6/9/2024

**[Advanced Programming]**

[Final Project]

LIYAN AQEL

[21110405]

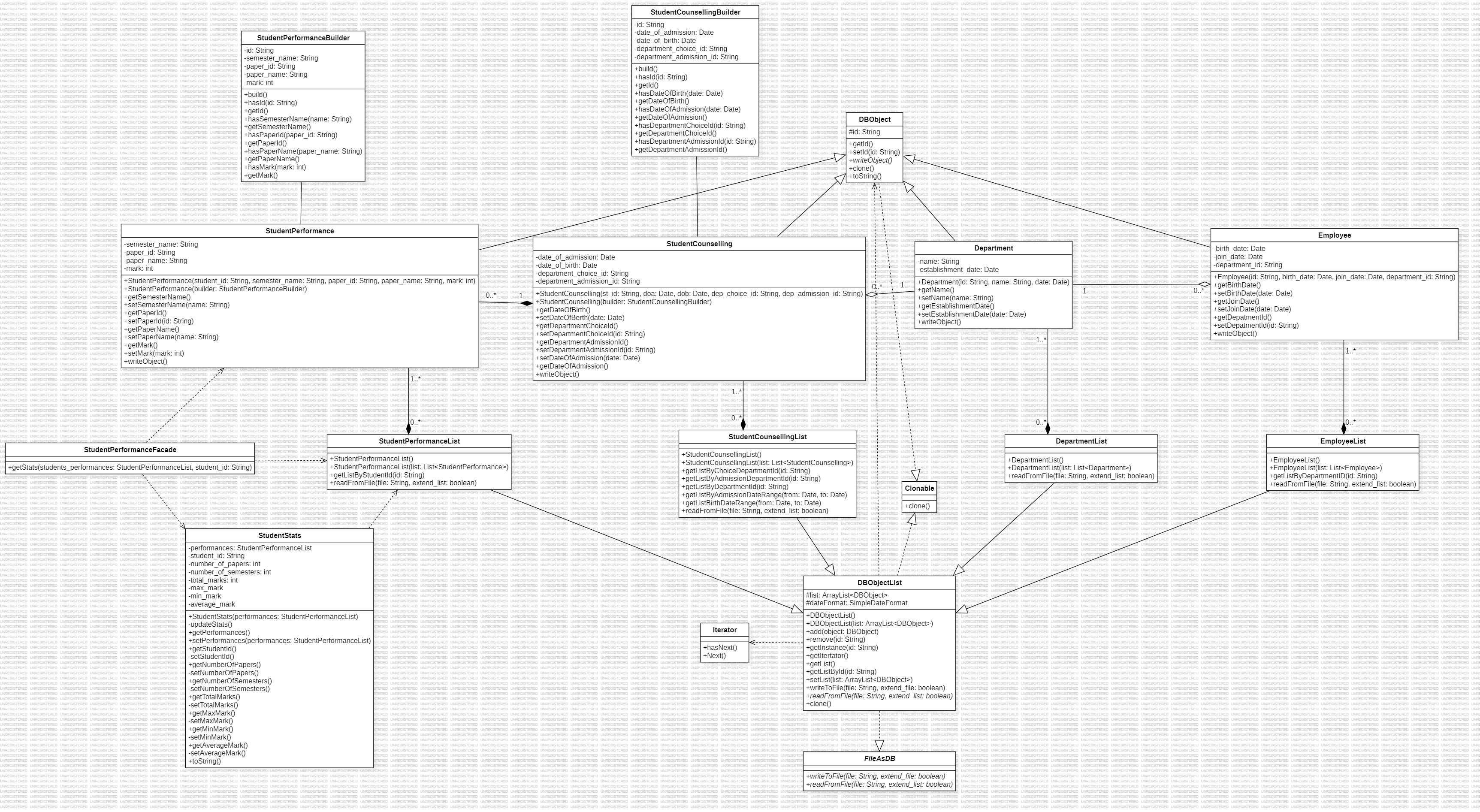
Computer science

1. UML class diagram that represents a dataset processing application for the dataset described above. P2

A computer screen with many white squares

Description automatically generated

1. Refining my design by using design patterns to enhance the operation of my software and justifying my choices. M2



**The design patterns that I am used.**

**Prototype Pattern:**

Classes: “DBObject “and” DBObjectList” and “StudentStats”

Methods: DBObject.clone (), DBObjectList.clone (), StudentStats.setPerformances ().

**Justify:**

I used this pattern to create a copy of any object quickly and with no need to know its type exactly, as well as helping the developers save time because they will not need to rewrite the code. By implementing “clone ()” method in “DBObject” class, I allow all subclasses such as “Employee”, “Department”, “StudentPerformance”, “StudentCounselling” to be clones quickly and easily. This pattern is valuable to my project because it enables for the efficient construction of object copies, and which can be useful in situations when objects must be copied repeatedly, such as deep copying lists in DBObjectList.

**Builder Pattern:**

Classes: StudentCounsellingBuilder, StudentPerformanceBuilder, StudentCounselling, and StudentPerformance.

Methods: build () and has\*().

**Justify:**

I used this pattern to provide my system flexibility to construct complex objects. So, when I applied this pattern, it will enable my system step-by-step generation of objects with several attributes. In my software, the classes builder which are StudentCounsellingBuilder, StudentPerformanceBuilder simplify the creation of object for StudentCounselling, and StudentPerformance classes by offering a simple and succinct way to configure multiple attributes before object construction. **As a result,** this pattern enhances my system maintainability, readability, and makes object generation simpler.

**Iterator Pattern:**

Classes: DBObjectList

Methods: getIterator(), getInstance().

**Justify:**

I used this pattern to access the elements of a collection of objects sequentially without identifying its underlying representation. In my software, I applied this pattern in “DBObjectList” class to access the list of DBObject instances**. As a result,** this pattern streamlines iteration across collections and hides the list's internal structure, allowing for more consistent navigation among objects.

**Facade Pattern:**

Classes: StudentPerformanceFacade, StudentPerformanceList and StudentStats

Methods: getStats(StudentPerformanceList students\_performances, String student\_id)

**Justify:**

I used this pattern to provide a simple interface to a complicated subsystem. In my system, I am interacting the faced class with StudentPerformanceList and StudentStats classes to reach the statistics of student performances. **As a result,** this pattern decreases complexity for the user by offering an easy-to-use interface for collecting student performance statistics while concealing the deep features of the underlying subsystem.

**Strategy Pattern:**

Classes: FileAsDB

Methods: writeToFile(String file, boolean extend\_file), readFromFile(String file, boolean extend\_list)

**Justify:**

I used this pattern to define a set of algorithms or methods, encapsulate each one, and make them reusable and interchangeable. In my system, the FileAsDB is a strategy interface that includes methods for file operations. Many subclasses of DBObjectList class implement these methods, which offer certain reading and writing strategies. **As a result,** this pattern enables the file operation algorithm to change independently of the classes that use it and without editing the main class which is “FileAsDB”, as well as increasing flexibility and extensibility.

**The design patterns that I am not used are singleton, factory, Proxy, adapter, bridge, and chain responsibility, and template.**

**Singleton pattern 🡪**

My current software does not require a global state or a centralized point for accessing any of the classes. For example, additional instances of DepartmentList, EmployeeList, and other list classes are generated as needed. Also, using singletons might limit flexibility, particularly in a multi-threaded system where different threads may require separate instances.

**Proxy Pattern 🡪**

My design doesn't need object access control. All objects are accessed directly, with no access limitations, logging. Also, introducing proxies can result in needless performance overhead for the current software.

**Adapter Pattern 🡪**

All my classes have a consistent interface design, especially given they extend DBObject and implement the FileAsDB interface.   
In my current architecture, no old or existing class needs to be modified to accommodate a new interface.

**Chain of Responsibility Pattern 🡪**

My current design does not provide a scenario in which a request must be processed by multiple handlers in order. Each class manages its own functionality independently. Also, methods and functionalities are managed directly within classes, therefore there is no need to transfer requests across objects.

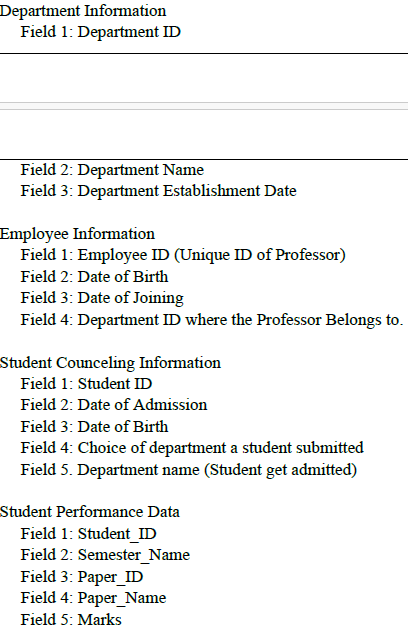
**Bridge Pattern 🡪**

The current design does not include different abstractions and implementations that must vary separately. As an example, the DBObjectList and its subclasses are structurally and functionally tightly coupled. Also, the additional complexity of separating abstractions and implementations is unnecessary for the existing basic system.

1. Analyze how the UML class diagram extracted from a given scenario, use the scenario above and your UML design as an example. D2

The following analysis explains how the project requirements are used in the UML class diagram. Each class and method were developed to meet certain functional requirements described in the scenario, ensuring that the software can efficiently manage university information. The design decisions, such as method implementation, attribute selection, encapsulation, and inheritance are explained by the project requirements.

**Classes:**

First, I started my design with **four essential classes** because my scenario mentioned that a university maintain information about departments, employee, students and students’ information, the information is arranged in 4 files as shown below: 

**The accessible methods for each class and its attributes:**

Then, I provide each class with its **attributes** according to above scenario that mentioned them, as well as give them private access modifier to reduce access. So, I used [public] Getter and setter methods to follow the encapsulation principles , allowing limitted access to the attributes, ensures data integrity while hiding the internal state.

**Department Class**

**Attributes:**

Id: String

Name: String

Establishment\_date: Date

**The constructor 🡪 Department(id:Steing,name:String,date:Date)**

**Methods:**

getId() , setId(id:String)

getName() , setName(name:String)

getEstablishmentDate(), setEstablishmentDate(date:Date)

writeObject() 🡪 A method for writing the object from department type.

