**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](#_Toc136554215)

[II. OOP GENERAL CONCEPTS 5](#_Toc136554216)

[1. Introduce OOP 5](#_Toc136554217)

[2. OOP General Concepts 5](#_Toc136554218)

[A, Encapsulation: 6](#_Toc136554219)

[B, Polymorphism: 6](#_Toc136554220)

[C, Inheritance: 7](#_Toc136554221)

[D, Data Abstraction: 8](#_Toc136554222)

[E, Classes and Objects 10](#_Toc136554223)

[F. Access Modifiers 11](#_Toc136554224)

[III. OOP SCENARIO 12](#_Toc136554225)

[1. Scenario 12](#_Toc136554226)

[2. Usecase Diagram 13](#_Toc136554227)

[3. Class Diagram 15](#_Toc136554228)

[A. Important Classes Explain 17](#_Toc136554229)

[B. OOP features 21](#_Toc136554230)

[C. UML Relationships 25](#_Toc136554231)

[D. Multiplicity: 27](#_Toc136554232)

[IV. DESIGN PATTERNS 29](#_Toc136554233)

[1. Creational pattern 29](#_Toc136554234)

[2. Structural pattern 32](#_Toc136554235)

[3. Behavioral pattern 33](#_Toc136554236)

[V. Design Pattern vs OOP 34](#_Toc136554237)

[VI. CONCLUSION 35](#_Toc136554238)

[REFERENCE: 36](#_Toc136554239)

[Figure 1: Encapsulation OOP 6](#_Toc136554240)

[Figure 2: Polymorphism OOP 7](#_Toc136554241)

[Figure 3: Inheritance OOP 8](#_Toc136554242)

[Figure 4: Abtraction OOP 9](#_Toc136554243)

[Figure 5: Example Abtraction 10](#_Toc136554244)

[Figure 6: Classes and Objects 11](#_Toc136554245)

[Figure 7: Code Access Modifiers Example 12](#_Toc136554246)

[Figure 8: UseCase Diagram 14](#_Toc136554247)

[Figure 9: Class Diagram 17](#_Toc136554248)

[Figure 10: Employees Class 17](#_Toc136554249)

[Figure 11: Position Class 18](#_Toc136554250)

[Figure 12: Salary Class 19](#_Toc136554251)

[Figure 13: Project Class 20](#_Toc136554252)

[Figure 14: Employee Controller 21](#_Toc136554253)

[Figure 15: Project Controller 21](#_Toc136554254)

[Figure 16: Abtraction Explain 22](#_Toc136554255)

[Figure 17: Inheritance 24](#_Toc136554256)

[Figure 18: Person Class with Encapsulation 25](#_Toc136554257)

[Figure 19: Association Unary 26](#_Toc136554258)

[Figure 20: ASsociation Binary 27](#_Toc136554259)

[Figure 21: Example For multiplicity 28](#_Toc136554260)

[Figure 22: Example Creaional - Abstract Factory Pattern UML 30](#_Toc136554261)

[Figure 23: Example Creaional - Abstract Factory Pattern Code 31](#_Toc136554262)

[Figure 24: Template Method - Structural Pattern UML 33](#_Toc136554263)

[Figure 25: Observer Behavioral Pattern 34](#_Toc136554264)

