OO Design3

- Sample of software design issues
  - Class hierarchies
    - Use of polymorphism
    - Liskov substitutability principle
    - Avoiding potential problems
  - Protected variations
    - Mechanisms to protect code from later changes
    - Encapsulation, abstraction, polymorphism, indirection

Class Hierarchies

- Class hierarchies use is-a inheritance
  - How to ensure the correctness of extensions?
- Java: class B extends A { … }
  - Single inheritance (one superclass)
- Every member (i.e., method and field) of A is inherited by B
- B may override inherited methods

Polymorphism

- A variable may refer to (point to) objects of different classes
  - Calls through this variable may invoke different methods
  - Can use *to specialize code by type* or to choose between alternatives based on type
- Call: x.area() in Java
  - If x points to a Rectangle object, this call invokes method area() in Rectangle
  - if x points to a Square object, invokes an overriding method area()

Liskov Substitution Principle (LSP)

- A subclass object must be substitutable for an object of its superclass
  - Named after Barbara Liskov (MIT)
- E.g.: class A, subclass B
  - In class X declare method m(A a) { … }
  - If m() behaves correctly when given an A object, it should also behave correctly when given a B object (without m() knowing about the existence of B)

Liskov Substitution Principle (LSP)

- Polymorphism: if we add a new subclass of A, we don't need to recompile m()
  - Programming language mechanism
- LSP is stronger: if we add a new subclass of A, m() will still be correct

A Classic Example

```
class Rectangle {
   protected double h,w;
   protected Point top_left;
   public void setHeight(double x) { h = x; }
   public void setWidth(double x) { w = x; }
   public double getHeight() { return h; }
   public double getWidth() { return w; }
   public double area() { return h*w; } … }
```

- Suppose we have written a lot of code that uses Rectangle (e.g. graphics code)

Adding a Square

- The customer decides that we need a new class Square
  - *A square is a kind of rectangle, right?*
  - class Square extends Rectangle { … }
- With polymorphism: don't have to recompile existing code
  - **e.g.** void m(Rectangle x) { … } **does not have to be recompiled**

Problems

- Square doesn't really need both w and h
  - Wasted memory (relatively minor issue)
- Square inherits setHeight() and setWidth(), but this may lead to incorrect behavior

```
Square s = new Square();
…
Rectangle r = s;
…
r.setHeight(5);
r.setWidth(10);
???
```

One Solution

- Put guards in client code of Rectangle class
  - Before changing the size of a Rectangle, check if is it a Square

```
Rectangle r;
. . .
r.setHeight(5);
if (r instanceof Square) r.setWidth(5);
```

- For client code: increases coupling, reduces cohesion, makes it fragile -- Bad idea

Another Solution

- Override setHeight() and setWidth() in Square class

```
class Square extends Rectangle {
   public void setHeight(double x)
       { h = x; w = x; }
   public void setWidth(double x)
   { h = x; w = x; }
}
```

- BUT problems are possible when store a Square in a Rectangle reference

More Problems

```
void m(Rectangle r) {
        r.setHeight(5);
        r.setWidth(4);
        assert (r.area() == 20);
}
```

- When we only had Rectangle objects, this was valid code, but now it may break. Who's to blame?
- The programmer of m() is justified in writing this code
- There is something wrong with Square …

Back to LSP

- If m was written correctly with respect to a superclass, it should also be correct for the subclass
  - Square violates LSP!

Postcondition for Rectangle.setHeight()
  h == x and w == w_{old}
Postcondition for Square.setHeight()
  h == x and w == x
The subclass postcondition does not imply the superclass postcondition

LSP is about Behavior

- The *behavior* of Square.setHeight() does not conform to the *behavior* of method Rectangle.setHeight()
- LSP: Inheritance should guarantee conformance of behavior
  - Called behavioral subtyping
  - It may be OK to violate it, but the violation should be examined carefully and may depend on the clients of the hierarchy

Ensuring LSP

- One way: consider preconditions and postconditions
  - Whenever a subclass inherits or overrides an operation from a superclass
- Precondition for the superclass should imply the precondition for the subclass
- Postcondition for the subclass should imply the postcondition for the superclass
  - Contravariant conditions

