Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 8:

Modularity and More Design Patterns

Last Week

- Exceptions
- Packages
- Nested Classes

Lecture 8a: Overview

- Modularity Principles (1)
- Modularity Principles (2)
- Factory Design Patterns
- Strategy Design Pattern

Modularity

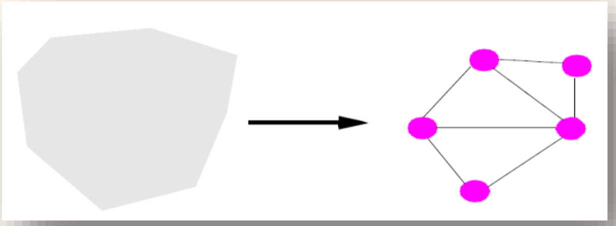
- A Modular design results in a software that can be broken down to several individual units, denoted **modules**
- Modular programs have several benefits
  - Easy to maintain (debug, update, expand)
  - Allow breaking a complex problem into easier sub-problems
  - Allow to easily divide the project into several team members or groups

Modularity Principles

- A design method which is “modular” should satisfy 4 fundamental requirements:
  - Decomposability
  - Composability
  - Understandability
  - Continuity

Decomposability

- A software design satisfies *Modular Decomposability* if it:
  - Decomposes a software problem into a small number of **less complex** sub-problems
  - These sub-problems are connected by a **simple structure**, and independent enough to allow further work to proceed separately on each of them

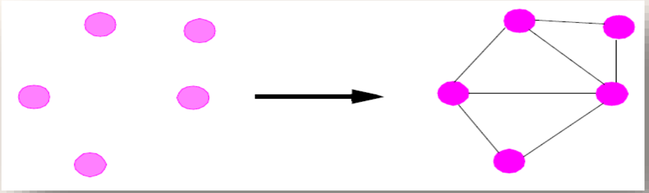


Decomposability

- A corollary of decomposability is division of labor
  - A decomposed system is easier to distribute among different people or groups
- **A good example**: Top-Down Design
- **A typical counter-example**: a software system that includes a global initialization module

Composability

- A method satisfies *Modular Composability* if it produces software elements which may be freely ***combined with each other to produce new systems***
  - Possibly in an environment quite different from the one in which they were initially developed



Composability

- Elements should be sufficiently **autonomous**
- Composability is directly connected with the goal of reusability
  - Design software elements performing well-defined tasks and usable in widely different contexts
- Example: Software libraries (or packages)

Composability Vs. Decomposability

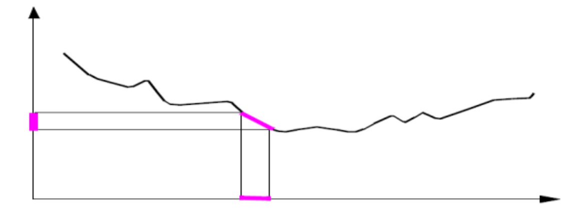
- The principles of composability and decomposability are **independent**
  - In fact, these criteria are often at odds
  - Top-down design, for example, which we saw as a technique favoring decomposability, tends to produce modules that are not easy to combine with modules from other sources

Understandability

- A software design favors *Modular Understandability* if it produces software in which a **human reader** can **understand** each module without having to know anything about the **others**
  - At worst, by having to examine only a few of the others
  - Important for the maintenance process
  - Rule of thumb: can you describe each module in a few words?
- This is **not** the same as readability

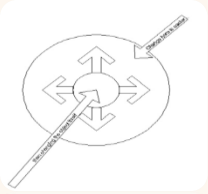
Modular Continuity

- **Continuity** – A method satisfies *Modular Continuity* if a small change in the problem specification triggers a change in just one module, or a small number of modules
  - Minimize dependencies between different modules
  - The term “continuity” is drawn from an analogy with the notion of a continuous function in mathematical analysis



