**Student Assessment Submission and Declaration**

When submitting evidence for assessment, each student must sign a declaration confirming that the work is their own.

|  |  |  |  |
| --- | --- | --- | --- |
| Student name: Rashed Hassan | | Assessor name: | |
| Issue date (1St Submission): | Submission date (1St Submission): | | Submitted on: |
| In case of resubmission | | | |
| Issue date (2nd Submission): | Submission date (2nd Submission): | | Submitted on: |
|  | | | |
| Assignment number and title: 2: Design selection and implementation | | | |

**Plagiarism**

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

**Student Declaration**

|  |
| --- |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.  Student signature: Rashed Hassan Qahah Date: 31\12\2022 |

**Task 1**

Based on the given scenario, answer the following questions:

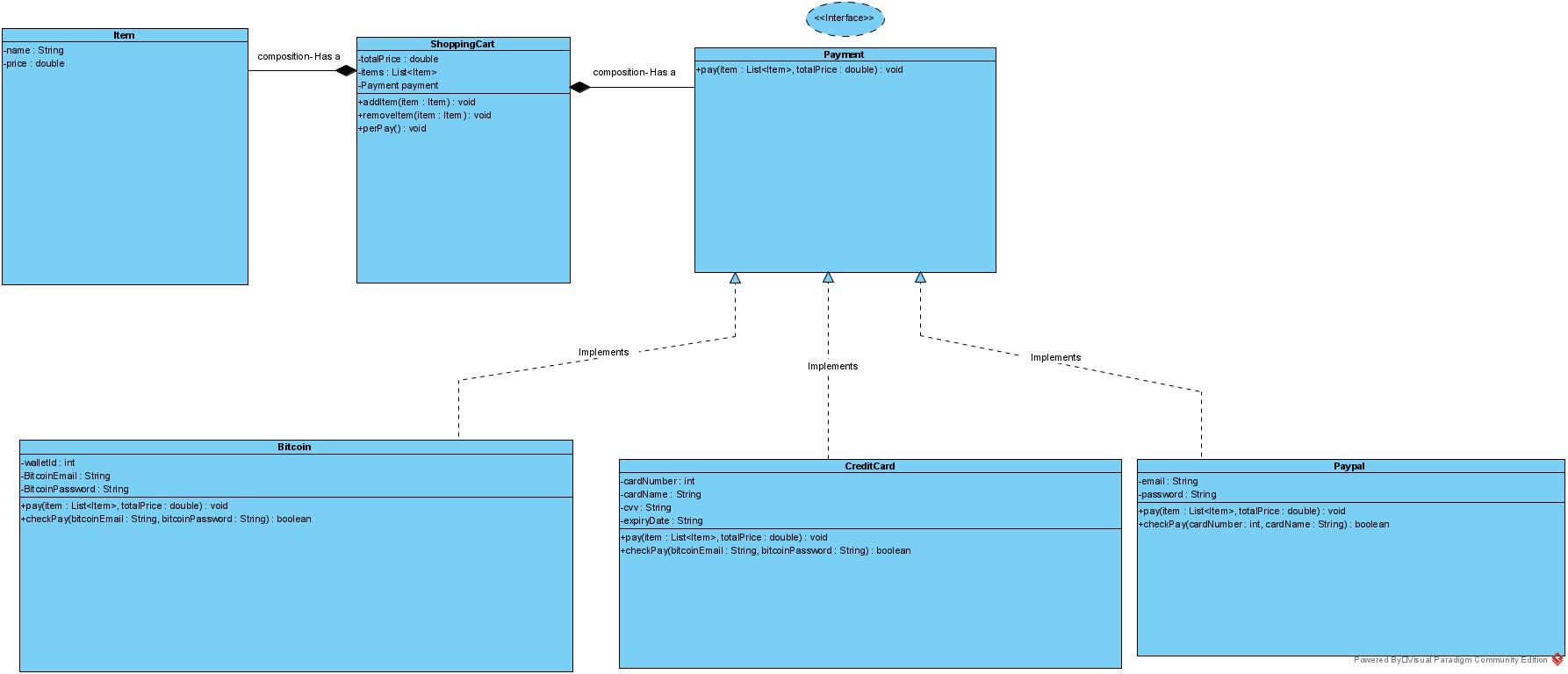
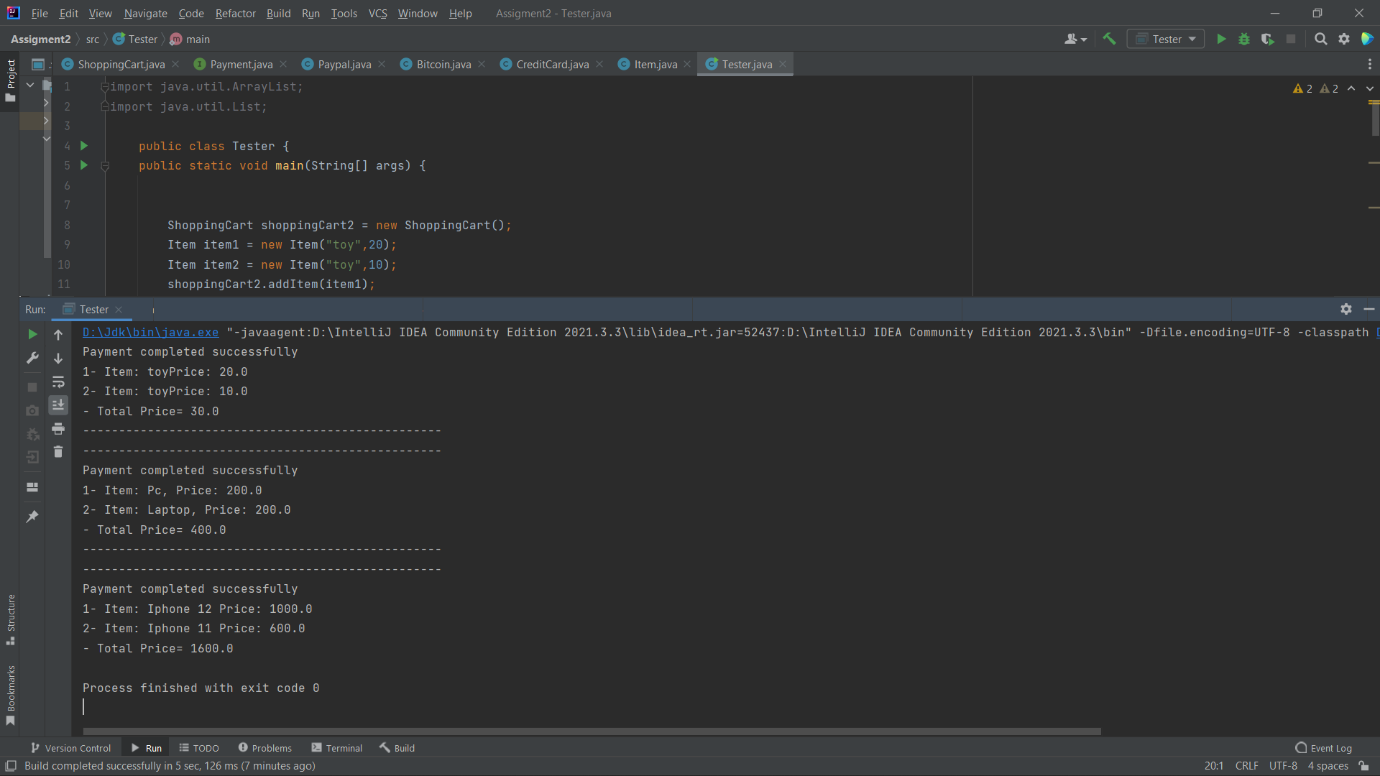
1. Assess a design pattern that can be used to implement this project, explain your choice by examining the strengths and weaknesses.

