Software Engineering – Architectures, Designs, and Code

L00 on Coding
Focus on the strategy pattern in Java

Giancarlo Succi
Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna
g.succi@unibo.it

Location of the material:
https://github.com/GiancarloSucci/UniBo.SE.A2023

Structure of the Part on Coding

- Part 0:
  - Revision of the strategy pattern in Java (and, a very little bit, on C++)
- Part 1:
  - Introduction on Python as a self-proclaimed multiparadigm language
- Part 2:
  - Patterns in Python with an eye on ChatGPT and around

Premise

- Here we will reflect on the concepts we saw on architecture and design looking at the code
- We will focus on a concrete example, which could be used also in the overall course
- We will see many examples of code
- Also very detailed, step-by-step examples
- Our focus will be on the software engineering side, though
- That is in understanding why and how people achieved what they achieved
  - not on hacking solutions

Motivation of this section

- Here we reflect substantially on how the strategy pattern is achieve in Java
- This is at the core of polymorphism via overriding and late binding
- Indeed, this is also because we do not use pointers to functions

Let's look at the reference

```
testScope
  | t |
  t := 42.
  self testBlock: [Transcript show: printString]

testBlock: aBlock
  | t |
  t := nil.
  aBlock value
```

- This chunk of SmallTalk code goes well beyond simple pointers to function, and it is the ultimate goal of the strategy pattern
- Evaluating testScope returns 42 ... why?

Classes in inner scopes

- In Java and in C++ it's possible to place a class definition within another class definition
- In C++ this is just related to visibility and naming
- In Java the matter is deeper; we can distinguish between
  - nested classes, which are just put inside other classes for naming/packaging concerns
  - inner classes, whose objects depend on the existance of object of the nesting class
  - local classes, that are declated locally within a block of Java code, rather than as a member of a class

Nested classes

- It is possible to place a class definition within another class definition
  - This is called a nested class
- Nested classes allow to hide the existence of a class from the outside world
- Moreover, they allow to group classes that logically belong together and to control the visibility of one within the other
- The name of a nested class is local to its enclosing class
- The nested class is in the scope of its enclosing class
- Declarations in a nested class can use only static members from the enclosing class

Inner classes

- Java takes this approach further, defining so called inner classes
- In Java simply declaring a class inside the body of another one, i.e., without any further qualification, results in an inner class
- In Java to declare a nested class we need to add the keyword `static` in front of its declaration

Examples of nested and inner classes in Java

```
public class X {
  int instanceVar; // Belongs to an instance of class X
  static int staticVar; // Belongs to class X
  private class Inner {
    int y;
    Inner() { y = instanceVar + staticVar; }
  }
  private static class Nested {
    int x;
    Nested() {
      x = staticVar;
      // x += instanceVar; Error!
    }
}
```

Examples of nested classes in C++

```
class X {
  public:
  int instanceVar; // Belongs to an instance of class X
  static int staticVar; // Belongs to class X

  class Nested {
    int x;
    Nested() : x(staticVar) {
    // instanceVar++; Error!
    }
  };
};
```

Comments on inner classes

- Each instance has an enclosing instance, and can use its members
- Inner classes cannot have static members
- Inner classes cannot have the same name as the enclosing class
- If an inner class is a local class (see later), it has access to the members of the enclosing class, final local variables and parameters
- Inner classes can also be anonymous classes (see later), i.e., unnamed classes defined within an expression
  - They are similar to local classes but can have only one instance

Local classes

- Both Java and C++ allows definition of a class inside the body of a function
- This is called a local class
- Local classes are very useful when we need code and data structures to perform specific tasks inside a function, but we don't want to make them available to the outside world
- Therefore local classes are a mean to enforce stricter encapsulation and data hiding at function scope level
- The name of a local class is local to its enclosing scope
- The local class is in the scope of the enclosing scope, and has the same access to names outside the function as does the enclosing function
- In Java, local classes are indeed inner classes, and not simply nested classes

Interfaces

- An interface declaration introduces a new reference type whose members are classes, interfaces, constants, and methods
- This type has no instance variables, and typically declares one or more abstract methods; otherwise unrelated classes can implement the interface by providing implementations for its abstract methods
- Interfaces cannot be instantiated – they can only be implemented by classes or extended by other interfaces
- Interfaces contain only constants, method signatures, default methods, static methods, and nested types
- Programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to Object

