Review – Unit Testing

Equivalence Classes

- With blackbox testing, we don't attempt to test all inputs
- Instead, we divide the input into equivalence classes so that
  - the classes cover the entire valid input space
  - the classes are disjoint
  - all values within an equivalence class behave similarly with respect to specification

Equivalence Classes

- Note: there is often more than one acceptable way of partitioning the valid input space
- Once we have determined our equivalence classes, we select at least one typical value and one boundary value for each class

Example 1

```
class Account {
   …
   /**
   * @pre amount >= 0
   * …
   */
   public void deposit(double amount ) { … }
}
```

- One equivalence class that satisfies the precondition: *amount >= 0*
  - Select at least one typical member of the class, *amount = 200*
  - Select values at boundaries, only one boundary, *amount = 0*
- Test cases are then: {amount = 200, amount = 0 }

Example 2

```
class Account {
   ...
   /**
   * @pre true
   * ...
   * @throws IllegalValueException when amount < 0
   */
   public void withdraw(double amount) { … }
}
```

- Two equivalence classes. What are they?

- What test cases would you specify?

Class Design II: Class Diagrams

You should be able to:

- interpret UML class diagrams to identify relationships between classes
- draw a UML class diagram to represent the design of a software system
- describe the basic design principles of low coupling and high cohesion
- design a software system (expressed in UML) from a given specification that adheres to basic design principles (low coupling and high cohesion)
- identify elements of a given design that violate the basic design principles of low coupling, high cohesion

Reading:

2nd Ed:

Chapter 9: 9.1, 9.2

Chapter 17: 17.2, 17.3, 17.4

3rd Ed:

Chapter 8: 8.1, 8.2

Chapter 12: 12.2, 12.3, 12.4

Some ideas in this section come from:

"Practical Object-Oriented Development with UML and Java"

R. Lee, W. Tepfenhart, Prentice Hall, 2002.

"Object-Oriented Software Development Using Java",

Xiaoping Jia, Addison Wesley, 2002

Heuristics for Finding Classes

- We usually start with the problem description and map each *relevant* word as follows:

nouns	→ classes or attributes
is/are	→ inheritance
has/have	→ aggregation or association
other verbs	→ methods
must	→ constraint
adjective	→ attribute, relation

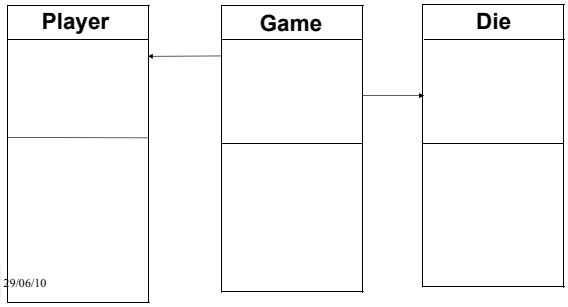
- This is called Abbott's heuristics for natural language analysis
- This is not always very accurate but it provides a good start

Simple Design Example

- Problem Description:
  We want to simulate a simple betting game. In this game, a player with money to gamble makes a bet and then rolls a single die. If a 1 is rolled, the player wins an amount equal to the bet, otherwise (s)he loses the bet.

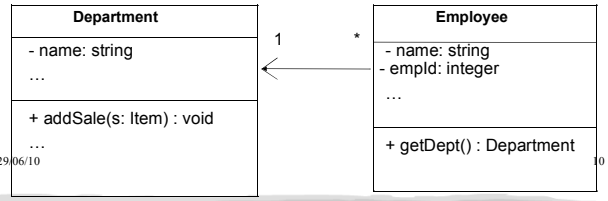
- Let us try to identify the classes and their behaviour…..
- Nouns:
  - **game**, **player**, **money**, **bet**, **die**, **amount**, **bet**
- Verbs :
  - **gamble, makes (a bet), rolls, wins, loses**

Putting it Together



Relationship 1: Association

- **Association**:
  - A *structural* relationship that describes a connection between objects: each object of one type contains reference(s) to objects of the other type.
- Example: Unidirectional association
  - employee stores a reference of a department



Relationship 1: Association

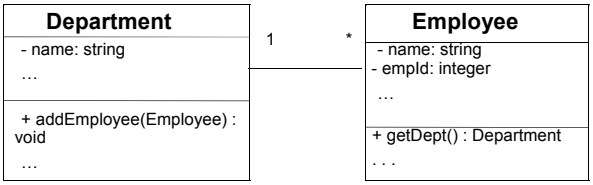
- "*Associations* are stronger than dependencies and typically indicate that one class retains a relationship to another class over an extended period of time. The lifelines of two objects linked by associations are probably not tied together (meaning one can be destroyed without necessarily destroying the other)."
  - UML 2.0 In a Nutshell

Relationship 1: Association

- Typically read as a "has a" relationship
- Associations have explicit notation to indicate navigability
- The arrows indicate whether you can navigate from one class to the other
- Relationship indicated by solid line, open arrow (no arrow if bidirectional...)
- Line may be adorned with a phrase or symbols to add information