**Answer:**

If you choose the Strategy design pattern because in my view it is the design pattern that best suits scenario and because it is also A strategy is a behavioural design pattern that transforms behaviours into objects and makes them interchangeable within the original context. It is used when we have multiple algorithms for a specific purpose and the client takes the actual implementation to use it in operation. In other words, you can define a set of algorithms and put each of them into a separate class and makes its objects interchangeable instead of executing a single algorithm and receives instructions during runtime directly regarding which of the set of algorithms should be used, so the behaviour of the class or its algorithm can be changed during runtime, and this type of design pattern comes under the design pattern behaviourist.

Also, the main strengths are that it helps to swap operation behaviour at run time, and each strategy follows one responsibility principle, also represented by the principle of open / closed - the introduction of new strategies that have been purchased and that have an impact on the customer, it allows separation of interests or strategies between components that are different in the system, if the context object does not need that The strategy object uses it to know the implementation details of the strategy object, which allows a high level of modularity and flexibility in the design and implementation of the system.

Also, however, one of the weaknesses in the strategy design pattern is that it can increase the number of classes that are in the system so that we must implement each strategy as a separate class, which can lead to a more complex design and more difficult to understand and maintain the system. The number of strategies has increased dramatically, and it is difficult for us to manage and maintain it.

1. Specify class diagram for your suggested design by drawing an appropriate UML.
2. Evolve the UML from the previous question by implementing it using an appropriate programming language.

**Task 2**

1. Consider three real life examples each represent a design pattern from the three categories which are Creational, Behavioural and Structural.

**Answer:**

* First, creational design patterns.
* Singleton pattern.

Suppose, for example, that we have a ship and the ship has more than one captain, captain 1 and Captain 2, and each ship, since it has captains, must have sailors who carry out the captains' orders.

Suppose that the first captain tells the sailors to move the ship towards the north, and then the second captain who is in the same ship comes and tells the sailors to move the ship towards the south, and here we will face the problem. From the first and second captains, do the sailors carry out the orders of the first captain or the second captain? And then suppose that the sailors worked and carried out the orders of the first captain, then what would they say to the second captain? Here, it is very clear that there is a problem and a mistake.

The solution here must be one captain in the ship, so that the sailors carry out orders through the captain, and that there should not be another captain.

Supposedly, in some cases that we need, there must be one place that issues commands. Suppose that the captain is a class. I only need one instance with my code, and here comes the idea of a singleton pattern. We want to create a class, then we create one object for this class only, and we cannot After that create another object of the same class, this applies to the most used design pattern which is the singleton pattern.

* Secondly, Behavioural patterns.
* Observer patterns.

Let's assume that we have a student and we also have two platforms, the educational platform and the training platform that offer courses. In the normal situation, courses may or may not be available for both platforms, and here the student searches for these platforms so that he checks if they have downloaded the courses they have or not, so using observer patterns we do on the contrary, so that if the platforms download or provide courses, we will send a notification to the student that the educational or training platform has downloaded or provided the courses.

In short, if any change occurs, such as downloading or providing courses from the two platforms, a notification will go to the student.

So, the observer clause is defined as one-to-many relationships. You have a single object that a group of people depend on. If the state of this dominant being changes, then all the people associated with that being know that its state has changed. This object when its state changes must be required to wake up all people, which is the idea of the observer pattern.

* Third, structural design patterns
* Decorator Pattern.

For example, we have a person named Ali who wants to eat noodles and go to a restaurant. The restaurant naturally gives Ali the menu in the restaurant, then Ali looks at the menu and chooses the meal he wants, but Ali wants to order the meal the way he wants, but the problem of the restaurant does not deal with the type of orders that come with the special or customized order, that is, the restaurant does not order A custom request, as some restaurants have a problem, that is, how can it be a bill or cost after adding the additions to the meal, that is, the restaurant offers basic or main meals and the customer must choose one of them and there is no custom request in the system they have. The system in the restaurant does not provide or there is no custom ordering option, and this is a problem:

If the problem is solved in agreement with the restaurant with a programmer named Rashid, the programmer will solve the problem so that there is a custom request in the restaurant system, so when I have something and I want to decorate the object with the additions that the user wants, until the appropriate structure is added to our code, that is, if I have Basic pasta, which is the ingredient, and I want to decorate it or add it to basic pasta, for example:

1- Adding cheese = $1

2- Add chicken = $3

3 - add sauce = 2 dollars

Basic pasta = $12

So, the idea of a decorator pattern is that it inherits properties of a class and then adds other actions to it without changing anything in the base class. Therefore, it is a structural pattern that allows us to add additional functionality to an object dynamically and is also used to change the functionality of an object during operation. Other examples of the same class will not be affected, and because it is a structural pattern if it falls under Structural Design Patterns.

1. Based on your examples, investigate the connection between Object Oriented paradigm and design patterns.

**Answer:**

Based on the examples we have illustrated, the relationship between the object-oriented paradigm and design patterns:  
The object-oriented paradigm is a programming methodology that organizes code between objects, relationships between objects, and design patterns. Design patterns are designed to provide or suggest solutions to common problems or errors that arise when designing and implementing object-oriented programs. Design patterns build on object-oriented programming principles such as polymorphism, encapsulation, and inheritance, and also provide a way to structure code or information in a way that is maintainable, scalable, and easy to understand. Using design patterns, developers or programmers can design and implement programs in a way that follows best practices and takes advantage of the strengths of the object-oriented model. Objects and design patterns have proven successful in creating many types of objects to solve scenario problems in specific programs.

**Task 3**

1. Based on the previous scenario, what is the suitable design pattern to be used in designing this project and how it will fulfils the scenario‘s requirements?

