ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Truong Tan Phuc	**Student ID**	GCD210070
Class	GCD1101	**Assessor name**	Pham Thanh Son
Student declaration I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.			
		Student's signature	Phuc

Grading grid

P1	P2	M1	M2	D1	D2

❒ **Summative Feedback:**		❒ **Resubmission Feedback:**
Grade:	**Assessor Signature:**	**Date:**
Lecturer Signature:		

Contents

I. INTRODUCTION 5

II. OOP GENERAL CONCEPTS 5

1. Introduce OOP 5

2. OOP General Concepts 5
A, Encapsulation: 6
B, Polymorphism: 6
C, Inheritance: 7
D, Data Abstraction: 8
E, Classes and Objects 10
F. Access Modifiers 11
III. OOP SCENARIO 12
1. Scenario 12
2. Usecase Diagram 13
3. Class Diagram 15
A. Important Classes Explain 16
B. OOP features 20
C. UML Relationships 24
D. Multiplicity: 26
IV. DESIGN PATTERNS 28
1. Creational pattern 28
2. Structural pattern 33
3. Behavioral pattern 34
V. Design Pattern vs OOP 36
VI. CONCLUSION 39
REFERENCE: 40

Figure 1: Encapsulation OOP 6

Figure 2: Polymorphism OOP 7
Figure 3: Inheritance OOP 8
Figure 4: Abtraction OOP 9
Figure 5: Example Abtraction 10
Figure 6: Classes and Objects 11
Figure 7: Code Access Modifiers Example 12
Figure 8: UseCase Diagram 14
Figure 9: Class Diagram 16
Figure 10: Employees Class 16
Figure 11: Position Class 17
Figure 12: Salary Class 18
Figure 13: Project Class 19
Figure 14: Employee Controller 20
Figure 15: Project Controller 20
Figure 16: Abtraction Explain 21
Figure 17: Inheritance 23
Figure 18: Person Class with Encapsulation 24
Figure 19: Association Unary 25
Figure 20: ASsociation Binary 26
Figure 21: Example For multiplicity 27
Figure 22: Example Creaional - Abstract Factory Pattern UML 29
Figure 23: Example Creaional - Abstract Factory Pattern Code 33
Figure 24: Template Method - Structural Pattern UML 34
Figure 25: Observer Behavioral Pattern 35

I. INTRODUCTION

This report of this project is to use C# to construct a thorough personnel management solution that uses OOP design patterns and principles. The solution will effectively manage personnel data and streamline numerous business processes for a corporation. This project will create a reliable and approachable solution for managing employee information through the diligent use of OOP concepts and intelligent design patterns.

II. OOP GENERAL CONCEPTS

1. Introduce OOP

A programming paradigm known as object-oriented programming is based on the idea that objects should take precedence over action or reasoning. It enables users to build objects according to specifications and then build methods to interact with those things.

The aim of object-oriented programming is to manipulate these objects in order to get the desired outcome.

- Object-Oriented Programming offers several advantages over the other programming models like:
- The precise and clear modular approach for programs offers easy understanding and maintenance.
- Classes and objects created in the project can be used across the project.
- The modular approach allows different modules to exist independently, thereby allowing several different developers to work on different modules together.

2. OOP General Concepts

Here are some general concepts of OOP:

A, Encapsulation:

Encapsulation is an object-oriented programming concept that allows programmers to wrap data and code snippets inside an enclosure. By using the encapsulation program, you can hide the members of one class from another class. It's like encircling a logical item within a package. It allows only relevant information available and visible outside and that too only to specific members.

Encapsulation is implemented by using access specifiers. Access Specifier is used for defining the visibility and accessibility of the class member in C#.

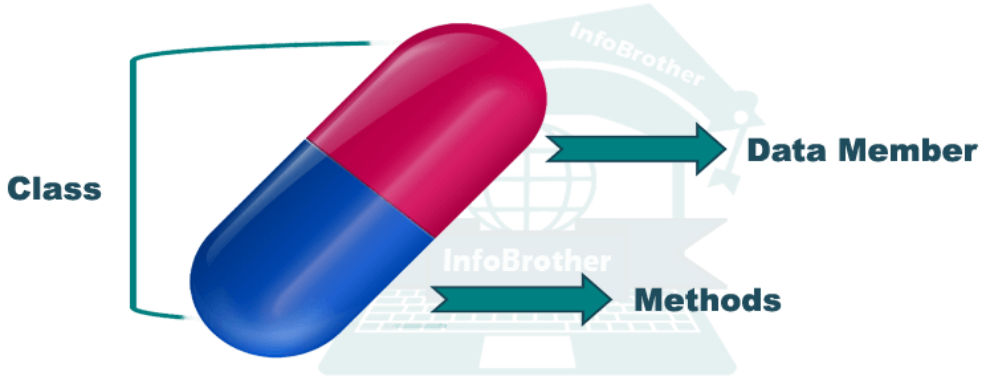


Figure 1: Encapsulation OOP

B, Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

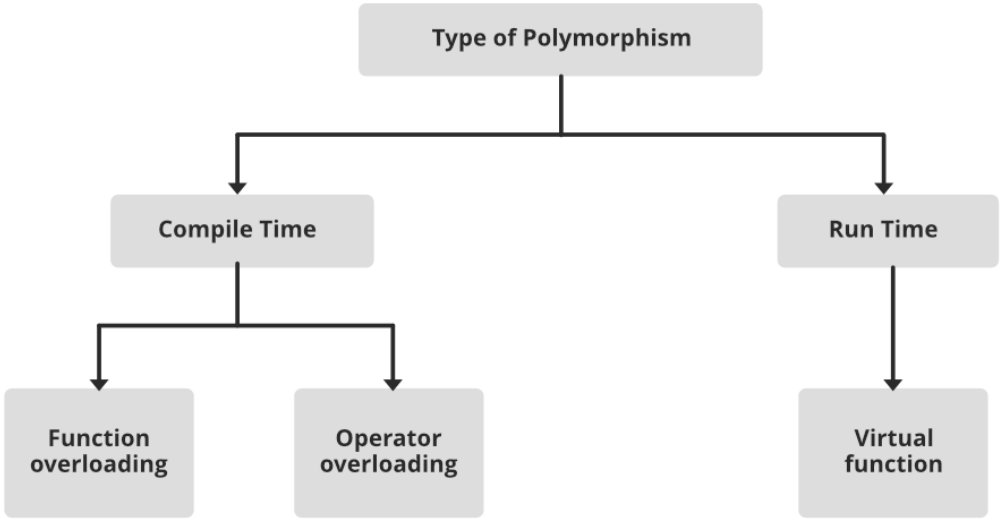


Figure 2: Polymorphism OOP

C, Inheritance:

Inheritance is an important pillar of OOP(Object-Oriented Programming). The capability of a class to derive properties and characteristics from another class is called Inheritance. When we write a class, we inherit properties from other classes. So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.

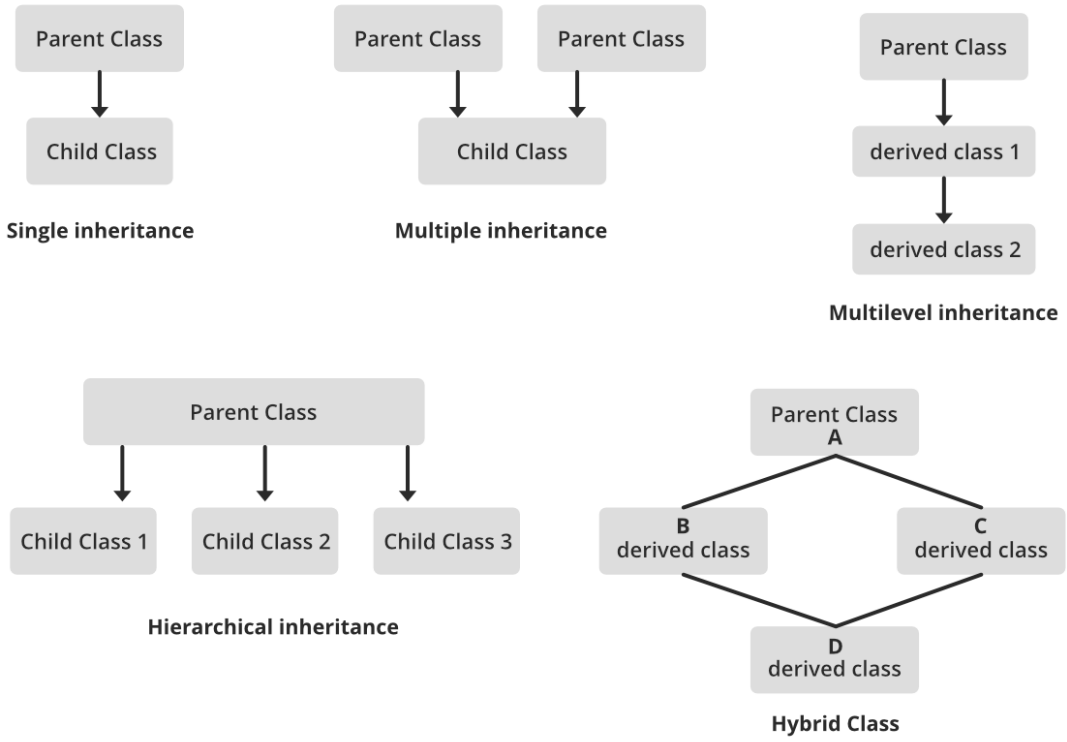


Figure 3: Inheritance OOP

D, Data Abstraction:

Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car.