Bidirectional Association

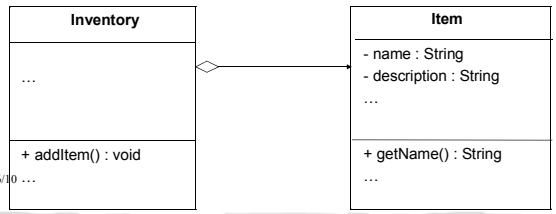
- Indicates that both classes reference each other
- Shown with a line without arrows
- Example:



- 1 and * in the previous examples are called ***multiplicities***
  - indicate the number of objects of this side that are associated by each object of the other side
  - * means any number

Relationship 2: Aggregation

- **Aggregation**:
  - A special form of association that specifies a whole-part relationship between the aggregate (the whole) and a component (the part)
- Example:

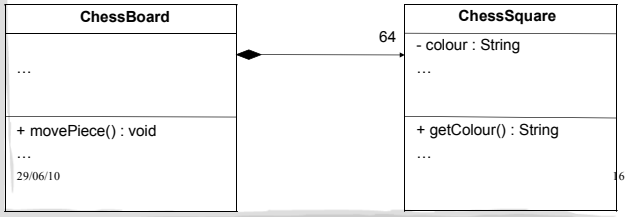


Relationship 2: Aggregation

- "*Aggregation* is a stronger version of association. Unlike association, aggregation typically implies ownership and may imply a relationship between lifelines."
  - UML 2.0 In a Nutshell
- Typically read as a "owns a" relationship
- Aggregation indicated by diamond shape next to owning class and solid line to owned class

Relationship 3: Composition

- **Composition**:
  - a *form of aggregation*, where the composite (whole) strongly owns the parts
  - when the whole is deleted (dies) the parts are also deleted (die)
  - A part is in exactly one whole (implicit multiplicity of 1)
- Example:

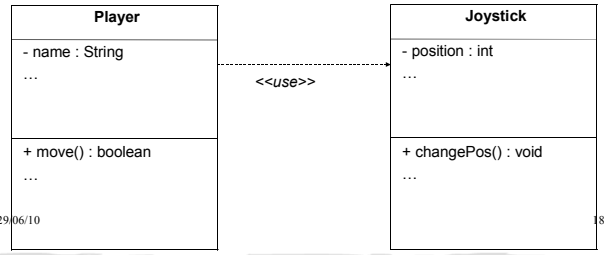


Relationship 3: Composition

- "*Composition* represents a very strong relationship between classes, to the point of containment. Composition is used to capture a whole-part relationship. The "part" piece of the relationship can be involved in only one composition relationship at any given time."
  - UML 2.0 In a Nutshell
- Typically read as a "is part of" relationship
- Indicated by filled diamond next to owner class and solid line to owned class

Relationship 4: Dependency

- **Dependency**:
  - A relationship describing that a change to the target element may require a change in the source element.
- Example:



Relationship 4: Dependency

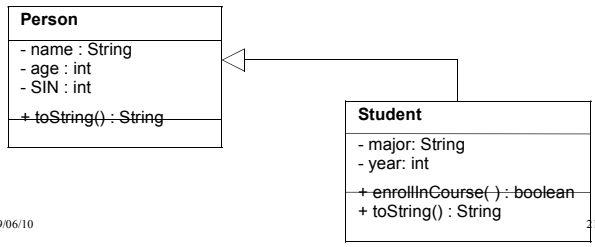
- "The weakest relationship between classes is a *dependency* relationship. Dependency between classes means that one class uses, or has knowledge of, another class. It is typically a transient relationship, meaning a dependent class briefly interacts with the target class but typically doesn't retain a relationship with it for any real length of time."
  - UML 2.0 In a Nutshell

Relationship 4: Dependency

- Typically read as a "uses a" relationship
- Indicated by a dashed line with an arrow pointing from the dependent class to the class that is used.

Relationship 5: Generalization

- **Generalization**:
  - An **inheritance** relationship, where a subclass is a specialized form of the superclass
- Example:



Relationship 5: Generalization

- "A *generalization* relationship conveys that the target of the relationship is a general, or less specific, version of the source class"
  - UML 2.0 In a Nutshell
- Typically read as a "is a" relationship
- Indicated by a solid line with a closed arrow, pointing from the specific class to the general class

Relationship 5: Generalization

- Note: UML allows for multiple inheritance, but Java does not
- If we want to simulate multiple inheritance, we can use *interfaces*

Example

```
class Car extends Vehicle
{
   . . .
   private Tire[] tires;
}
```

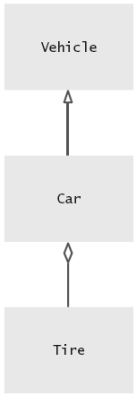
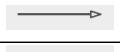
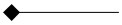


Figure 6
UML Notation for
Inheritance and Aggregation

UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open
Association		Solid	Open
Composition		Solid	Filled Diamond