**Justify:**

The scenario specifically mentions attributes like id, name, and establishment\_date. Also, methods are used to encapsulate data such as Getters and setters (get and set methods), allowing for restricted access and alteration.   
The writeObject() method is required for saving department data to a file, as indicated by the necessity of handling university information.

**Employee class**

**Attributes:**

Id: String

birthDate: Date

joinDate: Date

departmentId: String 🡪 The employee's departmental ID.

**The constructor 🡪 Employee (id:String, birth\_date:Date, join\_date:Date,department\_id:Steing)**

**Methods:**

getId() , setId(id:String)

getBirthDate(), setBirthDate(date: Date)

getJoinDate(), setJoinDate(date: Date)

getDepartmentId(), setDepartmentId(id: String)

writeObject() 🡪 A method for writing the object from department type.

**Justify:**

Attributes are directly related to the scenario's necessity to preserve personnel information.   
Getters and setters encapsulate data.   
The writeObject() method assists in handling employee information by enabling serialized for storage.

**StudentCounselling class:**

**Attributes:**

studentId: String

dateOfAdmission: Date

dateOfBirth: Date

choiceDepartmentId: String

departmentAdmittedId: String

**The constructor 🡪 StudentCounselling(st\_id:String, doa:Date, dob:Date,dep\_choice\_id:String,dep\_admission\_id:String)**

**Methods:**

getStudentId() , setStudentId(id:String)

getDateOfAdmission(), setDateOfAdmission(date: Date)

getDateOfBirth(), setDateOfBirth(date: Date)

getDepartmentChoiceId(), setDepartmentChoiceId(id: String)

getDepartmentAdmissionId(), setDepartmentAdmissionId(id: String)

writeObject()

**Justify:**

Attributes are consistent with the scenario's demand to keep student counseling information.  
Getters and setters assure encapsulation.  
The writeObject() method is required for storing student counseling data.

**StudentPerformance class:**

**Attributes:**

studentId: String

semesterName: String

paperId: String

paperName: String

marks: int

**The constructor 🡪 StudentPerformance(student\_id:String, semester\_name; String, paper\_id, String, paper\_name:String, mark:int)**

**Methods:**

getStudentId(), setStudentId(id: String)

getSemesterName(), setSemesterName(name: String)

getPaperId(), setPaperId(id: String)

getPaperName(), setPaperName(name: String)

getMark(), setMark(mark: int)

writeObject().

These previous methods offer controlled access to performance data.

**Justify:**

Attributes rely on the requirement to keep student performance data.  
Getters and setters allow for controlled access to data.  
The writeObject() method enables serializing of student performance records.

**DepartmentList Class**

**Attributes:**

list: ArrayList<Department>: List of departments.

dateFormat: SimpleDateFormat: For dealing with date formats.

**The constructors 🡪**

DepartmentList(in list:List<Department>)

DepartmentList()

**Methods:**

add (object: Department): Adds a department to the list.

remove (id: String): Removes a department by ID.

getInstance(id: String): Retrieves a department by ID.

getList()

getListById(in id:String)

setList(in list:ArrayList<Department>)

writeToFile(file: String, extend: boolean): Writes the list to a file.

readFromFile(file: String, extend: boolean): Reads the list from a file.

**Justification:**

This class controls a list of departments as specified by the scenario.   
Methods such as add, remove, and getInstance perform some of the necessary operations on departments depending on the scenario.   
Persistent storage requires the ability to write to and read from files.

**EmployeeList Class**

**Attributes:**

list: ArrayList<Employee>: List of employees.

dateFormat: SimpleDateFormat: For dealing with date formats.

**The constructors 🡪**

EmployeeList()

EmployeeList(in list:List<Employee>)

**Methods:**

add(object: Employee): Adds an employee to the list.

remove(id: String): Removes an employee by ID.

getList()

getListById(in id:String)

setList(in list:ArrayList<Employee>)

getInstance(id: String): Retrieves an employee by ID.

getListByDepartmentID(id: String): Retrieves employees by department ID.

writeToFile(file: String, extend: boolean): Writes the list to a file.

readFromFile(file: String, extend: boolean): Reads the list from a file.

**Justification:**

This class handles a list of employees, which meets the need for managing employee data.  
GetListByDepartmentID retrieves personnel from a given department.  
File operation methods ensure that data is stored persistently.

**StudentCounselingList Class**

**Attributes:**

list: ArrayList<StudentCounseling>: List of student counseling records.

dateFormat: SimpleDateFormat: For handling date formats.

**The constructors 🡪**

StudentCounsellingList()

StudentCounsellingList(in list:List<StudentCounselling>)

**Methods:**

add(object: StudentCounseling): Adds a student counseling record to the list.

remove(id: String): Removes a student counseling record by ID.

getInstance(id: String): Retrieves a student counseling record by ID.

getListByDepartmentId(in id:String)

getListByChoiceDepartmentId(id: String): Retrieves records by chosen department ID.

getListByAdmissionDepartmentId(id: String): Retrieves records by admitted department ID.

getListByAdmissionDateRange(from: Date, to: Date): Retrieves records within an admission date range.

getListBirthDateRange(from: Date, to: Date): Retrieves records within a birth date range.

writeToFile(file: String, extend: boolean): Writes the list to a file.

getList()

getListById(in id:String)

setList(in list:ArrayList<StudentCounselling>)

readFromFile(file: String, extend: boolean): Reads the list from a file.

**Justification:**

As described in the scenario, this class controls student counseling records.   
Methods such as getListByChoiceDepartmentId and getListByAdmissionDepartmentId enable retrieval depending on department ID.   
The date range methods (getListByAdmissionDateRange, getListBirthDateRange) retrieve records between dates.   
File operations ensure data permanence.

**StudentPerformanceList Class**

**Attributes:**

list: ArrayList<StudentPerformance>: List of student performance records.

**The constructors 🡪**

StudentPerformanceList()

StudentPerformanceList(in list:List<StudentPerformance>)

**Methods:**

add(object: StudentPerformance): Adds a student performance record to the list.

remove(id: String): Removes a student performance record by ID.

getInstance(id: String): Retrieves a student performance record by ID.

getListByStudentId(id: String): Retrieves records by student ID.

getStats(student\_id: String): Performs statistical operations on a student's performance records.

getList()

getListById(in id:String)

setList(in list:ArrayList<StudentPerformance>)

writeToFile(file: String, extend: boolean): Writes the list to a file.

readFromFile(file: String, extend: boolean): Reads the list from a file.

**Justification:**

This class handles student performance records.   
getListByStudentId retrieves records according to a student ID.   
getStats conducts necessary statistical operations such as the number of papers, semesters, and average grades.   
File operations ensure long-term storage.

**The enhancement:**

The Department, Employee, StudentCounseling, and StudentPerformance classes all inherit from DBObject because they have similar attributes such as id and methods such as writeObject().

I applied the **protected access modifier** to the id attribute of the DBObject class to ensure that it is available inside the class, its subclasses (Department, Employee, StudentCounseling, and StudentPerformance), and the same package. This enables subclasses to easily access and manipulate the id attribute, simplifying inheritance and code reuse while maintaining a level of encapsulation that precludes direct access from external classes. This approach addresses the scenario's requirement for controlled access to common properties while increasing code maintainability and readability.

I am using DBObject as an abstract class rather than an interface that makes sense since it allows for the definition of both shared attributes (such as id) and concrete methods (such as writeObject()). An abstract class can include abstract methods (which must be implemented by subclasses), concrete methods (which offer common functionality), and fields that can be inherited directly. This structure encourages code reuse and reduces duplication because common attributes and methods are declared just once in the DBObject abstract class and inherited by all subclasses such as StudentCounseling, Employee, Department, and StudentPerformance. Finally, an interface can only define abstract methods and constants, which prevents these classes from sharing concrete implementations.

This structure encourages code reuse, minimizes code duplication, improves maintainability and streamlines maintenance by consolidating shared functionality into a single superclass.

**Encapsulation:**   
Getters and setters are employed to encapsulate attribute values. This makes sure attribute values can be accessed and changed in a controlled manner, thus preserving their confidentiality and consistency.

The following describes how the requirements of the project appear in the UML design, with an emphasis on the StudentStats class.

**StudentStats Class**

**Attributes:**

performances: StudentPerformanceList.

student\_id: String

number\_of\_papers: int

number\_of\_semesters: int

total\_marks: int

max\_mark: int

min\_mark: int

average\_mark: double

**The constructors 🡪**

**StudentStats(in performances:StudentPerformanceList)** 🡪 The constructor initializes the object with a list of student performance records.

**Methods:**

updateStats() 🡪 Updates statistics data according to performance records.

getPerformances()

setPerformances(performances: StudentPerformanceList)

getStudentId(), setStudentId(id: String)

getNumberOfPapers(), setNumberOfPapers(count: int)

getNumberOfSemesters(), setNumberOfSemesters(count: int)

getTotalMarks(), setTotalMarks(total: int)

getMaxMark(), setMaxMark(mark: int)

getMinMark(), setMinMark(mark: int)

getAverageMark(), setAverageMark(average: double)

toString() 🡪 Retrieves a string representation of the student's statistics.

* In this class, the setter methods that access and modify the attributes are private because the user not authorized to edit these attributes.

**Justify:**

The attributes are required to store and manage detailed information about a student's performance, which is consistent with the need to carry out statistical operations on student performance data.

The constructor StudentStats(performances: StudentPerformanceList) sets up the object with essential information.

UpdateStats() is required to calculate and update statistics depending on performance data.

Getters and setters enable enclosed access to attributes, maintaining data integrity.

The toString() method is important for presenting statistical data in an understandable manner.