In my opinion, the appropriate design pattern is the decorator design pattern because it is the appropriate choice that meets the requirements, and because the idea of the decorator pattern is that it inherits the properties of a specific class and then adds other procedures to it without changing anything in the base class, so it is a structural pattern that allows us to add Additional functions to an object dynamically and also used to change the functionality of the object during operation, other examples of the same class will not be affected, and because it is a structural pattern if it falls under the structural design patterns, then we can define the base car class that is represented in the various types of cars available for rent that the scenario put forward Which are SUVs, sedans, sports cars, then we can create a separate category to decorate or add specific upgrades like tinted windows and stereo systems, for example we can define a category

The VehicleDecorator class has a getDescription() method to include a description of what stereo systems will be upgraded to the StereoSystemsDecorator class implements VehicleDecorator because it is an interface and applies to tinted windows. Also, as mentioned in the scenario, there is a requirement that no client be able to log in from different devices at the same time, and that the system must have only one administrator, so we can use Singleton design. pattern because it allows or guarantees that only one instance of the class is created i.e., one object is created for that class only and we can't then create another object of the same class, which allows you to enforce the rule that the client can only be logged in from one machine at a time Also, we can use the Singleton design pattern to ensure that there is only one administrator in the system by creating an Admin class that is implemented as Singleton.

1. Critically judge your decision by drawing a comparison between the design patterns and give a valid example for each design?

**Answer:**

So, after choosing the appropriate design pattern for a scenario, which is the Singleton design pattern and the decorator design pattern, I found that they are the most appropriate to use to achieve their requirements based on the requirements of the scenario and as I mentioned. The previous question, based on the requirements, I chose the decorator and Singleton, which was the idea of ​​the decorator pattern is that it inherits the characteristics of a specific class and then It adds other actions to it without changing anything in the base class, so it's a structural pattern that allows us to add additional functionality to an object dynamically and also used to change the functionality of the object on run, other examples of the same class won't be affected, and because it's a structural pattern if it falls under structural design patterns Then we can define a base car category that is represented in the different types of cars available for rent that the scenario put forward which are SUVs, sedans, sports cars, then we can create a separate category to decorate or add certain upgrades such as tinted windows and stereo systems, for example We can define a category the VehicleDecorator class has a getDescription() method to include a description of the stereo systems to which it will be upgraded.

The StereoSystemsDecorator class implements VehicleDecorator because it is an interface and applies to tinted windows.

So, it was also, as mentioned in the scenario, there is a requirement that no client can log in from different machines at the same time, and the system should only have one administrator, so we can use the Singleton design. Pattern because it allows or guarantees that only one instance of the class is created, i.e., one object is created only for that class and we can then no longer instantiate another object of the same class, which allows you to enforce the rule that a client can only be logged in from one machine in Also each time, we can use the Singleton design pattern to ensure that there is only one administrator in the system by creating an Admin class that is implemented as Singleton.

Also, in the decorator style, if I don't use it in the car rental system, I won't be able to add upgrades or dynamically add new behavior to existing objects such as cars by wrapping them in decorator objects, which will make it more difficult to add new features or upgrades to the system. As this will require modifying the code of the basic objects directly, which may lead to the system not working as intended.

If I do not use the Singleton design pattern in the car rental system, it does not guarantee that there is only one instance of the Administrator class, as I mentioned earlier, and it is also possible or possible to create multiple instances of the Administrator class, and this may cause many problems with the system that wants it, as mentioned, the scenario must That there is only one administrator, which may lead to the system not working as intended.

Therefore, it is important that we use design patterns in the code when it is appropriate to the requirements of the scenario so that it can help us write code that is more organized, maintainable, and also able to expand and avoid errors. Using the wrong design pattern or using unnecessary or useless design patterns exposes us to lead to complex and unnecessary code.

Other design patterns that can be used in scenario include a car rental system, for example, the factory design pattern is a pattern that is often used in software design and adds a method that enables us to instantiate an object without specifying which class we want to use, which means it allows the factory method in the class to defer instantiation of one One or more concrete subclasses that can be used to create different types of cars.

Including the observer pattern and his idea that it is one-many relationships. You have a single object that a group of people depend on. If the state of this dominant being changes, then all the people associated with that being know that its state has changed. Should this object change its condition, it is required to alert all persons, and may be used to notify customers of changes to their tenancies.

If the Factory pattern or Observer pattern is not used in the car rental system, this may lead to the system being less flexible, and without using the Factory pattern, it may be difficult to add new types of cars to the pattern without modifying the current code, and also without the Observer pattern, the administrator may not have A way to notify it of changes to the registry status of a client, which makes it difficult to administer the system effectively.