Printing an Invoice – Requirements

- Task: print out an invoice
- Invoice: describes the charges for a set of products in certain quantities
- Omit complexities
  - *Dates, taxes, and invoice and customer numbers*
- Print invoice
  - *Billing address, all line items, amount due*
- Line item
  - *Description, unit price, quantity ordered, total price*
- For simplicity, do not provide a user interface
- Test program: adds line items to the invoice and then prints it

Sample Invoice

INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description	Price	Qty	Total
Toaster	29.95	3	89.85
Hair dryer	24.95	1	24.95
Car vacuum	19.99	2	39.98

AMOUNT DUE: $154.78

Printing an Invoice

- Discover classes
- Nouns are possible classes

```
Invoice
Address
LineItem
Product
Description
Price
Quantity
Total
Amount Due
```

Printing an Invoice

- Analyze classes

```
Invoice
Address
LineItem // Records the product and the quantity
Product
Description // Field of the Product class
Price // Field of the Product class
Quantity // Not an attribute of a Product
Total // Computed - not stored anywhere
Amount Due // Computed - not stored anywhere
```

- Classes after a process of elimination

```
Invoice
Address
LineItem
Product
```


Printing an Invoice – UML Diagrams

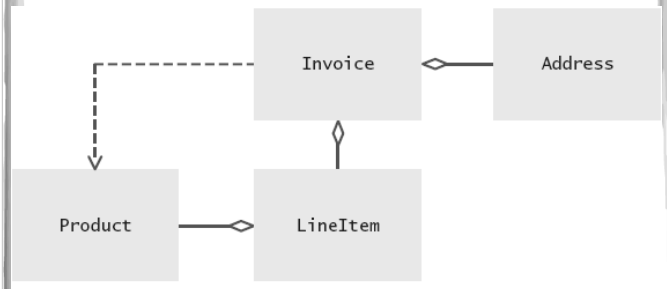


Figure 7 The Relationships Between the Invoice Classes

Implementation

- `Invoice` aggregates `Address` and `LineItem`
- Every invoice has one billing address
- An invoice can have many line items:

```
public class Invoice
{
   . . .
   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```


ch12/invoice/InvoicePrinter.java

```
01: /**
02:    This program demonstrates the invoice classes by printing
03:    a sample invoice.
04: */
05: public class InvoicePrinter
06: {
07:    public static void main(String[] args)
08:    {
09:       Address samsAddress
10:             = new Address("Sam's Small Appliances",
11:                "100 Main Street", "Anytown", "CA", "98765");
12:
13:       Invoice samsInvoice = new Invoice(samsAddress);
14:       samsInvoice.add(new Product("Toaster", 29.95), 3);
15:       samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16:       samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17:
18:       System.out.println(samsInvoice.format());
19:    }
20: }
21:
22:
23:
```

ch12/invoice/Invoice.java

```
01: import java.util.ArrayList;
02:
03: /**
04:    Describes an invoice for a set of purchased products.
05: */
06: public class Invoice
07: {
08:    /**
09:       Constructs an invoice.
10:       @param anAddress the billing address
11:    */
12:    public Invoice(Address anAddress)
13:    {
14:       items = new ArrayList<LineItem>();
15:       billingAddress = anAddress;
16:    }
17:
18:    /**
19:       Adds a charge for a product to this invoice.
20:       @param aProduct the product that the customer ordered
21:       @param quantity the quantity of the product
22:    */
```

Continued

ch12/invoice/Invoice.java (cont.)

```
23:    public void add(Product aProduct, int quantity)
24:    {
25:       LineItem anItem = new LineItem(aProduct, quantity);
26:       items.add(anItem);
27:    }
28:
29:    /**
30:       Formats the invoice.
31:       @return the formatted invoice
32:    */
33:    public String format()
34:    {
35:       String r =  "                        I N V O I C E\n\n"
36:             + billingAddress.format()
37:             + String.format("\n\n%-30s%8s%5s%8s\n",
38:                "Description", "Price", "Qty", "Total");
39:
40:       for (LineItem i : items)
41:       {
42:          r = r + i.format() + "\n";
43:       }
44:
```

Continued

ch12/invoice/Invoice.java (cont.)

```
45:       r = r + String.format("\nAMOUNT DUE: $%8.2f",
getAmountDue());
46:
47:       return r;
48:    }
49:
50:    /**
51:       Computes the total amount due.
52:       @return the amount due
53:    */
54:    public double getAmountDue()
55:    {
56:       double amountDue = 0;
57:       for (LineItem i : items)
58:       {
59:          amountDue = amountDue + i.getTotalPrice();
60:       }
61:       return amountDue;
62:    }
63:
64:    private Address billingAddress;
65:    private ArrayList<LineItem> items;
66: }
```


ch12/invoice/LineItem.java

```
01: /**
02:    Describes a quantity of an article to purchase.
03: */
04: public class LineItem
05: {
06:    /**
07:       Constructs an item from the product and quantity.
08:       @param aProduct the product
09:       @param aQuantity the item quantity
10:    */
11:    public LineItem(Product aProduct, int aQuantity)
12:    {
13:       theProduct = aProduct;
14:       quantity = aQuantity;
15:    }
16:
17:    /**
18:       Computes the total cost of this line item.
19:       @return the total price
20:    */
```

Continued

ch12/invoice/LineItem.java (cont.)

