Chapter 3-Integrative Coding

Email:begna19mrblgo@gmail.com
visit: https://begnafrique.wordpress.com/

Contents

Part 1: Design Patterns

Part 2: Interface

Part 3: Inheritance

Part 4:Versioning and version control

Design Patterns

What are Design Patterns

- Designing is an art and it comes with the experience.
- But there are some set of solutions already written by some of the advanced and experienced developers while facing and solving similar designing problems.
- These solutions are known as Design Patterns.
- The Design Patterns is the experience in designing the object oriented code.
- Design Patterns are general reusable solution to commonly occurring problems.
- Patterns are not complete code, but it can use as a template which can be applied to a problem.

Design Patterns

- What are Design Patterns
  - Patterns are re-usable; they can be applied to similar kind of design problem regardless to any domain.
  - In other words, we can think of patterns as a formal document which contains recurring design problems and its solutions.
  - A pattern used in one practical context can be re-usable in other contexts also.

Design Patterns

- It is recurring solution to recurring problems in software architecture.
- It is a description or template for how to solve a problem that can be used in many different situations.
- A Lower level framework for structuring an application than architectures (Sometimes, called *micro-architecture*).
- Reusable collaborations that solve sub problems within an application.
- ***Why Design Patterns?***
  - Design patterns support ***object-oriented reuse*** at a high level of abstraction
  - Design patterns provide a “**framework**” that guides and constrains object-oriented implementation

Design Patterns

- Organizing of Design Patterns
  - Design patterns can be categorized in the following categories:
    - ***Creational patterns*** : used to help make a system independent of how its objects are created, composed and represented.
      - Creational design patterns are used to design the instantiation process of objects. The creational pattern uses the inheritance to vary the object creation.

Design Patterns

- ***Structural patterns*** are concerned with how classes and objects are organized and composed to build larger structures.
  - Structural class patterns use inheritance to compose interfaces or implementations
  - As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

- .

Design Patterns

- Organizing of Design Patterns
  - Design patterns can be categorized in the following categories:
    - ***Behavioral patterns*** are used to deal with assignment of responsibilities to objects and communication between objects.
      - Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
      - Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

Design Patterns

List of Patterns	Types	Description
Creational patterns	Abstract factory	Creates an instance of several families of classes
	Builder	Separates object construction from its representation
	Factory Method	Creates an instance of several derived classes
	Prototype	A fully initialized instance to be copied or cloned
	Singleton	A class of which only a single instance can exist
Structural Patterns	Adapter	Match interfaces of different classes
	Bridge	Separates an object's interface from its implementation
	Composite	A tree structure of simple and composite objects
	Decorator	Add responsibilities to objects dynamically
	Façade	A single class that represents an entire subsystem
	Flyweight	A fine-grained instance used for efficient sharing
	Proxy	An object representing another object

Design Patterns

Behavioral patterns	Chain of Responsibility	A way of passing a request between a chain of objects
	Command	Encapsulate a command request as an object
	Interpreter	A way to include language elements in a program
	Iterator	Sequentially access the elements of a collection
	Mediator	Defines simplified communication between classes
	Memento	Capture and restore an object's internal state
	Observer	A way of notifying change to a number of classes
	State	Alter an object's behavior when its state changes
	Strategy	Encapsulates an algorithm inside a class
	Template Method	Defer the exact steps of an algorithm to a subclass
	Visitor	Defines a new operation to a class without change

Design Patterns

Example(Façade)

- **Facade:**
  - **Provides a unified interface to a set of interfaces in a subsystem.**
  - Façade defines a higher-level interface that makes the subsystem easier to use.
  - This structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

Interface

- Application programming interface
  - Are sets of requirements that govern how one application can talk to another
  - applications to share data and take actions on one another's behalf without requiring developers to share all of their software's code
  - define exactly how a program will interact with the rest of the software world—saving time, resources
  - Eg:- **System-level APIs-** cut and paste LibreOffice document into an Excel spreadsheet
  - Eg:-FacebookAPIs- Facebook users sign into many apps and Web sites using their Facebook ID
  - Eg:-Web APIs – games let players chat, post high scores and invite friends to play via Face book, right there in the middle of a game

