back, & length of prior example.
- Why limit access?
  - Protects module against outside interference.
  - Prevents other modules from depending on implementation details.
  - Modification is easier: Local effects of changes.

Encapsulation

- Think of an ADT as a unit or object: don't worry about implementation details (from a higher level of abstraction).
- Again: The combination of data (characteristics) with the methods (behavior) for manipulating an object.
- Ex: Queue class.

Queue Java Implementation

```
class QueueNode  {
      DataItem  item ;
      QueueNode link ;
   }

   class Queue {
      private QueueNode front ;
      private QueueNode back ;
      private int length ;
            // default no-arg constructor
      public boolean Empty() {
         return (length == 0) ;
      }
```


Java Queue (2)

```
public void Enqueue (DataItem itemvalue) {
    QueueNode temp = new QueueNode() ;
    temp.item = itemvalue ;
    temp.link = null ;
    if (back == null) {
        front = back = temp ;
    } else {
        back.link = temp ;
        back = temp ;
    }
    length ++ ;
 }
 ...
}
```


A Class

- The template for an object.
- The primary ADT.
- Follows OOD nomenclature.
- Provides encapsulation.
- Supports information hiding: Separates interface and implementation.

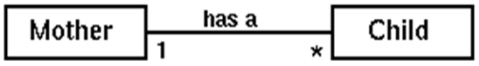
Other OO Terms

- Destructors: methods that free the dynamic storage used to store an object's state.
  Not needed in Java; required in C++
- Message passing: mechanism used to communicate with objects.
  Call to an objects method, with parameters.

Class Links

- Represent connections between objects.
- Link types represent relationships:
  - Non-hierarchical associations.
  - Part-of associations: *aggregation & composition.*
  - Is-a relationship: inheritance.
  - Use dependencies: transient connections.

Non-hierarchical Associations in a Class Diagram



Associations:

- Can have multiplicity indicators.
- Can have direction.

Association Implementation

```
class Child {
   ...
   private Mother mom;
   ...
}
```

- Variable *mom* references mother object.
- Relationship created by setting *mom variable.*

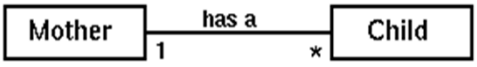
Association Implementation

```
class Mother {
  …
  private Child[] theKids
        = new Child[20];
  …
  }
```

Array *theKids* stores references to the children. Need a container class when there are more than one to reference.

Does the implementation match the class diagram?

```
class Mother {
  …
  private Child[] theKids
        = new Child[20];
  …
  }
```



Creating the Relationships

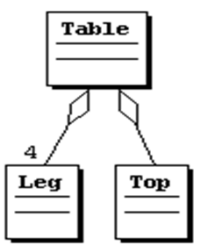
```
Mother theMom = new Mother();
Child sue = new Child();
Child tom = new Child();
theMom.addChild(tom);
theMom.addChild(sue);
tom.setMom(theMom);
sue.setMom(theMom);
```


Keep Instance Variables Private

- You can modify the representation without changing client code.
  - You might change the array representation of *theKids* to an implementation of the Java Collection interface.
  - Keeping *theKids* private allows this change without affecting an unknown number of clients.

Composition

- One software entity is built out of other entities.
- Composition: a stronger form of *aggregation*.
- Precise definitions of aggregation continue to be debated, but composition is clear.

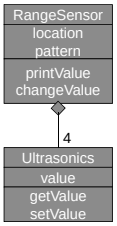


Implementing Composition

- Component object instantiated by containing class

```
class Table {
   private Legs[] legs = new Legs[4];
   private Top theTop = new Top();
   ...
  }
```


Composition: Objects Composed of Other Objects



- Range sensors have a radial or array pattern; they are placed in a location on a robot; users can print or change sensor values.
- Range sensors consist of 4 ultrasonic sensors.
- Each ultrasonic sensor has a value. Users can get or set values.

Composition Example: Ultrasonic Class

```
class Ultrasonic {
   private double value;
              // instance variable

   public Ultrasonic(float v) {
           value = v; }
   public double getValue() {
    return value;
   public void setValue(double v)
    {value = v;}
```

Ultrasonic has no reference to its containing RangeSensor object.