```
21:    public double getTotalPrice()
22:    {
23:       return theProduct.getPrice() * quantity;
24:    }
25:
26:    /**
27:       Formats this item.
28:       @return a formatted string of this item
29:    */
30:    public String format()
31:    {
32:       return String.format("%-30s%8.2f%5d%8.2f",
33:             theProduct.getDescription(), theProduct.getPrice(),
34:             quantity, getTotalPrice());
35:    }
36:
37:    private int quantity;
38:    private Product theProduct;
39: }
```


ch12/invoice/Product.java

```
01: /**
02:    Describes a product with a description and a price.
03: */
04: public class Product
05: {
06:    /**
07:       Constructs a product from a description and a price.
08:       @param aDescription the product description
09:       @param aPrice the product price
10:    */
11:    public Product(String aDescription, double aPrice)
12:    {
13:       description = aDescription;
14:       price = aPrice;
15:    }
16:
17:    /**
18:       Gets the product description.
19:       @return the description
20:    */
```

Continued 38

ch12/invoice/Product.java (cont.)

```
21:    public String getDescription()
22:    {
23:       return description;
24:    }
25:
26:    /**
27:       Gets the product price.
28:       @return the unit price
29:    */
30:    public double getPrice()
31:    {
32:       return price;
33:    }
34:
35:    private String description;
36:    private double price;
37: }
38:
```


ch12/invoice/Address.java

```
01: /**
02:    Describes a mailing address.
03: */
04: public class Address
05: {
06:    /**
07:       Constructs a mailing address.
08:       @param aName the recipient name
09:       @param aStreet the street
10:       @param aCity the city
11:       @param aState the two-letter state code
12:       @param aZip the ZIP postal code
13:    */
14:    public Address(String aName, String aStreet,
15:          String aCity, String aState, String aZip)
16:    {
17:       name = aName;
18:       street = aStreet;
19:       city = aCity;
20:       state = aState;
21:       zip = aZip;
22:    }
```

Continued 40

ch12/invoice/Address.java (cont.)

```
23:
24:    /**
25:       Formats the address.
26:       @return the address as a string with three lines
27:    */
28:    public String format()
29:    {
30:       return name + "\n" + street + "\n"
31:             + city + ", " + state + " " + zip;
32:    }
33:
34:    private String name;
35:    private String street;
36:    private String city;
37:    private String state;
38:    private String zip;
39: }
40:
```


Properties of a Good Design

- In most case there are many ways to break a problem into classes that would provide a solution (implementation) for it
- Some of these ways are better than the others in the sense that
  - it is easier to understand the software system
  - it is easier to revise and modify the code
  - it is easier to extend the code (add new functionality)

Basic Design Principles

- How would we know if our design is good?
  - it must satisfy some properties
- Two basic principles (properties) of a good design:
  1. design should have ***high cohesion***
  2. design should have ***low coupling***

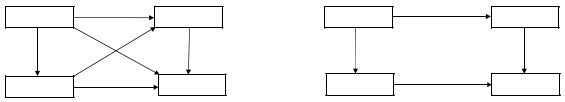
Cohesion

- A class should represent one concept; it should be *cohesive*
  - its public interface must be cohesive
- If a class is not cohesive (i.e. represents many concepts)
  - there's a greater chance that it might have to change in the future
  - changing one concept may inadvertently break an unrelated concept
- Violations of this rule are acceptable in some special cases:
  - *utility classes* that contain only static methods and constants (like the Math class)
  - classes that contain just a main method

Coupling

- A class A depends on another class B if it uses instances of the class B
- If many classes of a program depend on each other, we say that they have high coupling
- The coupling among classes is low, if there are only a few dependencies among them

high coupling *low coupling*

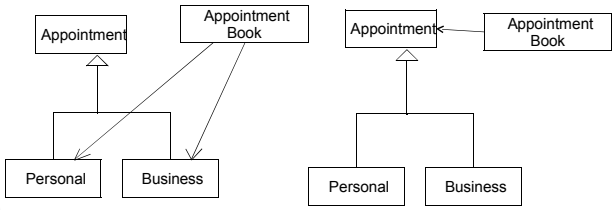


- **We want classes with high cohesion and low coupling**

Use Interfaces to Reduce Coupling

- Use most **general classes** and **interfaces** to reduce coupling
- For instance:

instead of this we should do this



Use Interfaces to Reduce Coupling

- A couple of lectures ago we reviewed interfaces and noted that they can reduce coupling
- Since we were doing single class design at that point, we didn't go in to that topic in detail
- Let's revisit those examples now

Interfaces

- When we define a class that **implements** an **interface**, we are committed to providing definitions for the abstract methods listed in the interface
- The interface itself contains no method definitions, it just tells you what you need to do
- So if you need to implement an interface (e.g. for an assignment, hint hint), look at the interface definition and it will tell you which methods your class will need

Interfaces vs. Classes

An interface type is similar to a class, but there are several important differences:

- *All methods in an interface type are abstract; they don't have an implementation*
- *All methods in an interface type are automatically public*
- *An interface type does not have instance fields*

Syntax 9.1 Defining an Interface