# 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#.

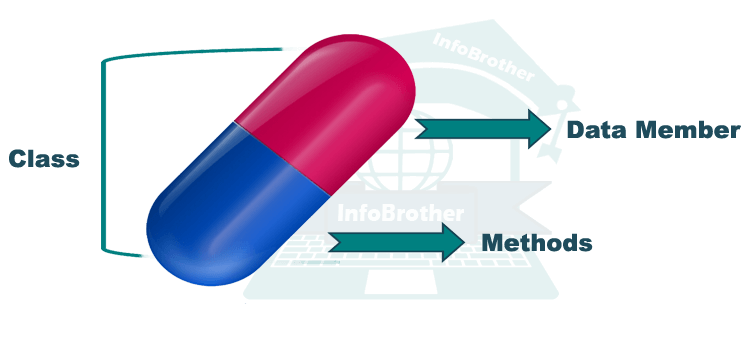


Figure 1: Encapsulation OOP

### 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.

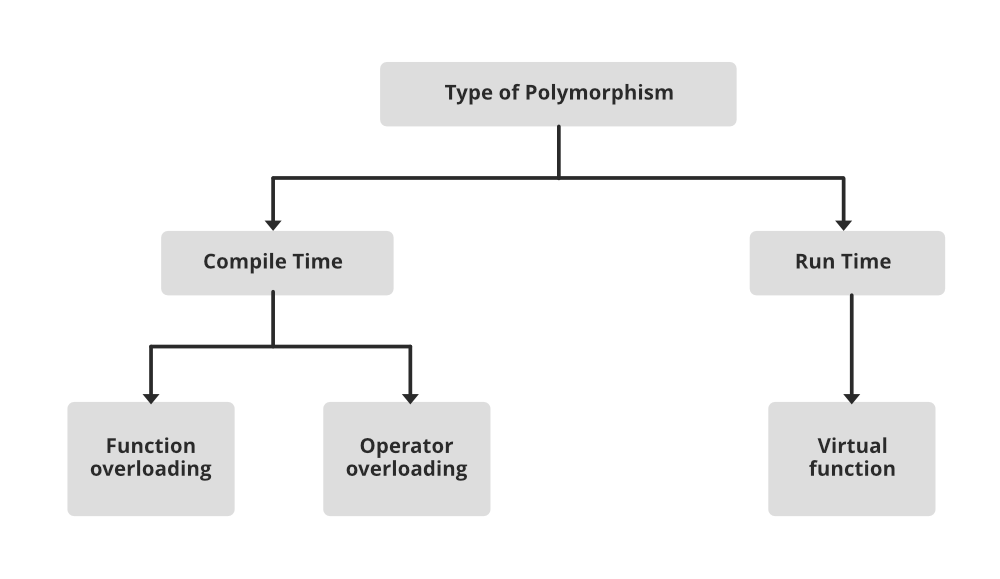


Figure 2: Polymorphism OOP

