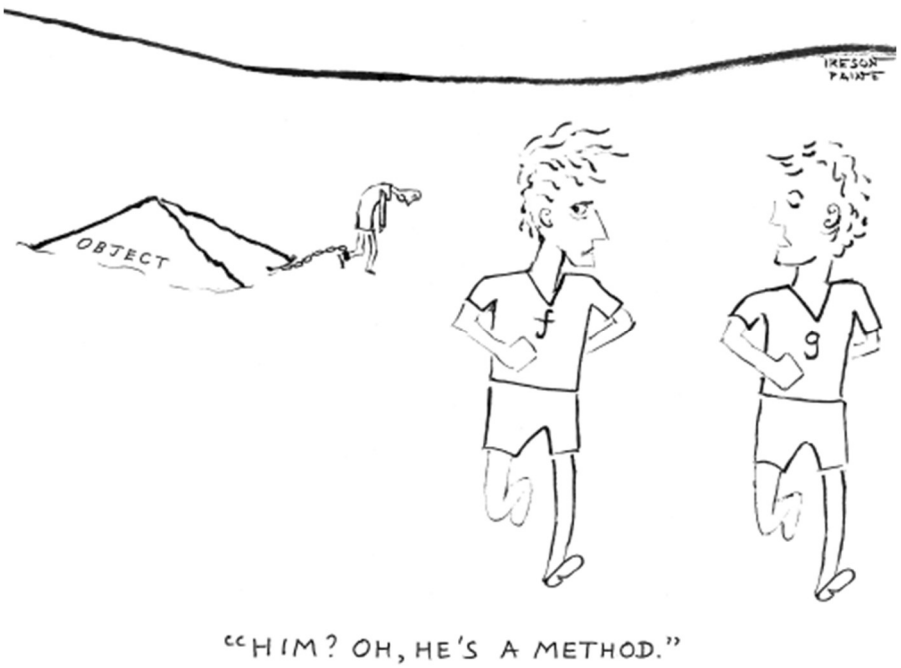

OBJECT-ORIENTED PROGRAMMING

Using Polymorphism Well

Lecture #5

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2024/2025

Polymorphism

- Polymorphism means **having many forms**
- How is this related to object-oriented programming?
  - Polymorphic reference
- Since a polymorphic reference can refer to different types of objects over time the specific method it invokes can also change from one invocation to the next

Forms of polymorphism

- **Ad-hoc polymorphism**
  - Polymorphic functions which can be applied to arguments of different types, but which behave differently depending on the type of the argument to which they are applied
  - Function **overloading** (e.g. overloading of constructors and methods)
- **Parametric polymorphism**
  - Allows a function or a data type to be written generically, so that it can handle values identically without depending on their type
  - **Generic** programming
- **Subtype polymorphism** (inclusion polymorphism)
  - Allows a function to be written to take an object of a certain type *T*, but also work correctly if passed an object that belongs to a type *S* that is a subtype of *T*
  - **Subclassing**, **inheritance**

Overloading

- Providing more methods with the same name but different parameter types
  - **Statically** select most specific matching method of a type

```
public class Utility {
  public String whatAmI(Object o) { return "object"; }
  public String whatAmI(String s) { return "string"; }
  public boolean isString(Object o) { return false; }
}

public class Overloaded extends Utility {
  public boolean isString(String s) { return true; }
}
```

Overloads but NOT overrides inherited method **boolean** `isString(Object o)` from class `Utility`

Overloading

```
public static void main(String[] args) {
  Overloaded overloaded = new Overloaded();

  System.out.println(overloaded.whatAmI(overloaded));
  System.out.println(overloaded.whatAmI(new String("test")));

  Object o = new String("test");

  System.out.println(overloaded.whatAmI(o));
  System.out.println(overloaded.isString(new String("test")));
  System.out.println(overloaded.isString(o));

  Utility utility = overloaded;

  System.out.println(utility.isString(new String("test")));
}
```

Overloading

```
public static void main(String[] args) {
  Overloaded overloaded = new Overloaded();

  System.out.println(overloaded.whatAmI(overloaded));
  System.out.println(overloaded.whatAmI(new String("test")));

  Object o = new String("test");

  System.out.println(overloaded.whatAmI(o));
  System.out.println(overloaded.isString(new String("test")));
  System.out.println(overloaded.isString(o));

  Utility utility = overloaded;

  System.out.println(utility.isString(new String("test")));
}
```

object

string

object

true

false

false

Overkill

- Overloading (and overriding together) can be overwhelming
- Avoid overloading whenever possible
  - Names are cheap and plentiful
- One place you cannot easily avoid it – constructors
  - They all have to have the same name