Interface Declaration

- In its most common form, an interface is a group of related methods with empty bodies.
- Interfaces have the same access specifiers as classes
- Interfaces support single and multiple inheritance, since the absence of instance data eliminates the ambiguities of, say, C++

```
public interface Bicycle {
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

Source of these and the following slides:
https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Interface Declaration

- Default methods are defined with `default`, and static methods with the `static`.
- All abstract, default, and static methods are implicitly public
- An interface can contain constant declarations. All constants are implicitly public, static, and final

```
public interface GroupedInterface extends Interface1,
    Interface2, Interface3 {
    // constant declarations
    // base of natural logarithms
    double E = 2.718282;
    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

Interface Implementation

```
class BianchiBicycle implements Bicycle {
  int cadence = 0, speed = 0, gear = 1;
// The compiler will now require that methods
// changeCadence, changeGear, speedUp, and
// applyBrakes all be implemented. The compilation will
// fail if those methods are missing from this class.
  void changeCadence(int newValue) {
    cadence = newValue;
  }
//...
  void printStates() {
    System.out.println("cadence:" +
      cadence + " speed:" + speed + " gear:" + gear);
    }
}
```

Methods in Interfaces

- Default methods and abstract methods in interfaces are inherited like instance methods.
- However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict.
- These rules are driven by the following two principles:
  - Instance methods are preferred over interface default methods
  - Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor

Instance vs. default interface methods (1/2)

Instance methods are preferred over interface default methods

```
public class Horse {
  public String identifyMyself() { return "I am a horse.";}
}
public interface Flyer {
  default public String identifyMyself() {
        return "I am able to fly."; }
}
public interface Mythical {
    default public String identifyMyself() {
        return "I am a mythical creature."; }
}
```

Instance vs. default interface methods (2/2)

```
public class Pegasus extends Horse implements Flyer,
    Mythical {
     public static void main(String... args) {
         Pegasus myApp = new Pegasus();
         System.out.println(myApp.identifyMyself());
     }

}
```

The method Pegasus.identifyMyself() returns the string “I am a horse”

Overriding default methods (1/8)

Methods that are already overridden by other candidates are ignored

```
public interface Animal {
    default public String identifyMyself() {
        return "I am an animal."; }
}
public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs."; }
}
public interface FireBreather extends Animal { }
```

Overriding default methods (2/8)

```
public class Dragon implements EggLayer, FireBreather {

    public static void main (String... args) {
        Dragon myApp = new Dragon();
        System.out.println(myApp.identifyMyself());
    }

}
```

The method Dragon.identifyMyself() returns the string “I am able to lay eggs”

Overriding default methods (3/8)

- If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.
- Consider the example about computer-controlled cars that can now fly. You have two interfaces (OperateCar and FlyCar) that provide default implementations for the same method:

Overriding default methods (4/8)

```
public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

Overriding default methods (5/8)

A class that implements both OperateCar and FlyCar must override the method startEngine(). You could invoke any of the of the default implementations with the super keyword.

```
public class FlyingCar implements OperateCar, FlyCar {

    // ...

    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }

}
```

Overriding default methods (6/8)

- The name preceding super (in this example, FlyCar or OperateCar) must refer to a direct superinterface that defines or inherits a default for the invoked method
- This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature
- You can use the super keyword to invoke a default method in both classes and interfaces

Overriding default methods (7/8)

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

```
public interface Mammal {
    String identifyMyself();
}

public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}
```

Overriding default methods (8/8)

```
public class Mustang extends Horse implements Mammal {

    public static void main(String... args) {
        Mustang myApp = new Mustang();
        System.out.println(myApp.identifyMyself());
    }
}
```

- The method Mustang.identifyMyself() returns the string "I am a horse"
- The class Mustang inherits the method identifyMyself() from the class Horse, which overrides the abstract method of the same name in the interface Mammal

Summary on Default Methods

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Why defaults and static?

- Static methods are for general behaviours of the interface and cannot allow access to non-static data nor overriding
- Default methods can be overridden and they are a powerful mechanism to allow evolution of interfaces, supporting the non-mandatory implementation of all methods inside classes

Sources: `https://softwareengineering.stackexchange.com/questions/233053/why-were-default-and-static-methods-added-to-interfaces-in-java-8-when-we-alread`

Example from Oracle (1/3)

- Suppose you have