**Relationships:**

|  |  |  |  |
| --- | --- | --- | --- |
| classes | Relation that was used | Relation that was not used | Justify |
| DBObject(whole)-other 4 classes(part) | generalization | Realization | The DBObject is providing shared attributes such as id and methods such as witeObject(). I don’t use realization relation between them because Realization is used when classes constantly must implement abstract methods from an interface. |
| Department (part)- Employee (whole) | Aggregation | Composition | Department can exist independently of the employee. The department’ lives are not inextricably linked to the employee’s lifecycle. Also, if the employee is destroyed, the department won’t get discarded. |
| Department (whole)- Student Counselling (part) | Aggregation | Composition | Department may exist independently of Student Counselling. The department lives are not inextricably linked to the Student Counselling’s lifecycle. Also, if the Student Counselling is destroyed, the department won’t get discarded. |
| Student Counselling (whole)- Student Performance (part) | Composition | Aggregation | Student Performance cannot exist independently of Student Counselling. Student performance lives are inextricably linked to the Student Counselling 's lifecycle. Also, if the Student Counselling is destroyed, the student performance will get discarded. |
| Department (part)- DepartmentList (whole) | Composition | Aggregation | The department cannot exist independently of DepartmentList class. Also, if the DepartmentList is destroyed, the Department will get discarded. |
| Employee (part) – EmployeeList (whole) | Composition | Aggregation | The employee cannot exist independently of EmployeeList class. Also, if the EmployeeList is destroyed, the Employee will get discarded. |
| Student Counselling (part) - Student CounsellingList (whole) | Composition | Aggregation | The Student Counselling cannot exist independently of Student CounsellingList class. Also, if the Student CounsellingList is destroyed, the Student Counselling will get discarded. |
| Student Performance (part)- Student- PerformanceList (whole) | Composition | Aggregation | The Student Performance cannot exist independently of Student PerformanceList class. Also, if the Student PerformanceList is destroyed, the Student Performance will get discarded. |
| StudentStats (receive object)- StudentPerformanceList (send object) | Dependency | Composition | I used dependency relation because StudentStats is not the owner of StudentPerformanceList, nor is it responsible for any aspect of its lifecycle. Also, StudentStats simply relies on StudentPerformanceList to execute calculations and update data; however, StudentPerformanceList can exist independently of StudentStats. |

The UML design matches the project's requirements by including the appropriate attributes, methods, and interactions between classes. Each class and method are carefully developed to meet specific functional requirements, ensuring that university information is managed efficiently. The use of encapsulation and inheritance improves the design's resilience and maintainability, in line with guidelines in object-oriented programming. The StudentStats class responds to the need for thorough statistical analysis of student performance, demonstrating the design's capacity to handle complicated requirements.

1. Examine the characteristics of Object-Oriented Design and class relationships, provide examples of your project. P1

**Object oriented programming:** It is a type of Programming Paradigm which focuses on software design of data, object, class, inheritance, encapsulation, and polymorphism. An object means a data field that has unique attributes and behaviors. It is good for programs which are large and complex and programs for manufacturing and mobile applications. It is also useful for collaborative development in which projects are divided into groups. It reduces complexity and increases efficiency and security because it divides the program into classes and objects.

**Object oriented programming characteristics:**

**Encapsulation:**

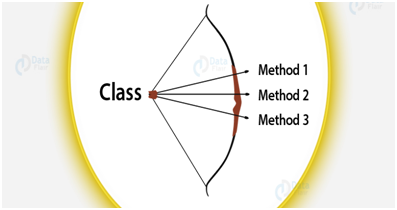
Encapsulation in Java is a crucial concept in object-oriented programming (OOP) that refers to the grouping of data and methods that operate on that data into a single unit known as a class in Java. Java Encapsulation is a technique for preventing outside access to a class's implementation details while providing only a public interface by which to interact with the class.

Encapsulation in Java is performed by defining a class's instance variables as private, meaning they can only be accessible within the class. To provide external access to the instance attributes, public methods called getters and setters are established, which are used to get and edit the instance variables' values, respectively. Employing getters and setters, the class may implement its own data validation rules and keep its internal state constant.

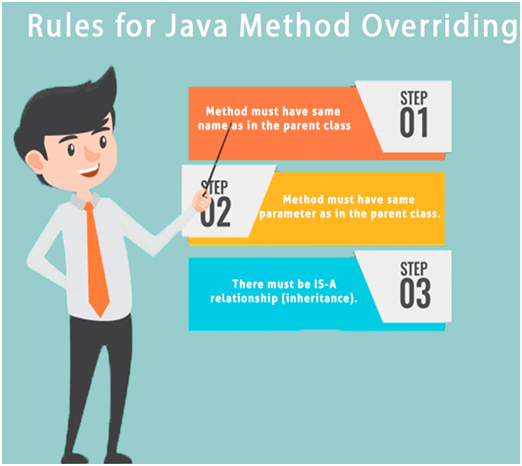
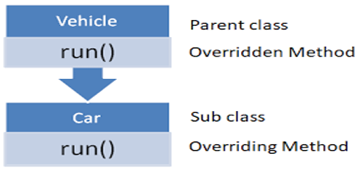
**Polymorphism:**

Polymorphism is regarded as one of the most significant elements of object-oriented programming. Polymorphism allows us to do the same action in multiple ways. In summary, polymorphism allows you to declare a single interface while having various implementations. "Poly" implies numerous, and "morphs" indicates shapes. It can be done in two different ways such as overloading and overriding.

Java allows polymorphism in various ways, including method overloading. Method overloading occurs when there are many methods with the same name but differing types/order/number of parameters in the same class.



Overriding is another approach to achieve polymorphism in Java.   
Overriding occurs when a method in a subclass has the same name and type signature as a method in its superclass. In this case, the method in the subclass overrides the method in the superclass.



**Relationships:**

Object-oriented programming (OOP) is an effective method for developing and structuring code. At the core, OOP is based on four essential class relationships: inheritance, association, composition, and aggregation. These relationships allow us to understand complicated systems and develop software that is stable, maintainable, and scalable.

**Inheritance:**

In object-oriented programming (OOP), inheritance allows a new class to inherit properties and behaviors from an existing class. It demonstrates an “is-a" relationship and encourages code reuse. **For example,** a "Dog" class may inherit from an "Animal" class. It is useful to reuse the code when it has common feature of other classes. It is something like a son inherits his qualities from his father.

**Association:**

Association is a relationship among classes in which each class exists independently but is loosely connected. It is frequently used to express a simple relationship between objects. Also, this relationship defines a relationship between two or more items that is connected by a link. It may be a one-to-one, or many-to-many, or one-to-many relationship. **For example,** an enrollment relationship can connect a "Student" object to a "Course" object.

**Composition:**

Composition is a strong relationship in which one class is made up of other classes, and each part is necessary for the whole to exist. It is used to build complex items by merging simpler ones. This relationship is a more powerful form of aggregation in which the component (child) cannot exist without the whole (parent). **For example,** a "House" object can be made up of "Room" objects, and if the house breaks down, the rooms are also lost.

**Aggregation:**

Aggregation is a weaker relationship in which one class includes others, but the components can exist independently. It is utilized when individual parts have their own life cycle and can be shared by numerous whole objects. This relationship is a type of association in which objects have a part-whole relationship, with the part (child) existing independently of the whole (parent). **For example,** a "Car" object can have a "Engine" object as a component, but the engine can exist independently.

**Dependency:**  
Dependency is a relationship in which one class (client) depends on another class (supply) for functionality. If a supplier class modification, the client class might require to be updated. For example, a "Person" class may require a "Car" class to function, yet the person can exist without the car.

**Abstract class vs Interface**

**Final Variables:** By default, the variables defined in a Java interface are final. Non-final variables may exist in an abstract class.

**Method implementation:** Some methods in an abstract class can be implemented, while others remain abstract, implying that they have no execution and must be overridden by concrete subclasses. whereas all methods in an interface are abstract by default and must be executed by any class that conforms to the interface.

**Type of methods:** Interfaces can only have abstract methods. An abstract class can have both abstract and concrete methods. It is possible to have default, Private, and static methods starting.

**Variable types:** An abstract class can have final, non-final, static, and non-static variables. The interface only contains static and final variables.

**Implementation:** An abstract class can implement the interface. An interface cannot implement abstract class.

**implements vs. extends:** The keyword "implements" is used to implement a Java interface, while the keyword "extends" is used to extend an abstract class.

**Multiple implementations:** An interface can extend one or more Java interfaces, but an abstract class can extend another Java class while implementing several Java interfaces. Also, a class can only inherit from one abstract class, but it may implement many interfaces. This is due an abstract class describes a type of object, whereas an interface provides a collection of behaviors.

**Create object:** Abstract classes and interfaces cannot be directly instantiated; therefore, you cannot create objects of either.

**To summarize,** abstract classes act as a foundation class from which concrete subclasses can be inherited, whereas interfaces provide a set of methods that a class needs to implement. Abstract classes can include both implemented and abstract methods, but interfaces can only contain abstract methods. Classes can only inherit one abstract class, but they can implement several interfaces.

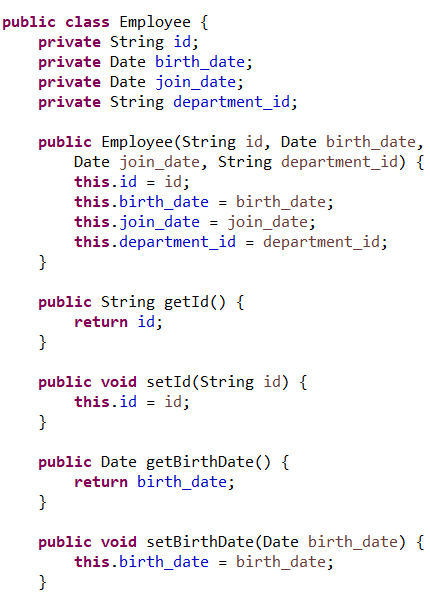
As we all know, abstraction is hiding the internal implementation of a feature and simply displaying the functionality to users.  displaying only the necessary features while disguising how those features are executed behind the scenes. In contrast, an interface is another approach to achieve abstraction in Java. Both abstract classes and interfaces are utilized for abstraction.

**The principles of object-oriented design (OOD)** include encapsulation, inheritance, polymorphism, and abstraction. These ideas are critical for developing reusable, modular, and maintainable systems. Here are concepts that I am using in my project.

1. **Encapsulation**

Encapsulation involves integrating the data and the methods that operate on the data into a class. To preserve the data's integrity, access to some components is also restricted.

**In the bellow example,** “Employee” class encapsulates its attributes and provides public methods (getter and setter) for accessing and modifying the attributes.