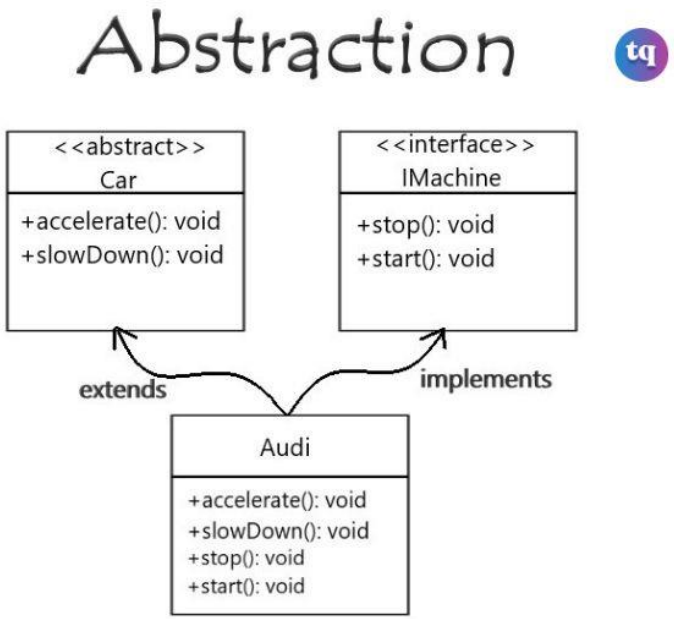


Figure 4: Abtraction OOP

```
using System;
public abstract class Animal
{
    public abstract void MakeSound();

    public void Sleep()
    {
        Console.WriteLine("Zzzz...");
    }
}
public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meow!");
```

```
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        Animal cat = new Cat();
        cat.MakeSound(); // Output: Meow!
        cat.Sleep();    // Output: Zzzz...

        Animal dog = new Dog();
        dog.MakeSound(); // Output: Woof!
        dog.Sleep();    // Output: Zzzz...
    }
}
```

Figure 5: Example Abtraction

E, Classes and Objects

A class is nothing more than a representation of an object type. The blueprint, plan, or template is what outlines an object's specifics. The building block from which unique objects are produced is a class. Three elements make up a class: a name, characteristics, and operations.

An item that is capable of carrying out a number of connected actions is referred to as a "thing". The behavior of an object is determined by the set of actions it takes. A Hand (an item) might be able to grasp something, or a student (an object) might be able to provide their name or address.

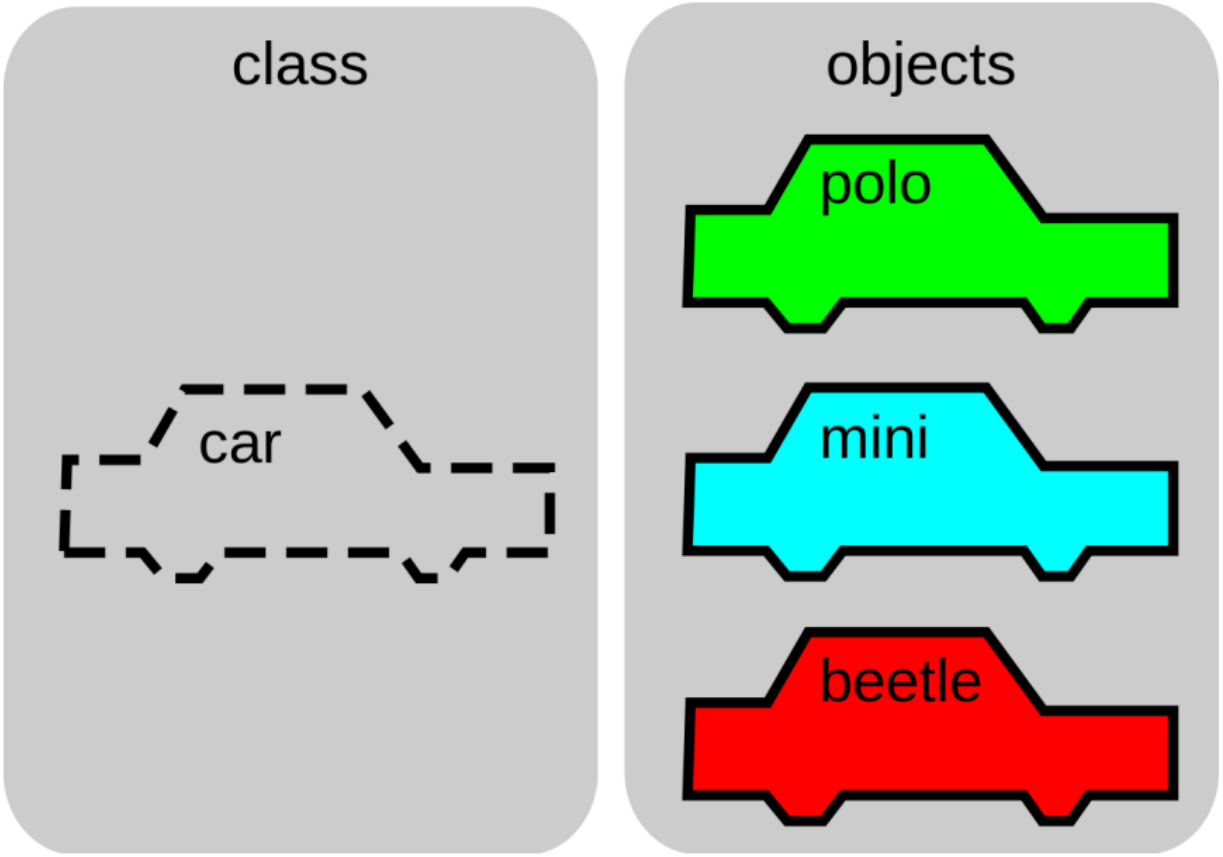


Figure 6: Classes and Objects

F. Access Modifiers

Access modifiers specify the accessibility of types (classes, interfaces, etc) and type members (fields, methods, etc).

```
class Student {

  public string name;

  private int num;

  }
```

Figure 7: Code Access Modifiers Example

There are 4 basic types of access modifiers.

- Public: When we declare a type or type member public, it can be accessed from anywhere.
- Private: When we declare a type member with the private access modifier, it can only be accessed within the same class or struct
- Protected: When we declare a type member as protected, it can only be accessed from the same class and its derived classes. For example,
- Internal: When we declare a type or type member as internal, it can be accessed only within the same assembly.

III. OOP SCENARIO

1. Scenario

❖ Employee:
  - Attributes: ID, TimeJoined, Name, Birthday, Age, Sex, HomeTown, Phone, Status, ListProject, Salary, Position.
  - Methods: Includes getter and setter methods for the attributes mentioned above, as well as methods to add, delete, and modify other objects.

❖ Position:

- Attributes: PositionID, PositionName, PositionSalary.
- Methods: Includes methods to display the existing positions and their information, as well as methods to add, delete, and modify positions.

❖ Salary:
- Attributes: Employee, Position, DayWork, OverTime, PaidSalary, TotalSalary, RemainingSalary.
- Methods: Includes getter and setter methods for the attributes, automatic salary calculation functions as required, and a function to display the information of the attributes.

❖ Project:
- Attributes: ProjectID, Employee, ProjectName, ProjectDetail, ProjectBonus, Status.
- Methods: Includes getter and setter methods for the attributes, as well as methods to add, delete, and modify projects.

2. Usecase Diagram

You don't need to log in. The menu options will include:

❖ Employee Management
- In the Employee Management section, I will have the ability to add, delete, and modify the positions of employees. Employees will have two positions: Manager and Newbie. Managers will be responsible for managing the Newbies.
- Employee Management will directly retrieve data from the Employee file to access and use the data attributes. When a new employee is added, a new corresponding Salary will be created for that person. Similarly, when an employee is deleted, their Salary will be removed as well.

❖ Project Management
- Next, I will be the one creating the projects. Once a project is created, nobody, including myself, can modify it (this helps prevent fraud).
- Projects will retrieve attributes from the Project table, and when a project is created, an automatic function will generate a unique ID for the new project. The default status of a newly created project will be "incomplete."

❖ Salary Management
- Lastly, in the Employee Salary Management section, as someone who is not proficient in calculations, I need an automatic function to calculate the overall salary of employees and determine the remaining unpaid amount.

- The salary table will retrieve attributes from the Salary data table, and it will include automatic functions for calculating salaries and other relevant functions.

