|  |  |
| --- | --- |
| **Distributed System Programming – Assignment 1** | **SDH3-A**  **Laiba Asif (R00201303)**  **04/11/2023** |

Table of Contents

[Question 1: Prototype Design Pattern 2](#_Toc149734431)

[Code 2](#_Toc149734432)

[Explanation: 2](#_Toc149734433)

[Code Run 4](#_Toc149734434)

[Explanation: 4](#_Toc149734435)

[Question 2: Decorator Design Pattern 5](#_Toc149734436)

[Code 5](#_Toc149734437)

[Explanation: 5](#_Toc149734438)

[Code Run 7](#_Toc149734439)

[Explanation: 7](#_Toc149734440)

[Question 3: Strategy Design Pattern 8](#_Toc149734441)

[Code 8](#_Toc149734442)

[Explanation: 8](#_Toc149734443)

[Code Run 9](#_Toc149734444)

[Explanation: 9](#_Toc149734445)

# Question 1: Prototype Design Pattern

Code A screenshot of a computer program

Description automatically generated A screen shot of a computer program

Description automatically generated

### Explanation:

Prototype Design Pattern is a creational design pattern that creates new objects by cloning existing ones. This pattern uses a prototypical instance to create new objects by copying this prototype. It avoids the cost of creating a new object from scratch by copying or cloning the original object. In this case, I will create an object called 'document' that will be cloned using shallow and full recursive deep (deep copy) method. The object will consist of a list of 2 lists and I will create 4 copies of the object using the Prototype Design Pattern.

Here's the code explanation:

* Importing the `copy` module, provides functions for creating copies of objects.
* A class `Document` is defined. It has the following methods:

1. `\_\_init\_\_(self)`: This is the constructor method that initializes the `data` attribute of the `Document` object with a 2D list containing some values.
2. `clone(self)`: This method returns a shallow copy of the `Document` object using the `copy.copy()` function from the `copy` module.
3. `deep\_clone(self)`: This method returns a deep copy of the `Document` object using the `copy.deepcopy()` function from the `copy` module. Deep copy creates a new object and recursively copies the entire object hierarchy, including all nested objects.
4. `set\_data(self, data)`: This method allows the `data` attribute of the `Document` object to be updated with a new value.
5. `\_\_str\_\_(self)`: This is a special method that returns a string representation of the `Document` object. In this case, it simply returns the string representation of the `data` attribute.

* An instance of the `Document` class called `original\_document` is then created.
* The code then creates 4 copies of the `original\_document` using both shallow (`clone()`) and deep (`deep\_clone()`) copy methods. Each copy is assigned to a separate variable: `copy1`, `copy2`, `copy3`, and `copy4`.
* The `set\_data()` method is called on each copy, passing in a different 2D list as a new value for the `data` attribute.
* Finally, the code prints the contents of the original document and each copy of the document to observe the changes made by the `set\_data()` method.

Code Run A screen shot of a computer

Description automatically generated

### Explanation:

Conclusion: This code demonstrates how to create shallow and deep copies of an object using the `copy` module in Python.

In the above screenshot, The original document is initialized with the data `[[1, 2, 3], [4, 5, 6]]`. copy1 is a shallow copy of the original document, and its data is updated but the original document remains the same and copy2 is a deep copy of the original document, and its data is updated but again the original document remains the same. Therefore, the output shows the state of each copy after the `set\_data()` method is called. The original document remains unchanged, while each copy has its data modified accordingly.

# Question 2: Decorator Design Pattern

Code A screenshot of a computer

Description automatically generated A screen shot of a computer code

Description automatically generated

### Explanation:

The Decorator Design Pattern is a structural design pattern that attaches additional responsibilities to an object dynamically. The pattern is aimed to provide flexibility in adding, removing, or modifying the behavior of an object without affecting the behavior of other objects in the same class. In this case, I will create a custom class called Value that will hold a number. Then, I will add and subtract to a number (Value) using the decorators. The addition and subtraction decorators can accept integers directly, a custom Value object, or other Addition and Subtraction decorators. I will implement the IValue interface in Addition, Subtraction, and Value classes.

Here's the code explanation:

* `from abc import ABC, abstractmethod`: This imports the Abstract Base Class module from the abc package. ABC is a module that allows to create abstract base classes.
* `class I\_Value(ABC):`: This defines an abstract base class called I\_Value. The class is derived from ABC, indicating that it is an abstract base class.
* `@abstractmethod`: This decorator is used to mark the `check\_value` method as an abstract method. Abstract methods are methods that are declared in the base class but have no implementation. They must be implemented by any concrete subclasses.
* `class Value(I\_Value):`: This defines a concrete class called Value, which inherits from the abstract class I\_Value. The class has an initializer method (`\_\_init\_\_`) that takes a value parameter and assigns it to the `\_value` attribute.
* `def check\_value(self):`: This is the implementation of the `check\_value` method in the Value class. It simply returns the value stored in the `\_value` attribute.
* `class Addition(I\_Value):`: This defines another concrete class called Addition, which also inherits from the abstract class I\_Value. The class has an initializer method (`\_\_init\_\_`) that takes a value and an operand parameter and assigns them to the `\_value` and `\_operand` attributes.
* `def check\_value(self):`: This is the implementation of the `check\_value` method in the Addition class. It calls the `check\_value` method of the `\_value` object and adds the `\_operand` to the result.
* `class Subtraction(I\_Value):`: This line defines yet another concrete class called Subtraction, which also inherits from the abstract class I\_Value. The class has an initializer method (`\_\_init\_\_`) that takes a value and an operand parameter and assigns them to the `\_value` and `\_operand` attributes.
* `def check\_value(self):`: This is the implementation of the `check\_value` method in the Subtraction class. It calls the `check\_value` method of the `\_value` object and subtracts the `\_operand` from the result.
* `initial\_value = Value(50)`: This creates an instance of the Value class with an initial value of 50.
* `addition\_decorator = Addition(initial\_value, 20)`: This creates an instance of the Addition class with the `initial\_value` object and an operand of 20.
* `subtraction\_decorator = Subtraction(addition\_decorator, 5)`: This creates an instance of the Subtraction class with the `addition\_decorator` object and an operand of 5.
* `output = subtraction\_decorator.check\_value()`: This calls the `check\_value` method on the `subtraction\_decorator` object and assigns the result to the `output` variable.
* `print(f"The output of the use case diagram is '{output}' ")`: Then this prints out the value of the `output` variable.

Code Run A screen shot of a computer

Description automatically generated

### Explanation:

Conclusion: This code (Decorator Design Pattern) demonstrates a flexible way of modifying the behavior of an object dynamically without affecting the behavior of other objects in the same class. It allows to add or remove functionalities at run-time without changing the class structure.

How the output is achieved:

1. Value object with an initial value of 50
2. Then, Addition decorator by passing the `initial\_value` object and an operand of 20 to the `Addition` class constructor.
3. Next, I Subtraction decorator (`subtraction\_decorator`) by passing the `addition\_decorator` object and an operand of 5 to the `Subtraction` class constructor.
4. The `check\_value()` method is then called on the `subtraction\_decorator` object, which in turn calls the `check\_value()` method of the `addition\_decorator` object, and then finally calculates the sum of the value returned by `check\_value()` and the `\_operand` value (i.e., 20 - 5 = 15).
5. "The output of the use case diagram is '65'".

# Question 3: Strategy Design Pattern

Code A screenshot of a computer program

Description automatically generated

### Explanation:

Strategy Design Pattern is used to define a family of algorithms or behaviors and encapsulate each one of them so that they are interchangeable. In this case, I have 2 strategies: aggressive and passive. I will create a Stock\_Investment class for this purpose. The Stock\_Investment class will have 2 methods: set\_strategy and purchase\_stock. The purchase\_stock method will implement the strategy defined by set\_strategy. The set\_strategy method will set the strategy to be used for purchasing stocks.

Here's the code explanation:

* The `Stock\_Investment` class represents the context that will use different strategies for purchasing stocks. It has an attribute `strategy` that holds the current strategy being used.
* The `set\_strategy` method allows to change the strategy at runtime.
* The `purchase\_stock` method is the main method that uses the current strategy to purchase stocks. It calls the `purchase\_stock` method of the current strategy object and returns the result.
* The `Aggressive` and `Passive` classes represent 2 different strategies for purchasing stocks. Both classes have a `purchase\_stock` method that takes the investment amount and the current stock price as parameters and returns the invested amount and the target price based on the respective strategy.
* In the sample run, `aggressive\_strategy` is initialized as an instance of the `Aggressive` class, and `passive\_strategy` is initialized as an instance of the `Passive` class.
* Then, a `Stock\_Investment` object called `purchaser` is created with the initial strategy set as `aggressive\_strategy`.
* The `purchase\_stock` method is called on the `purchaser` object with an investment amount of 2500 and a current stock price of 70. This calls the `purchase\_stock` method of the `aggressive\_strategy` object and returns the invested amount and the target price. These values are then printed.
* The `set\_strategy` method is called to change the strategy to `passive\_strategy`.
* The `purchase\_stock` method is called again with the same investment amount and current stock price. This time, it calls the `purchase\_stock` method of the `passive\_strategy` object and returns the invested amount and the target price. These values are then printed.

## Code Run

A screen shot of a computer

Description automatically generated

### Explanation:

Conclusion: This code demonstrates how the Strategy design pattern can be used to encapsulate different strategies for purchasing stocks and switch between them at runtime.

In the above screenshot, the first line shows the output when the aggressive strategy is used. The second line shows the output when the passive strategy is used. I can tell from the code run, that the investment and price returned by the Stock\_Investment method depend on the strategy being used.