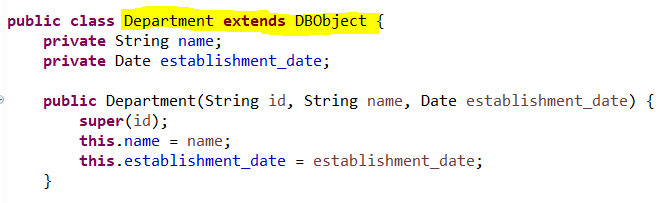
 A computer code with text

Description automatically generated

1. **Inheritance**

Inheritance enables one class to inherit the attributes and methods of another, which encourages code reuse.

**In the bellow example,** all these classes “Employee”, “Department”, “StudentPerformance”, “StudentCounselling” inherited the “DBObject” abstract class.

 A computer code with yellow text

Description automatically generated A computer screen shot of a code

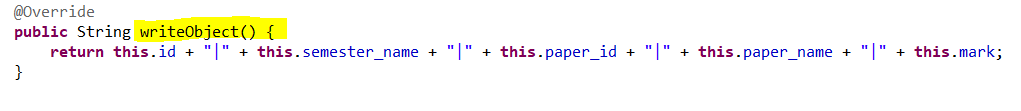
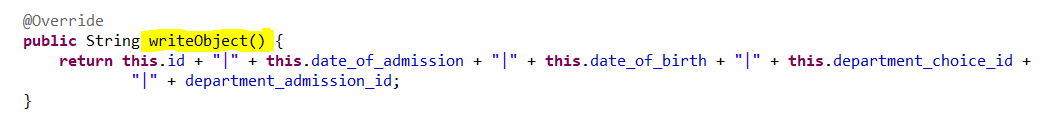
Description automatically generated A white background with black text

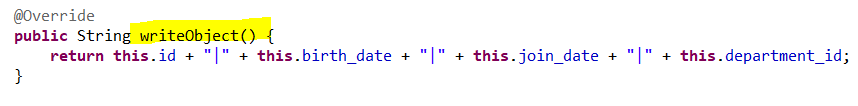
Description automatically generated

1. **Polymorphism**

Polymorphism enables objects to be considered instances of their parent class, rather than their own class. It enables a single interface to be used for a wide variety of tasks or actions.

**In the bellow example,** I have the “writeObject” method in my system that is overridden in all subclasses such as “Employee”, “Department”, “StudentPerformance”, “StudentCounselling” to provide a certain implementation in each class.

A computer code with text

Description automatically generated with medium confidence

1. **Abstract**

Abstraction is the process of hiding complicated implementation details and displaying only the object's fundamental characteristics.

**In the bellow example,** the “DBObject” abstract class defines a standard interface for its subclasses that implements the “writeObject” method while abstracting the specifics of how every type of object is written.

A screenshot of a computer code

Description automatically generated A computer code with text

Description automatically generated

**Class relationships**

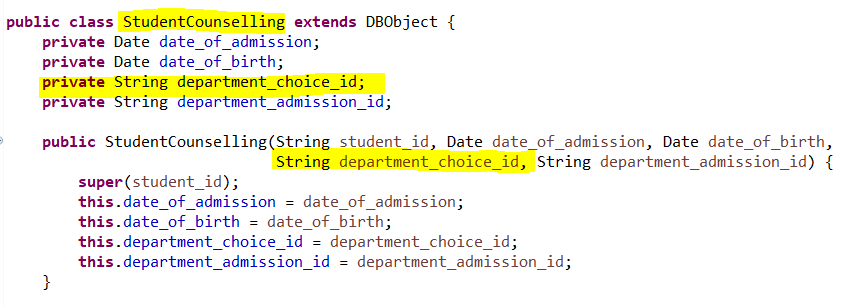
Relationships establish how classes interact and rely on one another, which helps to structure my project properly.

1. **Aggregation**

Aggregation is a type of association relationship in which objects have separate lifecycles but share ownership. However, the child object can exist independently of its parent.

A computer code with yellow text

Description automatically generated



1. **Composition**

Composition is a more robust form of aggregation. The child object has no independent lifecycle in this case, and if the parent object breaks down, the child object is destroyed with it.

A computer screen shot of text

Description automatically generated

A yellow text with black text

Description automatically generated

A yellow and black text

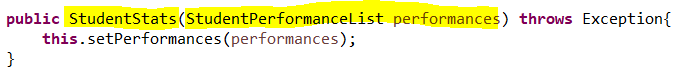
Description automatically generated

A computer screen shot of text

Description automatically generated

1. **Dependency**

Dependency is a less strong connection in which one class relies on another to function.



1. Determine the design Pattern for each of Creational, Structural and Behavioral pattern types. M1

A variety of design patterns are routinely implemented in software development. These patterns can be divided into 3 main groups:

|  |  |  |
| --- | --- | --- |
| **Creational Design Patterns** | **Structural Design Patterns** | **Behavioral Design Patterns** |
| **factory pattern** | **facade pattern** | **Chain of responsibility pattern** |
| **singleton pattern** | **Adapter Pattern** | **Iterator pattern** |
| **Prototype pattern (Clone)** | **Bridge Pattern** | **Template pattern** |
| **Builder Pattern** | **proxy Pattern** | **Strategy pattern** |

**Creational Design Patterns:**

The creational design pattern abstracts the instantiation process. They help to make a system independent of how its objects are formed, constructed, and displayed. This pattern uses inheritance to modify the class that is instantiated, while an object creational pattern delegated instantiation to another object. Creational patterns allow for a great deal of flexibility in terms of what is created, how it is created, who creates it, and when. The main purpose of this design pattern is to deal with object creation methods, attempting to generate objects in a manner appropriate for the situation.

**The types of creational Design Patterns:**

**The factory pattern:** this pattern defines interface to generate objects, but subclasses can change the type of objects that are created, encouraging loose dependency (coupling) and extensibility. Also, a class needs to generate objects without knowing what type of object will be created. This Pattern handles this by introducing a separate factory class for object creation. Clients communicate with the factory class to generate objects, and the factory class selects the subclass to be instantiated based on the client's request.

**The singleton pattern:** this pattern assures that a class has only one instance and gives a global interface to that instance. It implements this by creating a static method or attribute to access the one instance and making the class's constructor private, prohibiting other objects from generating instances of the class directly. This design is appropriate when only one object is required to coordinate actions throughout the system, such as a configuration administrator or logging service.

**Prototype pattern (Clone):** this pattern generates new objects by cloning an existing object, referred to as the prototype. In programming, the existing object is frequently cloned to generate new instances. It implements this by creating a prototype interface that has a method for cloning itself. Then, define a concrete class that implements this interface and include cloning logic. Users construct new objects by copying an existing object, which eliminates the need for specific constructors or sub classing. This pattern improves testability by streamlining object generation and encouraging modular testing approaches. Also, it shortens development iterations by allowing for the rapid production of new object instances, encouraging agile development and faster reaction to modified requirements.

**The Builder Pattern:** this Pattern separates the creation of a complicated object and its representation, enabling the same construction to produce several representations. To use the Builder Pattern, a director class manages the construction process through a builder interface. Concrete builder classes implement this interface by providing methods for creating the object's various components. The director class directs the construction process by invoking these methods in a predefined sequence. When building is finished, clients can pick up the completed thing from the constructor. This pattern is beneficial when an item requires extensive construction with numerous possible options or configurations, since it allows for a more flexible and readable way to create objects.

**Structural Design Patterns:**

Structural Design Patterns focus on how classes and objects are combined to generate larger structures. Structural class patterns apply inheritance to combine interfaces or implementations. Analyze how multiple inheritances combine two or more classes. The result is a class with the attributes of its parent classes. The main purpose of this design pattern is to deal with the way classes and objects are organized or composed.

**The types of Structural Design Patterns:**

**The facade pattern:** this Pattern integrates a subsystem's interfaces. It specifies a higher-level interface which makes the subsystem faster and easier to use. This design hides the subsystem's intricacies and provides a simpler interface with which clients can communicate, eliminating constraints, dependencies and making the system easier to comprehend and operate.

**The Adapter Pattern:** this Pattern enables objects with not matching interfaces to collaborate. It serves as a connection between two different interfaces, transforming the interface of one class into the interface that a client expects. This approach is useful for integrating current or external code that is not compatible with the interface needed by other parts of the system.

**The Bridge Pattern:** this Pattern separates an abstraction from its implementation, enabling both to change independently and allowing them to be changed or extended independently. It is beneficial when we want to prevent an ongoing connection between an abstraction and its implementation. Also, when we have different implementations that must be dynamically combined.

**The proxy Pattern:** this Pattern creates a replacement or placeholder for another object to control access to it. It functions as a sort of placeholder for an object, enabling the proxy to manage access to the original object. This approach is excellent for adding access control, logging, and monitoring without modifying the actual object's code.

**Behavioral Design Patterns:**

Behavioral Patterns deal with algorithms and the distribution of responsibility among objects. Behavioral patterns define not only patterns of objects or classes, but also patterns of interaction between them. These patterns represent complicated control flow that is difficult to understand at runtime. The main purpose of this design pattern is to deal with how classes and objects communicate and interact with each other.

**The types of Struct Behavioral Design Patterns:**

**Chain of responsibility pattern:** this pattern enables numerous objects to deal with a request without the sender knowing which one will handle it. It creates a chain of objects, with each one containing a reference to the following one in the chain, so this pattern helps to distribute responsibility. When a request is sent out, it is forwarded down the chain until it is handled by an object, or the chain reaches its end.

**Iterator pattern:** this pattern allows you to access the components of a collection continually without providing the fundamental representation. It is implemented by defining an iterator interface that includes methods for accessing elements, and collection objects contain methods for generating iterators. This pattern is useful for exploring a number of objects without sharing their internal structure.

**Template pattern:** this pattern defines the basic structure of an algorithm in a method, with some phases delegated to subclasses. It includes a template method that describes the structure of the algorithm and abstract methods that subclasses must implemented in order customize the algorithm's behavior. This approach is helpful when you wish to establish a general algorithm structure while allowing subclasses to offer unique implementations for certain stages.

**Strategy pattern:** this pattern defines a set of algorithms that are described as separate classes that implement a shared interface or inherited from the same superclass, then encapsulates them and makes them reusable. It enables the algorithm to change independently of the clients that use it. This pattern entails developing a collection of strategy classes that execute an interface that is common and allows clients to select the right strategy at runtime.

1. Design Patterns have a positive impact on Object Oriented Design, Analyze the impact of Design Patterns on object-oriented design listing advantages and strengths, support your findings with evidence and examples. D1

**Object-oriented programming [OOP]** is a common way to create complex software, but it is sometimes confusing, mainly when you have repeated design problems. In this case we **design patterns**, which have positive impact on OOP and enable developers to deal with these problems.