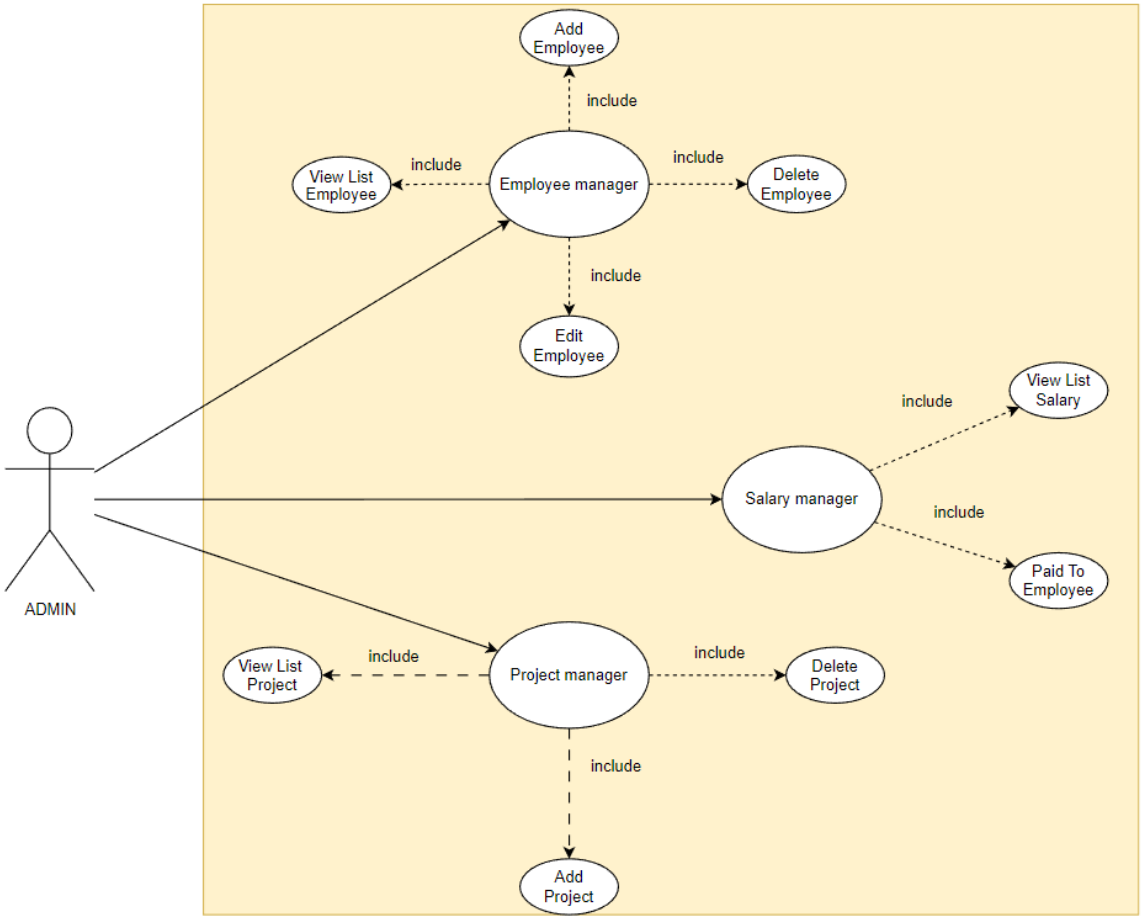


Figure 8: UseCase Diagram

3. Class Diagram

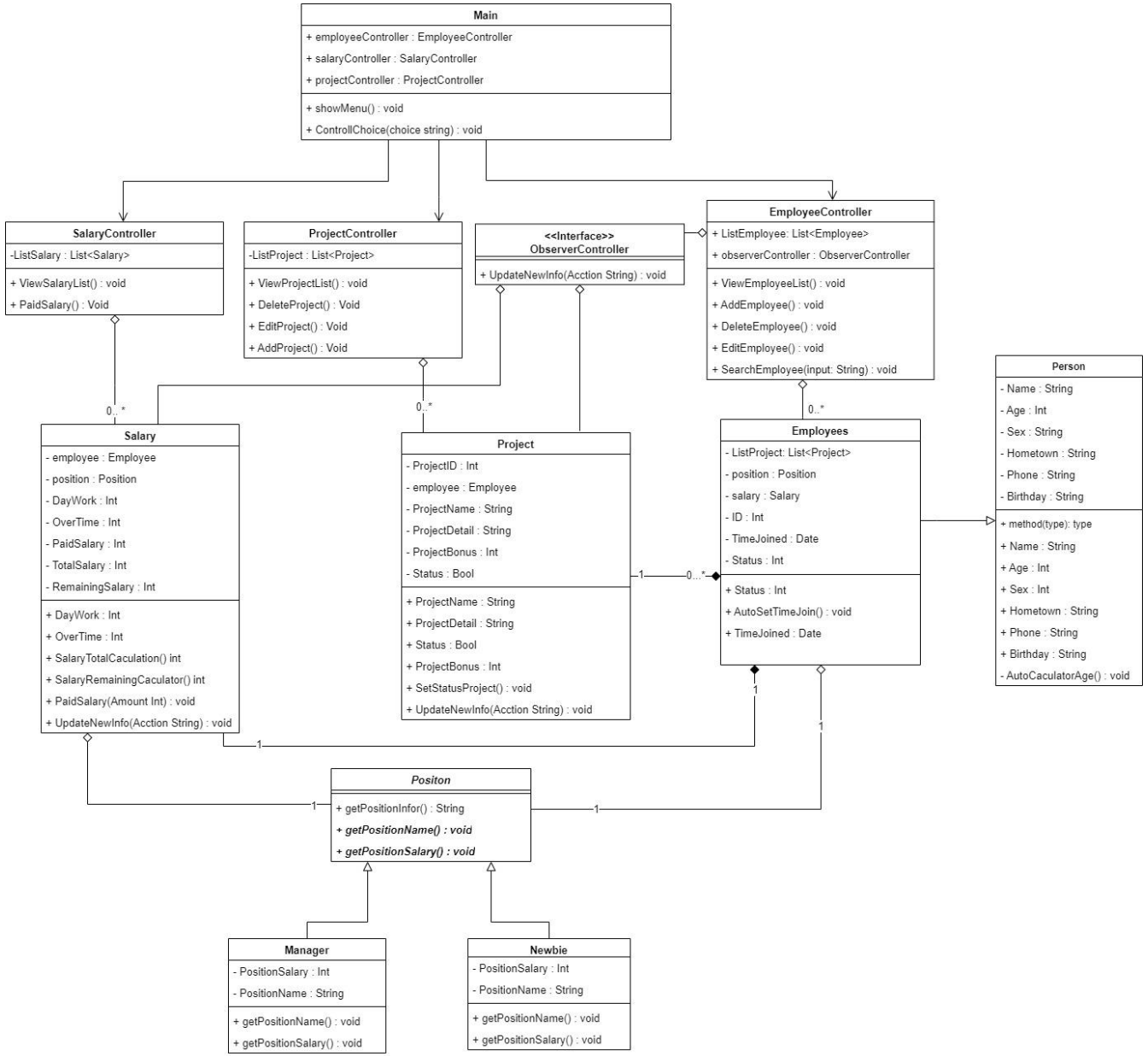


Figure 9: Class Diagram

A. Important Classes Explain

- Employee:
  - Attributes: ID, TimeJoined, Name, Birthday, Age, Sex, HomeTown, Phone, Status, ListProject, Salary, Position.
  - Methods: Includes getter and setter methods for the attributes mentioned above, as well as methods to add, delete, and modify other objects.

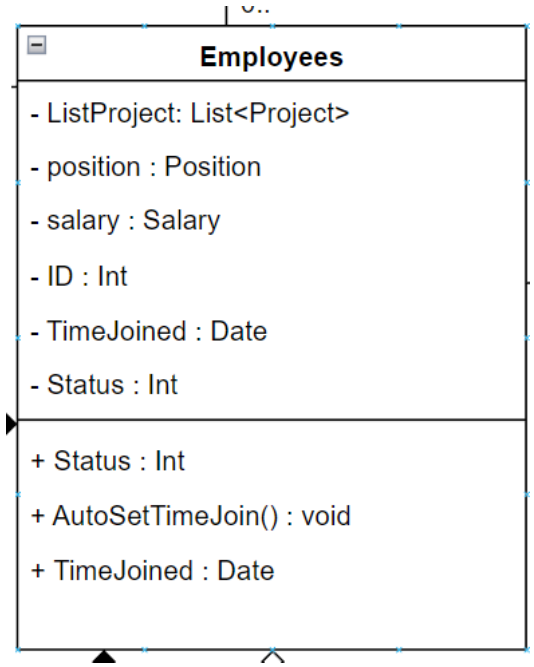


Figure 10: Employees Class

- Position:
  - Attributes: PositionID, PositionName, PositionSalary.
  - Methods: Includes methods to display the existing positions and their information, as well as methods to add, delete, and modify positions.