```
public interface DoIt {
    void doSomething(int i, double x);
    int doSomethingElse(String s);
}
```

Sources: https://docs.oracle.com/javase/tutorial/java/IandI/nogrow.html

Example from Oracle (2/3)

- Then you want it to evolve into:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);
}
```

- The code of the old implementations would now break, since it does hot have the method `didItWork`

Sources: `https://docs.oracle.com/javase/tutorial/java/IandI/nogrow.html`

Example from Oracle (3/3)

- But with default everything is fixed:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
   default boolean didItWork(int i, double x, String s) {
       // Method body
   }
}
```

Sources: https://docs.oracle.com/javase/tutorial/java/IandI/nogrow.html

Anonymous Classes

- Anonymous classes enable you to make your code more concise
- You can declare and instantiate a class at the same time
- Like local classes except that they do not have a name
- Common usage pattern: when you need to use a local class only once
- The class is defined inside another expression

Listeners And Anonymous Classes

- An event listener is used to process events. For example, a graphical component like a `JButton` or `JTextField` are known as event sources. This means they can generate events - when a user clicks on the `JButton` or types text into the `TextField`. The event listeners job is to catch those events and do something with them.
- Event listeners are actually different types of interfaces. There are many types: the `ActionListener`, `ContainerListener`, `TextListener`, `WindowListener` to name a few.

Listeners and Anonymous Classes (1/2)

- Each interface defines one or more methods that must be implemented by a class in order for the event to be processed.
- However there is a nice flexibility about event listeners in that more than one graphical component can be associated with the same event listener. It means if you have a similar set of components that are basically doing the same thing their events can all be handled by one event listener.
- For example, a `JButton` needs an object of a class that implements an `ActionListener` to process its button. If there are several buttons that do similar tasks when they are clicked then they can all be assigned the same `ActionListener`.

Listeners and Anonymous Classes (2/2)

- When you want to listen a button click you have to have a class that implements the `ActionListener` interface, a simple interface with only one method:

```
public interface ActionListener extends
    EventListener{
      public void actionPerformed(ActionEvent e);
}
```

- To implement the `ActionListener` interface the class needs to have a method `actionPerformed()`. This method is called when a button click event occurs
- The `ActionEvent` object called `e` holds information about the event. For example, you can find out which button was clicked by looking at the method `e.getActionCommand()` method

Motivation for anonymous classes (1/3)

Let's say we want to build a simple calculator (class `SimpleCalc`). First, we need to create a GUI, containing our buttons and a text field to provide the information to the users:

```
public SimpleCalc() {
  guiFrame=new JFrame();
  numberCalc=new JTextField();
  buttonPanel = new JPanel();
  //Make a Grid that has three rows and four columns
  buttonPanel.setLayout(new GridLayout(4,3));
  guiFrame.add(buttonPanel,BorderLayout.CENTER);
  for (int i=1;i<10;i++) { //Add the number buttons
    addButton(buttonPanel, String.valueOf(i));
  }
```

Source for this and the following slides: `https://www.bitspedia.com/2012/12/simple-calculator.html`

Motivation for anonymous classes (2/3)

```
  JButton addButton = new JButton("+"); //Add the operation
     buttons
  addButton.setActionCommand("+");
  buttonPanel.add(addButton);
  buttonPanel.add(subButton);
  buttonPanel.add(equalsButton);
  guiFrame.setVisible(true);
}
private void addButton(Container parent, String name) {
  JButton but = new JButton(name);
  but.setActionCommand(name);
  parent.add(but);
}
```

Motivation for anonymous classes (3/3)

- With `SimpleCalc` we implement the `ActionListener` interface in three different ways to show the different options you have for listening for events:
  - inside the containing class
  - as an inner class
  - as an **anonymous** inner class

Implementation inside the containing class (1/3)

- This approach can be handy shortcut if you have several graphical components that all act in exactly the same way when they are clicked
- In this case the numbered buttons 1 to 9 will all act in the same way.
  - When any of them are clicked the number they represent will be placed in the `JTextField`
- We implement the interface inside `SimpleCalc`, which means that `SimpleCalc` class also needs to implement the method `actionPerformed()`

Implementation inside the containing class (2/3)