Containing Class: RangeSensor

Reference to contained objects.

```
public class rangeSensor {
   private Ultrasonic ultras[4] = new Ultrasonic[4];
   private double  height;
   private double offset;
   public rangeSensor (double iVal, double h, double o){
      for (i=0; i < MAX; i++)  ultras[i].set_value(ival);
      height=h; offset=o;
    }
   void printValues() {…}
   void changeUltraValue(int ultraNum, double val) {…}
};
```


Ultrasonic Client Program

```
someClientMethodBody() {
   RangeSensor rangeA =
                  new  RangeSensor(10.0,3.0,0.0);
   rangeA.print_values();
   rangeA.changeValue(0, 6.75);
   rangeA.changeValue(1, 11.9);
   rangeA.changeValue(2, 3.5);
   rangeA.changeValue(3, 5.5);
}
```


Polymorphism

- "Ability to hide different implementations behind a common interface" (Taylor, 1990).
- "Single interface, many implementations" (Entsminger, 1995).
- "Literally, the ability to have many forms" (Graham, 1991).

Polymorphism in OO Software

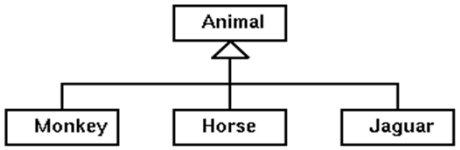
- Supported by dynamic binding (done at run-time, not at compile-time) and overloading.
  - Objects of a declared class can be replaced at run time with on objects of any of its subclass.
  - Dynamic Binding: the method that is invoked depends on the object that is bound at run time to a variable rather than the declared type of the variable.
- Example:

```
public int m(RangeSensor r){
   r.printValues();
}
```

At runtime any RangeSensor subclass object may be bound to r. A printValues method defined in the subclass is the one invoked.

Inheritance Supports Polymorphism

- Defines a class that is a specialization of another class.
- Generalization/Specialization in UML.



Inheritance Implements Specialization.

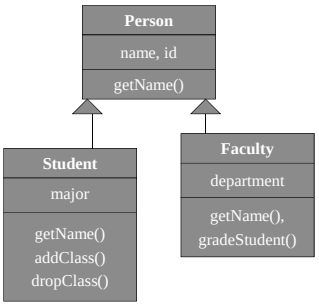
In Java, use the key word *extends:*

```
class Monkey extends Animal {
   ...
   }
```


Inheritance Terms

- Inheritance: a mechanism to implement a generalization-specialization relationship.
- Subclass: the specialized, extended or derived class.
- Superclass: the more general class in the relationship. The class that is extended.

Hierarchy Example



Generalizaion Class (Base Class or Superclass)

```
class Person  {
 private String name;
 private String id;
 public Person(String n, String ident) {
     name = new String(n);
     id = new String(ident);
  }
 public String getName() { return name; }
}
```


Specialization Class Derived Class / Subclass

```
class Student extends Person {
  private  String major;
  public  Student(String name, String id, String m){
     this.super(name, id); major = m;
  }
  public String getName() {
     return "Student name: " + this.super.getName()
           + ";  Major: " + major ;
    }
  public addClass (…) {…}
  public dropClass(…) {…}
}
```


Another Specialization Class

```
class Faculty extends Person {
  private  Department dept;
  public  Student(String name, String id, String d){
     this.super(name, id); dept = d;
  }
  public String getName() {
     return "Faculty name: " + this.super.getName()
           + ";  Department: " + dept ;
    }
  public gradeStudent (…) {…}
}
```


Creating & Using Specialization Class Objects in Client Code

```
{
Person t = new Person("Tom", "1234");
Student s = new Student("Sally", "4321", "CS");
Faculty j = new Faculty("Jim", "987", "CS");
t.getName();
s.getName();
s.addClass(…);
j.gradeStudent(…);
Person s2 = new Student("Audrey", "789", "Math");
s2.getName();  //                    ← Which getName() runs?

// s2.addClass();                    ← would not compile.  Why?
}
```


Inheritance Details

- A subclass cannot directly access private members of its base class.
- Creating a subclass does not affect its base class' s source code.
- To resolve polymorphism: At runtime, the class hierarchy is searched upward to find the first definition of a member function --- specialization wins!

Dynamic Binding (Yo-Yo) Example

```
public class A  {
  private int x = 1;
  public String doIt() {return hi() + "x = " + x;}
  public String hi() {return "Hi, I'm A. ";}
}

public class B extends A{
  private int x = 2;
  public String doIt() {
    return hi() + "x = " + x + " " + super.doIt();}
  public String hi() {return "Hi, I'm B. ";}
}

public class C{
  public String m(A a){ return a.doIt();}

  public static void main (String[] args){
   A testAObj = new B();
   C c = new C();
   System.out.println( c.m(testAObj) );
  }
```

What happens when it runs?

