

**Advanced Programming**

9/6/2024

Amr Adi

**Table of Contents**

[Task 1: UML class diagram 2](#_Toc210312654)

[Task 3: 3](#_Toc210312655)

[Task 4: characteristics of Object-Oriented Design and class relationships 4](#_Toc210312656)

[Task 5: the design Pattern 5](#_Toc210312657)

[Task 6 6](#_Toc210312658)

[Task 9: 7](#_Toc210312659)

[Task 10: Design patterns 8](#_Toc210312660)

[Task 11 10](#_Toc210312661)

[1- The constructor of a class has many parameters 10](#_Toc210312662)

[2- object is created and used in many different location 10](#_Toc210312663)

[3- a large object that should be loaded on demand and avoids duplication of the same object 10](#_Toc210312664)

[4- Choosing payment method at runtime in a shopping site 10](#_Toc210312665)

[Task 12: 10](#_Toc210312666)

[References 11](#_Toc210312667)

## Task 1: UML class diagram

UML

A grid of lines with text

Description automatically generated with medium confidence

## Task 3:

scenario and the UML design

* Entities (Department, Employee, StudentCounseling, StudentPerformance): These classes act as the main data structures in the system to store the fundamental information. It stores information about various departments, employees, the students’ guidance & counseling, and their performances. The classes define attributes and instance methods for the data contained within each one.
* Factories (DepartmentFactory, EmployeeFactory, StudentCounselingFactory, StudentPerformanceFactory): These classes are chargeable for generating the instances of the relevant entities. Therefore, by dividing the creation of objects to a set of factories, the system follows the Factory Method approach. This pattern carries on the sequence that the creator has a relatively close tie with the product, which makes extending or revising it possible.
* Observer Pattern (Observer): Observer Pattern is used to inform all the observers about change in departments. The DepartmentSubject is also a subject that has a list of observers and sends out a message to all the observers note that some observers of this class are members of all the classes that implement this subject, when the department is updated. This pattern enables the observer subject to maintain multiple observers in a one-to-many relationship hence they can all update on changes in the same subject without necessarily being much coupled with it.
* Singleton Pattern (DataStorage): The DataStorage class implements the Singleton design pattern, so the only exists one instance of storage across the application. This pattern is used where there should naturally be only one way through which it could be accessed; e. g. a shared resource or data.

## Task 4: characteristics of Object-Oriented Design and class relationships

characteristics of Object-Oriented Design.

* **Encapsulation**: This is a way of associating data with the procedures acting upon that data into one object known as a class. Encapsulation assists in concealing object’s state and only providing access to services in strategies of methods.

Example in my code:

Classes having encapsulated data (attributes) and methods, such as Department, Employee, StudentCounseling, and StudentPerformance, are examples of encapsulation in action.

* **Abstraction**: Abstraction aims at the ability of concealing essential inner mechanisms working, and only revealing its core characteristics. This is done through use of abstract classes and interfaces as will be explained later in the paper.

Example in my code:

The Data Loader interface allows various data types (department, employee) to build their own data loading mechanism by abstracting the process of loading data from various sources.

* **Inheritance**: It is a feature that enables one class, the subclass to reuse, extend and sometimes modify components contained in another class, the superclass. It makes code more reusable and creates a superclass relationship between classes.

Example in my code:

The offered code sample does not explicitly illustrate inheritance. To prevent code duplication, inheritance may be taken into consideration if entities contain similar properties or methods (Employee and StudentCounseling both have a dob attribute).

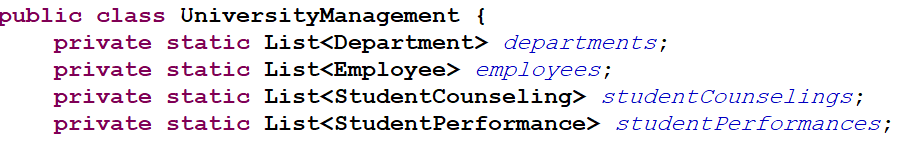
* **Polymorphism**: polymorphism one object of one class is replaceable by an object of another class derived from it. This provides simple and heterogeneous nature of coding methods for flexibility and more space for extension.