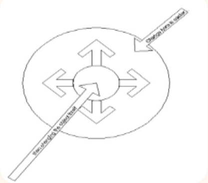
Lecture 8b: Overview

- Modularity Principles (1)
- Modularity Principles (2)
- Factory Design Patterns
- Strategy Design Pattern



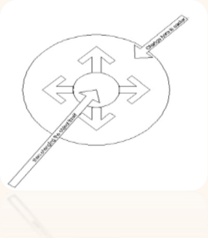
The Open-Closed principle

- Software entities (Classes, Modules, Functions, etc.) should be **open for extension but closed for modification** (Meyer, 1988)
- A single change to a program ➔ a cascade of changes to dependent modules ➔ **“bad” design**
  - The program becomes fragile, rigid and un-reusable
  - Violation of the *Continuity* principle
- The solution: design modules that **never change**
  - Requirements change ➔ extend the modules by **adding new code**, not by **changing old code** that already works!



The Open-Closed principle

- Modules that conform to the open-closed principle have two primary attributes:
  - They are “**Open for Extension**”: the behavior of the module can be extended; we can make the module behave in new and different ways as the requirements of the application change
  - They are “**Closed for Modification**”: the source code of such a module is inviolate. No one is allowed to change it



The Open-Closed principle

Shape example

- We have an application that must be able to draw circles and squares on a standard GUI
  - The circles and squares are drawn in a particular order
- The program traverses and draws a list of circles and squares in a given order
- Possible solutions:
  - Procedural solution – shape type is queried each time we draw it
  - OOP solution – a common interface is used

Shape Example

Procedural Solution

```
void drawAll(Shape[] list) {
    for (int i=0; i < list.length; i++) {
        Shape s = list[i];
        int type = getType(s);
        switch (type){
            case SQUARE:
                    drawSquare((Square)s);  break;
            case CIRCLE:
                    drawCircle((Circle)s);  break;
        }
    }
}
```

Procedural Solution

What's Wrong Here?

- The *drawAll*() method doesn't conform to the open-close principle
  - It is not **closed** against new kinds of shapes
  - If we wanted to **extend** this function to draw a list of shapes that included **triangles**, we would have to **modify it**
- This program is only a simple example
  - This **switch** statement could be **repeated** **over and over** again in various functions, each one **doing something a little different**
  - Adding a new shape means hunting for every such **switch** statement and adding the new shape

The Shape Example

OOP Solution

```
public interface Drawable{
	public void draw();
}
public class Square implements Drawable {
	public void draw() {..}
}
public class Circle implements Drawable {
	public void draw() {..}
}
public void drawAll (Drawable[] list) {
	for (Drawable drawable: list)
		drawable.draw();
}
```

OOP Solution – Advantages

- Extending the behavior of the *drawAll*() method to draw a new kind of shape, is done by adding a new implementation of the *Drawable* interface
- Our **design** is **open** to changes in the software's requirements, while *drawAll*() is **closed** to the changes
- ➔ *drawAll*() now conforms to the open-closed principle
  - Its behavior can be extended without modifying it

The Single-Choice Principle

- If a software system **must** support a set of alternatives, **one and only one** module in the system should know their exhaustive list
- By doing this, we prepare the scene for later changes:
  - If variants are added, we only have to update the module which has the information — the point of single choice
  - All others, in particular its clients, are able to continue their business as usual
  - This principle interacts with the **open-closed** principle:
    - Keep our exhaustive list of options in one place, so that this is the only place that needs to be changed upon updates

The Shape Example

OOP Solution

```
public Drawable[] loadAll (String[] list) {
	Drawable[] drawables = new Drawable[list.length];
	for (int i = 0 ; i < list.length ; ++i) {
		if (list[i].equals("Square")) {
			drawables[i] = new Square();
		} else if (list[i].equals("Circle")) {
			drawables[i] = new Circle();
		} …
	}
	return drawables;
}
```

One method must know all the options.
It **cannot** be closed for changes

The Shape Example

OOP Solution

```
void drawAll(Drawable[] list) {
	for (Drawable drawable: list)
		drawable.draw();
}

void deleteAll (Drawable[] list) {
	for (Drawable drawable: list)
		drawable.delete();
}
```