% java C

Hi, I'm B. x = 2 Hi, I'm B. x = 1

Use Inheritance Only for "is a" Relationships

- Use inheritance only when the subclass is a true specialization of the superclass.
  - There should be a generalization-specialization relationship.
  - Subclass objects should be clearly a specialization.
- A *student* is a specialized *person*; it has all of the properties of an animal and some additional ones.

Java Interfaces

- Java interfaces define operations & type signatures

```
interface PointI {
    public float x();   /* Show my x coordinate */
    public float y(); /* Show my y coordinate */
    public Point add(PointI p); /* Point addition */
    }
```

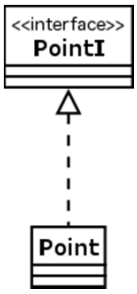

Implementing an Interface

```
class Point implements PointI {
  /* representation: x, y are Cartesian coordinate values.  */
   private float x, y;

  /* Construct myself as an origin point */
   public Point() {}

  /* Construct myself with given x & y coordinates */
   public Point(float xval, float yval) {
   x = xval;
   y = yval;   }
```

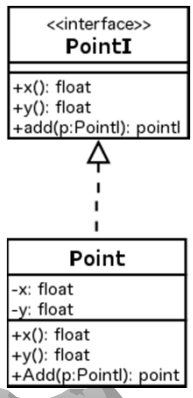

Class Point Methods

```
/* Show my coordinates */
    public float x() {return x;}
    public float y() {return y;}

  /* Point addition */
  public Point add(Point p) {
    float sumX = x + p.x();
    float sumY = this.y() + p.y();
    return new Point(sumX,sumY);   }
```


Class Point *Implements* Interface PointI (in UML)



Class Point & Interface PointI With More Details



Interfaces Can Inherit From Other Interfaces

- Pmult extends PointI with Point multiplication:

```
interface Pmult extends PointI {
    public Point mult(Pmult p);
        /* Point multiplication */
    }
```


Use Links / Use Dependencies

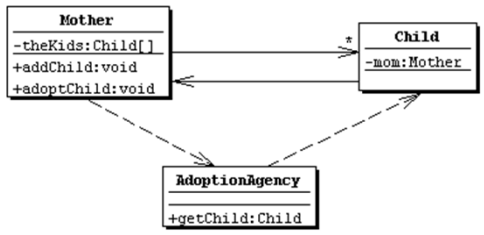
- Represent a transient connection:
  - Link not represented in class state.
  - Link active during method activation only.
- Object whose services will be used is passed in as a method parameter.

Ex. Use Dependency/Link: Adoption

- Class Mother has a method adoptChild, which adds an adopted Child to the family:
- Use an AdoptionAgency method:

```
public void adoptChild(AdoptionAgency
  theAgency) {
     addChild(theAgency.getChild());
   }
```


Use Link btween Mother, Child, & AdpotionAgency in UML



Class Link Summary

- Non-hierarchical association:
  - Relationship: no clear whole-part or specialization relationship.
  - Duration: part of the class state; It persists over the lifetime of the class object.
  - Implementation: define an instance variable that is a reference to the associated link. Use container class to support multiplicity.

Class Link Summary (2)

- Composition association:
  - Relationship: clear whole-part relationship.
  - Duration: part of the class state; It persists over the lifetime of the class object.
  - Implementation: define an instance variable that is a reference to the associated link. Use a container class to support multiplicity.

Class Link Summary (3)

- Inheritance link:
  - Relationship: generalization-specialization.
  - Duration: permanent part of the static definition of the subclass.
  - Implementation: use inheritance. In Java, the subclass *extends* the superclass.

Class Link Summary (4)

- Use links/ use dependencies:
  - Relationship: one class object uses the services of another class object.
  - Duration: transient; Exists only while the client or server methods are active.
  - Implementation: client class method has a formal parameter which is a reference to the server class. The client invokes a server method.

Example Design: Cave Game Now Adventure

- Player visits a cave looking for treasure.
- Move from room to room.
- Purely text based: predates GUI's.
  - Rooms described via textual description only.
  - Players must construct their own maps, on paper.

Cave Game Player Options

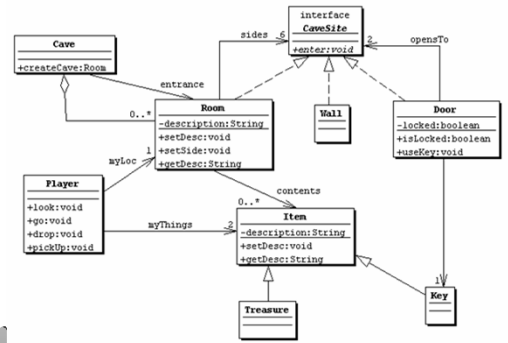
- As you go from room to room, you can:
  - Look at the room,
  - Go into an adjacent room or through an adjacent door,
  - Pick up an object in the room, or
  - Drop an object that you are carrying.

