The Liskov Substitution Principle
(and Design By Contract)

Prof. Dr. Eric Dubuis
Berner Fachhochschule,
Engineering and Information Technology @ Biel

Course "UML and Design Patterns" of module "Software Engineering and Design",
version October 2008 (X)
Session's Goals

You'll learn:

- Review of polymorphism
- “IS-A” inheritance relation is not always good enough
- It is the client view that determines whether a given class hierarchy is a good one
- The ultimate criterion is for a good class hierarchy is:

  The Liskov Substitution Principle

Readings

Introduction to the Problem

Consider a program being able to move shapes. Suppose the following two shape classes:

```java
1 public class Rectangle {
2    public int x0 = 10;
3    public int y0 = 10;
4    public int width = 40;
5    public int height = 20;
6    // constructors not shown...
7 }
8 public class Line {
9    public int x0 = 10;
10   public int y0 = 20;
11   public int x1 = 20;
12   public int y1 = 5; // constructors not shown...
13 }
```
Introduction to the Problem (cont'd)

Consider a method `moveAllShapes(shapes)` of a class `Client` wanting to move `Line` and `Rectangle` shapes:

1. `Object[] shapes = ...;`  // Array is created and initialized with `Rectangle` and
2. `...`  // `Line` objects.
3. `// Now let's move the shapes.
4. `moveAllShapes(shapes);
5. `6. `// continued on next page`
Introduction to the Problem (cont'd)

private static void moveAllShapes(Object[] shapes) {
    int dx = ...; int dy = ...; // static method is used
    for (int i = 0; i < shapes.length; i++) {
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle) shapes[i];
            moveRectangle(r, dx, dy); // definition see below
        } else if (shapes[i] instanceof Line) {
            Line l = (Line) shapes[i];
            moveLine(l, dx, dy); // definition see below
        } else {
            System.err.println("Error: Cannot handle shape type: "+ shapes[i]);
        }
    }
}
Introduction to the Problem (cont'd)

The helper methods to move the rectangles and lines are:

```java
private static void moveRectangle(Rectangle r, int dx, int dy) {
    r.x0 = r.x0 + dx;
    r.y0 = r.y0 + dy;
}

private static void moveLine(Line l, int dx, int dy) {
    l.x0 = l.x0 + dx;
    l.y0 = l.y0 + dy;
    l.x1 = l.x1 + dx;
    l.x1 = l.x1 + dx;
}
```
Discussion

What if:

- we change the representation of a line:

  from: $(x_0 / y_0)$ to: $(x_1 / y_1)$

- we add new kind of shapes

Big disadvantage of this approach:

Changing the representation of a shape or adding other kind of shape type requires the modification of all client code!

Another issue is: You can add a new kind of shape, but forget the client to handle it!
A First Improvement

Who knows best how to move a shape? ==> The shapes themselves!

```java
public class Client {
    ...
    private static void moveLine(Line l, int dx, int dy) {...}
    private static void moveRectangle(Rectangle r, int dx, int dy) {...}
}
```

```java
public class Line {
    public void move(int dx, int dy) {
        this.x0 += dx;
        this.y0 += dy;
        this.x1 += dx;
        this.y1 += dy;
    }
}
```

```java
public class Rectangle {
    public void move(int dx, int dy) {
        this.x0 += dx;
        this.y0 += dy;
    }
}
```
A First Improvement (cont'd)

However, the case distinction in the Client moveAllShapes(shapes) method does not disappear:

```java
private static void moveAllShapes(Object[] shapes) {
    for (int i = 0; i < shapes.length; i++) {
        int dx = ...; int dy = ...;
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle) shapes[i];
            r.move(dx, dy);
        } else if (shapes[i] instanceof Line) {
            Line l = (Line) shapes[i];
            l.move(dx, dy);
        } else {
            System.err.println("Error: Cannot handle shape type: " + shapes[i]);
        }
    }
}
```

Still a distinction on types of shapes ...!
Introducing Inheritance to Solve the Shapes Problem

Let's introduce the abstract base class `Shape`:

```java
define public abstract class Shape {
    // could be an interface, too
    public abstract void move(int dx, int dy);
}
```

The class hierarchy for shapes now looks like:

Could be an interface, too ...
Adapting Line and Rectangle

```java
public class Rectangle {
    private int x0, y0, x1, y1;
    private void move(int dx, int dy) {
        this.x0 = this.x0 + dx;
        this.y0 = this.y0 + dy;
        this.x1 = this.x1 + dx;
        this.y1 = this.y1 + dy;
    }
}
```

```java
public class Line extends Shape {
    private void move(int dx, int dy) {
        this.x0 = this.x0 + dx;
        this.y0 = this.y0 + dy;
        this.x1 = this.x1 + dx;
        this.y1 = this.y1 + dy;
    }
}
```
Method moveAllShapes(shapes) of the Client