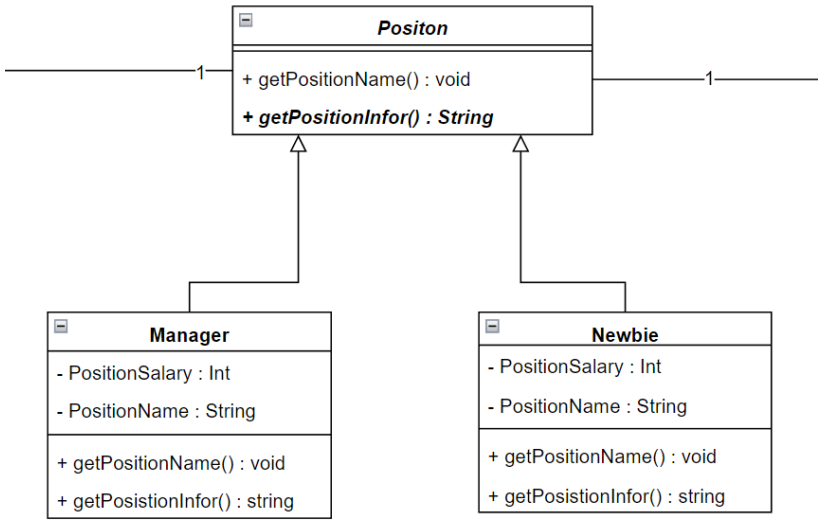


Figure 11: Position Class

- Salary:
  - Attributes: Employee, Position, DayWork, OverTime, PaidSalary, TotalSalary, RemainingSalary.
  - Methods: Includes getter and setter methods for the attributes, automatic salary calculation functions as required, and a function to display the information of the attributes.

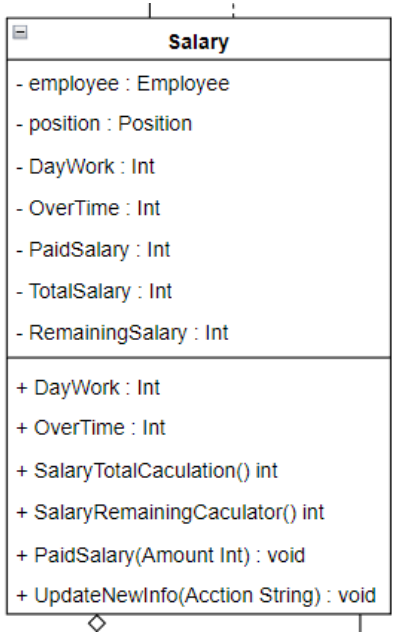


Figure 12: Salary Class

- Project:
  - Attributes: ProjectID, Employee, ProjectName, ProjectDetail, ProjectBonus, Status.
  - Methods: Includes getter and setter methods for the attributes, as well as methods to add, delete, and modify projects.

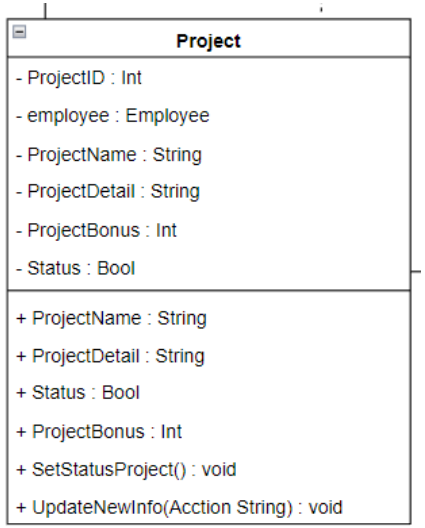


Figure 13: Project Class

❖ Controller Classes:

The Controller Class will mostly be the same, all have functions to handle the objects mentioned above such as: View, Edit, Delete, Add

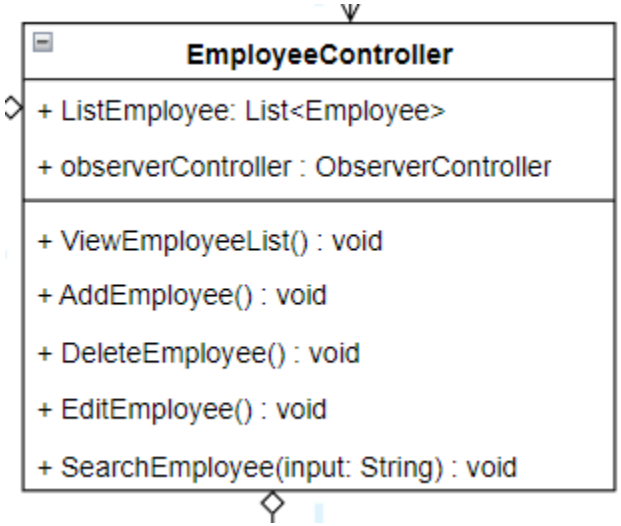


Figure 14: Employee Controller

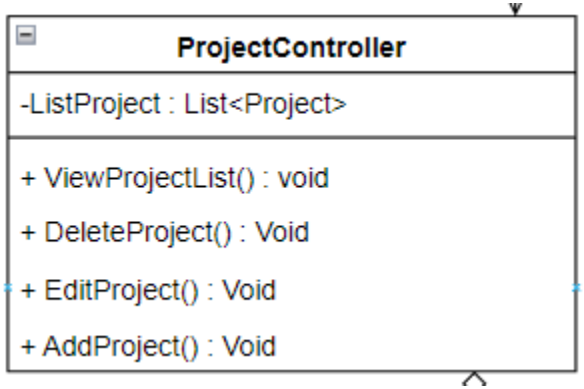


Figure 15: Project Controller

B. OOP features

❖ Abstraction:

In my program I used Postion as Abtract class with getPositionName() method.

And then 2 classes Manager and Newbie will inherit that method to edit again.

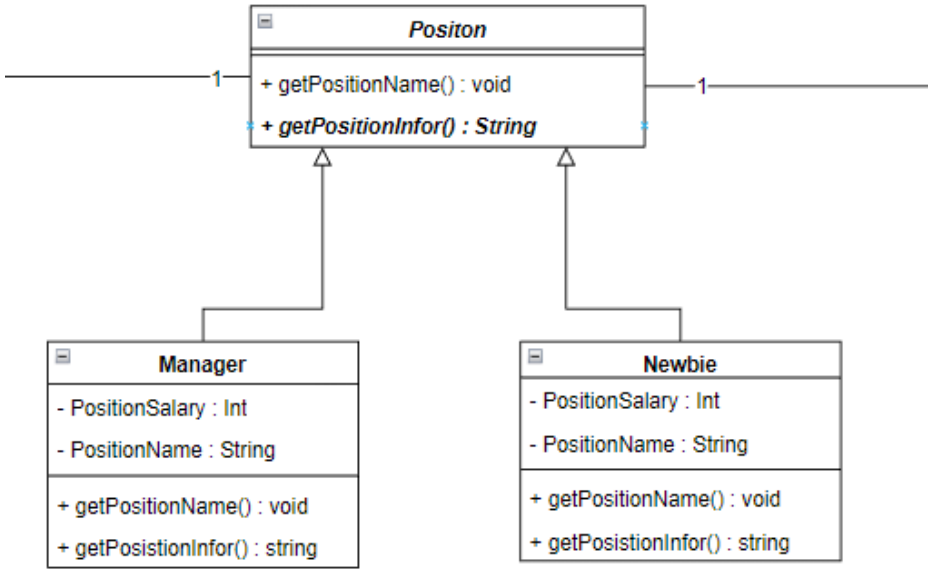


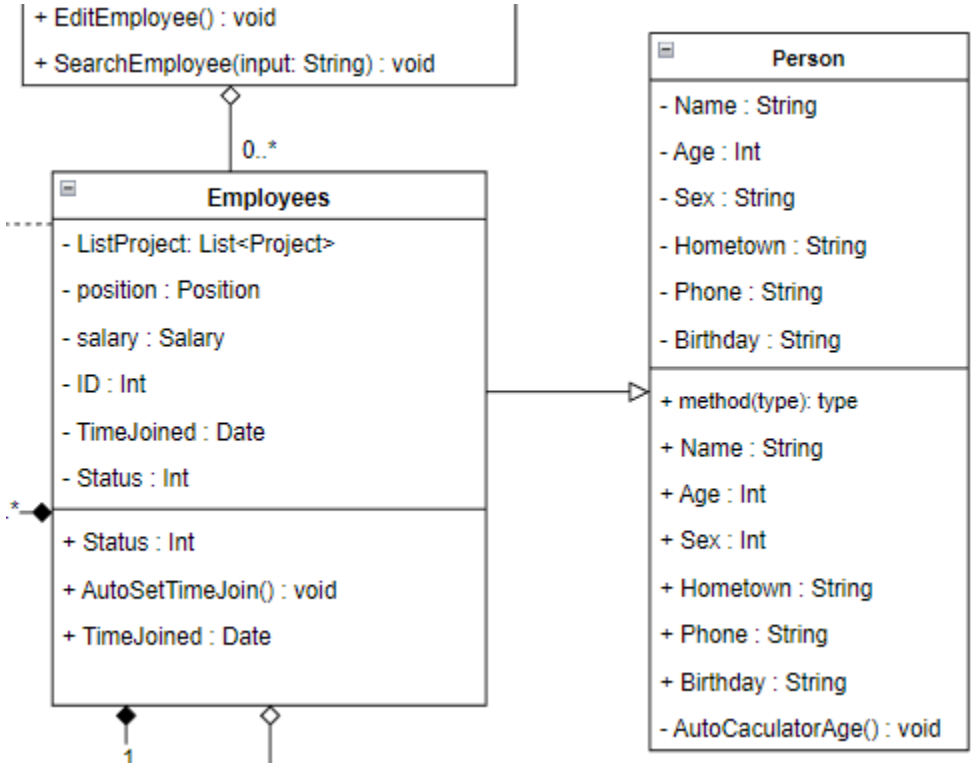
Figure 16: Abtraction Explain

❖ **Polymorphism:**

Similar to the Abstraction I mentioned above, in my Class Positon I contain an override method GetPosition() : String. And Class Manager, Newbie will inherit that method and rewrite it to match the position it holds

❖ **Inheritance:**

In my post, I used the Person Class to store the basic characteristics of a person, and then used Employee to inherit it, as well as between Position and Manger, Newbie, I did the same thing.


