***PART I: Introduction***

This software project consists of creating a Graphical User Interface (GUI) in Java, painting shape objects in the GUI, and sorting shape objects with a sorting algorithm and the Comparable interface. The goal of this project is to apply Object Oriented Design (OOD) principles and design patterns to produce stable software that fulfills the provided specifications. The OOD principles used throughout the software project are Inheritance, Abstraction, and Encapsulation. Abstraction is used to hide details of how shape objects are instantiated and sorted from other classes, thus easing their interaction. Encapsulation is used to hide variables from other classes so that they can only be accessed and modified from within their class. Inheritance is used to describe the relationship between the Shape class, Circle class, Square class, and Rectangle class. Inheritance represents an “is-a” relationship, and the Circle, Square, or Rectangle “is-a” shape, so these classes will inherit the states and methods of the Shape class. The design pattern used throughout the software project is the Singleton pattern. The Singleton pattern ensures that only single objects of a class get created. This report will be structured into four parts: an introduction, design of the solution, implementation of the design, and conclusion.

***PART II: Design of the solution***

Diagram

Description automatically generatedThe first UML class diagram of the system is shown below. The DisplayShapes class creates the GUI and paints the Shape objects. The DisplayShapes class inherits the add() method from the parent class JPanel. The DisplayShapes class has an association relationship with the ShapeFactory and SortingTechnique classes because instances of these classes are stored in the fields sortList and creator. The Shape class has subclasses Circle, Rectangle, and Square. The Shape class has an association relationship with the DisplayShapes and ShapeFactory classes because instances of the Shape class are stored in the fields shapeList. The Shape class has a composition relationship with the SortingTechnique class because SortingTechnique objects are instantiated with a List of Shape objects. Therefore, they (SortingTechnique objects) live and die when the Shape class lives and dies. The Shape class implements the Comparable interface.

The Object Oriented Design principles used in the class diagram are Encapsulation, Inheritance, and Abstraction. Encapsulation is used to hide variables from other classes so they can only be accessed and modified within their class. Variables are encapsulated by specifying their access modifier as private. This is represented by a “-” symbol in front of the variable name. Inheritance represents an “is-a” relationship whereby a class has subclasses that inherit variables and methods of the superclass. This is represented by a solid line with a hollow arrowhead from the child to the parent class. The Shape class has subclasses Circle, Rectangle, and Square that inherit the variables and methods of the superclass. Abstraction is used to hide the internal implementation of a class, thus easing the interaction between classes. Abstraction is present in the methods createShapes() and selectionSort() in classes ShapeFactory and SortingTechnique respectively. The methods createShapes() and selectionSort() have a private access modifier so that access is restricted to method calls made within the class.

A picture containing diagram

Description automatically generatedThe alternative UML class diagram of the system is shown below. This alternative design has a discrete Conductor class that coordinates the operations of all other classes. The Conductor class has an association relationship with the ShapeFactory, SortingTechnique, DisplayShapes, and Shape class because instances of these classes are stored in the fields creator, sortShapes, display, and shapeList within the Conductor class. There is no association or composition relationship between the Shape, ShapeFactory, SortingTechnique and DisplayShapes class because instances of the Shape class are not stored in fields of the other classes. This simplifies class interactions because they are all handled by the Conductor class. Abstraction can no longer be used to hide the internal implementation of the methods createShapes() and selectionSort(), since they must return the list of Shape objects directly to the Conductor class. This is necessary because the ShapeFactory and SortingTechnique classes do not have an associative relationship with the Shape class. Thus, they cannot store instances of the Shape class within a field of their class.

The second class diagram yields a more simplistic design that the first class diagram. This is a result of the Conductor class handling all class interactions, thus simplifying the system. In the second class diagram there are no composition relationships meaning objects of each class have distinct lifetimes. Additionally, all association relationships involve the Conductor class. However, as a result of this simplification, the Abstraction design principle can no longer be used to hide the internal implementation of a class.

***PART III: Implementation of the solution***

The algorithm implemented in the SortingTechnique class to sort the Shape objects is the selection sort algorithm. Selection sort is an in-place sorting algorithm that traverses a list (starting from a reference index) to find the smallest element and swap it with the element at the reference index. The reference index starts at zero and moves one element to the right whenever an element is swapped. The reference index acts as a boundary between the sorted list (to the left) and the unsorted list (to the right). The sorting algorithm is finished when the reference index is at the last element in the list.

The first UML class diagram was implemented in Java. The first classes to be implemented were the simplest – the Shape superclass and Circle, Rectangle, and Square subclasses. These classes were written into a single .java file. The next class to be implemented was the ShapeFactory class in order to instantiate objects of the Shape class. The next class to be implemented was the DisplayShapes class to display the instantiated Shape objects on a GUI. Once the Shape objects were displayed correctly on the GUI, the SortingTechnique class was implemented. The load and sort buttons in the GUI were added later. The Comparable interface was implemented in the Shape class after the SortingTechnique class was confirmed to be functioning properly. The SortingTechnique class was modified to use the Comparable interface. The code-and-fix model was used to implement the classes of the UML diagram due to the limited initial planning, immediate development, and troubleshooting errors as they occurred.

The Eclipse IDE version JavaSE-16 was used during the implementation to write and run code.

Below is a snapshot of the execution of code (i.e., of the interface).

|  |  |
| --- | --- |
| Chart, waterfall chart  Description automatically generated | Chart, waterfall chart  Description automatically generated |

Attached is a short video showing how to launch the application and run it.

<https://youtu.be/_FzC18f50k8>

***PART IV: Conclusion***

Elements of this software project that went well were implementing Object Oriented Design principles and design patterns in the system. The inheritance relationship between the Shape class and Circle, Rectangle, and Square subclasses were easily implemented. Encapsulation was easy to implement by setting access modifiers to private and using setters and getters to access the variables. Abstraction was easy to implement by privatizing methods to ensure that they may only be accessed from within their class. The Singleton design pattern was simple to implement by privatizing the constructor and adding a getInstance() method. The challenges of this software project were creating a Graphical User Interface with responsive buttons and displaying the Shape objects when the user required them to be loaded in. Creating the interface was challenging because I did not understand how the paintComponent() method worked, when it was called, or where it was being called from. Additionally, I did not know how the add() method worked, or what a JFrame and JPanel was. I did not familiarize myself with the concepts of a Java GUI before developing the software for this project. This resulted in errors, methods not working as intended, interfaces not responding to repaint() calls, and other obstructions. I have learned that it is best to familiarize myself with a concept before applying that concept in a piece of software. My first recommendation for completing this software project is to learn more about the Graphical User Interfaces in Java. Specifically, JFrames, JPanels, add() method, paintComponent() method, Graphics objects, and the repaint() method. This will simplify what I believe was the hardest part of the software project. My second recommendation would be to have a clear vision as to how your classes should interact. Creating a UML class diagram before coding saves time effort. My second recommendation would be to understand the principles of Object Oriented Design, and the Singleton and Factory Design patterns. Since these are necessary implementations within the software project, it is best to fully understand these concepts before the planning stages.