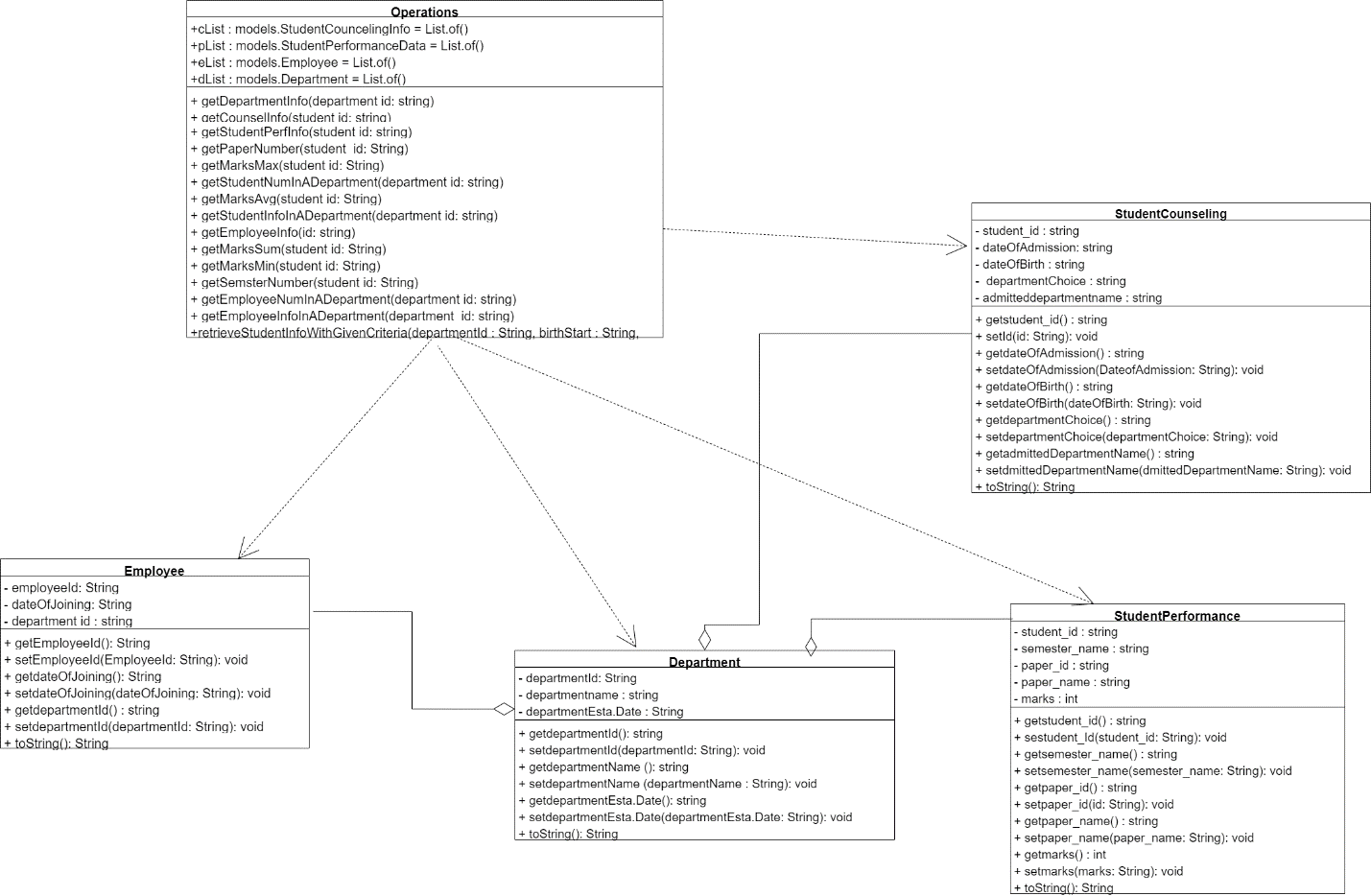
AHMAD OBEID

[21110079]