Inheritance

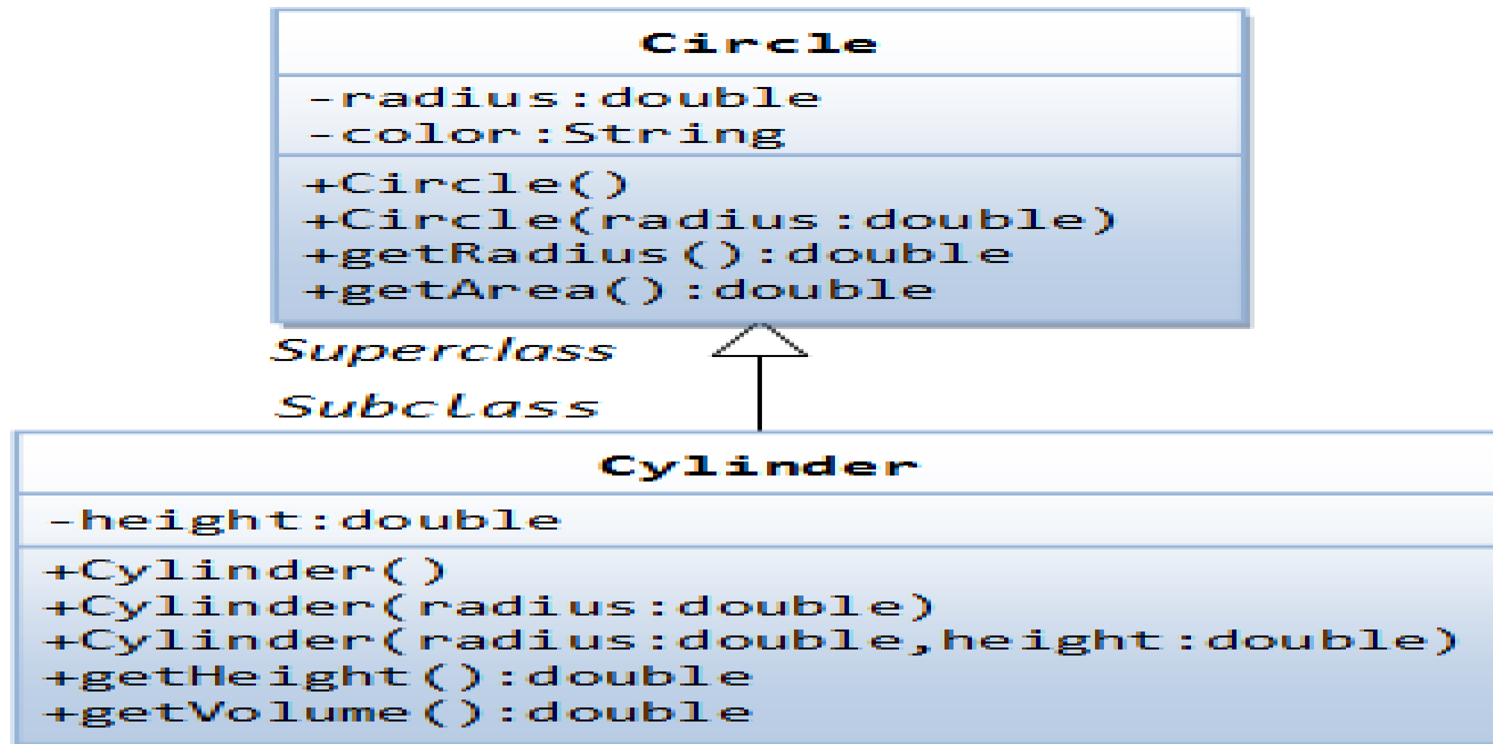
- derive a new class based on an existing class, with modifications or extensions
- A subclass inherits all the variables and methods from its super classes, including its immediate parent as well as all the ancestors
- *avoid duplication and reduce redundancy*
- **Types of Inheritance**
  - Simple , Multilevel, Multiple, hierarchical and Hybrid
- **Inheritance and Abstract class**
  - **Abstract Method:-** a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body).
  - use the keyword abstract to declare an abstract method

