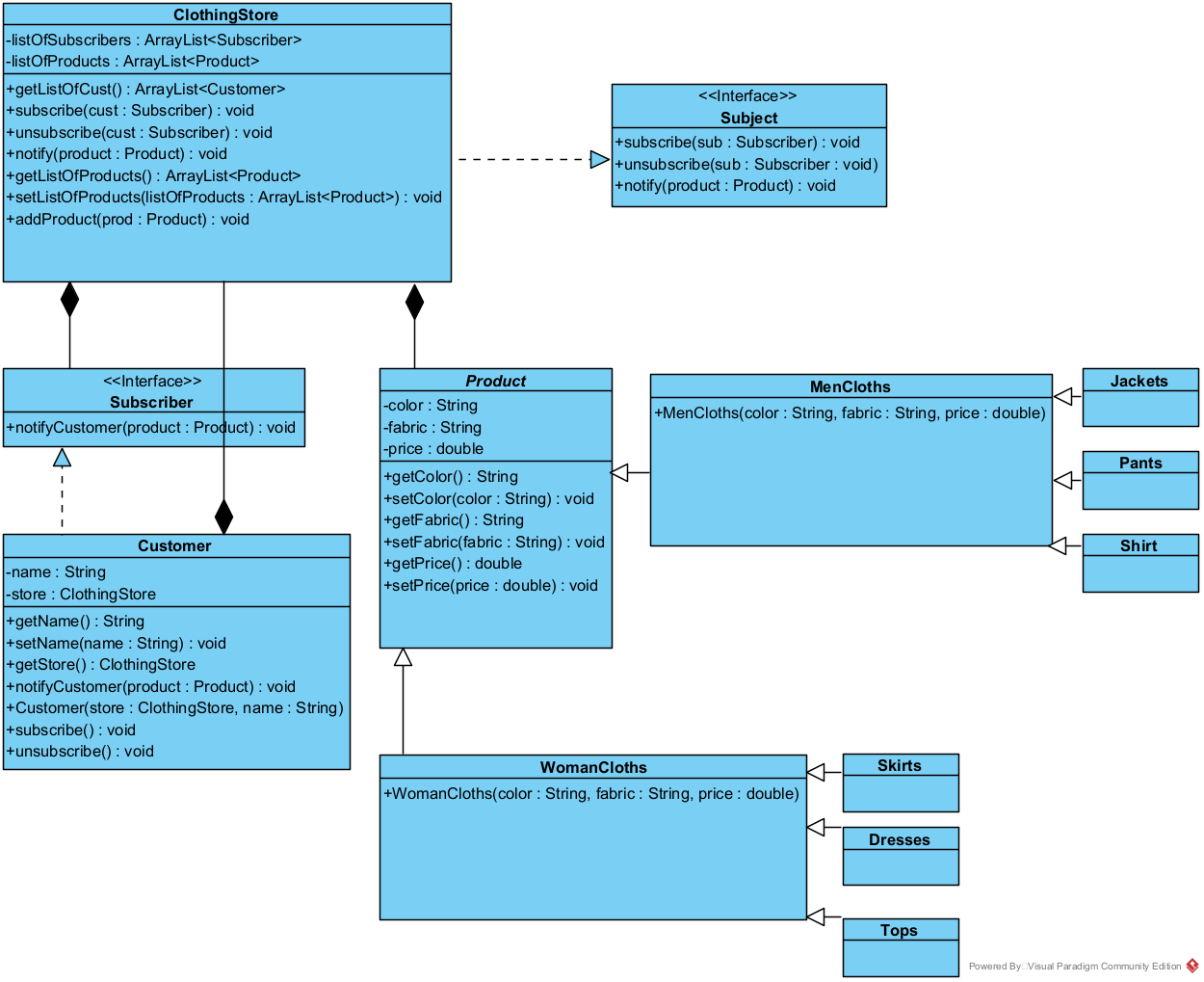
**Student Assessment Submission and Declaration**