The design patterns are considered guidelines which help developers to deal with software problems in an efficient and flexible way and show the interactions and relationships among classes or objects. Also, they help developers use solutions together to design software that is easy to maintain, adapt, and reuse solution for frequent or common problems in software design. This leads to higher-quality software and quicker development cycles.

The main goal is to enhance the development process by offering well-tested and reliable, validated development, and design paradigms. Also, Design patterns are approaches that independent of programming-language for solving several common problems. This means that a design pattern reflects an idea rather than a specific implementation. It is not necessary to always use design patterns in my projects. Design patterns aren't intended for project development. Design patterns are intended to facilitate common problem resolution.

**The benefits of design patterns:**

* Design Pattern focuses on generating solutions that are not related or specific to a single project but can be reused across multiple projects. This encourages code reuse and consistency.
* Patterns provide ready-made solutions to frequent problems, which speeds up the development process. Developers can concentrate on specific application logic rather than low-level design issues.
* Design patterns help to create a flexible and well-organized code base, making it easier to fix and update. Changes in particular patterns can be applied locally without affecting the overall system.
* Design Patterns offer an organized approach for developing, making it simpler to grow a system using comprehensive solutions to new elements or modules. This provides a consistent architecture throughout the application.
* Patterns provide best practices and established solutions, leading to improved code quality. So, they enable developers to use time-tested techniques that have been developed and approved by the software engineering society resulting in higher-quality, more reliable code.
* Design Patterns simplify maintenance and debugging by isolating specific sections of code. Organizing code into clearly defined patterns makes it easier to find and solve issues. Each pattern covers a specific function, making the system more predictable and more straightforward to debug.
* Design Patterns improve adaptability to changing requirements such as Strategy and Factory Method, have been developed to be adaptable, enabling new behaviors or products to be implemented with little impact on existing code.
* **Here are examples and evidence related to how design pattern impact on OOP Positively:**

**The singleton design pattern** in object-oriented programming is essential to make sure that the class has one instance and gives global access to this instance. Thers is always one instance that can be accessed globally regardless of how often we create instances for the class. **An example of this pattern** is a government in the country. This government represents a singleton class and there is only one government for a specific country. When the government is created, we are really creating the only instance of the government for the country. The interaction with the government does not mean creating another government each time we need something, but instead we access the current government. Because there is only one instance from class, the actions and decisions of the government are centralized and consistent. Also, this can prevent inconsistencies that may appear from creating multiple instances of the class.

**This pattern has some advantages** such as ensuring a single instance, having global access point, and ability to make consistent decisions making. These advantages enhance the reliability and efficiency of using this pattern in software design, which ensures that essential components in the system are managed in a centralized and controlled way.

**The factory design pattern** in object-oriented programming. This pattern creates the interface for creating an object, but it makes subclasses change the type of the object that is created. It gives an abstract method, as well as the subclasses of the interface which implement this method to create objects of a certain type. **An example of this pattern** is a vehicle manufacturing company. This company produces several types of vehicles such as bikes, cars, such motorcycles. The company represents a factory class which creates an abstract method. The products of the company are different subclasses for each product, which is responsible for a specific type of vehicle. These subclasses use the abstract method when we need to separate the creation of the object from its usage.

**This pattern has some advantages** such as **decoupling objects** which is the ability to separate the creation of the object from its usage.

**Flexibility:** it can enable subclasses to change the type of object and make addition and modification of the types of the object easier with no need to edit the main code.

**Reusability of the code:** it enhances reusing the object creation logic, encourages maintainability, as well as reducing duplication.

**Scalability:** it enables expansion of the product line by adding new objects, adapting the future growth and changes in needs and requirements.

**The Proxy design pattern** in object-oriented programming offers a substitute element of another object in order to control access to this object, which means that it acts as representative or intermediary of another object. This helps to add more functionality such as access control, caching, or slow loading with no need to change the underlying object. **An example of this pattern** is the access system of a smart card. In this scenario, the access control system is the real object. This system authenticates the customers and either denies or grants access to the system depending on the credentials. The proxy is the guard which stands in the middle between the access control system and the customers. This security guard can check credentials of customers and verify identities and then decide whether to grant access or not. So, customers interact with the guard instead of interacting with the access control system. The security guard offers more services like verifying IDs, controlling flow of clients, logging access without the need for users to be aware of the previous operations. This is like the way in which a proxy in a software offers access control, caching, or slow loading with no need to change the underlying object.

**This pattern has some advantages** such as controlling access by acting as representative which ensures that access to the system is restricted and managed to enhance security. Also, it offers more functionality like verifying IDs, controlling the flow of clients, logging access without changing the main code. This makes it flexible to apply more features such as access control, caching, or slow loading. In addition, it enhances security because it verifies users credentials and then decides whether to grant access or not. Hence, it reduces the risk of illegal access to the access control system. It separates clients from the real system, which allows for changes with no need to modify the main code. This enhances maintainability and modularity of the system. **Finally,** implementing the previously mentioned additional features leads to better performance by postponing the loading of resources until users need them or by storing data frequently and locale, which enhances the efficiency of the real system.

**The adapter design pattern** in object-oriented programming acts as a bridge between two different interfaces. It helps objects with these uncompilable interfaces to work effectively together. This makes them work in agreement with no change of the actual class. This pattern includes one class (the adapter class) which is accountable for joining functionalities of different interfaces. **An example of this pattern** is the electrical outlet which are different types in different countries. The laptop is a client that is designed for an outlet in the USA. It is typing A and B that have flat pins. The European outlet is considered the target interface which is type C and F that have round pins. The target interface is the interface that the users need to connect with. In this case **the adapter** is the physical device that is responsible for converting the US outlet into the European outlet.

**This pattern has some advantages** such as compatibility because it helps uncompilable interfaces to work effectively together, which enables interaction and communication without making changes to the current interfaces or classes. Also, it offers flexibility because it allows current objects to be used again with different interfaces. This reduces the need to create new objects or edit the current objects. It allows interoperability between different systems with incompatible interfaces to work together, interact, as well as exchanging data effectively. Finally, it encourages code reusability by giving a reusable solution to adapting interfaces, which deploys effort and time.

**The strategy design pattern** in object-oriented programming, which defines a group of algorithms (classes) and makes them interchangeable, as well as encapsules them. It hides the details of algorithms implementation from the details by separating the algorithm from users. As a result, users can choose one class from a group of classes at runtime.

**An example of this pattern** is a navigation application which offers different routing choices to users. This application is the context, which needs various routing options, as well as representing the user. The routing algorithms represent classes such as Avoiding Traffic Route, Shortest Route, and Scenic Route. The users can choose their favorite routing strategy. So, if they chose Avoid Traffic Route, the application will use this particular routing algorithm. This authentic example explains who this pattern helps users to choose a particular routing algorithm with no need to understand the internality of application or algorithm. Also, this pattern is used in situations which have different algorithms (classes) for a certain task, and the user needs to choose one of these algorithms at runtime.

**This pattern has some advantages** such as its **flexibility** because it helps users to choose the algorithm that is most suitable for their needs from different algorithms (classes) at runtime. This improves user experience, and it is appealing for different preferences. Also, it is **easy to maintain** and **improves scalability** because this pattern encapsulates every algorithm to separate strategies. In this pattern, adding a new algorithm or editing the current algorithm is done independently from other algorithms. As a result, it makes understanding and maintaining the code easier, as well as reducing the danger of bugs. In addition, it **encourages reusability** of the code because it encapsulates the algorithms into strategies which can be reused. Also, they can be shared among different applications or contexts that need similar functionality, which reduces duplication. Finally, it **improves testability** because it enables easier testing of isolated strategies to ensure their effectiveness and correctness, which enhances the quality of the whole software.

**The template design pattern** in object-oriented programming defines the main structure of the algorithm and postpones some steps into subclasses. This helps the code to define some steps of the algorithm again without changing the internality of the algorithm. This encourages reusability of the algorithm by offering a common structure for a group of related algorithms.

**An example of this pattern** is preparing a hot drink. The template method is the common strutter of making the hot drink. It defines the algorithm which includes essential steps such as boiling water, brewing drinks and serving them. While other optional or extra steps such as adding milk or sugar are left to be done or overridden by subclasses when needed.

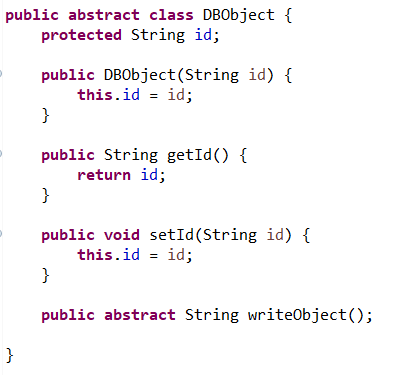
**This pattern has some advantages such as** promoting **code reusability** because it provides a common structure for a group of related algorithms (template method), which reduces redundancy. Also, it provides **flexibility** because it helps subclasses to override some steps of the algorithm without changing the main structure, which keeps **consistency** of the whole process. It is easy to maintain because updates and modification to the main structure can be done in the same way across all subclasses, which ensures consistency and easy maintenance. Finally, it hides implementation details from users because it encapsulates the common structure of the algorithm in the template method, which encouraged modularity, abstraction, as well as helping customers to use the algorithm without understanding its internality.