+ EditEmployee() : void
+ SearchEmployee(input: String) : void
0..*
Employees
- ListProject: List<Project>
- position : Position
- salary : Salary
- ID : Int
- TimeJoined : Date
- Status : Int
+ Status : Int
+ AutoSetTimeJoin() : void
+ TimeJoined : Date
1
Person
- Name : String
- Age : Int
- Sex : String
- Hometown : String
- Phone : String
- Birthday : String
+ method(type): type
+ Name : String
+ Age : Int
+ Sex : Int
+ Hometown : String
+ Phone : String
+ Birthday : String
- AutoCaculatorAge() : void

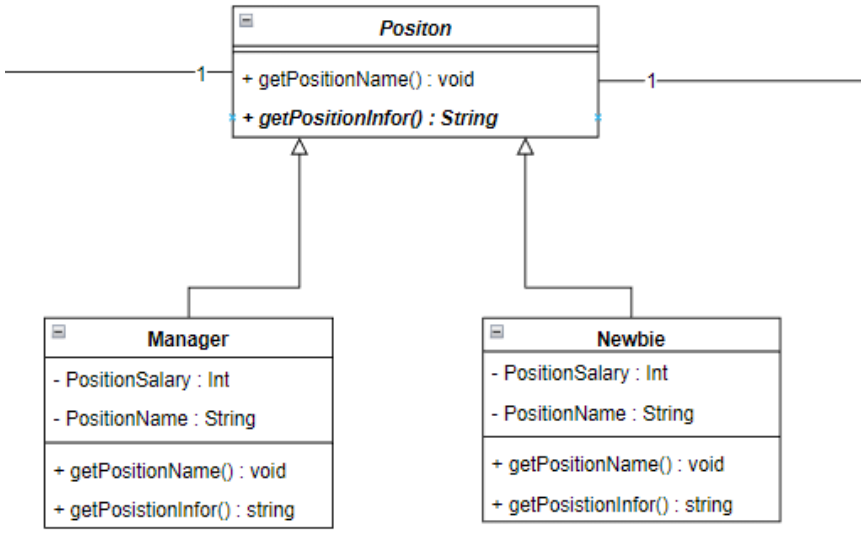


Figure 17: Inheritance

❖ **Encapsulation:**

In terms of encapsulation I have used a lot for classes, in order to hide the original data and avoid user changes. The user cannot call those data stubs without the Set and Get functions.

For example:

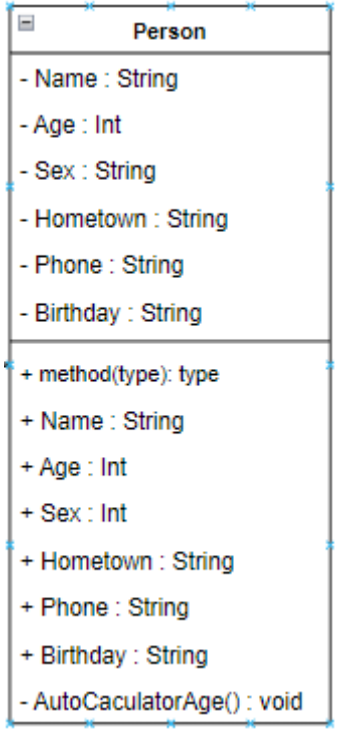


Figure 18: Person Class with Encapsulation

We can see that the AutoCaculatorAge() variables and functions with a (-) sign in front are intended to let those data be used only in the Person class, not for other classes to call.

C. UML Relationships

- Association:
  - Unary:

    Association Unary occurs when an object in a class has a relationship with itself. This means that an object is attached to another object in the same class. This relationship is often used to describe the states, features, or behavior of an object. In a UML diagram, the Association Unary is represented by a line connecting the object to itself.

    For example:

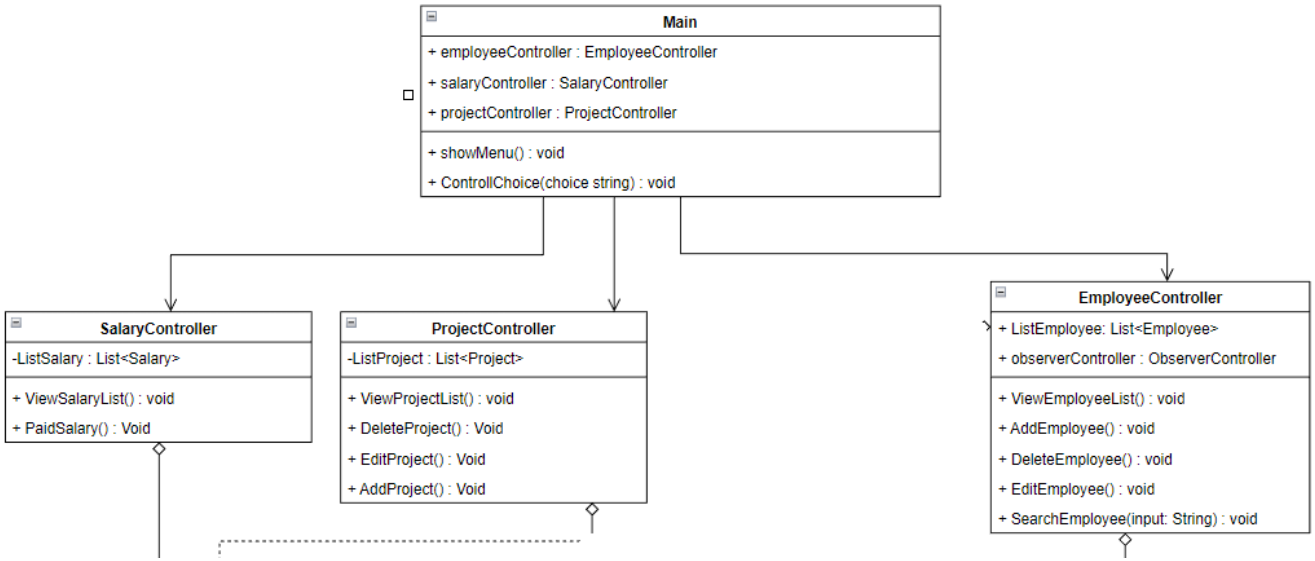


Figure 19: Association Unary

- Binary:

  Association Binary is the relationship between two different objects in the system. It denotes a relationship or a link between two different objects. An Association Binary can be a single or multiple (many participating) relationship. In a UML diagram, the Association Binary is represented by a connecting line between two objects.

  For example:

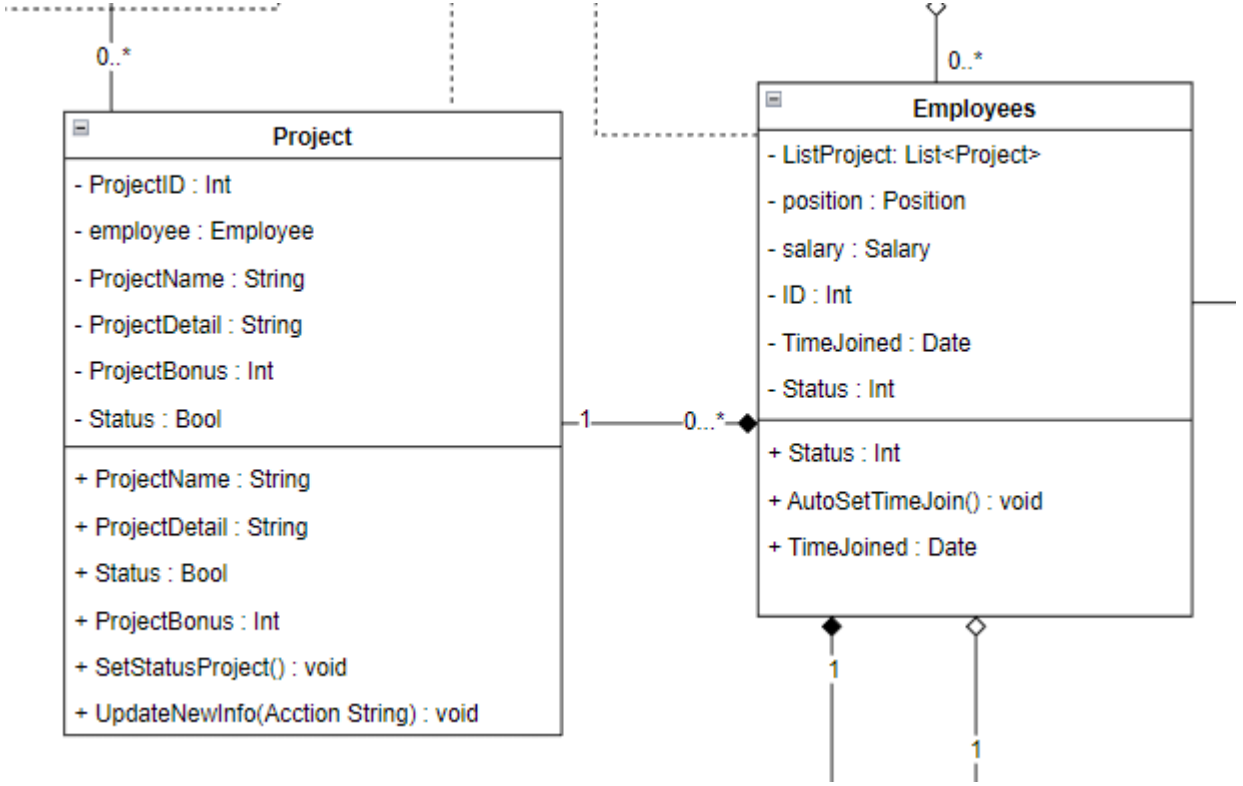


Figure 20: ASsociation Binary

D. Multiplicity:

The multiplicities I use in my Program are 0...* and 1.

Example:

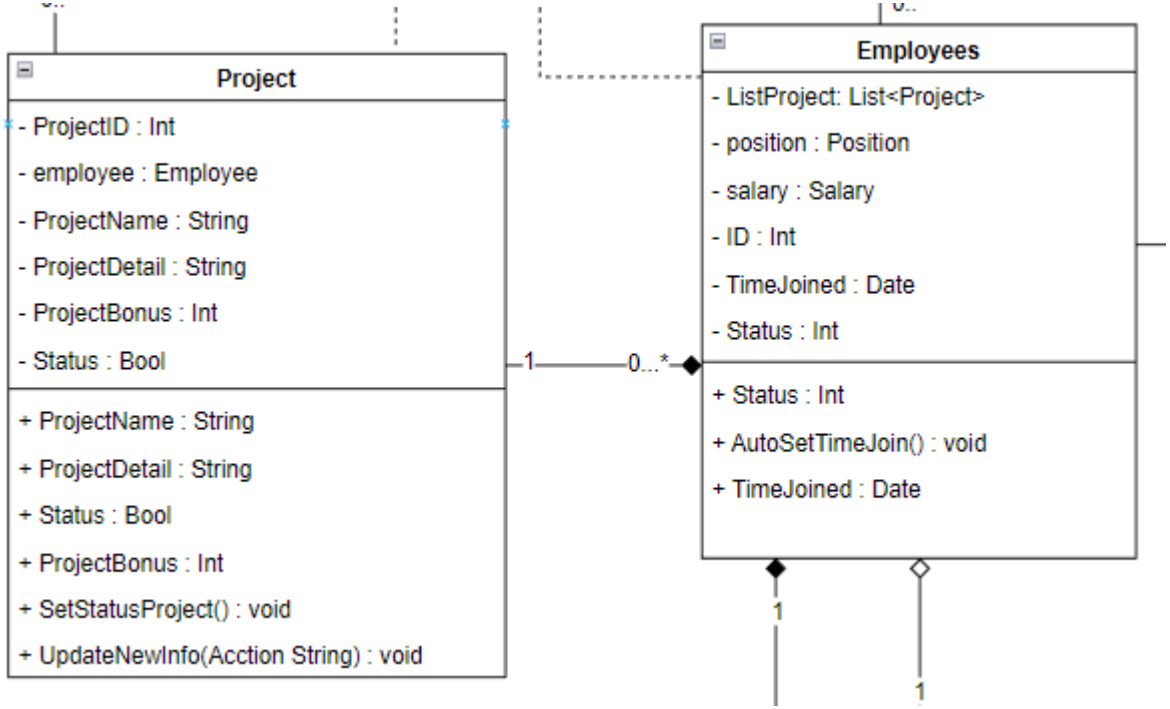


Figure 21: Example For multiplicity

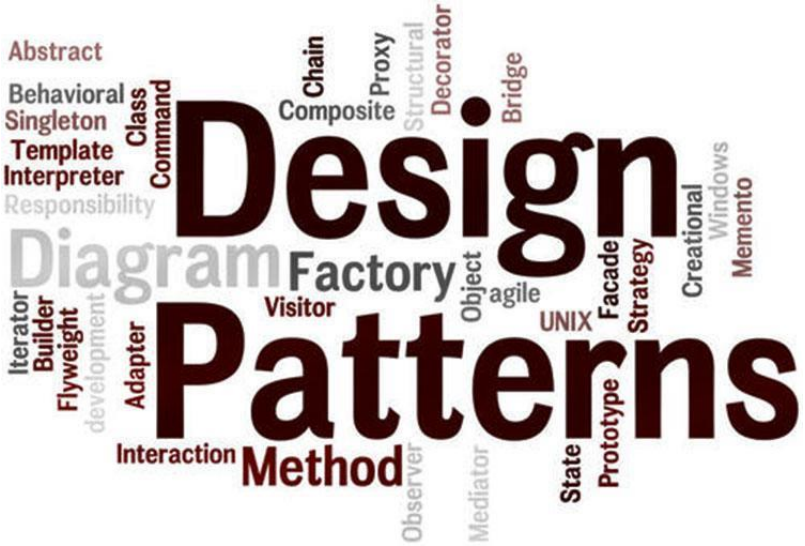
As shown in the picture, the relationship from Project to Employee is 0...* because an employee can have many Projects and also no Project. For 1 Project, only 1 Employee should be allowed to do it.

Link Image: https://drive.google.com/file/d/16bpqPK2YtL3H5cbaPlnfX3aswfqF3VRr/view?usp=sharing

IV. DESIGN PATTERNS

Explain what ITNavi's definition of Design Pattern is as follows: Currently, in software engineering, a design pattern is considered a total solution to help solve common problems in the software design process. .

Each design pattern will not be considered a finished design but has been directly converted into different codes. It is seen as a description or problem-solving framework that is used in a variety of situations. These object-oriented design patterns often show the relationships and interactions between classes or objects; this does not need to specify other application-specific classes or objects. Most often, algorithms are not considered design patterns, because they only deal with computational problems rather than design divination problems. Therefore, Design pattern will be able to make your design more flexible and easier to change and maintain.



1. Creational pattern

Creational Pattern (initialization group – 5 patterns) and includes: Abstract Factory, Factory Method, Singleton, Builder, Prototype. Design patterns in this category are often provided with a solution to be able to create objects and assist in hiding the logic of its creation. Instead, it is possible to create objects directly using the most appropriate new methods. This will make the program more flexible in deciding which objects should be created in other given scenarios.

- Factory Method is used to define the interface that creates an object, but only to let the subclass decide which class to use to create the object of that class. It can be used to create the class interface first and specify the concrete class later.
  Example UML:

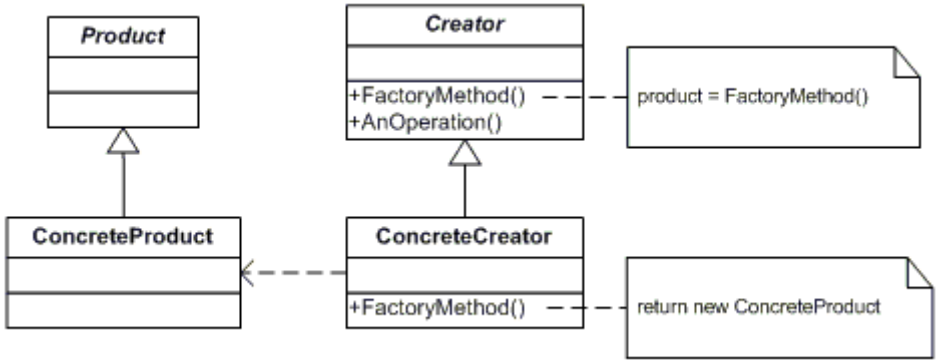


Figure 22: Example Creaional - Abstract Factory Pattern UML

Example Code:

```
using System;

namespace DoFactory.GangOfFour.Factory.Structural
{
  /// <summary>

  /// MainApp startup class for Structural

  /// Factory Method Design Pattern.

  /// </summary>

  class MainApp

  {
    /// <summary>
```

```
  /// Entry point into console application.

  /// </summary>

  static void Main()
  {
    // An array of creators

    Creator[] creators = new Creator[2];

    creators[0] = new ConcreteCreatorA();
    creators[1] = new ConcreteCreatorB();

    // Iterate over creators and create products

    foreach (Creator creator in creators)
    {
      Product product = creator.FactoryMethod();
      Console.WriteLine("Created {0}",
        product.GetType().Name);
    }

    // Wait for user

    Console.ReadKey();
  }
}

/// <summary>

/// The 'Product' abstract class

/// </summary>
```

```
abstract class Product

{
}

/// <summary>

/// A 'ConcreteProduct' class

/// </summary>

class ConcreteProductA : Product

{
}

/// <summary>

/// A 'ConcreteProduct' class

/// </summary>

class ConcreteProductB : Product

{
}

/// <summary>

/// The 'Creator' abstract class

/// </summary>

abstract class Creator
```

```
{
  public abstract Product FactoryMethod();
}

/// <summary>

/// A 'ConcreteCreator' class

/// </summary>

class ConcreteCreatorA : Creator

{
  public override Product FactoryMethod()
  {
    return new ConcreteProductA();
  }
}

/// <summary>

/// A 'ConcreteCreator' class

/// </summary>

class ConcreteCreatorB : Creator

{
  public override Product FactoryMethod()
  {
    return new ConcreteProductB();
  }
}
```

```
}
```