### 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.

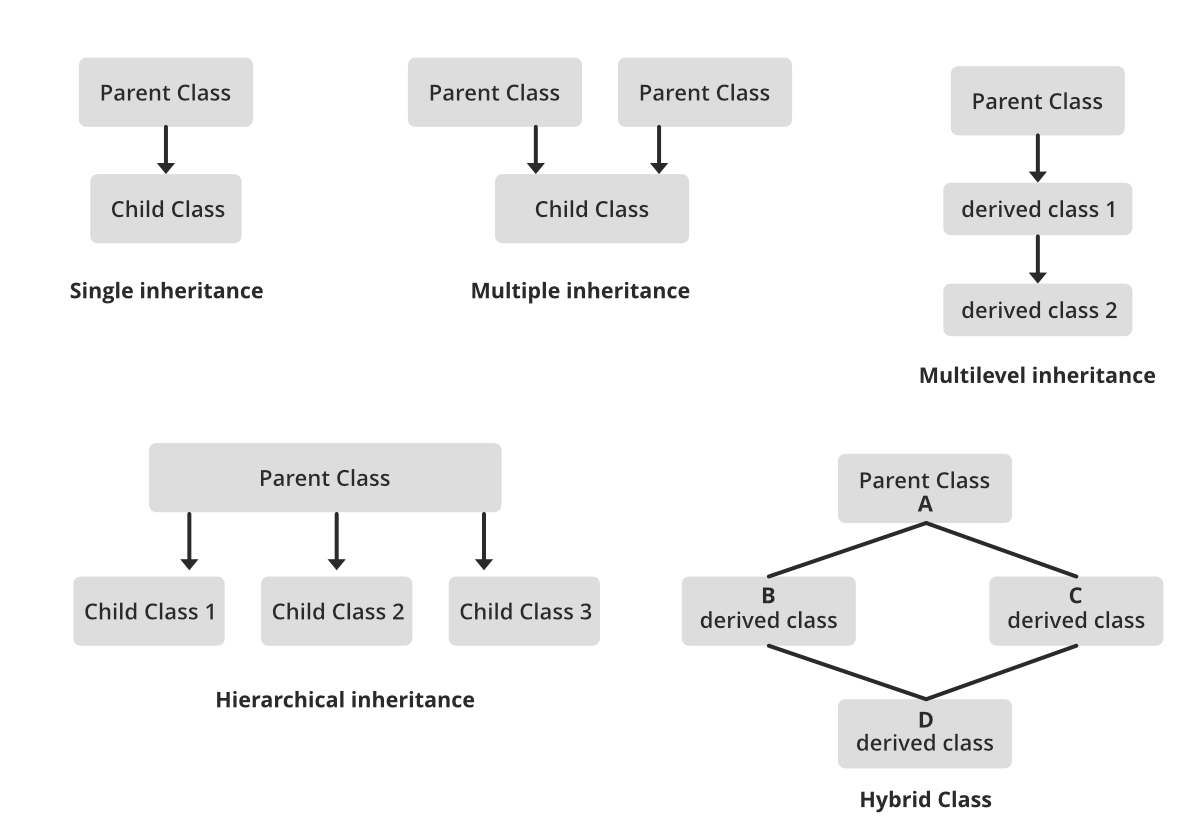


Figure 3: Inheritance OOP

### 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.

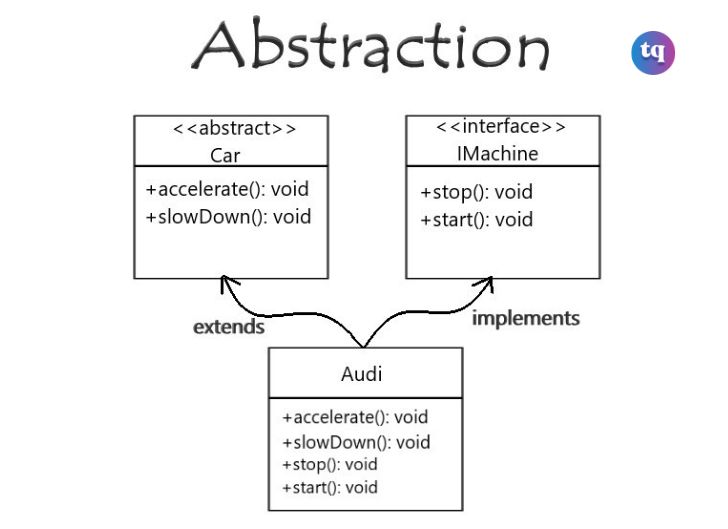


Figure 4: Abtraction OOP

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.