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

|  |  |  |  |
| --- | --- | --- | --- |
| Student name: Abdelrahman Saleh | | Assessor name:  **Abdelbaset Assaf, Hazem Al-Najjar** | |
| Issue date (1St Submission):  **09/05/2023** | Submission date (1St Submission):  **27/05/2023** | | Submitted on:  **27/05/2023** |
| In case of resubmission | | | |
| Issue date (2nd Submission): | Submission date (2nd Submission): | | Submitted on: |
| Programme: Higher National Diploma in computing – Software Engineering, Intelligent Systems | | | |
|  | | | |
| Assignment number and title: 2: Design patterns usage and implementation | | | |

**Task 1)**

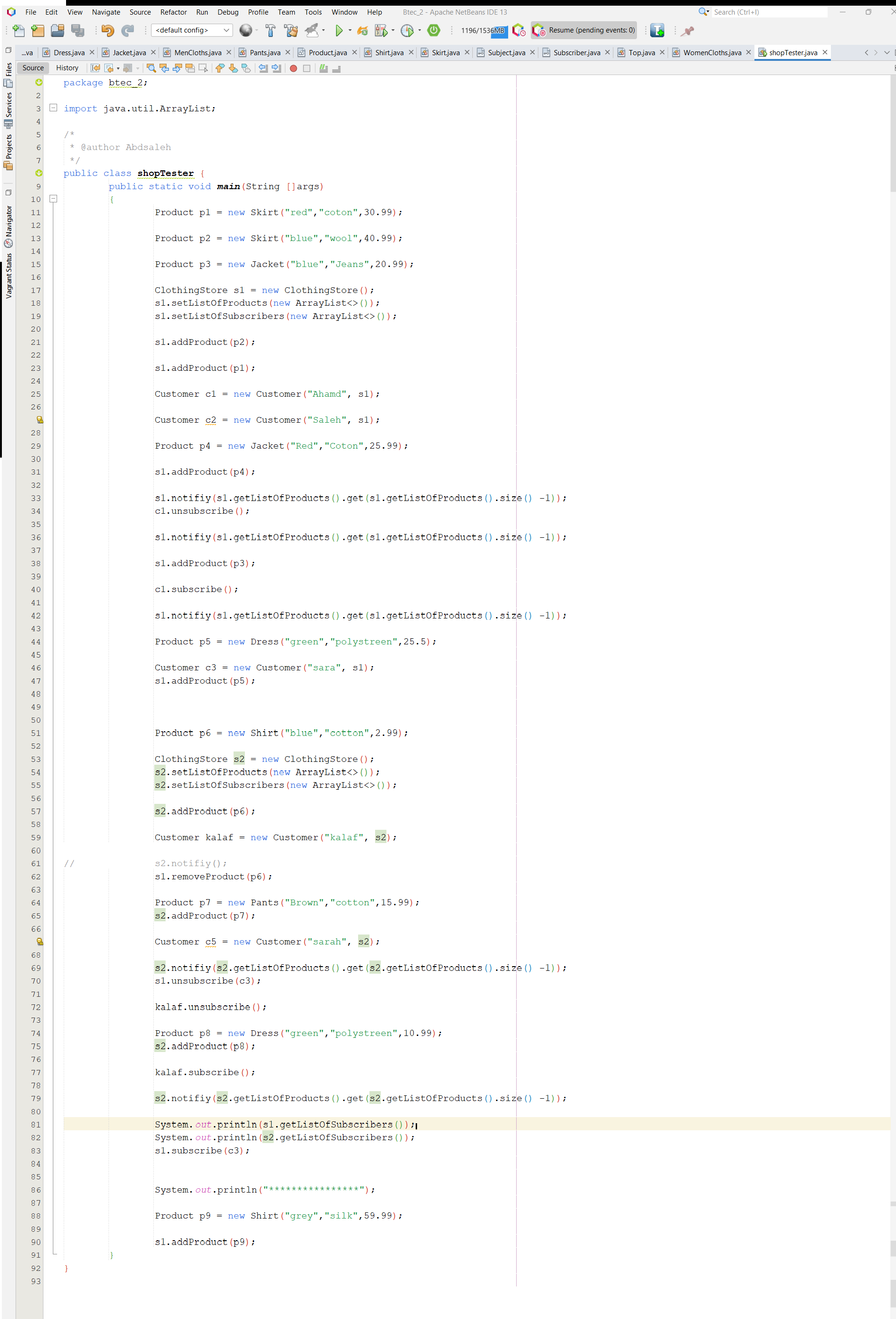
1. By reading the scenario I concluded that I need to use the observer design pattern, why do I concluded that? Well you see in the scenario we have multiple signs and cases that fits perfectly for this pattern, like having a group of customers subscribed to the store social page for example and whenever a new product is released we notify all the customers that are subscribed or interested in that product that a new product is released so that they can check it, and not to mention that this pattern provides us with high flexibility dealing with new product that we may add/remove to/from stock in future and its really flexible in adding or removing members out and into the notification list, doing so this pattern gives us a loose coupling approach that gives us high flexibility and usability in our code and as I mentioned it gives an easy way to extend the project easily without doing a big changes to the code, on the other hand It has some disadvantages too like it may be harder to debug due to the multiple observers involved and the layers of classes for each product that we may create throughout the process which may overwhelm the developer, and it may encounter some performance issues like if we have a huge customers(observers) base it can be overwhelming for the system to notify each and every one of them not to mention that it may take along time and a lot of resources out of the system that eventually may lead the system to crash.