Figure 23: Example Creaional - Abstract Factory Pattern Code

When We run this code, We have this:

```
Created ConcreteProductA
Created ConcreteProductB
```

Factory Melthod is a very widely used and useful pattern in many projects, so you can consider applying this Pattern if it makes sense

2. Structural pattern

Structural Pattern (structural group - 7 patterns) includes: Adapter, Facade, Bridge, Composite, Decorator, Flyweight and Proxy. These types of design patterns are often related to object and class components. Therefore, it is used to establish and define relationships between objects.

The Template Method separates the specific steps of an algorithm from the so-called "template methods" in the base class. These specific steps are implemented in subclasses, but the general structure and order of the steps are controlled by the template method in the base class.

The Template Method pattern enhances design flexibility and reuse, as it allows subclasses to customize and extend specific steps without changing the overall structure of the algorithm.

In my program I applied the Template Method to the Structural Patten as follows:

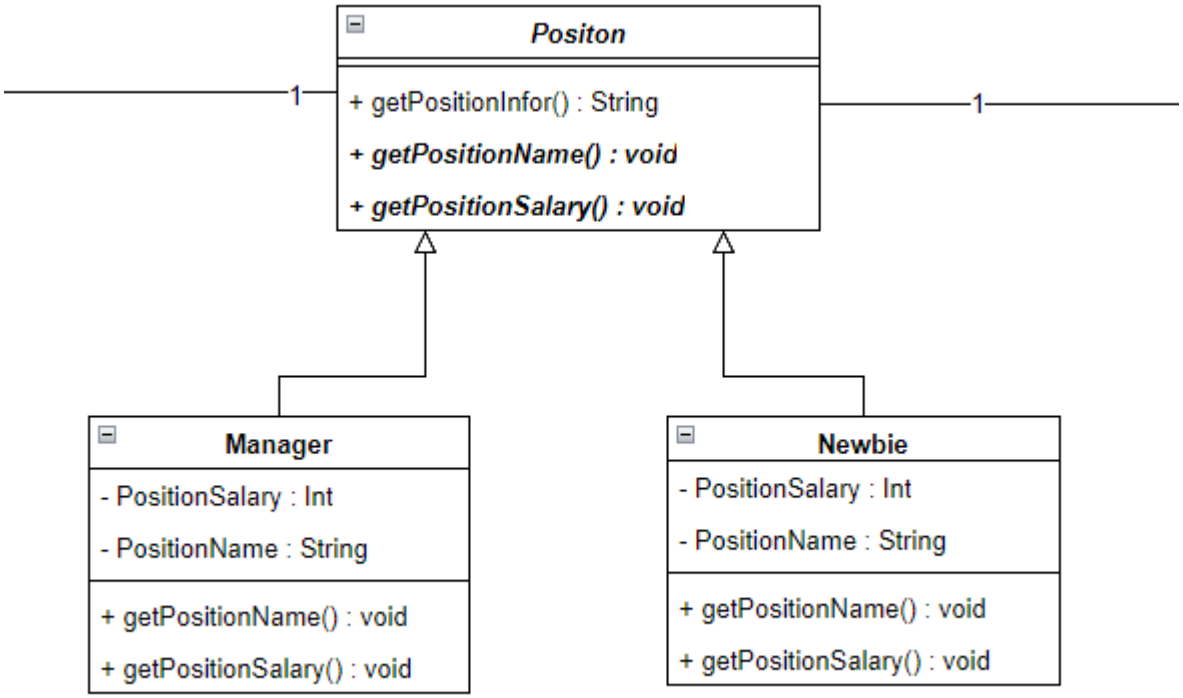


Figure 24: Template Method - Structural Pattern UML

In my project, ***PositonClass*** defines a sample method **getPositionInfor()** and implements the fixed steps of the algorithm. The two subclasses Manager and Newbie inherit from **PositonClass** and implement **getPositionName()** and **getPositionSalary()** methods in their own way. When running the sample method **getPositionInfor()** for 2 subclasses and we will have the correct information for each class.

3. Behavioral pattern

Behavioral Pattern (group of interactions/behavior - 11 patterns) includes: Interpreter, Chain of Responsibility, Template Method, Command, Iterator, Mediator, Memento, Observer, State, Strategy and Visitor: This is the group used for implementation object behavior as well as the communication between objects.

In C#, Observer is a Behavioral Design Pattern that allows objects (observers) to monitor and react to changes in the state of a subject object. It provides a way to build systems where components can communicate without knowing about each other's existence.

In C#, the Observer pattern is usually implemented using an interface or delegate to define the notification and update methods. Observers subscribe to receive notifications from the subject object and are automatically notified when the subject object's state changes.

In my program I applied the Observer to the Behavioral Pattern as follows:

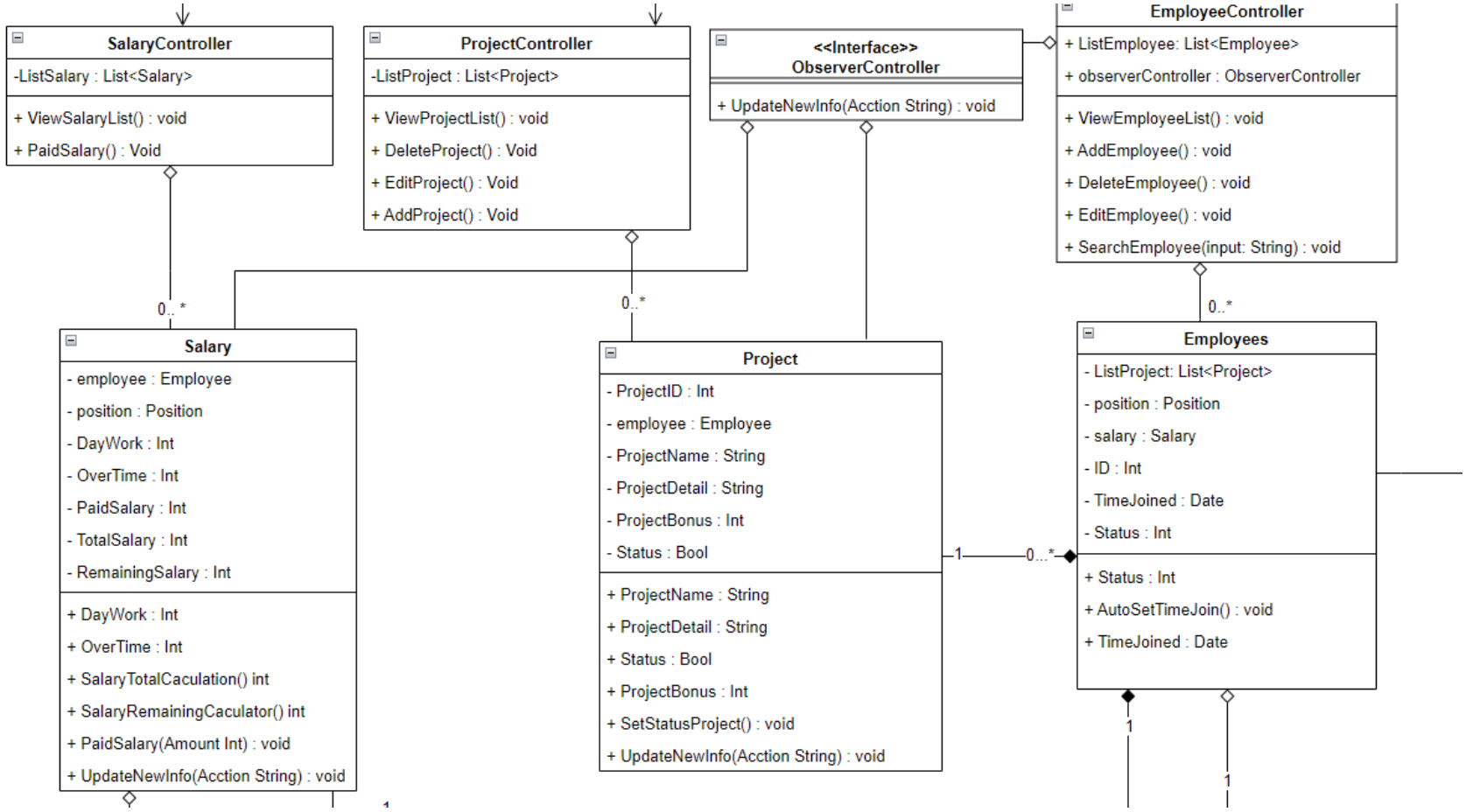


Figure 25: Observer Behavioral Pattern

In my Project, I use Class **ObserverController** to monitor activities from **EmployeeController.** And in functions**: AddEmployee()**, **EditEmployee()**. **DeleteEmployee(),** there will be **UpdateNewInfo()** method to update information to **Salary, Project** to change employee information to match the updated information.