Example in my code:

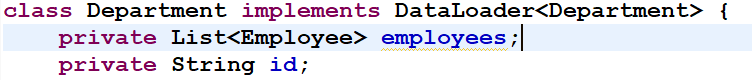
The offered code sample does not directly show polymorphism. However, polymorphism might be used through method overriding if there are actions that can be carried out on distinct sorts of objects (computing performance measures for both students and workers).

Class Relationships:

* **Association**: An association is a connection in which items of two different classes are associated to each other. There can be one-to-one, one-to-many, or many-to-many relationships in this one.

****

* **Aggregation**: When two classes aggregate, it signifies a "has-a" connection in which one class is a subset of the other. The child class might exist separately from the parent class in this weaker type of relationship.



## Task 5: the design Pattern

1- **Creational**: ﻿Creational Design Patterns attention on powerful methods to create objects at the same time as making sure flexibility and performance inside the instantiation process. The Singleton Pattern, for example, guarantees that a class has best one example globally accessible, making it perfect for situations which include coping with logging instances or database connections. On the alternative hand, the Factory Method Pattern permits subclasses to decide the precise magnificence instantiation, providing flexibility in object creation. It's broadly used in eventualities just like the Java Calendar elegance, in which the getInstance() approach returns an item based at the contemporary locale. Abstract Factory Pattern takes this a step in addition by means of imparting an interface for developing families of associated gadgets with out specifying their concrete lessons. For example, in Java's DocumentBuilder, extraordinary varieties of Document gadgets like HTMLDocument and XMLDocument can be produced with out exposing their implementations. These creational styles useful resource in keeping a clean and adaptable object introduction technique, critical for scalable and maintainable software structures.

2- Structural: Structural; Structural Design Patterns are used to design the composition of objects into larger structures, such Applying this pattern methods to achieve the code organization and reusability. Carrier Pattern : to make an classes with incompatible interface work together. Converting one interface of a class into another interface expect by the client. A notable example is in Java's Arrays: x, y post as parameters for accessing asList method (which makes an array appear as a list. The Decorator Pattern (GOF) makes it possible to individually add new methods and properties to objects, extending them without subclassing them. Decorators (decorator for bigger input) With the Composite Pattern, you can compose objects into tree structures to represent part-whole hierarchies in your models, where individual (Atomic) objects can be treated in the same way as Composition objects. The JPanel from Swing is an example of an instance of this pattern, existing as a container whose children are other components, and these children too can have more parents, so a hierarchical organization forms.

3- Behavioral: Each of the behavioral design patterns tackles the question of how objects can work together for the benefit of the entire system and its users; they are primarily concerned with object communication. Observer- The Observer pattern is used for establishing one-to-many dependency between objects where a change in one object will automatically provide notification and updates all its dependants Event handling in Java, The Observer and Observable classes in Java, uses the listener facility for Swing GUI components. Strategy Pattern: Strategy pattern defines a family of algorithms, encapsulates each one, and make them interchangeable. This then permits the algorithms to evolve independently of the clients that use them, as with Java's Collections. Calling sort() method sort an array in different way The object-oriented approach to wrapping a request is the key to the Command Pattern, it allows clients to be parameterized with queues, requests, and operations, and it supports undoable operations. For example, GUI frameworks like Swing handle cut, copy and paste operations through commands, so using commands as first-class objects means you can just encapsulate something with an action in a command object and register in a common place to manage actions. Behavioral patterns are there for communicating the objects and also help us in the arranging the object communication.

## Task 6

1**- Enhanced Code Maintainability:** These patterns, often driven by the Observer, Strategy and Decorator provide a framework that enable our code to be more readable, modular and scalable. One example is the Observer pattern which is applied, among other things, in event handling systems where you can register new observers without touching any existing code. Which helps to maintain and update the codebase easily and make it more Angular to understand for fresher so the learning curve is reduced as well as the mistakes are becomes less.