```
public interface InterfaceName
{
   // method signatures
}
```

Example:

```
public interface Measurable
{
   double getMeasure();
}
```

Purpose:

To define an interface and its method signatures. The methods are automatically public.

Syntax 9.2 Implementing an Interface

```
public class ClassName
implements InterfaceName, InterfaceName, ...
{
      // methods
      // instance variables
}
```

Example:

```
public class BankAccount implements Measurable
{
   // Other BankAccount methods
   public double getMeasure()
   {
      // Method implementation
   }
}
```


Advantages of Interfaces

- Polymorphism
  - Classes that implement an interface x will have objects of type x with the methods listed in x, but the method definitions will differ
- Simulating multiple inheritance
- Reducing coupling between classes

A simple interface

```
public interface Moveable {

   public void moveForward();
   public void moveBackward();
}
```


Implementing the interface

```
public class Car implements Moveable {
public void moveBackward() {
System.out.println("Going 95 in reverse");
}

public void moveForward() {
System.out.println("Going 95 on the freeway");

}

}
```


Implementing the interface

```
public class Bike implements Moveable {
public void moveBackward() {
System.out.println("Pedaling backwards!");
}

public void moveForward() {
System.out.println("Pedaling forwards!");

}

}
```


Interfaces and Polymorphism

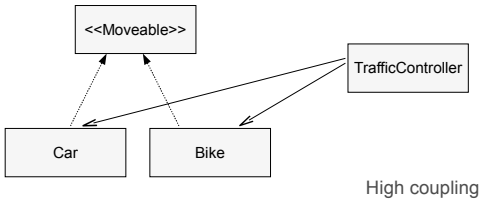
```
public class MoveTest {
public static void main(String[] args) {
Moveable[] moveArr = new Moveable[2];
moveArr[0] = new Bike();
moveArr[1] = new Car();
for (Moveable mover: moveArr)
{
mover.moveForward();
}
}
}
```

What gets printed?

Interfaces and Coupling

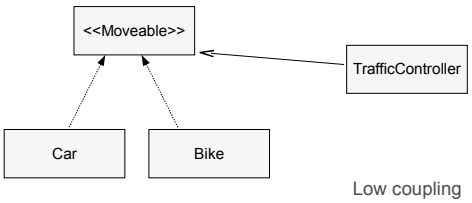
- Say we have a third class, TrafficController
- It has various methods for moving items (such as cars and bikes) around a city
- It would simplify things to have all of those items implement the Moveable interface
- That way TrafficController only needs to know about the Moveable interface and doesn't directly know about the classes that implement the interface
- All TrafficController cares about is that its methods can take Moveable objects and call moveForward() or moveBackward() on them

Interfaces and Coupling



High coupling

Interfaces and Coupling



Low coupling

Interfaces

- “The most important characteristic of interfaces is that they completely separate the definition of the functionality (the class's 'interface' in the wider sense of the word) from its implementation.”
  - Objects First with Java

A larger example: a Restaurant

- Suppose we want to simulate a restaurant.
- We will use the following classes:
  - Bill
  - CashPayment
  - CreditCardPayment
  - IndividualBill
  - Menu
  - MenuItem
  - Order
  - Payment
  - Table
  - TableBill
  - Waiter
- What are the relationships between these classes?

Summary

- UML diagrams help us to specify relationships between classes.
- Five important relationships:
  - Association
  - Aggregation
  - Composition
  - Dependency
  - Generalization
- The classes for a software system should be defined in such a way that they have
  - high cohesion
  - low coupling

In-Class Exercise I

- Given project description, use heuristics to identify **classes** and their **relationships**:
  - We want to create a graphical user interface (GUI) simulating an ATM machine. The GUI has a keypad. The ATM is linked with a bank. A bank has multiple customers. Each customer can have two accounts (savings and checking). The ATM can serve one customer at a time, and the customer can select one account at a time.

Learning goals review

You should be able to:

- interpret UML class diagrams to identify relationships between classes
- draw a UML class diagram to represent the design of a software system
- describe the basic design principles of low coupling and high cohesion
- design a software system (expressed in UML) from a given specification that adheres to basic design principles (low coupling and high cohesion)
- identify elements of a given design that violate the basic design principles of low coupling, high cohesion

Tea break!

Class Design III: Good Practices and Bad Practices

You should be able to:

- describe the open-closed principle, why it matters, and how it applies to object-oriented code.
- use overloading correctly and recognize inappropriate uses
- describe the Liskov Substitution Principle (LSP)
- explain whether or not a given design adheres to the LSP
- incorporate inheritance into the design of software systems so that the LSP is respected
- compare and contrast the use of inheritance and delegation
- use delegation and interfaces to realize multiple inheritance in design (e.g., to support the implementation of multiple types)
- identify elements of a given design that violate the basic design principles of low coupling and high cohesion

- Additional References
  - "Object-Oriented Software Development Using Java", Xiaoping Jia, Addison Wesley, 2002
  - "Core Java 2", Cay Hortsmann, Gary Cornell, Sun Microsystems Press, 2003

To Overload or Not to Overload

- **Overloading**: Same name is used for more than one method in the same class
- Mainly used for convenience
- Misuse may reduce program readability
- Should use overloading only in two situations:
  - There is a general description that fits all overloaded methods
  - All overloaded methods have the same functionality (some may provide default arguments)

Overloading Examples

Good:

```
public class StringBuffer
{
 public StringBuffer append(char c)
 {  …  }

 public StringBuffer append(int i)
 {  …  }

 public StringBuffer append(float f)
 {   …  }

 …
```

Bad:

```
public class Employee
{
  //sets  employee's name
  public void name(String s)
  {   …   }

  // returns employee's name
  public String name()
  {   …   }

  …
```

Do both fit under a common description?

Overloading Example – another problem

```
public class Employee {
   public void name(String s)
   {   …   }

   public void name(Object o)
   {   …   }
```

In Main:

```
   String stringAsString = new String("aString");
   Object stringAsObject = stringAsString;
   Employee e = new Employee();
   e.name(stringAsObject);    // what gets called?
   e.name(stringAsString);    // what gets called?
```

*** It is a very bad idea to overload methods in a way that all parameters of one method are subclasses of those in the other**

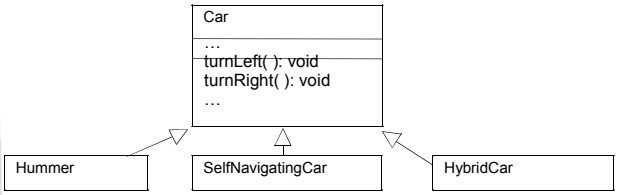
Open-Closed Principle

- Classes should be *open for extension* but *closed for modification*
  - Want to extend the behaviour of our system by adding subclasses
  - without having to modify the superclasses
- The principle suggests you should consider possible future subclasses when defining a class

is-a Style Inheritance : The **right** way

- Must be able to replace *any* instance of a superclass with an instance of *any* of its subclasses (Liskov Substitution Principle or LSP)
- Each class defines a ***type*** (the set of all instances of that class).
  A subclass following LSP defines a ***subtype*** (a subset of the superclass type).
- Example:
  - A Person class and a Student class

Is-A Style Inheritance: The right way…



- Client program should be able to create a number of cars, and then **control** each one without having to know exactly which car it is:

```
Car myToyota = new HybridCar();
myToyota.turnLeft();
```

Weakening the precondition

- A subclass method can *weaken the precondition (but it cannot strengthen it)* when overriding a method from its superclass.
  The subclass can accept a wider range of values as input.

```
abstract class Payment {
   /**
    * @pre amt >= 0
   */
   public void setPaymentAmount(int amt) {…}
}
class CreditCardPayment extends Payment {
   /**
    * @pre true
    */
   public void setPaymentAmount(int amt) {…}
}
class CashPayment extends Payment { … }
```


Weakening the precondition

- Why does it not make sense to strengthen the precondition?
- Suppose we set the precondition on the `setPaymentAmount` of `CreditCardPayment` to be:
  `@pre amt >= 25`
- Client should be able to do:

```
Payment p;
// substitute CashPayment for Payment
p = new CashPayment();
p.setPaymentAmount( 5 );

// substitute CreditCardPayment for Payment
p = new CreditCardPayment();
p.setPaymentAmount( 5 );    // oops!
```


Strengthening the postcondition

- A subclass's method can *strengthen the postcondition (but it cannot weaken it)*: a subclass's method can return a subset of the values returned by the method it overrides.

```
class Pump {
   /**
    * @post true
   */
   public double volumePumped() {…}
}

class PropanePump extends Pump {
   /**
    * @post value returned is integral and divisible by 5
    */
   public double volumePumped() {…}
}
```


Strengthening the postcondition

- Why does it not make sense to weaken the postcondition?
- Suppose the client writes code based on the postcondition of the superclass.
- That client code could break if we substitute a superclass object with an instance of one of its subclasses if the subclass' method has a weaker postcondition.
- Example:
  - client writes code assuming that a method returns a value that is positive
  - subclass overrides method to return *any* value (so postcondition is weakened)
  - client code is going to break if a negative value is returned.

Limitation Inheritance : The wrong way

- Subclass *restricts* rather than *extends* the behavior inherited from the superclass
- Violates *is-a* relationship
- Violates the Liskov Substitution Principle
- Usually used for implementation convenience (obviously in the wrong way)
- Example
  - `Square` defined as a subclass of `Rectangle` (next slide)
    - Methods `setHeight` and `setWidth` are not applicable to a square

Example: Rectangle Class

```
public class Rectangle {
   private double height;        // class invariant
height>0
   private double width;         // class invariant
width>0

   public Rectangle(){
      height = 1.0;  width = 1.0;
   }
   public Rectangle( double h, double w){
      height = h;  width = w;
   }
```


Example: Rectangle Class