1. Code 1. p3
2. Code 2. M3
3. Evaluate how the use of design pattern for enhancing application development of your code (provide evidence) from Task 8. D3

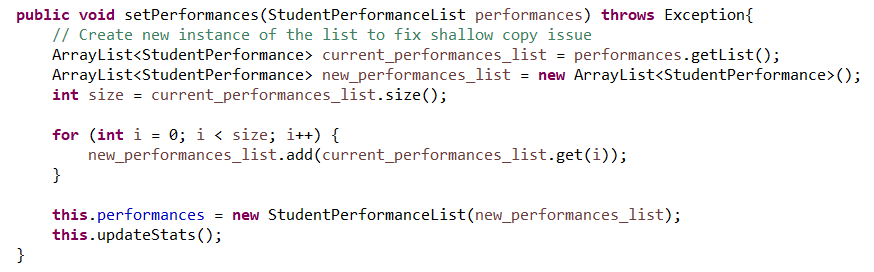
**Clone:**

**Before applying the clone:**

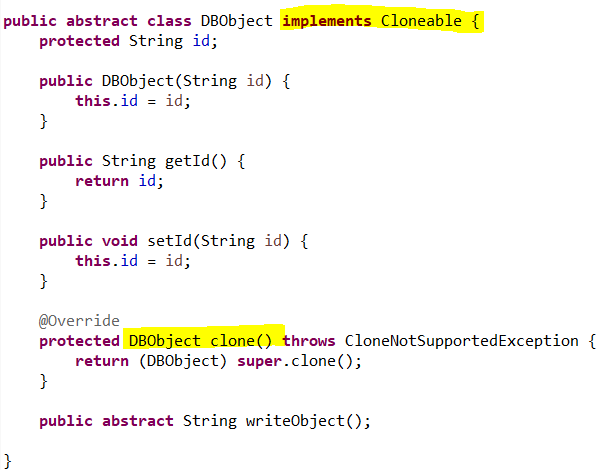
**DBObject Class:**



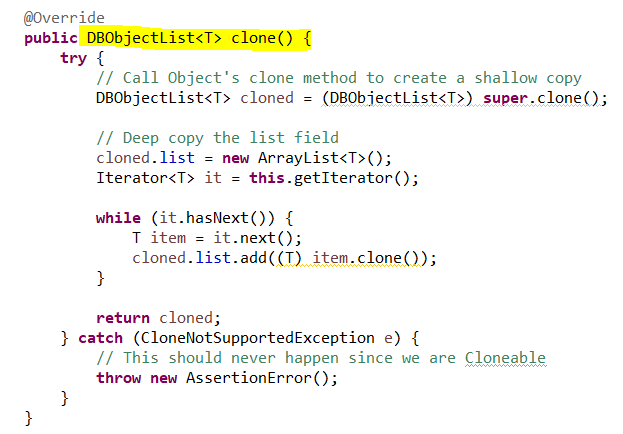
**StudentStats Class:**



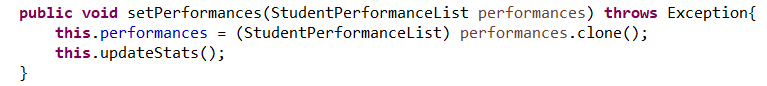
**After applying the clone:**







**StudentStats Class:**



**Evaluate 🡪**

**Flexibility 🡪**

**Without the clone () method,** creating duplicates of an object needed manually instantiating new objects and replicating their properties. This approach is stiff and error-prone, particularly when dealing with complicated objects.

**With the clone () method,** I can easily and efficiently make clones of objects. It improves the adaptability of your code by allowing you to quickly produce new instances without thinking about the complicated details of each object's type or properties. The StudentStats class can simply clone DBObjectList objects, which simplifies complex calculations that involve object duplication.

**Maintainability 🡪**

**Without the clone () method,** manually duplicating object properties can result in duplicated code throughout the project. Any modifications to the object's structure would involve updating all instances where manual copying happens, making maintenance difficult.

**With the clone () method,** this method keeps the copying logic within the object itself. Updates to the object's structure require only to be changed in the clone () method, making maintenance easier and lowering the possibility of problems. This pattern also helps make sure that all object copies are exact and consistent.

**Readability 🡪**

**Without the clone () method,** manually copying properties typically results in longer and more difficult-to-read code, particularly when dealing with several attributes or nesting objects.

**With the clone () method,** using this method helps the code become clearer and more understandable. It abstracts the cloning mechanism, letting developers emphasize higher-level reasoning rather than the specifics of object copying. The StudentStats class illustrates this by utilizing the clone () method to produce a copy of the studentPerformanceList object.

**Usability 🡪**

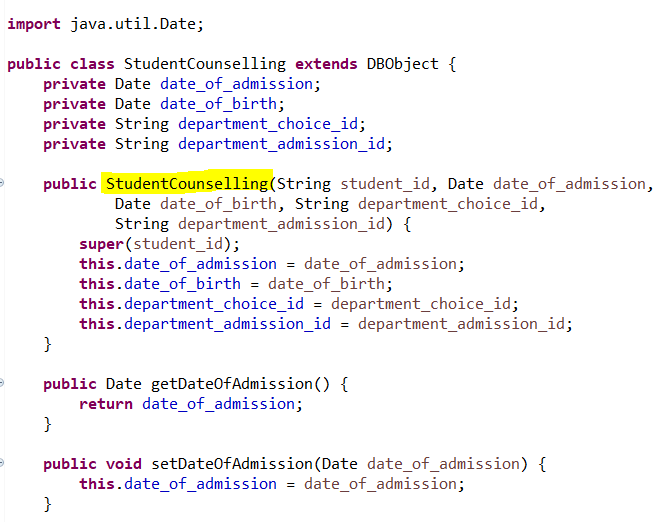
**Without the clone () method,** without a consistent cloning technique, different areas of the software may use copying in different ways, resulting in inconsistencies and potential problems.

**With the clone () method,** the clone () method duplicates objects in a standardized manner, encouraging consistency throughout the code base. It ensures that all cloned objects are produced appropriately and follow the same logic. This technique is very useful in cases where objects must be copied repeatedly, or deep copying needs to be performed.

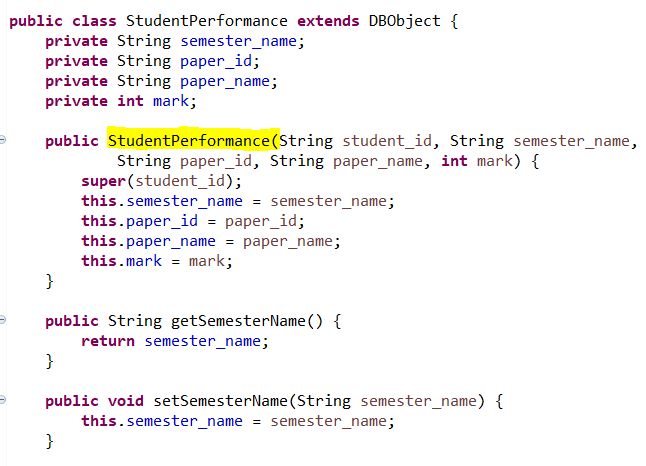
**Builder:**

**Before applying the builder:**

**StudentCounselling Class:**



**StudentPerformance Class:**

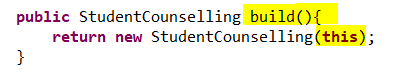


**After applying the builder:**

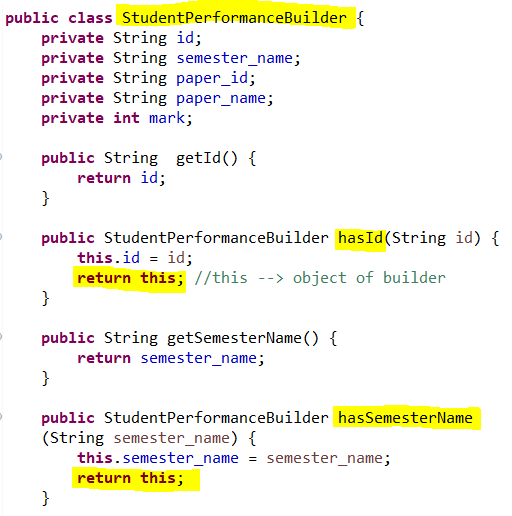
**StudentCounsellingBuilder Class:**

A computer code with text and images

Description automatically generated



**StudentPerformanceBuilder Class:**



A close-up of black text

Description automatically generated

**Evaluate 🡪**

**Flexibility 🡪**

**Without the builder pattern,** creating StudentCounselling and StudentPerformance objects required giving various parameters directly to their constructors. This approach was less adaptable because any changes to the object's structure required altering all instances in which the object was produced.

**With the builder pattern,** the Builder Pattern enables the step-by-step building of complex items. The StudentCounsellingBuilder and StudentPerformanceBuilder classes include methods for configuring specific properties before object creation. This approach adds flexibility by allowing for selective attribute setting and easy change of object creating without modifying client code.

**Maintainability 🡪**

**Without the builder pattern,** the direct use of constructors for object creation might result in code that is difficult to maintain, particularly when the number of attributes increases or changes over time. Any change to the constructor parameters requires modifying all constructor calls within the source code.

**With the builder pattern,** this Pattern increases maintainability by splitting construction behavior within builder classes. Changes in object attributes must be changed just in the builder class, not everywhere the object is created. This lowers the maintenance cost and makes the code more adaptable to changes.

**Readability 🡪**

**Without the builder pattern**, Constructor calls with several parameters might be challenging to read and understand, especially if the parameters are not named or are the same type. It may be unclear what each parameter signifies.

**With the builder pattern,** this pattern improves readability by offering a natural interface for object building. Each method in the builder class clearly states which attribute is being set, making the code more self-documenting and understandable.

**Usability 🡪**

**Without the builder pattern,** creating complicated objects required providing all relevant arguments in the correct sequence, which might result in errors and make the code less user-friendly.

**With the builder pattern,** this Pattern makes object building easier by including default values and optional options. It simplifies the creation process and reduces errors. The builder pattern allows users to simply construct objects without caring about parameter ordering or missing optional values.

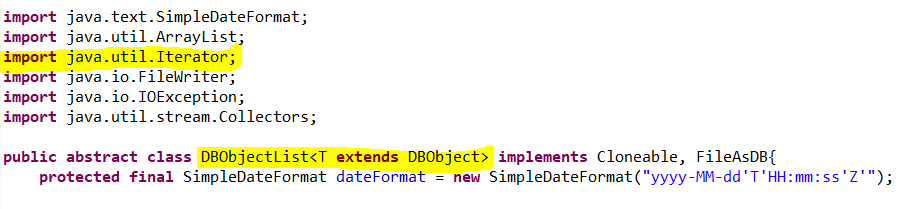
**Iterator:**

**Before applying the iterator:**

A computer code with colorful text

Description automatically generated

**After applying the iterator:**



A screen shot of a computer code

Description automatically generated

**Evaluate 🡪**

**Flexibility 🡪**

**Without the iterator pattern,** to access elements of the DBObjectList class, you needed to interact directly with the internal list structure. This approach was stiff and limited in adaptability, as any changes to the internal list structure could force adjustments to the client code that uses the list.