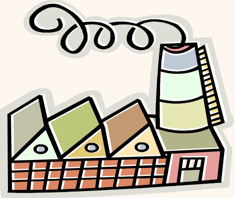
All other methods don‘t need to know the options.
They are **closed** to changes

Lecture 8c: Overview

- Modularity Principles (1)
- Modularity Principles (2)
- Factory Design Patterns
- Strategy Design Pattern

Reminder: Design Patterns Properties

- Describes a proven approach to dealing with a common situation in programming / design
- Suggests what to do to obtain an elegant, modifiable, extensible, flexible & reusable solution
- Shows, at design time, how to avoid problems that may occur much later
- Is independent of specific contexts or languages



Factory Design Patterns

- A **factory** is an object used to create other objects
  - A factory may be a **class** or a **method**
  - Factories may receive parameters that define the type and properties of the objects to be created
- Factories are used to decouple the creation of objects from the rest of the program
  - They are especially useful in situations where deciding which object to create is a complex task

Factory Design Patterns (2)

- Factories may create the object *dynamically*, return it from some *object pool*, do *complex initialization*, etc.
- Factory is not a concrete design pattern, but a family of **creational** design patterns
  - Abstract Factory, Factory Method, etc.

Factory Example – *loadAll*()

- Recall the shape example previously discussed
- The *loadAll*() method serves as a **factory** in this case
  - It handles the object creation for the system
  - This is the only part of the program responsible for creating object
- Using a factory allows us to maintain the **open/closed** and **single-choice** principles

```
public Drawable[] loadAll (String[] list) {
    Drawable[] drawables = new Drawable[list.length];
    for (int i = 0 ; i < list.length ; ++i) {
        if (list[i].equals("Square")) {
            drawables[i] = new Square();
        } else if (list[i].equals("Circle")) {
            drawables[i] = new Circle();
        } …
    }
    return drawables;
}
```

loadAll() – Take II

- Other implementations of loadAll() can use an object pool

```
public class ShapeFactory {
    private static final SQUARE_OBJECT = new Square();
    private static final CIRCLE_OBJECT = new Circle();
    …
    public Drawable[] loadAll (String[] list) {
            Drawable[] drawables = new Drawable[list.length];
            for (int i = 0 ; i < list.length ; ++i) {
              if (list[i].equals(“Square”))  drawables[i] = SQUARE_OBJECT;
              else if (list[i].equals(“Circle”)) drawables[i] = CIRCLE_OBJECT
               …
            }
            return drawables;
    }
```

The Singleton Design Pattern

- Intent
  - Ensure a class only has exactly **one instance**, and provide a global point of access to it
- A **creational** pattern
- The problem
  - If we want a single instance of a class to exist in the system
    - A single **window manager**, one **factory** for a family of products, etc.
  - We need to have that one instance easily accessible
  - Additional instances of the class cannot be created

Singleton: The Solution

- Store an instance of the class as a **static data member**
- Make the constructor **private**
- Create the (single) instance in the **static** *instance*() method
  - This method always returns a reference to the same single object
  - This way only one instance at most is created

Singleton Example

```java
public class Singleton {
    private static Singleton single =
                                   new Singleton();

    private Singleton () { … }

    public static Singleton instance() {
        return single;
    }
}
```

```java
Singleton s  = new Singleton();
// Illegal – constructor is private

Singleton s2  = Singleton.instance();
// single instance returned

Singleton s3  = Singleton.instance();
// s2 == s3
```

Singleton: Implications

- Using a single **private** constructor makes it **impossible** to subclass

Singleton vs. a Class of static Methods

- Why not use a class with **static** fields and methods instead of using a singleton?
  - Make constructor **private** and all methods / data members **static**
  - This way no instance of this class exists, but effectively, it is as is a **single instance** only exists
- Answer: singletons can implement interfaces
  - They can be used as regular classes, e.g., be up-cast (**polymorphism**)
  - Clients that use an up-cast singleton may not even be aware that they are using an object of a singleton class (**information hiding**)