The method moveAllShapes(shapes) now looks like:

```java
1 private static void moveAllShapes(Shape[] shapes) {
2     for (int i = 0; i < shapes.length; i++) {
3         int dx = ...; int dy = ...;
4         Shape s = shapes[i];
5             /////
6             s.move(dx, dy); ______________________
7             /////
8     }}
9 }
```

The method now works for every `T` which is a subtype of Shape!

```java
==> the shape's move() operation is polymorphic!
```
The Liskov Substitution Principle (LSP)

The LSP simplified:

Subtypes must be substitutable for their base types.

Barbara Liskov original wordings:\(^1\)

“What is wanted here is something like the following substitution property:

If

for each object \(o_1\) of type \(S\) there is an object \(o_2\) of type \(T\) such that for all programs \(P\) defined in terms \(T\), the behavior of \(P\) is unchanged when \(o_1\) is substituted for \(o_2\)

then

\(S\) is a subtype of \(T\).”

Rectangles and Squares, a (Subtle) Violation of LSP

Assume a program that deals with some (new!) Rectangle objects:

```java
public class Rectangle {
    private int width, height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getArea() {
        return this.getHeight() * this.getWidth();
    }
}
```

Client  \( \rightarrow \)  Rectangle
0..*
A Client Using Class Rectangle

Consider a piece of client code:

```java
1 // in some class, say Client:
2   public static void someMethodUsingGetArea(Rectangle r) {
3       r.setWidth(5);
4       r.setHeight(4);
5       assertTrue(r.getArea() == 20);
6   }
7 }
```

So far, so good.
Adding Square as an Extension of Rectangle

Let's introduce a Square class:

```java
public class Square extends Rectangle {
    public Square(int size) {
        this.setWidth(size);
        this.setHeight(size);
    }
}
```

Clearly, the above class breaks the following test:

```java
@Test
public void testSquareness() {
    // Demonstrates the squareness problem!
    Square s = new Square(5);
    s.setHeight(10);
    assertTrue(s.getWidth() == 10);
}
```
Fixing the “Squareness” Problem

It's easy to fix the “squareness” problem:

```java
public class Square extends Rectangle {
    // as above (perhaps optimized...)

    public void setWidth(int w) {
        super.setWidth(w);
        super.setHeight(w); // "correction"
    }

    public void setHeight(int h) {
        super.setHeight(h);
        super.setWidth(h); // "correction"
    }
}
```
The Real Problem

... occurs when using method `someMethodUsingGetArea(Rectangle)` in conjunction with a `Square` object:

1. `Rectangle r = new Square(...);`
2. `Client.someMethodUsingGetArea(r); // Allowed since square s IS-A rectangle!`

However, method `someMethodUsingGetArea(Rectangle)` fails!
Discussion

- Where lies the problem?
  - in Rectangle?
  - in `someMethodUsingGetArea()`?
  - in Square?

- Validity is not intrinsic!

- “IS-A” relationship breaks in `someMethodUsingGetArea(Rectangle)` when providing a `Square` object as argument.
**Post-Conditions for `setWidth()`**

- **of `Rectangle.setWidth(int newW)`:**
  - R1: \( \text{width} = \text{newW} \)
  - R2: \( \text{height} = \text{height}_{\text{old}} \)

- **of `Square.setWidth(int newW)`:**
  - S1 (= R1): \( \text{width} = \text{newW} \)
  - S2: \( \text{height} = \text{newW} \)

Is the post-condition \( \text{PC}_s \) stronger or equal than \( \text{PC}_r \)? That is:

\[
\{ \text{R1, R2} \} \subseteq \{ \text{R1, S2} \} \? 
\]

\( \text{PC}_r = \{ \text{R1, R2} \} \)
\( \text{PC}_s = \{ \text{S1, S2} \} = \{ \text{R1, S2} \} \)

X is equal or stronger than Y if X does enforce all constraints of Y (and may add additional constraints). [Martin]
Design by Contract¹

When overwriting (redeclaring) methods in extensions, the rules for pre- and post-conditions can be stated by B. Meyer as follows:

A routine redeclaration [in a derivative] may only replace the original pre-condition by one equal or weaker, and the original post-condition by one equal or stronger.²

Specifying Contracts in Unit Tests

Unit tests can take the role of specifying contracts. Users of a class may review the unit tests in order to know what to reasonably assume about the class they are using.

---