**With the iterator pattern,** The DBObjectList class offers a standardized method for accessing its elements without showing the underlying representation. This adds flexibility by allowing the underlying structure to be modified without changing the external code that iterates through the collection.

**Maintainability 🡪**

**Without the iterator pattern,** direct access to the internal list structure reduces the code's maintainability. Modifications to the list's implementation may cause errors in any code that uses it directly. Also, this method results in code duplication if multiple areas of the codebase must iterate over the list.

**With the iterator pattern,** this Pattern increases maintainability by centralizing iteration logic. The iterator () method provides a standardized way of accessing elements, reducing the need for repeated iteration code while making it easier to control changes to the list's structure.

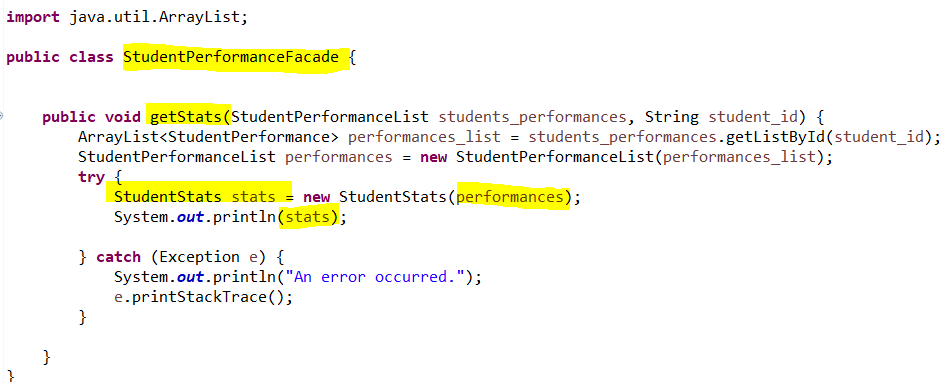
**Readability 🡪**

**Without the iterator pattern,** manual iteration of the list with “**for loop** “or other techniques can result in complex and error-prone code. Iteration logic might be difficult to understand, especially if it is distributed throughout the source code.

**With the iterator pattern,** The Iterator Pattern improves readability by giving a simple and straightforward way to iterate across a list. The usage of the iterator () method or an extended “**for-each loop**” makes the code more understandable and easier to follow.

**Facade:**

**After applying the Facade:**



**Evaluate 🡪**

**Flexibility 🡪**

**Without the Façade pattern,** Users of the StudentPerformanceList and StudentStats classes had to interact directly with one other. This technique is less flexible since it tightly couples the user's code to the construction specifics of the classes. Any changes to the core logic of StudentPerformanceList or StudentStats would necessitate adjustments to all client code.

**With the Façade pattern,** this Pattern introduces the StudentPerformanceFacade class, which offers a streamlined interface for interacting with both StudentPerformanceList and StudentStats. This separates the client code from the implementation details of these classes, allowing for internal modifications that do not affect the client code. This increases the system's flexibility by allowing the underlying implementations to change independently of client interactions.

**Maintainability🡪**

**Without the Façade pattern,** direct interaction with the StudentPerformanceList and StudentStats classes makes the code more difficult to maintain. Any modifications to the functions or structure of these classes require updates to all client programs, resulting in higher maintenance costs and an increased chance of defects.

**With the Façade pattern,** The Facade Pattern enhances maintainability by consolidating the interaction logic into the StudentPerformanceFacade class. Changes to StudentPerformanceList or StudentStats require only adjustments to the facade class, not the client code. This lowers maintenance costs and makes the system easy to manage.

The Facade Pattern enhances maintainability by consolidating the interaction logic into the StudentPerformanceFacade class. Changes to StudentPerformanceList or StudentStats require only adjustments to the facade class, not the client code. This lowers maintenance costs and makes the system easy to manage.

**Usability 🡪**

**Without the Façade pattern, to** use StudentPerformanceList and StudentStats efficiently, users must first understand how they work internally. This raises the learning curve and renders the code less user-friendly.

**With the Façade pattern,** this Pattern facilitates usage by offering a simple interface for typical operations. Users can perform operations without having to comprehend the underlying complexity, keeping the code more usable and minimizing the possibility of errors. The facade encapsulates the logic essential to manage student performance and statistics, hence improving the user experience.

**Strategy:**

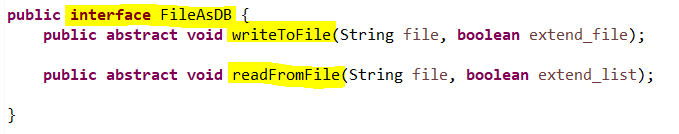
**Before applying the Strategy:**

These methods are repeated in all classes List.

A screenshot of a computer code

Description automatically generated

**After applying the Strategy:**





**Evaluate 🡪**

**Flexibility 🡪**

**Without the Strategy pattern,** all list classes used the same file operation techniques. This technique is rigid since any modification to the file operation logic will necessitate adjustments in several places.

**With the strategy pattern:** this Pattern provides a FileStrategy interface and specific implementations such as FileAsDB. This separates the file operation logic from the list classes, allowing many file operation strategies to be used interchangeably. This provides flexibility because new strategies can be implemented without changing the list classes.

**Maintainability 🡪**

**Without the Strategy pattern,** repeating file operation methods across all list classes increases maintenance expenses. Any changes to the file operation logic must be reproduced across all classes, increasing the chance of mistakes and inconsistencies.

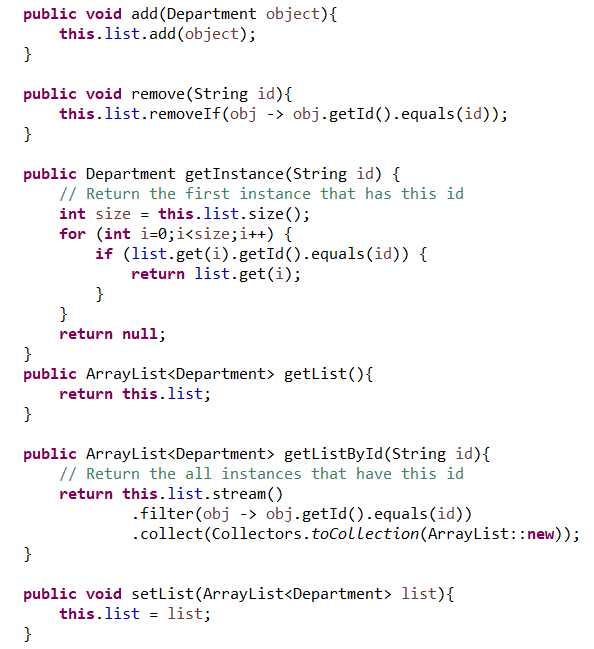
**With the Strategy pattern,** this Pattern organizes the file operation logic into strategy classes. Changes to file operation techniques can only be made in one location, simplifying maintenance and lowering the possibility of bugs. This method makes the codebase more manageable and extensible.

**Usability 🡪**

**Without the Strategy pattern,** users must deal directly with the file operation methods for each list class. This technique is less user-friendly because it necessitates understanding the implementation specifics of each class.

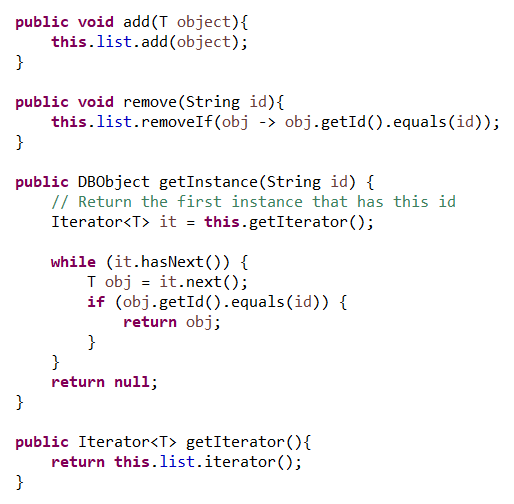
**With the Strategy pattern,** this Pattern makes usage easier by providing a clear interface for file operations. Users can quickly switch between multiple file operation strategies without changing the list classes, making the code very user-friendly and reducing the possibility of errors. The pattern encourages code reuse and consistency throughout the application.

**The bellow code is repeated in all classes List such as EmployeeList, DepartmentList, StudentPerformanceList, StudentCounsellingList before applying the design patterns.**



**After Applying design patterns:**

**The bellow code is applied in one class which is DBObjectList class and other classes inherited from it.**



A computer code with text

Description automatically generated with medium confidence

**Evaluate 🡪**

**Flexibility 🡪**

**Before 🡪**

The code contained duplicated methods and logic for different list classes (EmployeeList, DepartmentList, StudentPerformanceList, and StudentCounsellingList). This rendered the codebase inflexible because updates to the common logic required several modifications.

**After 🡪**

The code structure became more modular and flexible after implementing design patterns. The shared functionality was contained in strategy classes and iterators, allowing for simple modifications and extensions without affecting the client code. This enhanced flexibility enables the system to respond to changes more effectively.

**Maintainability 🡪**

**Before 🡪**

Repeated code across numerous classes resulted in too much maintenance costs. Any changes or bug fixes to the shared logic had to be repeated in each class, which increased the possibility of errors and inconsistencies.

**After 🡪**

Design patterns like the Strategy Pattern and the Facade Pattern centralized shared logic, making the code easier to maintain. Updates to the logic now only need to be made in one location, lowering maintenance costs and improving consistency. This strategy streamlines the maintenance process and improves code reliability.

**Readability 🡪**

**Before 🡪**

The use of code that was repeated made the source code more complex and difficult to read. Each list class has similar methods, complicating the code and making it difficult to understand the general structure and flow.

**After 🡪**

The use of design patterns increased readability by creating concise and straightforward interfaces for common actions. For example, the Iterator Pattern standardized collection traversal, whereas the Facade Pattern facilitated interaction with complicated subsystems. This makes the code more intuitive and easier to follow, increasing developer productivity and lowering the learning curve.

1. When to use creational, structural, and behavioral Design patterns, Provide an example of. P4

* **Creational design patterns:**

**When to use 🡪**

This pattern is used when we want to deal with object creation techniques, hide the logic of instantiation, trying to create objects that are appropriate for the scenario, and make the program doesn’t depend on how its objects are created, formed, and represented. Also, this pattern is used when you want to separate the object generation process from other parts of your code or when you need additional flexibility in object creation.