Inheritance

- **Abstract Class**
  - A class containing one or more abstract methods is called an abstract class.
  - must be declared with a class-modifier abstract
  - provides a ***template for further development***
  - **Notes:**
  - An abstract method **cannot be declared final**, as final method **cannot be overridden.**
  - An abstract method must be **overridden in a descendent** before it can be used.
  - An abstract method **cannot be private** (which generates a compilation error, because private method is **not visible to the subclass and thus cannot be overridden.**
  - **In Java, define a subclass using the keyword "extends", e.g.,**
    - class MyApplet **extends** java.applet.Applet {.....}

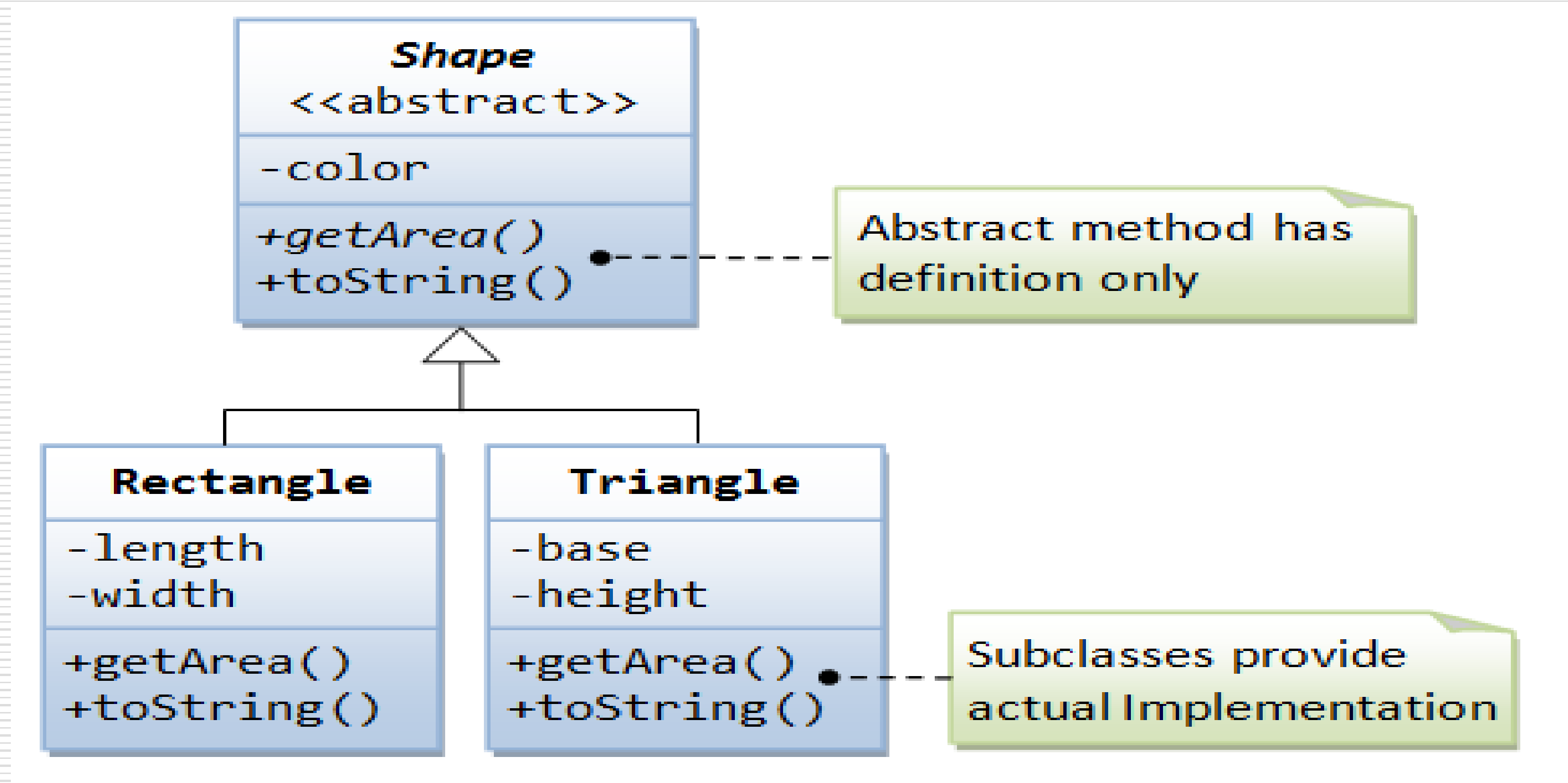
Inheritance

- Inheritance example



Inheritance

- Example for Abstract class and Inheritance



Inheritance

- Abstract class and Inheritance in Java
  - Shape.java

```java
abstract public class Shape
        {
        private String color; // Private member variable
        public Shape (String color)    // Constructor
         {       this.color = color;
         }
             public String toString()
         {
              return "Shape of color=\"" + color + "\"";
         }
        // All Shape subclasses must implement a method called getArea()
           abstract public double getArea();
    }
```

Inheritance

- Abstract class and Inheritance in Java
  - Rectangel.java

```
public class Rectangle extends Shape
{
    //Private member variables
    private int length;
    private int width;

    //constructor
    public Rectangle(String color,int length,int width)
    {
        super(color);
        this.length=length;
        this.width=width;
    }
    @override
    public double getArea()
    {
        return length*width;

    }
```

Inheritance

- Abstract class and Inheritance in Java
  - traingle.java

```java
public class Triangle extends Shape
{
	//Private member variables
	private int length;
	private int base;

	//constructor
	public Triangle(String color,int length,int base)
	{
		super(color);
		this.length=length;
		this.base=base;
	}
	@override
	public double getArea()
	{