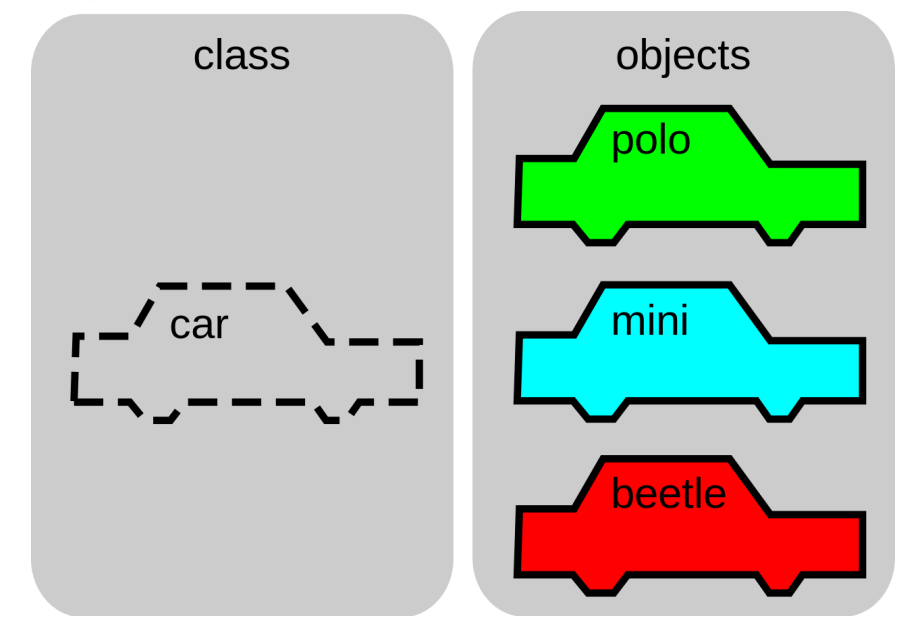


Figure 6: Classes and Objects

### 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.

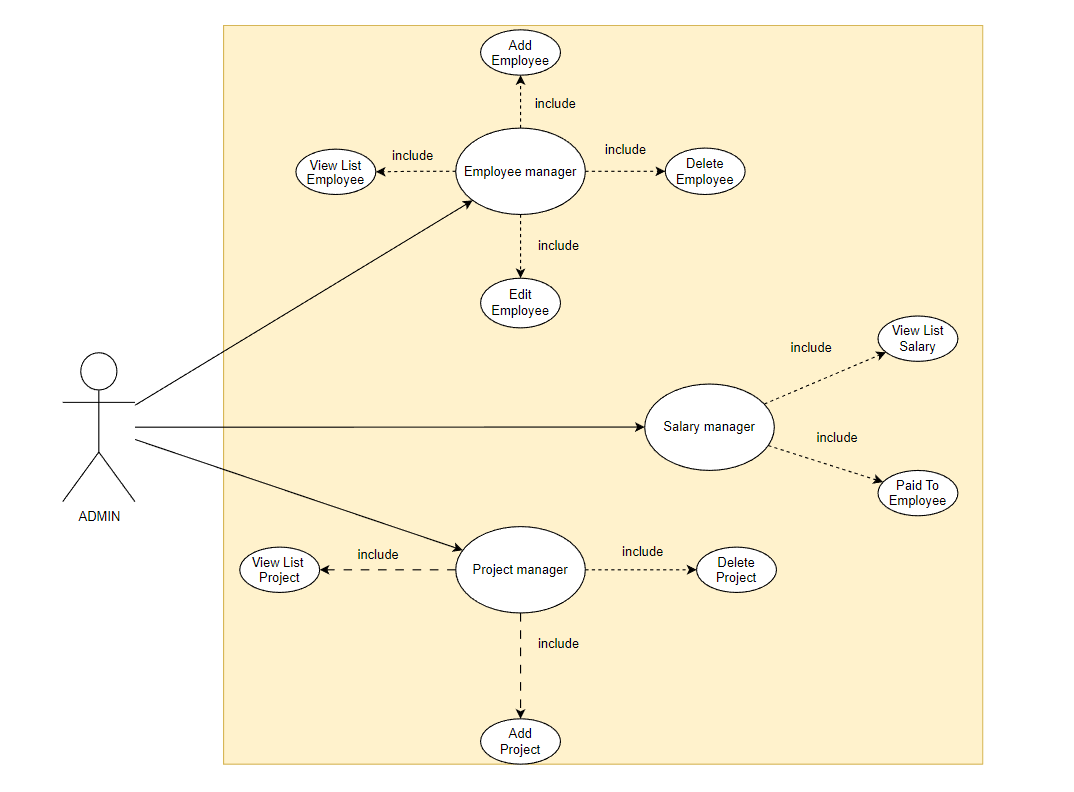