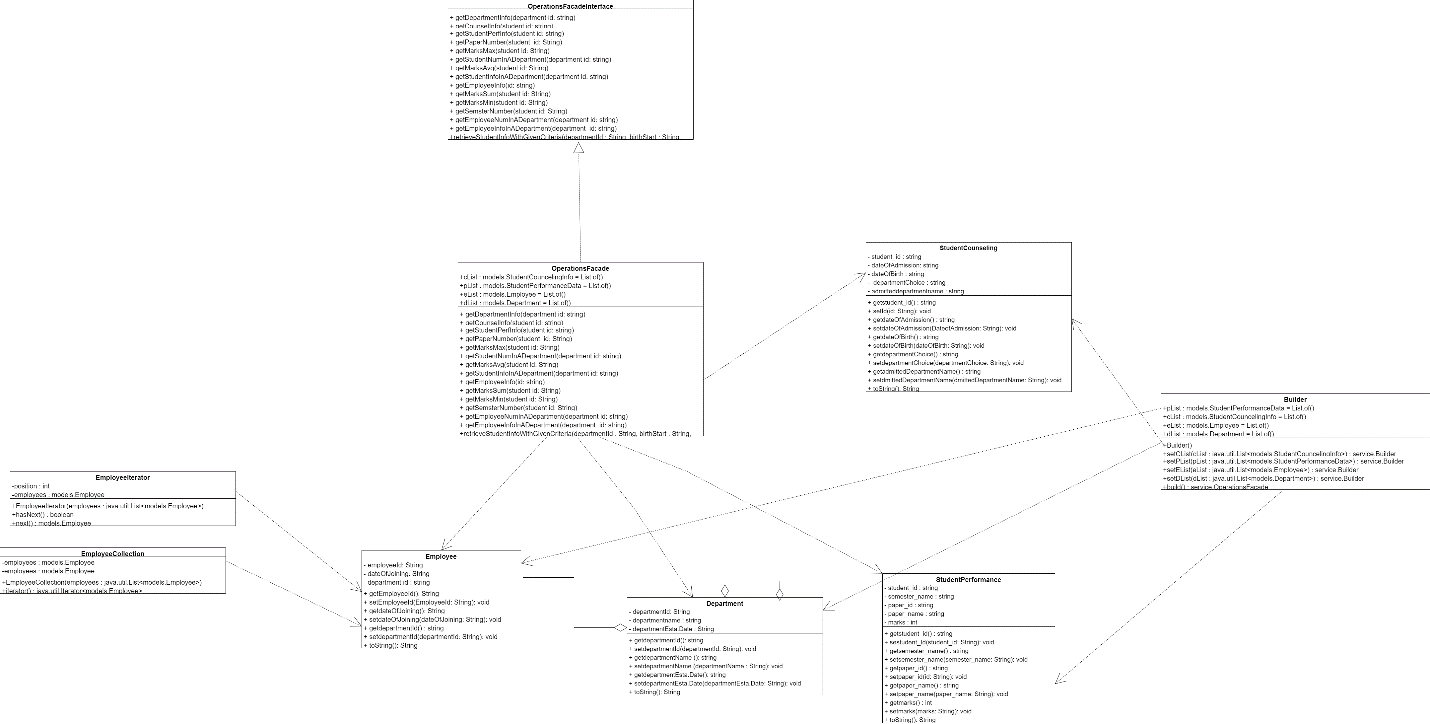
[Advance programming]

[Dr. Ahmad Bataineh]