A Simple Example

- **class Employee, with subclass Manager**
- **Operation** double calcBonus(int performeval)
  - **Calculates a bonus percentage**
  - **Defined in Employee; overridden in Manager**
- **The operation in the subclass**
  - **should not expect something more restrictive (pre-condition)**
  - **should not produce something less restrictive (post-condition)**

Preconditions and Postconditions

- **Employee.calcBonus:**
  - **Precondition: 0 ≤ eval ≤ 5**
  - **Postcondition 0% ≤ bonus ≤ 20%**
- 0 ≤ eval ≤ 5 **should imply the precondition for Manager.calcBonus**
  - **e.g., this cannot be** 1 ≤ eval ≤ 3, **can be** 0 ≤ eval ≤ 6,
- **Postcondition for Manager.calcBonus should imply** 0% ≤ bonus ≤ 20%
  - **e.g., this cannot be** 0% ≤ bonus ≤ 30% **but can be** 0% ≤ bonus ≤ 15%

Principle of Protected Variations

- Fundamental problem in design: current and future variations
- Protect the rest of the system from these variations
- Typical mechanisms for protection
  - Encapsulation
  - Abstraction
  - Polymorphism
  - Indirection

Object-Oriented Encapsulation

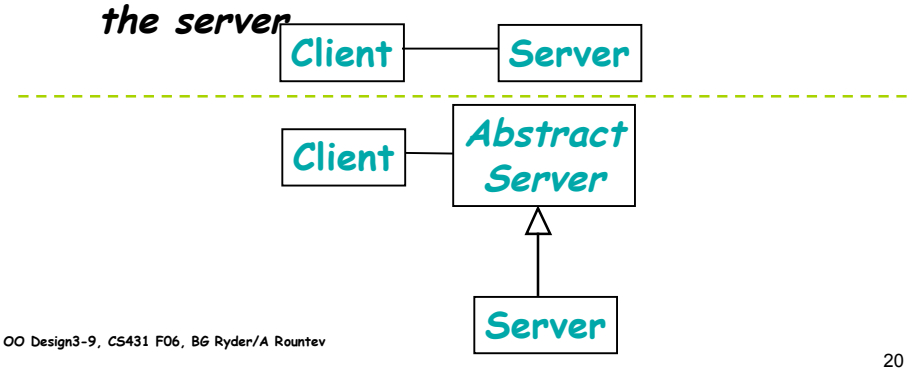
- Packaging of operations and attributes
- Attributes represent internal state that is not directly accessible
  - Hidden behind a "wall" of operations
- State is accessible and modifiable only via the operations
- Protection against
  - Changes in data representation
  - Changes in algorithm

Abstraction

- Common theme in software design
- Low-level abstractions
  - Procedural abstractions: subroutines
  - Data abstractions: classes
- Higher-level abstractions for object-oriented design
  - e.g., abstract classes
  - Protection against multiple classes needed for later extension

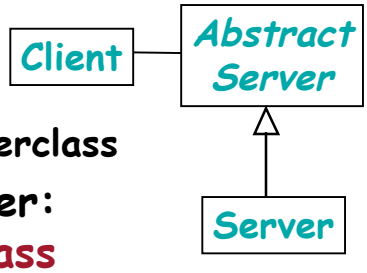
Using Abstractions

- Principle: write the client code against an abstraction of the server
  - In case of current or future *variations of the server*



Abstraction Through Subclasses

- Write the client against a superclass
  - Often an abstract superclass
- Variations in the server: introduce a new subclass
- If a new kind of server is added, client code does not need to be changed
  - Protection against new kinds of servers



Similar Example: POS System

- External tax calculator
  - Invoked across the network
    - e.g., through TPC, SOAP, Java RMI, etc.
- Want to be able to plug different tax calculators from different vendors
  - Adaptability is specified in the requirements
  - Cannot anticipate the interfaces we will encounter in the future

Indirection and Abstraction

- **Different interfaces: can create** adapter classes **running on the same machine as the POS system**
  - **Will deal with component-specific issues**
  - **Provide a level of indirection**
- **All adapter classes will implement a common interface that will be used by the POS system**
  - **Provides an abstraction of an adapter**
  - **Add a new tax system by adding a new adaptor**

Example