**Example 🡪**

1. Assume you have a class reflecting a company's CEO. No matter how many times you visit the office of the CEO, you will always engage with one particular person. This is equivalent to the **Singleton pattern**, which uses a single instance of a class within the program.
2. If a user produces a circle with specified parameters such as radius, size, and color, they can clone it to make more circles with the same attributes. This allows users to quickly construct various shapes without having to define all their properties again.   
   **The Clone pattern** enables users to clone current shapes as prototypes and change them as needed, streamlining the process of producing new shapes in the application.
3. Assume you're developing an application and need to create several characters such as archers, warriors, and mages. Instead of directly constructing instances of these classes in your application logic, you could use a **factory function** that determines which class to instantiate based on a certain parameter, such as the player-selected character type.
4. **The builder design pattern** is used to create complex objects which have many optional configurations or parameters. For example, if we need to a system to create customized laptops. Every laptop may have different features such as the type of processor, size of the RAM and screen, storage capacity, and touch screen keyboard. In this case, the builder pattern can be used instead of constructor with different parameters because it can be impractical and hard to maintain. The builder class gives functions which can set every parameter individually. This can help clients to only specify the parameter that is needed, which will result in a flexible and readable construction process. This pattern is ideal for objects with many configurations’ options and different parameters combinations.

* **Structural design patterns:**

**When to use 🡪**

This pattern is used when we want to deal with class or object structure and ensure that if one component of the system changes, other parts of the system do not have to. This pattern tries to simplify the structure of the program by determining the relationship between the objects.

**Example 🡪**

1. The main e-commerce program contains a number of subsystems, including order and payment processing, as well as inventory management. Client interactions can be simplified by implementing a Facade. This Facade provides a uniform interface that brings together the functionality of different subsystems. Clients are able to communicate with the Facade without having to learn the complexity of each subsystem separately. For example, a customer wants to buy a product. Instead of communicating with the payment subsystems, inventory, and order independently, they use a single interface provided by the Facade. The Facade manages all the complicated nature internally, such as placing purchases, checking availability of goods, and accepting payments.
2. An adapter design pattern is used to make two different interfaces that work together. For example, when we have a copy of a software component that has a different interface, but the current system depends on the old interface. We can use an adapter class that can translate requests on the old interface to the new interface instead of modifying the current system. This ensures seamless integration with no need to change the existing code.
3. Proxy design pattern is used to **control** access to the object. For example, if we have a large file that takes a long time to load, we can use a proxy object which can load the file only if it is necessary, instead of immediately loading the file. This process improves performance by postponing the leading process until it is needed. Also, it provides extra functionality such as access control and caching.
4. Bridge design pattern is used to separate the abstraction from its implementation, which allows them to differ independently. For example, if we have drawing applications with different shapes and displaying methods, a separate hierarchy for displaying methods and shapes can be created by using the bridge pattern This allows combing them dynamically at runtime with no need to modify the current code.

* **Behavioral design patterns:**

**When to use 🡪**

This pattern is used when we want to define how the objects interact with each other or wrapping algorithms and responsibilities. It is not only explaining the pattern between classes and objects. Also, it defines how they communicate with each other, so it is used to define how things interact in a flexible and contained manner.

**Example 🡪**

**Chain responsibility:** this pattern is used to separate the senders and receivers of requests. For example, when er have a customer support system in which queries of customers are dealt with by many support teams depending on their complexity. A chain of responsibility is implemented and so each team can pass it to the next team in the chain or can handle the query if it is his responsibility, instead of each team clearly knowing which query to handle. This offers an extendable and flexible approach to dealing with requests. This pattern is particularly used when the request of the customer is ambiguous, and not known by developers.

**Strategy:** this pattern is used for defining a group of algorithms, encapsulating each algorithm and making switch between them without modifying anyone. For example, if we are creating a sorting algorithm in the system, the strategy pattern can be used to encapsulate various sorting algorithms such as merge sort, bubble sort, and quick sort as separate classes instead of depending on one sorting algorithm. This helps to switch between classes at runtime dynamically depending on certain parameters.

**Iterator:** this pattern is used to access items of a collection by order without finding out their internal representation. For example, when we have a music player, and we want to go over the playlist to play the songs one by one, we can use this pattern to offer a unified interface to iterate over the playlist, instead of discovering internal representation of the playlist such as linked list or array list.

**Template:** this pattern defines the basic structure of an algorithm in a method, with some phases delegated to subclasses. It includes a template method that describes the structure of the algorithm and abstract methods that subclasses must implemented in order customize the algorithm's behavior. This approach is helpful when you wish to establish a general algorithm structure while allowing subclasses to offer unique implementations for certain stages. For example, if you have a recipe when baking a cake. The recipe outlines the processes for topping the cake, mixing ingredients, and baking the batter. However, you can change certain aspects, such as adding new flavors or decor, to fit your tastes. The recipe provides a pattern to follow while leaving opportunity for innovation.

In conclusion, creational patterns concentrate on object creation, structural patterns on object composition, and the third design pattern, which is behavioral patterns concentrates on object interaction. Each pattern serves its own purpose and may be applied in a variety of situations to enhance the architecture, flexibility, and scalability of the code.

1. 12. Choose a suitable design pattern for the following cases and justify your choice of Design pattern. M4 & D4

1. The constructor of a class have many parameters (6 or more) some of are mandatory the others

are optional.

**The suitable design pattern is: [ builder design pattern]**

**Justification:** When constructing a complex object with several parameters, this pattern gives a step-by-step systematic approach. This makes the creation process easier to handle and less dangerous than using a single constructor with a big list of parameters. The Builder pattern divides the constructing of an object from its actual representation. This separation makes the creation process independent of the exact representation of the thing being produced. In our example, it enables us to develop a builder class with methods for configuring particular parameters, separating away the complexities of object creation. Also, it smoothly handles instances in which certain parameters are necessary and others are optional. It enables user code to provide only the necessary parameters during object building and includes ways for setting additional parameters as needed. This flexibility improves the code's readability and maintainability. Several implementations of this pattern make the produced object immutable after it is formed. This offers the security of the thread and prevents any unwanted changes to the object after creation. In addition, it allows us to specify default values for extra parameters that are not provided by the client code. This reduces the load on the user's code and facilitates object formation. Finally, this pattern enhances the codebase's readability and maintainability by offering a clear and structured method for creating objects. It abstracts away the specifics of object building, allowing programmers to focus on code purpose rather than object creation processes.

2. An object is created and used in many different locations in the system and may be created by

many different developers.

**The suitable design pattern is: [ Singleton design pattern]**

**Justification: this pattern** confirms that an object exists only once in the entire system. This is useful when you want to have just one, sharing instance of an object that can be accessible from different locations throughout the system. Because there is one object within the system, its behavior is constant. This can be useful in applications demanding consistent behavior, such as logging, using database, managing shared resources, and caching. Also, it creates a global access point for the instance it maintains. This allows different developers of the system to use the same object without having to provide references or create new instances. The Singleton design isolates the object creation mechanism, allowing developers to quickly modify or expand the Singleton's behavior without affecting other parts of the system. This encourages code maintenance and extension. The implementation of this pattern uses lazy initialization, which means that the instance is created only once it is first requested. This saves resources and guarantees that the object is only instantiated when necessary. Finally, the implementation of this pattern frequently contains measures to maintain thread safety, particularly in multi-threaded settings. This ensures that the Singleton instance is correctly constructed and accessed by numerous threads without any race scenarios or synchronization difficulties.

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

object.

**The suitable design pattern is: [ Proxy design pattern]**

**Justification:** it helps the slow loading of a large object. It carries a reference to the real object and then loads it when the client orders it only, instead of the immediate loading of the whole object. This slow loading helps in keeping resources mainly when we deal with large objects that are not needed immediately. It makes sure that there is no repetition of the large object. When many customers want to access the object, the proxy can control this access and makes sure that one instance of the object is only generated and shared between customers, which will help improving memory usage, as well as preventing unwanted duplication of the available resources. Also, it plays as substitute or placeholder of the real object, as well as providing an interface that can be controlled for accessing it. This helps the proxy to carry out additional functionalities such as caching, validation, access control or logging before the request to the real object is delegated. For example, the proxy can make sure that the object is loaded previously and bring back the cached instance instead of loading the large object again. Finaly, this pattern encourages scalability because it allows the system to deal with big numbers of users effectively. Also, it can prevent overload situations and manage resources effectively because it can control access to the real object.

4. on a shopping site: the user adds items to the basket and by the end on checkout, the user can choose the payment method in runtime: PayPal, Credit Card and so on.

**The suitable design pattern is: [ Strategy design pattern]**

**Justification:** this pattern allows me to divide payment methods into classes like PayPal\_Payment\_Strategy and CreditCard\_Payment\_Strategy. Each of these payment methods belongs to a specific payment mechanism. Also, it allows me to choose between them at runtime. This flexibility allows users to select the method of payment they prefer throughout the process of checkout, without tying the client program to a particular payment solution. This pattern encourages a division of algorithms (classes) by dividing the payment methods into classes. As a result, each class has specific and one function that supports the single responsibility principle. This makes the source code more modular and maintainable, as modifications to implementation of one payment method have no effect on the others. In addition, the Strategy design complies to the Open/Closed Principle, enabling for the simple addition of additional payment methods without changing existing code that makes the codes more extendable and flexible. Each class may be reused across the code. For example, if similar payment methods are utilized in other parts of the code, the payment method classes can be used without duplication. The strategy pattern increases code maintainability by separating client code from payment method implementation specifics.

# References:

* <https://www.geeksforgeeks.org/introduction-to-pattern-designing/>
* <https://www.geeksforgeeks.org/design-patterns-in-object-oriented-programming-oop/>
* <https://www.geeksforgeeks.org/what-are-the-relationships-among-objects-in-ooad/>
* <https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/>
* <https://www.geeksforgeeks.org/encapsulation-in-java/>
* <https://medium.com/@laibamemon94/polymorphism-in-oop-and-java-3644d5aeb81a>
* <https://www.geeksforgeeks.org/design-patterns-understand-the-importance-with-real-life-examples/>