Lecture 8d: Overview

- Modularity Principles (1)
- Modularity Principles (2)
- Factory Design Patterns
- Strategy Design Pattern

The Strategy Design Pattern

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Let the algorithm vary independently from the clients that use it
- A **behavioral** pattern

Strategy: The Problem

- Case 1: There exist different variants of an algorithm
  - Each of these algorithms uses data that clients shouldn't know about
  - Clients may prefer to switch variants during runtime, depending on the input from its environment
- Case 2: A class defines many behaviors, and these appear as multiple conditional statements in its operations

Strategy: The Solution

- Define a strategy API (**abstract class** or **interface**)
  - Let each behavior / algorithm variant implement this API
- Instead of many conditionals, move branches into their own Strategy class
  - Use this pattern to avoid exposing complex, algorithm-specific details
- Let the client select the right behavior / algorithm upon creation
  - Use a **factory** to select the strategy

Strategy Example

- A class needs to select an algorithm for sorting an array
  - Many different sorting algorithms are available
- Solution: encapsulate the sort variants with the Strategy pattern!

Strategy Example

```
public class SomeCollection {
    private Comparable[] contents;
    private SortStrategy sorter;

    public SomeCollection() {
        this.sorter =
            SortStrategyFactory.select(…);
    }
    public void sortContents() {
        this.sorter.sort(this.contents);
        ….
    }
}
```

```
public interface SortStrategy {
    void sort(Comparable[] data) ;
}
```

```
public class QuickSort implements SortStrategy {
    public void sort(Comparable[] data) {…}
}
```

```
public class MergeSort implements SortStrategy {
    public void sort(Comparable[] data) {…}
}
```

```
public class SortStrategyFactory {
    public static SortStrategy select(…) {…}
}
```

Strategy vs. Inheritance

- Why not create a class that **extends** the **client** base class for each algorithm variant / potential behavior?
  - Potential Strategy scenarios are hardly ever a case of “is-a” relation
  - Modularity – separate between client code and algorithm/behavior code
  - Information hiding – client doesn’t need to publicly declare its usage of the algorithm / behavior
  - Reuse algorithm code (different classes use sorting algorithms)
  - Change behavior during run-time
  - Clients can extend another classes

One more Note on Strategy

- Strategy relates to the *open/closed* and the *single-choice* principles

Strategy + Factory + Singleton

- In the last example, we saw how **Strategy** and **Factory** design patterns can work together
- As a matter of fact, this case is **Singleton** pattern

```
public class SomeCollection {
    private Comparable[] contents;
    private SortStrategy sorter;

    public SomeCollection() {
        this.sorter = SortStrategyFactory.select(…);
    }
    public void sortContents() {
        this.sorter.sort(this.contents);
        ….
    }
}
```

Strategy + Factory + Singleton

```
public class QuickSort implements SortStrategy {
    private static QuickSort instance =
                                    new QuickSort();
    private QuickSort () { … }

    public static QuickSort instance() {
        return instance;
    }

   public void sort(Comparable[] data) {…}
}
```

```
public interface SortStrategy {
    void sort(Comparable[] data) ;
}
```

```
public class SortStrategyFactory {
    public static SortStrategy select(…) {
        if (cond1)
            return QuickSort.instance();
        else if (cond2)
            return MergeSort.instance();
        ….
    }
}
```

Why use Singleton here?

- No need for more than one instance (saves runtime and memory)
- If at some point we would need more than one instance, the *instance*() method can be overridden to always create a new instance
  - Rest of code is unaffected and unaware of such a change
- Unlike a general class with static methods, singletons can implement interfaces (as in the example here)

So far…

- Among most important OOP design principles
  - Modularity
  - Open-closed
  - Single-choice
- A few design patterns
  - The Factory concept
  - Singleton
  - Strategy

Next Week

- Streams
- Decorator Design Pattern