For example: When **EmployeeController** implements **EditEmployee(),** it will implement the **ObserverController's UpdateNewInfo()** function to change the employee information to match the changed information of the Employee class.

V. Design Pattern vs OOP

1. Design pattern Template Method:

- The Temple Method I just described seems to be an implementation of an algorithm for retrieving position information in a project, using a class hierarchy with inheritance and polymorphism. Let's analyze the structure and behavior of the classes involved.
- PositionClass:
- This class serves as the base class or superclass for the position hierarchy. It defines the common behavior and implements the fixed steps of the algorithm through the getPositionInfor() method. It likely contains the core logic for retrieving position information.
- Manager and Newbie:
- These classes are subclasses or derived classes of PositionClass. They inherit the behavior and implementation of getPositionInfor() from the base class but provide their own implementation for the getPositionName() and getPositionSalary() methods. This approach allows each subclass to customize the behavior specific to their position type.
- getPositionInfor():
- This method in PositionClass encapsulates the fixed steps of the algorithm for retrieving position information. Since it is implemented in the base class, it can be called on instances of both Manager and Newbie classes. This is possible due to polymorphism, where the method can be invoked on objects of different types but exhibit different behaviors based on the actual type of the object at runtime.
- By running the getPositionInfor() method on instances of both Manager and Newbie, you can obtain the correct information for each class. The specific behavior of getPositionName() and getPositionSalary() will be determined by the subclass in each case.

- This design allows for code reuse and promotes flexibility, as you can create additional subclasses of PositionClass with their own custom behavior while maintaining the core algorithm defined in the base class. It also adheres to the principles of object-oriented programming, such as encapsulation, inheritance, and polymorphism.

❖ Relationship to OOP:

The Temple Method you described exhibits several key principles of object-oriented programming (OOP). Let's discuss how it relates to OOP concepts:

1. **Inheritance:** The relationship between **the PositionClass** superclass and the Manager and Newbie subclasses demonstrates inheritance. Both subclasses inherit the behavior and implementation of the getPositionInfor() method from the base class. Inheritance allows subclasses to reuse and extend the functionality of the superclass, promoting code reuse and modularity.
2. **Polymorphism:** The ability to call the getPositionInfor() method on instances of both Manager and Newbie classes showcases polymorphism. Despite the method being implemented in the base class, it exhibits different behaviors depending on the actual type of the object at runtime. This allows for dynamic and flexible behavior based on the specific subclass instance being used.
3. **Encapsulation:** The encapsulation principle is evident in the design of the classes. Each class encapsulates its own data and behavior, providing a clear separation of concerns. The details of how the position information is retrieved are encapsulated within the PositionClass, while the specific implementation of getPositionName() and getPositionSalary() is encapsulated within the respective subclasses.
4. **Abstraction:** The PositionClass acts as an abstraction of a general position, defining the common behavior and algorithm for retrieving position information. It allows for the creation of specific position subclasses that implement their own variations of certain methods. The abstraction provided by PositionClass enables code organization and modularity by separating the core algorithm from the specific details of each position.

By utilizing inheritance, polymorphism, encapsulation, and abstraction, the Temple Method aligns with the fundamental principles of object-oriented programming. This approach promotes code reuse, flexibility, modularity, and maintainability, making it easier to manage and extend the functionality of the project.

2. Design pattern Observer:

The design I just described involves the use of the Observer pattern to monitor activities in the **EmployeeController** class and update related information in the **ObserverController** class. Let's analyze the structure and behavior of the classes involved and discuss the implementation.

- **ObserverController:** This class acts as the observer in the Observer pattern. It monitors activities from the **EmployeeController** class and performs updates based on those activities. It likely has a method called **UpdateNewInfo()** that receives notifications from the EmployeeController and performs the necessary updates, such as updating the salary or project information.
- **EmployeeController:** This class acts as the subject in the Observer pattern. It is responsible for performing operations related to employees, such as adding, editing, or deleting employees. When specific actions, such as editing an employee, are performed, the **EmployeeController** notifies the **ObserverController** by calling its **UpdateNewInfo()** method.
- **AddEmployee(), EditEmployee(), DeleteEmployee():** These functions represent operations performed in the **EmployeeController** class. When these functions are executed, they trigger corresponding actions and, in the case of editing or deleting employees, notify the **ObserverController** by calling its **UpdateNewInfo()** method.
- **The UpdateNewInfo()** method in the **ObserverController** is responsible for updating relevant information, such as salary or project details, based on the changes made in the **EmployeeController.** For example, if the **EditEmployee()** function modifies an employee's information, it will call the **UpdateNewInfo()** method in the **ObserverController** to update the corresponding data.

This design allows for decoupling the **EmployeeController** and **ObserverController** classes, ensuring that updates to employee information are properly communicated to the observer. It promotes loose coupling, as the **EmployeeController** does not need to be aware of how the information is updated in the **ObserverController.**

By utilizing the Observer pattern, the design enables a flexible and scalable approach to handling updates across different components of the system. It promotes modularity and extensibility, as additional observers can be added to the **ObserverController** to handle different types of updates or perform specific actions based on the changes detected in the **EmployeeController.**

❖ Relationshop to OOP:

1. **Encapsulation:** The classes involved, such as **ObserverController** and **EmployeeController**, encapsulate their own data and behavior. Each class is responsible for a specific set of operations and maintains its own internal state. This encapsulation promotes modularity and separation of concerns.

2. **Abstraction:** The Observer pattern abstracts the relationship between the subject (EmployeeController) and the observer (ObserverController). The subject notifies the observer about specific activities, but the observer doesn't need to know the internal details of the subject. This abstraction allows for loose coupling and reduces dependencies between classes.
3. **Dependency Injection:** In the Observer pattern, the subject (EmployeeController) doesn't have a direct reference to the observer (ObserverController). Instead, it depends on an abstract interface or base class that represents the observer. This approach supports dependency injection, enabling flexibility and extensibility by allowing different observers to be injected into the subject.
4. **Polymorphism:** The UpdateNewInfo() method in the ObserverController is an example of polymorphism. Different observers can implement this method to perform specific actions based on the updates they receive. This allows for dynamic behavior based on the specific type of observer at runtime.
5. **Design Patterns:** The Observer pattern itself is a widely used design pattern in OOP. It provides a way to establish a one-to-many relationship between objects, ensuring that multiple observers can be notified of changes in the subject. The pattern promotes flexibility, modularity, and maintainability in systems.

By incorporating these OOP principles and using the Observer pattern, the design you described enables loosely coupled and modular code, making it easier to manage and extend the functionality of the system. It also promotes code reusability and separation of concerns, enhancing the overall design and maintainability of the project.

VI. CONCLUSION

Using the design pattern Template Method: Applying the design pattern Template Method in the Position class is a good way to separate the general and specific behaviors of child objects. This increases the flexibility and scalability of the system. You can efficiently reuse the source code and easily add new objects in the future without changing the existing code too much.

Using the Observer design pattern: Using the Observer design pattern in the ObserverController class helps you build a flexible communication mechanism between objects in the system. By registering and tracking events and changes from different objects like Project or Salary, you can take appropriate actions and update data in a timely manner.

Object Structure: Having key objects like Project, Salary, Position, and Person helps you organize and manage employee-related information in a structured way. These objects represent different aspects of employee management and allow you to perform operations and calculations accordingly.

However, the final evaluation of the project depends on many other factors such as detailed design, implementation, error management, and performance. Also, it depends on the specific requirements of the project and your mastery of programming principles and design patterns.

REFERENCE:

https://www.softwaretestinghelp.com/c-sharp/oops-concepts-in-csharp/

https://www.geeksforgeeks.org/introduction-of-object-oriented-programming

https://itnavi.com.vn/blog/design-pattern-la-gi

https://www.programiz.com/csharp-programming/access-modifiers

https://www.codingninjas.com/codestudio/library/what-are-the-features-of-object-oriented-programming#:~:text=The%20four%20main%20pillars%20or,recall%20all%20of%20them%20easily.