Events this week

All done.

The Undergrad Research Poster Competition was great

Events next week

**EADS Info Session**
- **Date:** Mon., Feb 1
- **Time:** 3:30 – 5 pm
- **Location:** CEME 1202

**RIM Info Session**
- **Date:** Thurs., Feb 4
- **Time:** 5:30 – 7 pm
- **Location:** DMP 110

**Administrivia**

- Lecture slides (day by day) are on the web:
- Assignment #1 was due yesterday
  - You can hand assignments in up to 24 hours late for a small penalty
- Assignment #2 is out
  - Due Thursday February 11, 10:00pm
Where are we?

- We are discussing class design – particularly with reference to inheritance.
  - Use overloading with care
  - Open-closed principle
    - Open for extension but closed for modification

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…

[Diagram showing inheritance hierarchy with classes Car, HybridCar, SelfNavigatingCar, and Hummer]

- 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.

```java
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:
  ```java
  @pre amt >= 25
  ```
- Client should be able to do:
  ```java
  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.

```java
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 behaviour 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

```java
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;
  }

  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)

```java
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

```java
public class GrowableRectangle extends Rectangle {
    public GrowableRectangle() {
        super(); }
    public GrowableRectangle( double h, double w){
        super(h,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; }
}
```
Example: Square Class (a correct way)

```java
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)

```java
public class Square {
    private Rectangle rectangle;
    private double side;

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

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

    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:

```java
TeachingAssistant ta = new TeachingAssistant();
ta.myMethod();
```