```
public class SimpleCalc implements ActionListener{
// All the buttons are doing the same thing.
// It's easier to make the class implementing the
// ActionListener controlling the clicks from one place
@Override
public void actionPerformed(ActionEvent event) {
  //get the Action Command text from the button
  String action = event.getActionCommand();
  //set the text using the Action Command text
  numberCalc.setText(action);
  // Do the job
}
```

Implementation inside the containing class (3/3)

- For this to work we also need to add the `ActionListener` to each button. As the numbered buttons are created using `addButton()` we just need to add in an extra line:

```
private void addButton(Container parent, String
    name) {
    JButton button = new JButton(name);
    button.setActionCommand(name);
    button.addActionListener(this);
    parent.add(button);
}
```

- `addActionListener` gets as parameter the class implementing `ActionListener` with the `this`.

Implementation with an inner class (1/4)

- We could implement `ActionListener` in a separate class but such class needs only to handle the event of a click click of the button.
- Therefore, we could use a class inside `SimpleCalc`.
- This also has the advantage of allowing the inner class access to the components defined within the outer class.
- When the arithmetic operation buttons (`+` and `-`) are clicked, the operations to perform are very similar.
- The only difference is whether numbers are going to be added or subtracted.
- This makes it ideal for both buttons to use an inner class implementing `ActionListener`.

Implementation with an inner class (2/4)

```
public class SimpleCalc{
  private class OperatorAction implements ActionListener {
    public void actionPerformed(ActionEvent event) {
    }
  }
}
```

- `OperatorAction` is the inner class and its main duty is to set the `currentCalc` and `calcOperation` of `SimpleCalc`.
- `currentCalc` holds the number in the `JTextField` and the `calcOperation` the `int` representing whether it is a sum (i.e., 1) or a subtraction (i.e., 2).
- And this is used then by the button `=`.

Implementation with an inner class (3/4)

```
private class OperatorAction implements ActionListener{
  private int operator;
  public OperatorAction(int operator) {
    this.operator = operator;
  }
  public void actionPerformed(ActionEvent event) {
    currentCalc = Integer.parseInt(numberCalc.getText());
    calcOperation = operator;
  }
}
```

Implementation with an inner class (4/4)

```
JButton additionButton = new JButton("+");
additionButton.setActionCommand("+");

OperatorAction additionAction = new OperatorAction(1);
additionButton.addActionListener(additionAction);

JButton subtractionButton = new JButton("-");
subtractionButton.setActionCommand("-");

OperatorAction subtractionAction = new OperatorAction(2);
subtractionButton.addActionListener(subtractionAction);
```

Implementation with an anonymous class (1/4)

- Often, every button triggers a different behaviour
- It is still possible to have a class associated to each of such behaviours,
  - It is enough that each one implements the `ActionListener` interface
- However, such approach triggers a proliferation of "standard" classes, making the code not readable
- We can then use anonymous classes
- With anonymous classes:
  - the code becomes simpler
  - the behaviour associated with a button is placed right where such button is instantiated

Implementation with an anonymous class (2/4)

- A click on the equals triggers
  - the execution of the arithmetic operation and
  - the display of the result in the `JTextField`.
- It is the perfect candidate for an anonymous (inner) class

```
JButton equalsButton = new JButton("=");
equalsButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
      // Do something
    }
});
```

Implementation with an anonymous class (3/4)

- Instead of writing a separate class implementing the `ActionListener` interface, an anonymous class is defined right there.
- We perceive the new anonymous class as a piece of code created on the fly for the `ActionListener` interface implementing the `actionPerformed` method
  - We get the impression that a piece of code could be the parameter of the call of the method `addActionListener`
  - But it is not so, it is a fully fledged (anonymous) class that is created, and an object of it is then instantiated and passed as a parameter

Implementation with an anonymous class (4/4)

```
equalsButton.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent event) {
    if (!numberCalc.getText().isEmpty()) {
      int number = Integer.parseInt(numberCalc.getText());
      if (calcOperation == 1) {
        ... // perform addition
      } else if (calcOperation == 2) {
        ... // perform substraction
      }
    }
  }
});
```

Questions?

End of the lecture on the strategy pattern in Java.