```
public void setHeight(double h) {
      height = h;
   }
public void setWidth( double w ){
      width = w;
   }
public double area() {
      return height * width;
   }
}
What happens to the area when the height is doubled?
What happens to the width when the height is doubled?
Can we rely on this?
```


Example: Rectangle Class

```
/**
   * @pre width > 0
   * @post width=w and height is unchanged */
public void stretchToWidth(double w) {
          width = w;  }
/**
   * @pre width > 0
   * @post width=w and ratio of height:width is unchanged */
public void growToWidth( double w ){
         height = height*(w/width);
         width = w;  }
}

.....is there other reasonable behaviour?
```

Example : Square Class (the wrong way)

```
public class Square extends Rectangle {

   public Square() {
      super();
   }
   public Square( double s) {
      super(s, s);
   }
}
```

What is wrong with this?

Example : Square Class (the wrong way)

```
   // What about stretch to width???
 /**
    * @pre width > 0
    * @post width=w and height is unchanged
*/
 public void stretchToWidth(double w) { }
    //  It could just throw a NoSuchMethodException
    // (defined in java.lang), instead
```


Example : Square Class (the wrong way)

```
  // Override setHeight and setWidth
public void setHeight(double l) {
          ??????
 }
public void setWidth(double l) {
         ???????
}
public void setSide(double s){
      super.setHeight(s);
      super.setWidth(s);
  }
}
```


Example: Rectangle Class (revised)

```
public class Rectangle {
   double height;       // class invariant height>0
   double width;        // class invariant width>0

   public Rectangle(){
      height = 1.0;  width = 1.0;  }
   public Rectangle( double h, double w){
      height = h;  width = w; }
   public double area( ){
            return = height*width; }
```

There are no assumptions of what happens if the height or width changes!

Example: GrowableRectangle Class

```
public class GrowableRectangle extends Rectangle {
/** @invariant width > 0 and height > 0 */
public GrowableRectangle() {
     super(); }
public GrowableRectangle( double h, double w){
     super(h,w);   }
/**
  * @pre w > 0
  *@post width=w and ratio of height:width is unchanged */
public void growToWidth( double w ){
         height = height*(w/width);
         width = w;  }
}
```


Example : Square Class (a correct way)

```
public class Square extends GrowableRectangle {

   public Square() {
      super();
   }
   public Square( double s) {
      super(s, s);
   }
   public void setSide(double s){
       super.growToWidth(s);
   }
}
```


Delegation – another form of re-use

•A method delegates the execution of a task to another object of a different type

•Think of the "other object" as a servant used to carry out the task

•In OO languages delegation can be:

- class-based (or static)

servant is a component of the class

  – method-based (or dynamic)

method creates a servant and delegates the service

•Example next slide:

  – Square defined using class based delegation

Square Class (a right way)

```
public class Square {
   private Rectangle rectangle;

   public Square() {
      rectangle = new Rectangle();
   }

   public Square(double s) {
      rectangle = new Rectangle(s, s);
   }
```

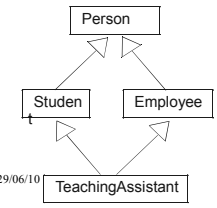
Square Class (a right way)

```
public void setSide(double s){
      rectangle.growToWidth(s);
   }

public double area() {
      return rectangle.area();
   }
}
```


Multiple Inheritance

- Multiple inheritance occurs when a class has more than one super-class.
- Multiple inheritance is supported by some programming languages (e.g., C++) but not others (e.g., Java).
- Multiple inheritance can lead to problems, for example, the classic *diamond* problem:



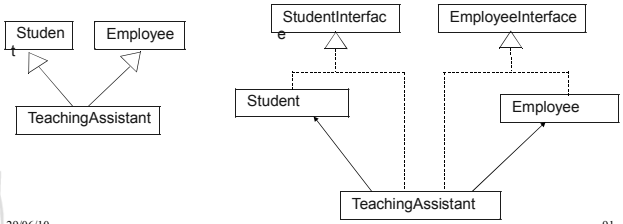
Suppose `Person` has a method `myMethod()` that's overridden in a different way in `Student` and `Employee` and that's not overridden in `TeachingAssistant`. Which version of the method should the following code call:

```
TeachingAssistant ta =
  new TeachingAssistant();

ta.myMethod();
```


Handling Multiple Inheritance in Java

- We can use delegation to implement multiple class inheritance if necessary
- For instance:
  instead of this: you can do this:



Multiple Inheritance Example

```
interface StudentInterface {
    public float getGPA();
}
interface EmployeeInterface {
    public float getSalary();
}
public class Student implements StudentInterface {
   protected float GPA;
   public float getGPA() {
      // code for GPA
   }
}
```

Multiple Inheritance Example (continued)

```
public class Employee implements EmployeeInterface {
   protected float salary;
   public float getSalary() {
      // code for Salary
   }
}

public class TeachingAssistant implements
                              StudentInterface,
EmployeeInterface {
   private  Student  student;
   private  Employee  employee;
```


Multiple Inheritance Example (continued)