**Task 1)**



**Task 2)**



**Task 4)**

1. Object-oriented design characteristics

* Encapsulation: a concept of encapsulating data and the operations performed on data in a single unit. This idea is frequently applied to conceal an object's internal state representation from view and prevent direct access to some object components to avoid (by mistake) modifications.
* Inheritance: The parent class is the class from which a class inherits its characteristics or capabilities. Since the code, which contains the properties of the parent class, can be inherited as previously mentioned, we can reuse it without having to write it every time the inheritance feature is implemented.
* Abstraction: Abstraction is the process of simplifying a complex reality by designing classes that are suited for the issue at hand and going up to a strong level of inheritance. This characteristic reduces the complexity of programming. It’s a technique of hiding an application's internal workings from the outside world and it's employed to draw a line between client programs and the application.
* Polymorphism: It concerns granting objects of various classes the same treatment as objects of a shared superclass. Methods like overriding are a basic way that it happens. It makes it easier to extend code and eliminate repetition by enabling the use of a single interface for a broad class of activities. Also, overloading is considered in polymorphism concept.
* Interface: An interface provides a class with a list of methods that it must implement; it does not supply the methods' implementation.

1. Class relations

* Association: a relationship in which every object has an independent life cycle and no owner. It symbolizes a "works-with" relationship.
* Aggregation: a particular kind of association in which each object has its own lifespan, ownership exists, and offspring objects are capable of existing separately from their parents.
* A relationship where one class depends on another because it uses its functionalities. Called dependencies

1. Examples from the project: I implement setters and getter in the main classes that I have like employee class, department, student counseling, and student performance. Also, used the aggregation relations between the classes, and dependency relation between the classes and the operations class.

**Task 5)**

1. Design patterns: Standardized, reusable solutions to frequent issues in software design are known as design patterns. They give programmers a tested blueprint for resolving particular design problems, enabling them to write more adaptable, scalable, and maintainable code. These patterns can be divided into three primary categories: creational patterns, which deal with object creation procedures and attempt to generate objects in a situation-appropriate way. structural, which address the arrangement or composition of classes and objects. Lastly, behavioral patterns that explain the interactions and communication between classes and objects.
2. Creational design patterns

* Factory: A method for creating objects without defining the precise class of the object to be created is the Factory Pattern. Subclasses overload the factory method to select which object to instantiate, in place of using a constructor. The system is updated with new classes and manages creation without affecting client code. Another name for it is Virtual Constructor.
* Builder: By separating a complicated object's representation from its production, the Builder Pattern makes it possible to produce several representations using the same construction method. It is very helpful when making items that take multiple steps to assemble or when there are a lot of optional parameters.
* Prototype: This pattern is about making a copy of an existing object as a new object, its cloning patterns. The need for using this pattern comes when the cost of creating an object from scratch and not copying it is high, and if the developer has a few objects that are copied frequently. Also, the cloning objects using this pattern is done without the need to pair the code to the object’s classes.
* Singleton: Of all the design patterns, the singleton pattern is the most well-known. It guarantees that a class has a single instance and offers a global point of access. The constructor will be private and there will be a static method that has return type object of this singleton class. getInstance(), this is a method to create an object instead of constructor.

1. Structural design patterns

* Proxy: provides a method for building a class that may be used to mimic the features of another class. In addition, it provides a surrogate for another object to control access to it and conceals the original object and control access to it. Without altering the code of the original object, this approach can be used to add features like remote access, logging, access control, and lazy initialization.
* Adapter: The adapter pattern changes a class's interface to one that a client would anticipate. An adapter, also called a wrapper, enables classes that would not otherwise be able to collaborate due to conflicting interfaces to do so. Because it allows for more flexibility, this technique is very helpful when combining legacy code with new code.
* Façade: the Facade pattern facilitates client interaction with a complex subsystem by offering a cohesive, simplified interface. This approach hides the intricate details of the subsystem from the client and simplifies interactions by wrapping the sophisticated subsystem in a facade class.
* Bridge: Separate an abstraction from its implementation to allow for independent variation in each. It also makes it possible to keep the implementation and abstraction apart. The two components of the bridge pattern are abstraction and implementation.

