SE2 DEAD

Abdrakhmanova Zh. (01449508), Jelbuldin R. (), Nikitina O. (a01467981), Zhamukhanova A. (01549494)
Team 303

I. Design Patterns

Iterator Pattern

An Iterator Pattern is a behavioral design pattern that makes it possible to consistently bypass the elements of composite objects without revealing their internal representation. It is used to bring the collection traversal behavior from the collection itself into a separate class. The iterator object tracks the status of the traversal, the current position in the collection, and how many elements are left to iterate. The same collection can simultaneously be iterated by different iterators, and the collection itself will not even know about it. The main advantage of using this pattern is well structured and encapsulated code . For example, if you need to add a new sequence iteration, you can create a separate iterator class without changing the existing collection code.

In our code we have two interfaces: ShapeIterator and ShapeContainer. In ShapeIterator there are defined two methods hasNext() and next() which are implemented in IteratorImpl class. In ShapeContainer there is function getIterator() defined. ShapeContainerImpl is a concrete class implementing the ShapeContainer interface. The method getShapeIfExists()in EditorManager class uses ShapeContainerImpl to get ShapeIterator by ID to iterate the list of shapes stored as a collection in ShapeContainerImpl.

Composite Pattern

Composite pattern is a structural design pattern that combines objects in a tree structure to represent the hierarchy from the particular to the whole. Linker allows customers to access individual objects and groups of objects in the same way. The composer is used when objects must be implemented as a hierarchical tree structure and when clients uniformly must manage a group of objects as one object. That is, the whole and its parts must implement the same interface. In our project this pattern is implemented in shape creation. Our Component is the Shape interface which is the base interface for all the shape objects in the composition. It has the common methods to manage the child composites. Circle, Triangle, Quadrangle, Line, Text, Star, Ngon and Ellipse are Leaf classes, which implement the default behavior of the base component. They doesn’t contain a reference to the other objects. EditorManager is our Composite element, which has a list of leaf elements (shapes). It implements the base component methods and defines the child-related operations.

Proxy Pattern

Proxy Pattern is a structural design pattern that provides a substitute object that controls access to another real object, performing the function of interface. A surrogate or substitute is an object whose interface is identical to the interface of a real object. At the first client request, the proxy creates a real object, saves its address, and then sends the request to tha real object. All subsequent requests are simply forwarded to the encapsulated real object. In our project Proxy Pattern is implemented in SVG interface and RealSVG, LayerManager classes. This pattern has SVG interface, which is implemented in sub classes. LayerManager class acts as Proxy substitute for RealSVG object. Proxy is implemented in saveSVG() method, when all objects created by user are saved into html String. If a user wants to generate and download the created shapes as svg file, the RealSVG object will be generated.

Abstract Factory Pattern

Abstract Factory Pattern is a generating design pattern, provides an interface for creating families of interrelated or interdependent objects, without specifying their specific classes. The template is implemented by creating an abstract factory class, which is an interface for creating system components (for example, it can create windows and buttons for a window interface). Then write classes that implement this interface. FactoryProducer is a factory creator class in our project. We implemented just one factory - shape factory, which is implemented in EditorManager class. Alternatively, the abstract factory can be extended by adding additional factories. EditorManager class implements AbstractFactory methods: getShapes(), iterator(), deleteShape(), scaleShape(), moveShape(), createShape() and changeColor().

Factory Method Pattern

The factory method is a creational design pattern that provides an interface for subclasses (child classes) to create instances of a certain class. At the time of creation, the heirs can determine objects of which class to create. In other words, this template delegates the creation of objects to the inheritors of the parent class. This allows to use in the program code not specific classes, but to manipulate abstract objects at a higher level. We implemented this pattern inside Abstract Factory Pattern. EditorManager class implements the createShape() method which is defined in AbstractFactory class. Ech Shape object is created in this method and then all the observers are being notified about the changes.

Decorator Pattern

Decorator Pattern is a structural pattern. This pattern dynamically adds new responsibilities to the object. Decorators are a flexible alternative to creating subclasses to extend functionality. If you need to add new responsibilities in the behavior or state of individual objects during program execution, the use of inheritance is not possible, since this solution is static and extends entirely to the whole class. Decorator pattern makes it possible to add a new functionality to the object without influencing the whole code. In our project the shapes can have one additional functionality, such as changing color of the object. We already have Shape interface and other shape objects, which implement Shape interface. ShapeDecorator is an abstract decorator class, which implements Shape interface and has Shape objects as its instance variable. FillColorDecorator implements ShapeDecorator enabling the shape object to have a changing color functionality.

Observer Pattern

This pattern allows the objects to monitor and respond to changes in other objects. As soon as the user adds a new figure or deletes an existing one, the Observer class will be notified immediately of the changes. The EditorManager class is a Subject component, and the IdOserver class is a dependent component in the Observer hierarchy that implements the Observer abstract class. In our project there is only one observer (IdObserver), which is added to the list of observers in EditorManager (Subject) and listens to the changes. As soon as a new figure is added, the Observer notifies our EditorManager via method update() that there is a new figure in the list and updates the list of IDs of the existing figures. Then EditorManager informs all listeners about the changes using notifyAllObservers() method. If required, however, additional observers can be added using the attach (observer: Observer) method. We fixed our mistakes in previous implementation of this pattern, viz. the observer pattern implementation doesn't have in the update() interface method parameters.