Downcasting

- Downcasting (type refinement) is the act of casting a reference of a base class to one of its derived class
  - When a variable of the base class has a value of the derived class, downcasting is possible

```
Article a = new DiscountArticle();
DiscountArticle da = (DiscountArticle) a;
```

This is possible since object referred by a is currently holding value of `DiscountArticle` class

- The danger of downcasting is that **it is not compile-time check**, but rather **it is run-time check**
  - Downcasting changes apparent type, so **during run-time it must be checked that actual type is a subtype of apparent type**

```
public String objectToString(Object o) {
  return (String) o;
}

String s = objectToString("a test string");
Object wrong = new Object();
s = objectToString(wrong);
```

This will only work when the `o` currently holding value is `String`. In compile-time it is OK, because `String` is subtype of `Object`

This will work since we passed in `String`, so `o` has value of `String`

This will fail since we passed in `Object` which does not have value of `String`

Downcasting example

- Class `java.util.Vector` has reusable methods

```
public void addElement(Object obj)
public Object elementAt(int i)
```

```
public class StringSet {
  private Vector elements;
  public void insert(String s) {
    elements.addElement(s);
  }
  public String choose() {
    return elements.elementAt(0);
    return (String) elements.elementAt(0);
  }
}
```

Downcasting is required here

A popular example of a badly considered object-oriented design is containers of universal supertypes, like the Java containers before Java generics were introduced, which requires downcasting of the contained objects so that they can be used again.

Avoid downcasting, since according to the LSP, an object-oriented design that requires it is flawed.

Some languages, such as OCaml, disallow downcasting altogether. In many cases downcasting can be replaced by using parametric polymorphism (e.g. generics in C++, C#, Java)

The Liskov Substitution Principle

- **Functions that use references to base (super) classes must be able to use objects of derived (sub) classes without knowing it**

LSP example

- Consider the following `Rectangle` class

```
public class Rectangle {
  private double width;
  private double height;
  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }
  public double getWidth() { return this.width; }
  public void setWidth(double width) { this.width = width; }
  public double getHeight() { return this.height; }
  public void setHeight(double height) { this.height = height; }
  public double area() { return this.width * this.height; }
}
```

LSP example

- Now, think about a `Square` class. Clearly, a square is a rectangle, so the `Square` class should be derived from the `Rectangle` class
- Observations
  - A square does not need both a width and a height as attributes, but it will inherit them from `Rectangle` anyway. So, each `Square` object wastes a little memory, but this is not a major concern
  - The inherited `setWidth()` and `setHeight()` methods are not really appropriate for a `Square`, since the width and height of a square are identical. So, we'll need to override `setWidth()` and `setHeight()`. (Having to override these very basic methods is a clue that this might not be an appropriate use of inheritance!)

LSP example

- Here's the `Square` class

```
public class Square extends Rectangle {
  public Square(double size) { super(size, size); }
  @Override
  public void setWidth(double width) {
    super.setWidth(width);
    super.setHeight(width);
  }
  @Override
  public void setHeight(double height) {
    super.setHeight(height);
    super.setWidth(height);
  }
}
```

LSP example

- Everything looks good but check this out!

```
public class Main {
  public static void main(String[] args) {
    Rectangle rectangle = new Rectangle(1.0, 1.0);
    testLSP(rectangle);
    Square square = new Square(1.0);
    testLSP(square);
  }
  public static void testLSP(Rectangle rectangle) {
    rectangle.setWidth(4.0);
    rectangle.setHeight(5.0);
    System.out.println("Width is 4.0 and Height is 5.0, so Area is " +
      rectangle.area());
    if (rectangle.area() == 20.0)
      System.out.println("Looking good!\n");
    else
      System.out.println("Huh? What kind of rectangle is this?\n");
  }
}
```

Now call method `testLSP()`. According to the LSP, it should work for either rectangles or squares. Does it?