1. Behavioral design patterns

* Chain of responsibility permits requests to be passed between possible handlers until a handler accepts them. The request is either handled by each handler in the chain or it is passed on to the one after them. This pattern facilitates flexibility and reusability in the handling process by severing the sender and recipients of a request.
* Iterator: used to obtain a sequential method of accessing a collection object's elements. The purpose of the iterator pattern is to transfer the burden of accessing and iterating through the collection's objects to the iterator object. By maintaining track of the current item and having a mechanism to determine which elements need to be iterated next, the iterator object will maintain the state of the iteration. Because a for loop makes use of indices, it isn't always feasible. For instance, because sets are unordered collections, they lack indices, but lists have. All types of iterable arrays and collections are compatible with iterators.
* Template: with the help of templates, an algorithm's general structure can be defined in a base class, and subclasses can override specific stages without affecting the algorithm's overall structure. By allowing specific behavior to be adjusted while allowing common behavior to be shared, it encourages code consistency and reuse.
* Strategy: The Strategy Pattern captures, specifies, and renders compatible a family of algorithms. The algorithm can change without affecting the clients that use it thanks to this design. Clients don't need to change their code to switch algorithms because all strategies use a common interface. It enables runtime modifications to an algorithm's behavior by the client.

**Task 6)**

Using and implementing design pattern provide high significant positive impact on object-oriented design, because design pattern improves development process as well as the quality of the software, when suggesting design solutions for design problems, design patterns role come to enhance code reusability and maintainability. design patterns offer shared vocabulary among the designers who work on a project, so the designers become able to collaborate with each other in effective way and helps the developers to convey complex ideas and requirements easily with gaining string understand of the software.

Design patterns offer the ability to improve code flexibility which will be written by the developer team, factory and strategy for example are types of design patterns, they decouple classes and their dependencies which lead to easy modifying and extending process on the system without causing big changes to the codebase. The Strategy pattern, for example, enables the selection of algorithms at runtime, improving the system's ability to adapt to changing requirements without modifying the context in which the algorithm functions.

Also, these patterns contribute to a more robust and reliable code by encapsulating recommended practices and lowering the risk of common errors, also lead to more resilient and dependable code. To avoid problems with many instances of a class that should be unique, the Singleton pattern, for instance, makes sure that a class has only one instance and offers a global point of access to it. This is especially helpful in situations when we need to manage connections to databases, for example.

Moreover, using these patterns allow the designer to understand and read the code in an efficient way, also by abstracting complex design concepts into more familiar patterns, the developer will become more able to understand the structure and intent of the code faster, because high readability means there are no problems when onboarding new team members and ongoing schedule for the software maintenance

A study in 2004 was conducted by Vokac to show that the systems when compared to those that did not use design patterns, those who did showed increased maintainability and lower error rates. Also, another conducted study by Prechelt in 2001, the writer found that the design pattern can lead to highly efficient problem solving, which is done by offer templates to help the developer to adapt specific scenario to meet the system requirements, which lead to faster development process and high software quality.

**Task 10)**

1. Creational design patterns

* Factory: when we need to create objects without giving specific class on the type of item to be created, we utilize a factory. For example, if a system for handling documents generates various document types, such as Word and PDF, we may use the factory pattern to construct a document based on user input without having to know the class of each document type. Example is that a document creation application where different types of documents such as PDF can be created without specifying the exact class of the object that will be created.
* Builder: when the object needs to be created in several steps, we use this pattern. For example, we use the builder to create complicated automobile objects step by step and let the user create several car models with a variety of component options.
* Prototype: in situations where we need to duplicate an existing object, we employ prototype patterns, and when creating new objects is resource-intensive. For instance, we can clone an existing shape in a graphic design program and alter it rather than starting from scratch each time we need to replicate a complex Shape object with small modifications using a prototype pattern.
* Singleton: a global point of access to a class and assurance that it has a single instance are provided by the singleton. To guarantee that all application components use the same configuration settings and that any changes are globally accessible, an application's configuration manager uses singletons. Example is logger class that ensures only one instance handles all log messages in an application.

To summarize, these patterns are usually used when the exact types and dependencies of objects need to be specified at runtime, when there is a need to manage and control the lifecycle of objects, and when there is a need to hide the complexities of creating objects.