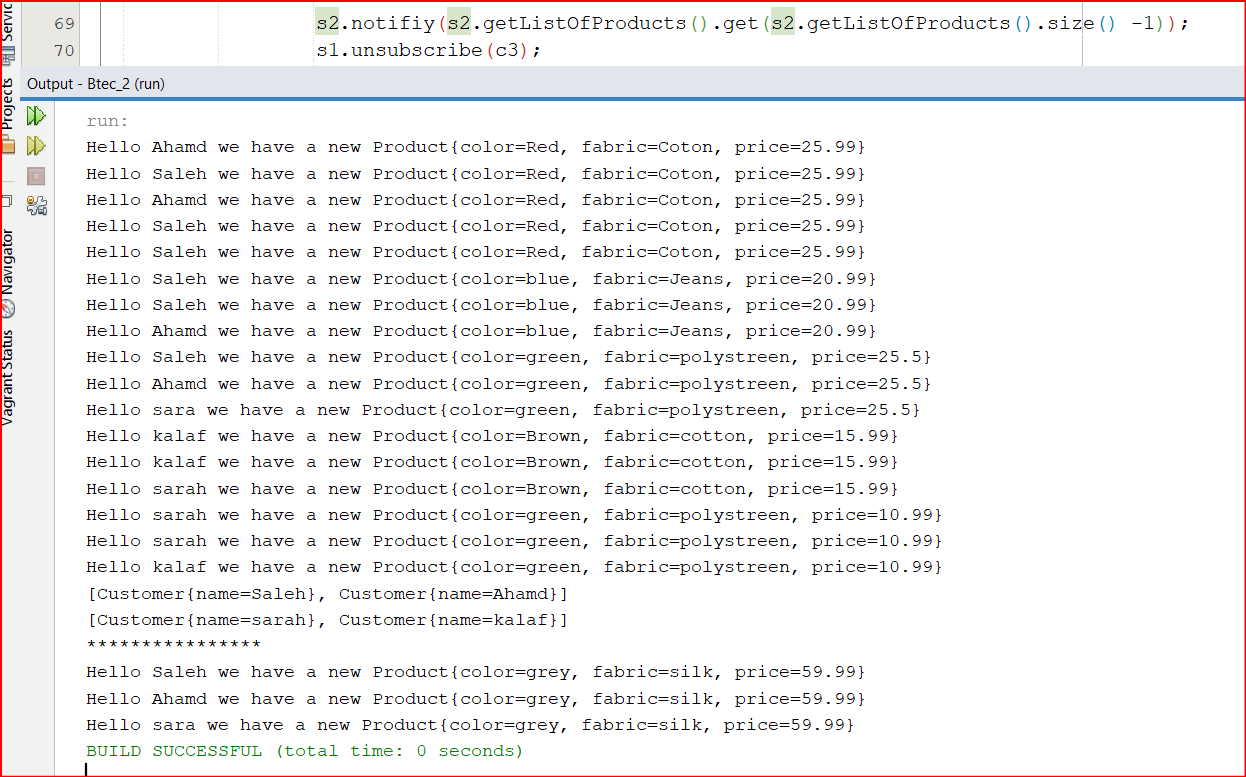


**III.**

**The source code has been attached.**

**And here is a screenshot to the testing class along with the names of all the classes that I’ve written.**

****

**Here is the output for the test class.**

**Task 2) Examples on patterns:-**

**1.**

**-> Creational(singleton) :- well as a real life example for this pattern I would use my younger brother as an example to illustrate my point, my younger brother Ibrahim uses the tawjihi online classes on jo academy and one of the things that drew my attention was that he cant log in to his account from another device not to mention that he can’t of course log in to his account again while his account is open, so i think that they are using singleton in the source code of their app to prevent multiple instantiation of the user’s account, so they can prevent account sharing among student and content leaks without people actually paying for the provided content, and they are restricting the opening of accounts by saving the mac address I think of the device that you first sign in to, then save the address in a static variable and every time you log in the instance of your account that you first created will still have that data in it then it will check if the new log in information is the same as the first instance that you created, if so it will grant access, otherwise it will block your account.**

public static CustomerAccount getInstance() {

if (instance == null) {

instance = new CustomerAccount();

}

return instance;

{

**Like here in this example the get instance method will check if the static variable instance is null or not (has been instantiated yet or not) if the variable(which we save the constructors data in it) is null we create a new one or initialize it but if its already initialized it will return the already existing value.**

* Behavioural design pattern(observer):- The first thing that comes to mind when I say observer pattern is youtube, youtube in my opinion is the most prefect example for describing this pattern, it has all the features resembled in it, from the channel it self to the observer and the videos are the products that the channel notify its users that a new one has been uploaded to the channel, the pattern itself serves these exact features, which are the platform, the observers then the product itself, youtube notify the subscribers of the channel but does not notify other subscribers so each channel has its own list of subscribers which it notify whenever a new video has been uploaded, and each observer can subscribe and unsubscribe at any moment and he will stop receiving new notification if unsubscribed.