Remember: If *B* is a subtype of *A*, everywhere the code expects an *A*, a *B* can be used instead

LSP example

- Test program output

```
Width is 4.0 and Height is 5.0, so Area is 20.0
Looking good!

Width is 4.0 and Height is 5.0, so Area is 25.0
Huh? What kind of rectangle is this?
```

- Looks like we violated the LSP!

LSP example

- What is the problem here? The programmer of the `testLSP()` method made the reasonable assumption that changing the width of a `Rectangle` leaves its height unchanged
- Passing a `Square` object to such a method results in problems, exposing a violation of the LSP
- The `Square` and `Rectangle` classes look self consistent and valid. Yet a programmer, making reasonable assumptions about the base class, can write a method that causes the design model to break down
- Solutions can not be viewed in isolation, they must also be viewed in terms of reasonable assumptions that might be made by users of the design

LSP example

- A mathematical square might be a rectangle, but a `Square` object is not a `Rectangle` object, because the behavior of a `Square` object is not consistent with the behavior of a `Rectangle` object!
- **Behaviorally, a `Square` is not a `Rectangle`! A `Square` object is not polymorphic with a `Rectangle` object**

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) makes it clear that the IS-A relationship is all about **behavior**
- In order for the LSP to hold (and with it the Open-Closed Principle) all subclasses must conform to the behavior that clients expect of the base classes they use
  - For example, usage of downcasting signal there is probably a problem (Why do we need explicit subclasses?)
- A subtype must have no more constraints than its base type, since the subtype must be usable anywhere the base type is usable
- If the subtype has more constraints than the base type, there would be uses that would be valid for the base type, but that would violate one of the extra constraints of the subtype and thus violate the LSP!
- **The guarantee of the LSP is that a subclass can always be used wherever its base class is used!**

Substitution principle guarantee

- **How do we know if saying *B* is a subtype of *A* is safe?**
- If *B* is a subtype of *A*, everywhere the code expects *A*, a *B* can be used instead
- For function *f*(*A*), if *f* satisfies its specification when passed an object whose actual type is type *A*, *f* also satisfies its specification when passed an object whose type is *B*

Subtype condition 1: Signature rule

class A {
  $R_A f(P_A p)$;
}
class B **inherit** A {
  override $R_B f(P_B p)$;
}

- R_B must be a subtype of R_A ($R_B \leq R_A$)
  - Covariants for results
- P_B must be a supertype of P_A ($P_A \leq P_B$)
  - Contravariants for parameters

Subtype condition 1: Signature rule

- OO languages are usually stricter than this, they does not allow any variations in types (novariant)
  - **Overriding method must have same return and parameter types**
- For example, in Java
  - Versions < 1.5 use **overloading** in this case, so the **methods are not actually overridden**
  - Versions ≥ 1.5 of Java use `@Override` clause to specify the variant method overrides the method from the superclass and in addition overriding method can throw fewer exceptions

Subtype condition 2: Methods rule

- Precondition of the subtype method must be **weaker** than the precondition of the supertype method ($pre_A \Rightarrow pre_B$)
  - The rectangle must have its width and height set to calculate its area
  - The square must have its width and height set and they must be the same to calculate its area
    - This means square has **stronger precondition** for calculating area than rectangle. **FAIL**
- Post condition of the subtype method must be **stronger** than the post condition of the supertype method ($post_B \Rightarrow post_A$)
  - The article gets actual price as its normal sales price
  - The discount article gets actual price as its normal sales price updated by the percentage discount
    - This means discount article has **stronger post condition** for getting actual price. **OK**

Subtype condition 3: Properties

- Subtypes must preserve all properties described in the overview specification of the supertype
- Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S where $S \leq T$
- $(pre_{super} \wedge post_{sub}) \Rightarrow post_{super}$
- Example: Let $B \leq A$
  - If A is immutable data type, can B be mutable?
  - If A is mutable data type, can B be immutable?

Eiffel's rule

- Another approach to type substitution
- Bertrand Meyer in object-oriented language Eiffel prefers **covariant** typing
  - The subtype replacement method parameter types must be **subtypes** of the types of the parameters of the supertype method
- The subtype method preconditions must be weaker than the supertype method precondition and the subtype post conditions must be stronger than the supertype post conditions
- Note that unlike the corresponding Liskov substitution principle, $(pre_{super} \wedge post_{sub}) \Rightarrow post_{super}$, there is no need for pre_{super} in the covariant rule since $post_{sub} \Rightarrow post_{super}$