Strategy Pattern

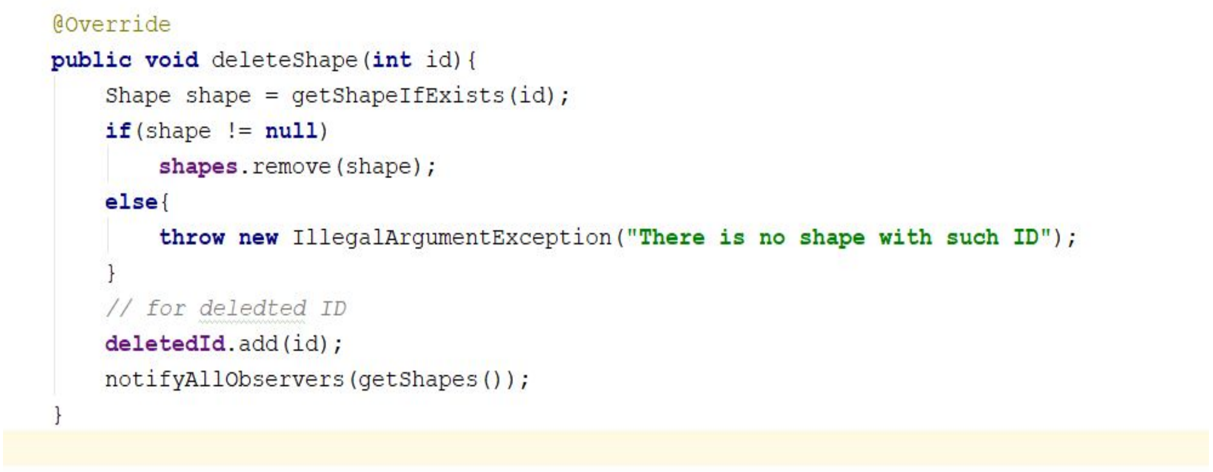
Strategy Pattern is a behavioral design pattern designed to define a family of algorithms, encapsulate each of them and ensure their interchangeability. This pattern allows to choose an algorithm by determining the appropriate class and to change the selected algorithm, independently of the client objects that use it. Strategy pattern is used when the program must provide different variants of the algorithm or behavior, when it is necessary to change the behavior of each class instance or when it is necessary to change the behavior of objects at the execution stage. The introduction of the interface allows the client classes to exist independently of the classes that implement this interface and encapsulate specific algorithms. Strategy pattern is contained in Strategy Pattern folder in our project. MoveStrategy is an interface in which the action move() is defined. MoveDown, MoveLeft, MoveUp, MoveRight are concrete classes which implement the move strategy declared in interface. MoveContext is a class that uses a MoveStrategy. The shape object can be moved in four different directions:right, left, up and down. Depending on the user choice the MoveContext behaviour will change according to the corresponding move strategy.

II. Coding Practices

Delivering a working program is the main task of developer, but it is not the only one. The code should be written according to the coding practices and techniques. We used techniques defined in the lecture as well as the agile programming best practices while implementing our group project. We tried to maintain readability, so that all team members could easily understand what a certain piece of code is doing. By implementing design patterns we ensured maintainability of code, as if it would have to be changed or improved, it can be easily implemented by adding new classes or methods without influencing the base structure. Nevertheless, the code was written by four team members, we tried to follow one style to make it look like it was written by a single developer. Class names, method names and variable names are meaningful and goal revealing. For example, class names of Strategy pattern: MoveRight, MoveUp, MoveDown, MoveLeft or Decorator pattern: ShapeDecorator, FillColorDecorator are easy to understand without opening the code. Also, such method names as getShapes(), createShape(), changeColor(), getShapeIfExists(), deleteShape(), moveShape() ecc. Are self-explaining and you don't need to "dive into the code" to find out what the concrete method is doing. Even though the code is easy to understand and there is a documentation, we also provide internal documentation, by writing javadoc. We also tried to minimize the code size implementing the same code in one function where it was possible.
While developing we tried to use agile programming practices, which enable more frequent delivery with higher quality. These agile best practices help the programmers and the code itself become more agile and therefore more efficient. For example, our team used Test-Driven Development, which helped us to find mistakes right away and correct them immediately. We experienced pair programming and sharing the codebase between the team members.