* Structural pattern(Decorator):- For the decorator pattern we have a ton of examples that we may look at, we have games, food and drinks, each one of them has a base or something that is the foundation , then comes the addition and extras that we may put or remove as we want, like simply it is like playing with a character in a game, that character has special characteristics like eye color, hair color, female or male and things like that, whatever you choose the base will remain the same which is the standard character but when you modify or add the new look the character gets a plus addition on their original look, let me give another example to be more clear imagine that you want to order a coffee from a coffee place you have a lot of option to pick from, you have coffee, latte , milk or special blends for example now these option are the base for any drink that we may order from that coffee place, we can have the drink without any additions but we can ask for extras like putting some nuts on it or add milk to the coffee or creame or whatever, every addition to the original drink acts as decoration to the original drink and gets blended or added to the original drink so we can imagine it like having nested loops (or at least I imagine it this way) we have the main loop then we can add as many loops as we want and every one will be performed and the result will accumulate on the original value and then result in a mesh of all the things that we crossed.

**2. The Connection between OOP paradigm and design patterns.**

**In my opinion the way we should approach this is to explain briefly what each on is to see the difference then make a connection, OOP is a way to provide a foundation for designing and implementing systems by using the commonly known main principles of it which are encapsulation, inheritance, composition and polymorphism to organize and structure the code, we use OOP which basically focuses on objects to solve problems and work our way to build systems, on the other hand the design patterns are the use of these concepts in an order fashion to solve a common problem or repeating one, every design pattern have its unique way to operate its like an algorithm to approach a certain problem that we face so we make the utmost use of the OOP concepts to come up with design patterns to solve the problem at hand, so the design patterns are proven solutions to recurring design problems, all these points gives us a strong connection between the two when we view them as each of them mutually beneficial to the other like the OOP without the design patterns will face a lot of problems that may hinder its functionality and use as well as the design pattern could not even exist without the concepts of OOP so it some how build on and have OOP as its core foundation and provides us with advanced and useful solutions.**