Cave Game Commands

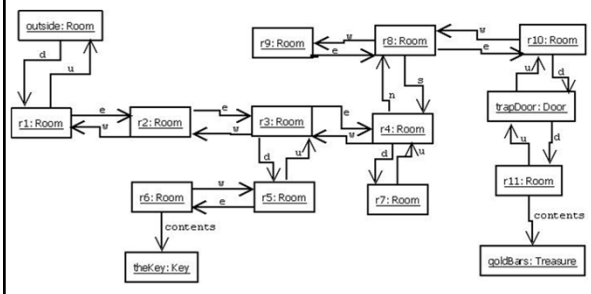
After reading the textual description a player types one of the following commands:

"n, s, e, w, u, d" for north, south, east, west, up, or down.

CaveGame Design-level Class Model



Instance Diagram of a Cave Layout



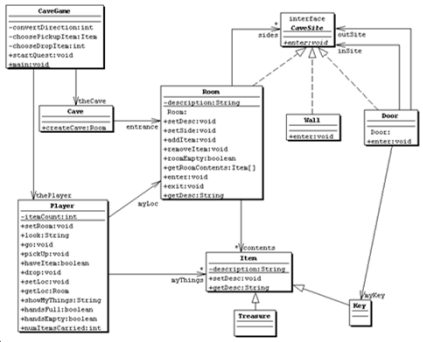
Implementing the Design

- The design evolves during implementation.
- Methods added to class Room:
  - void exit(int direction, Player p)
  - void addItem(Item i) and
    removeItem(Item i)
  - Item[] getRoomContents()

Implementing the Design (2)

- Methods added to class Player:
  - boolean haveItem(Item i)
  - Void setLoc(Room r)
  - Room getLoc()
  - String showMyThings()
  - boolean handsFull()
  - boolean handsEmpty()

Class Model Version 2



Class Player Code

```
class Player {
   private Room myLoc;
   private Item[] myThings = new Item[2];
   private int itemCount = 0;
   public void setRoom(Room r){
        myLoc = r;
    }
   public String look() {
      return myLoc.getDesc();
      }
```


Class Player Code (2)

```
public void go(int direction){
   myLoc.exit(direction,this);
}
public void pickUp(Item i){
   if (itemCount < 2) {
      myThings[itemCount] = i;
      itemCount++;
      myLoc.removeItem(i);
      }
   }
```


Class Room Code

```
class Room implements CaveSite {
   private String description;
   private CaveSite[] side
                        = new CaveSite[6];
   private Vector contents = new Vector();
   Room() {
      side[0] = new Wall();
      side[1] = new Wall();
      side[2] = new Wall();
      side[3] = new Wall();
      side[4] = new Wall();
      side[5] = new Wall();
      }
```


Class Room Code - Navigation

```
public void enter(Player p) {
  p.setLoc(this);
}
public void exit(int direction,
                 Player p){
  side[direction].enter(p);
}
```


Class Wall Code

```
class Wall implements CaveSite {
public void enter(Player p){
  System.out.println(
        "Ouch! That hurts.");
  }
}
```


Class Door Code

```
class Door implements CaveSite {
   private Key myKey;
   private CaveSite outSite;
   private CaveSite inSite;
   Door(CaveSite out, CaveSite in, Key k){
      outSite = out;
      inSite = in;
      myKey = k;
   }
```


Door Navigation Code

```
public void enter(Player p){
   if (p.haveItem(myKey)) {
      System.out.println(
         "Your key works! The door creaks open …");
      if (p.getLoc() == outSite) inSite.enter(p);
      else if (p.getLoc() == inSite) outSite.enter(p);
      }
      else {System.out.println(
         "You don't have the key for this door!");}
   }
}
```


Common Implementation Problems

- Distribution of class functionality.
- Not recognizing dynamic binding:
  - Objects know their class & will use the right method.
- Poor encapsulation.
- Separating the user interface with the guts of the system
  Model – View separation, or
  Model – View – Controller design pattern.
- Creating object configurations
  Factory or Abstract Factory design pattern.

Summary

- Review of OO concepts.
- UML class & instance models.
- Class links: non-hierarchical, whole-part, generalization-specialization, & use links.
- Design-to-code process.
- Note: UML models vary, depending on level of abstraction.