		return 0.5*length*base;

	}
}
```

Chapter 4

Versioning and version control

Versioning and version control

- Version control enables ***multiple people to simultaneously*** work on a single project.
- Each person ***edits*** his or her own copy of the files and chooses when to ***share those changes*** with the rest of the team.
- ***temporary or partial edits*** by one person ***do not interfere*** with another person's work.
- enables one person to ***use multiple computers*** to work on a project
- ***integrates work done simultaneously*** by different team members
- In rare cases, when two people make *conflicting edits* to the same line of a file, then the version control ***system requests human assistance*** in deciding what to do
- Version control gives ***access to historical versions of the project***

Versioning and version control

- If make a mistake, ***roll back to a previous version***. ***reproduce and understand a bug report*** on a past version of your software.
- ***undo specific edits without losing all the work*** that was done in the meanwhile.
- For any part of a file, ***determine when, why, and by whom it was ever edited.***
- Version control uses ***a repository*** (a database of changes) and a **working copy (checkout)** where you do your work
- *working copy* is your ***personal copy of all the files*** in the project.
- ***edits to this copy, without affecting your teammates.***
- ***commit your changes to a repository***
- *repository* is ***database of all the edits to, and/or historical versions (snapshots) of, your project***
- update your working copy to ***incorporate any new edits or versions***

Versioning and version control

- Two varieties of version control: *centralized* (one repository) and *distributed* (multiple repositories)
- Some popular version control systems are Mercurial (distributed), Git (distributed), and Subversion (centralized).
- The main difference between centralized and distributed version control is the number of repositories.
- In centralized version control, there is just one repository, and in distributed version control, there are multiple repositories.

Read more..

Thank you