**Task 3)**

**I.**

* The first scenario: - This scenario’s main points are: supporting product variations ,customization options and defining a family of algorithms that can be applied to different product attributes, these characteristics apply perfectly to the strategy pattern, this pattern fulfils the requirements by representing each behaviour in a separate class which this pattern represents it as a family of algorithms where each algorithm represents a specific product attribute variation such as size or color that are mentioned in the scenario, also this pattern fulfils the requirements by providing high customizable options that the customer can select from for each product attribute.
* The second scenario: - The main points that we have to highlight in this scenario are: single instantiation, reducing memory usage and save frequently used objects in memory , these are the main points in the given scenario, so we can conclude that the flywheight pattern is a great fit for this scenario as it fulfils these exact requirements, the flywheight pattern prevent multiple instantiation of a class so it ensures that we have a single variation instance of it and it reduces the memory usage, not to mention that flyweight operates by storing the instances that we created in a hash map so we can access them easily via their respective keys and ensure that we don’t create other instances of that object with the same values so it is basically preventing duplication, so we conclude that this pattern meets the requirements needed in this scenario.
* The third Scenario: - The main points In this scenario are: shared intrinsic properties, storing these interstices in a separate object to reduce memory usage and object sharing across multiple product instances, to fulfil these requirements the perfect design pattern is flyweight pattern which provide these properties and ensure the flexibility of the codebase, flyweight pattern does this by restricting making new instances of the same data to reduce memory usage and the instance created could be shared among multiple classes to be used without the need to create new instances to use in each class, this pattern does this by using a hash map to identify the key of each instance and restricting duplication so we can use the object multiple times in various classes.

**II.**

1. The strategy pattern is very helpful in this scenario as I mentioned earlier but what would happen if we didn’t use this pattern, the strategy pattern works on one instance on run time and make its behavior changeable depending on different implementation in various classes but the same interface will stay in use regardless of how many behaviors that we have, for example if we are going to look at Looka’s example we had a fighter class that Looka extended then we had an interface for each behavior like kicking and punching those interfaces had subclasses of different variations of that specific behavior like an upper cut punch or a straight punch same thing for kicking, this mechanism of working provides us with high flexibility during run time but the template pattern works in a slightly different way it works by doing this changes on design time and have different implementations for each class, it does this by pulling a totally different class and using its implementation and have an instance of that class not like the strategy which use single instance and work on it, the Template pattern does not provide the level of encapsulation that the strategy have for different algorithms in separate classes, the strategy allows switching of strategies in a more dynamic manner by using one instance and changing its behavior.

[(90) Design Patterns: Template Method - YouTube](https://www.youtube.com/watch?v=MfAvs0n9uMs&t=694s) (more info on the topic)

1. The flyweight pattern works by restricting the creation of a similar variation instance of a given class and it preserves some of the variables within that class to be shared across other classes (called intrinsic variables), as I mentioned earlier in the previous question it is a perfect fit for the scenario that we had, but if we didn’t want to use this pattern we can use a similar pattern like the object pool pattern this pattern is a bit weird I think it works by having a fixed number of instances so it essentially restricts the creation of new instances just like fly weight but the goal is different, the object pool does this to prevent the additional instances beyond the predefined pool, it uses those instances from the pool as they are already declared ,initialized with the necessary resources to be in a usable state then stored in the pool to be used when needed, and every change or alteration made to any instance will have an impact to the original instance as the variable are declared as static so it provides almost the same functionality as fly weight but as we saw the way they operate are totally different with how they declare and handle instances, one of the main differences between the two is the object pool pattern while indeed restrict the creation of a new instance but the at the same time it calls the already existing instance and use it whenever needed while the fly weight operates some how similar but it works in checking if the instances is present in our hash map or not, if it is no present it initialized it but if it exist it will return the already existing instances in the hash map and store its data in the new variable that you declared, whereas the object pool multiple users can simultaneously use different instances from the pool so it emphasize reusability of an existing instance to optimize resource usage and improve performance.
2. Well as you can see we can the same pattern that we used in the previous point in this point too, we can use the object pool pattern as an alternative of flyweight patter as it restrict the creation of new instances hence restricting duplication of an instance and it can allow the user to use the same instance multiple times and each time have different values for that instance but no at the same time, while the flyweight allows the creation of multiple instances at the same time but strongly focuses on not to have another object that have the same key as another instance while the object pool allows the user to use an instance and assign new value for it but these updates will take place for all the user that uses the same instance as well so we cant use two instances with two different values at the same time in object pool.

**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: Abdelrahman Saleh Date:27/05/2023 |