1. Structural design patterns

* Proxy: When we need to offer an intermediary for another object to manage access to it, we employ the Proxy Pattern. When creating an expensive object, such loading a large image file only when it is going to be displayed, we want to postpone creation until it is truly needed.
* Adapter: adapter pattern is used when we wish to enable conflicting interfaces to function together. When a modern ERP system has to access a legacy system component, an adapter is employed to change the legacy component's interface into a usable current component.
* Bridge: bridge is used when we need to support several implementations of the same abstraction. If I have multiple shapes (abstraction) that can be created in multiple ways (implementation) in the same graphic design program (we mentioned it in a previous example), I can combine shapes with the methods using the bridge pattern.
* Facade: when it's necessary to simplify a complex subsystem's interface and make a set of classes easier to use, facade is employed. Consider a home theater system that consists of several parts, such as a projector, DVD player, lighting, and sound system. By offering a single way to turn on the complete home entertainment system instead of addressing each component separately, a facade can streamline the process.

To summarize, these patterns are usually used when there is a need to compose objects into tree structures to represent part-whole hierarchies, when simplifying the structure by identifying the relationships, and when making sure that a system uses encapsulation to hide the complexities.

1. Behavioral design patterns

* Chain of responsibility: I used this pattern when I want to separate the request sender from the receiver, like in customer service, the request might go through and pass different employee until it is solved.
* Iterator: this pattern is used when there is a need to access a collection's elements in sequential way, like using the iterator in social media app to access user posts from a collection sequentially.
* Template: using this pattern to when there is a need to illustrate the structure of an algorithm in a method by deferring some steps to subclasses. In data processing applications, subclasses offer particular implementations for reading, processing, and writing the data, while template methods could describe the procedures for processing data.
* Strategy: When I need to choose an algorithm at runtime, I can build a family of algorithms and encapsulate them all using the Strategy Pattern. This is the situation where I need to use strategy. Example for using the pattern is in sorting algorithms like merge sort that can be selected at runtime for different collections.

To summarize, these patterns are usually used when defining how objects interact and how the responsibilities are distributed among them, when the designer need to improve communication flexibility, and when the designer want to encapsulate behaviors in objects.

**Task 11 & 12**

* The cases:

1. Builder pattern. Because there are many parameters and some of them are optional, so when we want to create an object with this number of parameters, we builder to provide a step-by-step process to construct that object to make sure that mandatory parameters are assigned and as much as needed from the optional or omitted them. Therefore, the code will be more readable, and we will avoid any issues that come with having a constructor with many parameters.
2. We should use singleton in this case because this pattern ensures that the class has only one instance and provides a global point of access to it. In this case, the object might be used in many different locations and by many different developers, and singleton is suitable for that, and when we implement singleton, we make sure that all part of the system use the same instance and we avoid multiple instances creation in the system to ensure the consistency across the system.
3. Proxy is suitable for this case. When expressing a huge object that needs to be loaded on demand, the Proxy Pattern works well. By using this pattern, it is possible to restrict access to the object and load it only when necessary. We also ensure that there is only one instance of the huge object utilized, preventing object duplication. This is especially helpful for resources that cost a lot to produce or require a lot of memory.
4. Strategy pattern. In this case, the user can choose different payment method for checking out in runtime, and strategy pattern is well-suited for that because it defines a family of algorithm, which is payment method in this case, encapsulate each one, and make them interchangeable, so the client can choose from these different options with flexibility. Also, the system will be extended in case of adding new method of payment in the future.

**References**

1. GeeksforGeeks (2024) Object Oriented Programming (OOPS) concept in Java, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/ (Accessed: 09 June 2024).
2. (No date) On inferring autonomous system relationships in the ... Available at: https://www.cs.princeton.edu/courses/archive/fall06/cos561/papers/gao01.pdf (Accessed: 09 June 2024).
3. (No date)Available at: https://link.springer.com/article/10.1023/B:EMSE.0000027774.30594.4b (Accessed: 09 June 2024).