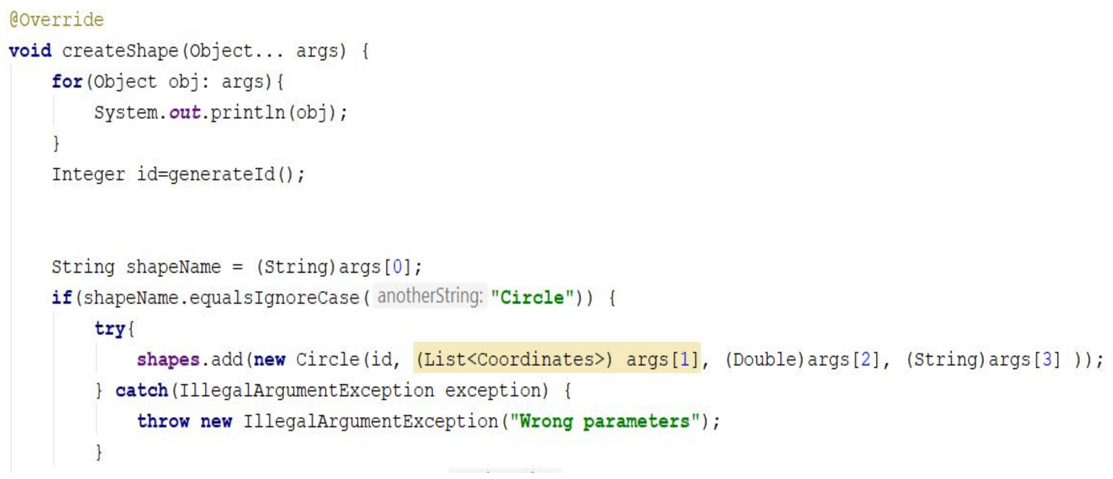
III. Defensive Programming

Defensive programming is a form of defensive design applied to software engineering that tends to ensure the behavior of any element of an application in any situation of use as incorrect or unpredictable as it may seem. Defensive programming techniques are used especially in critical components whose malfunction, whether due to carelessness or malicious attack, could lead to serious consequences or catastrophic damage. We used defensive programming in our team project as an approach that tends to improve software and source code in terms of quality (reducing the number of software failures and, consequently, problems), making the source code understandable (so that the source code is readable and understandable, proof of a code audit) and making the software behave in a predictable manner despite unexpected user input or actions.



```java
@Override
public void deleteShape(int id){
    Shape shape = getShapeIfExists(id);
    if(shape != null)
        shapes.remove(shape);
    else{
        throw new IllegalArgumentException("There is no shape with such ID");
    }
    // for deledted ID
    deletedId.add(id);
    notifyAllObservers(getShapes());
}
```

For example, in deleteShape() method shape object ID is checked and if a user entered an ID of the shape that doesn't exist, this method will throw an exception, which will be then processed by the front-end interface and alert message will pop up.



```
@Override
void createShape(Object... args) {
    for(Object obj: args){
        System.out.println(obj);
    }
    Integer id=generateId();


    String shapeName = (String)args[0];
    if(shapeName.equalsIgnoreCase( anotherString: "Circle")) {
        try{
            shapes.add(new Circle(id, (List<Coordinates>) args[1], (Double)args[2], (String)args[3] ));
        } catch(IllegalArgumentException exception) {
            throw new IllegalArgumentException("Wrong parameters");
        }
```

Defensive programming principles are also illustrated in the example above. If user enters the parameters that the program doesn't expect, IllegalArgumentException will be thrown by createShape() method. The thrown exception will be processed by front-end and an alert message will be shown to the user.

IV. Code Metrics

Number of packages	**3**
Lines of Code	**2113**
Comment Lines of Code	**126**
Number of classes	**32**

The following bugs were found with the help plugin SpotBugs.

Bug: In method saveSVG() the System.exit(..) were invoked, it is not advised to use it as it shuts down the entire virtual machine.

Resolution: throwing Exception instead.

Bug: Dead store to idObserver in method addAbstractFactory. The value is assigned to local variable but were never used. It is often indicates an error.
Resolution: Delete the unused local variable.

Bug: chooseLayer(String, Model) concatenates strings using + in a loop.
The method seems to be building a String using concatenation in a loop. In each iteration, the String is converted to a

StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration.

Resolution: Better performance can be obtained by using a StringBuffer (or StringBuilder in Java 1.5) explicitly.

Bug: MainController MyErrorController is not _static_ inner class
Resolution: It was unused method. We deleted it.

V. Team Contribution

- Abdrakhmanova Zh.: Line.java, Star.java, Iterator pattern, Strategy pattern.
- Jelbuldin R.: Text.java, Triangle.java, Composite pattern, Observer pattern, creating Layers.
- Nikitina O. : Ellipse.java, Quadrangle.java, Shape.java, Proxy pattern, Factory method pattern, delete shape functionality.
- Zhamukhanova A.: Circle.java, Coordinates.java, Ngon.java, Abstract factory pattern, Decorator pattern, scale shape functionality.

VI. HowTo

Requirements: Maven, Java

1. Go to the folder where the file pop.xml stored
2. Open Terminal
3. Use command to start application: mvn spring-boot:run
4. Wait till the application started
5. Go in browser the page http://localhost:8080/circle