2- **Improved Code Reusability:** Design patterns are concrete, well-defined answers to specific project headaches that assist makers to produce the software. The Factory Method pattern lets a class defer the instantiation to subclasses and create objects in a way that does not require knowing the class still and, like in the example, can be reused and not more coupling-made when the logic to create an object changes from time to time. By preventing code repetition, making it readable and usable with other parts encourages code reusability, which saves important development time and effort.

3- **Reduction of Code Complexity:** Design pattern is useful to solve the problem which is complex and also to make a complicated problem to be understandable by breaking the problem into its smaller part. One of them is the Composite pattern, which lets you treat individual objects and compositions of objects uniformly and tree structures formed by this. You can see an example of this pattern in file directories or GUI components, where every element of the tree can be a composition of others, but with the Code API, it also allows you forming only one tree. Code Complexity - As most of the patterns are design simplifies the design, it helps for implementing and understanding complex functionalities.

## Task 9:

1- Factory

* **Evidence from my code**: I have added the factory classes discovered in the documentation as DepartmentFactory, EmployeeFactory, StudentCounselingFactory, and StudentPerformanceFactory. These factories capture the business rules for the creation of objects such as department, employees, student counseling records and student performance records.
* **Impact**: As a direct outcome of utilising the Factory Method, the code that I have written becomes more flexible and easier to expand. If I require to alter how instances of entities are generated or even introduce new es, then I can update or add on to the factory for the specific class without modifying the client code. This is beneficial in that it facilitates the Open/Closed Principle whereby classes must be open for further enhancement but closed to alteration.

2- Observer

* **Evidence from my code**: with the Observer pattern, Regular classes are Observer, Subject and also DepartmentSubject. DepartmentSubject class holds reference of observers (listeners) and broadcast the event.
* **Impact**: Observer pattern allows for independence in participant objects while maintaining a system of interconnectivity. In my application if there are so many components who wanted to cone concerned when there are some changes in the department data for example, a UI component or an analytics module, then they can subscribe with the place of the data. This de-couples the subject (departments) from its observers implying that the system can be expanded with new observers or the existing observers can be altered to fit the new form and functionality without much effort being directed to the subject.

3- Singleton

* **Evidence** from my code: I have modified the DataStorage class to employ the Singleton design pattern; this makes for a singular instance of the data storage in the application.
* **Impact**: This makes it an important pattern when the idea is to have a global access point to the shared resources or the shared data. In my application, having one instance to store data will reduce problem like who lack of control on the data and who gets the last copy with more updated data. Moreover, since the instance is public, it makes it easy to exchange information with other areas of the framework.

Conclusion: The integration of design patterns in my code therefore improves the develoment of applications through principles such as modularity, flexibility, and maintainability. Finally, through the use of such container forms, I find that my code is more effective and compact and less likely to contain as many mistakes. In addition, design patterns help solve real-world problems and leverage past solutions so they can be reapplied through code reusability, which results in enhanced productivity and expandability of the development process.

## Task 10: Design patterns

1- Creational Design Patterns.

* The creational patterns are employed where there is an option that the objects can be created differently or where the system has to remain ignorant or the fact that how the objects were being created, assembled, or represented.
* They are also employed to encode information about the classes that the system employs and to define ways through which objects of these classes are instantiated and disposed.

Example:

* **Singleton Pattern:** Used when in a certain time, there is only requirement for a single object of the class and it is used in a logging process, in storing of configuration details, or in a database connection pool some times.
* **Factory Method Pattern:** It is used when they feel that a class must be inherited beyond several subclasses and they want subclasses to have the ability to define which concrete subclass should be created. For example, in a UI library, there are available types for button making, namely, WindowsButton and MacButton because its functions are constructed by a factory.
* **Abstract Factory Pattern:** Applicable in situations in which you need to generate families of related or child objects without deriving the classes. For instance, namely the factories can produce different parts of a game, for instance, characters, weapons and environments, wherein you can obtain different yields.

2- Structural Design Patterns.

* Used in cases where objects must be grouped into a large structure and where there exists efficient strategies to solve the relationships existing among the objects.
* They are also used when an object contains another object, and they need to be united, but you cannot alter the items that make up the other object.