```
  public TeachingAssistant() {
      student = new Student();
      employee = new Employee();
  }
  public float getGPA() {
      return student.getGPA();
  }
  public float getSalary() {
       return employee.getSalary();
  }
}
```


Name Collisions Among Interfaces

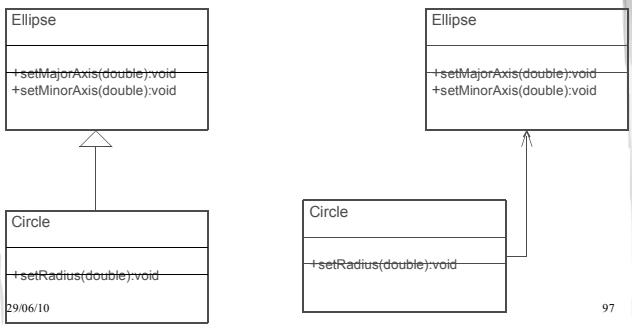
- A Java class may extend another class *and* implement one or more interfaces
- Inherited method from one interface may have same name as a method in another class or interface
- Name Collision procedure:
  - if methods have different signatures, they are considered overloaded
  - if they have same signature and return type, they are one method
  - if they have same signature, different return types, produce compilation error
  - if they have same signature and return type, but throw different exceptions, they are one method that throws the union of the exceptions thrown by each of them

General Design Guidelines for Inheritance

- Place common attributes and methods in the superclasses
- Use inheritance to model only *is-a* type relationships
- Use abstract classes and interfaces to design extensible families of objects with common properties
  - e.g., employees of different types
  - e.g., different types of objects to be drawn in a CAD application

Exercise

Which is the right way to define ellipse and circle?



Problem Description

- A TicketWizard Office needs a software system to track various events, their venues, and ticket orders for the events.
  - Each event has a name, description, date, time, a base ticket price and occurs at a single venue.
  - Each venue has a name, address, phone number.
  - Different events can have different seating plans. The seating plan consists of a number of sections and each section contains a number of seats. The price of a seat is determined by the base ticket price of the event and the section's price factor. A venue may host many different events, one event at a time, of course.

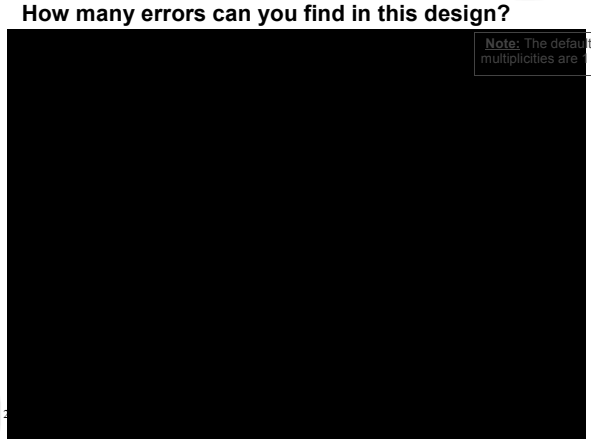
Problem Description (cont't)

- Customers can place orders, which are made up of one or more seats for one or more events. Ticket office employees can also place orders; they enjoy a 10% discount on any regular ticket price.
- Customers can pay for their orders by cash or charge them to a credit card. For each order, the system must track the type of payment.
- Finally, the system must track customer information so that customers can be notified if the event is changed or cancelled.

Some Issues to consider

- Does a venue need to know about events? If so, how?
- Does an event need to know about venue? If so, how?
- Do we need Seat objects?
- Do we need Ticket objects?
- Do we need Customer objects?
- Do we need Employee objects?
- What other objects do we need?

How many errors can you find in this design?



Key Concepts In This Lecture

- There are a lot of related concepts we covered today
  - When you design a superclass, think about whether it might be extended in the future (i.e., which methods should be protected instead of private, etc.). This is the **open-closed principle** in action.
  - In Java, a subclass is considered a subtype as is an implementation of an interface. To ensure an instance of a subclass (or a class that extends an interface) is substitutable for its superclass (or its interface) we need to follow the **Liskov Substitutability Principle (LSP)**. i.e., watch out that pre-conditions and post-conditions of overridden methods do the right thing.
  - If we want to reuse code but can't do it via a subclass because we'd violate the LSP, we can use **delegation** where we keep an object of the type from which we want the code and we call the object's methods to do the work we want done.
  - If we want one class to act like different types, use **interfaces** (and sometimes delegation too!)

Learning Goals Review

You should be able to:

- describe the open-closed principle, why it matters, and how it applies to object-oriented code.
- use overloading correctly and recognize inappropriate uses
- describe the Liskov Substitution Principle (LSP)
- explain whether or not a given design adheres to the LSP
- incorporate inheritance into the design of software systems so that the LSP is respected
- compare and contrast the use of inheritance and delegation
- use delegation and interfaces to realize multiple inheritance in design (e.g., to support the implementation of multiple types)
- identify elements of a given design that violate the basic design principles of low coupling and high cohesion