Example:

* **Adapter Pattern:** If a programmer having an existing class the time is used and come to know that it has a certain interface and the environment where to be implemented has different interface should use. For instance, renaming of a term in a class to fit with a new system or library can be performed.
* **Decorator Pattern:** Use when you would like to manage behaviors and responsibilities of specific objects of a class when created at run-time and do not wish all objects of that class should possess the behavior. For example, when the textual editor software could not have in its list of features spell check or formatting of the text.
* **Composite Pattern:** I use it when one element has to be relatively smallest in size to the second one, for example to represent the parts of an object in relation to the whole. There is where operationality comes into play if you have to make people treat elementary objects and also compositions of objects in a similar way. For instance, in file system, one can add both a file and other directories as the subdirectories of the current directory.

3- Behavioral Design Patterns.

* Behavioral patterns are employed when there is a necessity to control the algorithms, responsibilities as well as communication between objects.
* They are also used when you would like to make the system more adaptive and easier in the sending and receiving of requestors.

Example:

* **Observer Pattern:** Apply when you require objects to maintain a one-to-many relationship, where the existence of an object should inform lower-level objects regarding updates. For instance, using GUI frameworks, in cases of event handling, or when there is a need to update multiple views within the Model 2 architecture.
* **Strategy Pattern:** Use when you plan on differentiating between a set of algorithms, whenever you want to encapsulate each of the algorithms and when you want to swap between different algorithms. It is however useful when you want the running time to determine which algorithm is to be used or when you have different versions of an algorithm. For example, functions partitioning the list in a sorting library.
* **Command Pattern:** To be used any time you need to envelope a request as object so that you can parameterize your clients with the queues, requests and operations you require. It is useful for undoable operations or for reservations of requests for its further execution. For instance this is true in the case of providing a menu system in the graphical user interface.

## Task 11

### 1- The constructor of a class has many parameters

Use Builder design Pattern

### 2- object is created and used in many different location

Use Factory design Method Pattern

### 3- a large object that should be loaded on demand and avoids duplication of the same object

Use Proxy design Pattern

### 4- Choosing payment method at runtime in a shopping site

Use Strategy design Pattern

# Task 12:

Justification using design Pattern in task 11:

1 –

* Provides an understandable, clear-to-read way to construct objects when they are separated from each other
* Because it is convenient when you want to create objects that have a lot of parameters when some of them are optional and others are mandatory
* You can define a constructor class with methods for setting each optional parameter, allowing you to specify only the parameters you want
* It helps to avoid overloading and make the code readable

2-

* It is suitable when you need to create objects in more than one different place, as it works to create objects in one detailed place
* By delegating object creation to factory method classes, you can easily modify and extend the creation process without changing the client code, making it easier to maintain.

3-

* The Proxy Pattern is a technique utilized to manage access to massive objects that are retrieved only when required. The technique is like a doppelganger for the genuine item; it puts off its construction or loading up to the time it is demanded by the customer. The proxy verifies whether the article has been designed or loaded and when it hasn’t it starts this process and takes care of further inquiries personally. For instance, it can be applied by developers that deal with stuff that requires a lot of resources in terms of time, when creating or loading objects.

4-

* One use case where the Strategy Pattern is very effective in making it possible for clients to switch between different algorithms or behaviours (strategies) dynamically is when we have many payment method classes. The essence of the pattern is that we define an interface class named PaymentStrategyInterfaceClass, and then implement our different payment methods (strategies) as alternative sub-classes each representing a specific payment mechanism under this interface class. The core idea behind this pattern is constituted by what may be called the Plug & Play payment methods concept.

# References

Visual-paradigm.com. (2019). *What is Class Diagram?* [online] Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/.

‌

Khan, M.H. (2023). *Understanding Object-Oriented Relationships: Inheritance, Association, Composition, and Aggregation*. [online] Medium. Available at: https://medium.com/@humzakhalid94/understanding-object-oriented-relationships-inheritance-association-composition-and-aggregation-4d298494ac1c.

‌

refactoring.guru. (n.d.). *Creational Design Patterns*. [online] Available at: https://refactoring.guru/design-patterns/creational-patterns.

‌