Figure 8: UseCase Diagram

## 3. Class Diagram

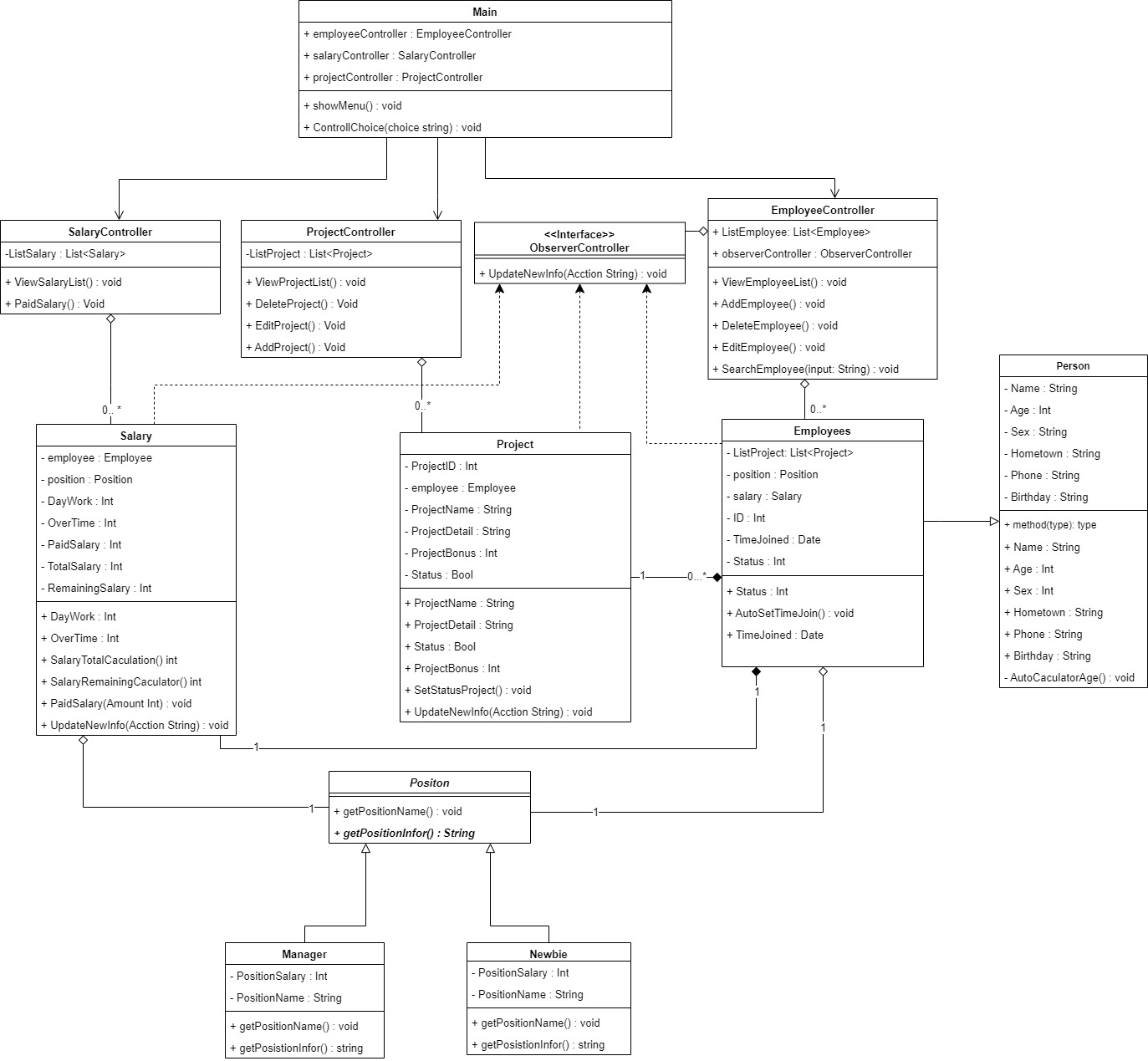


Figure 9: Class Diagram

### 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.

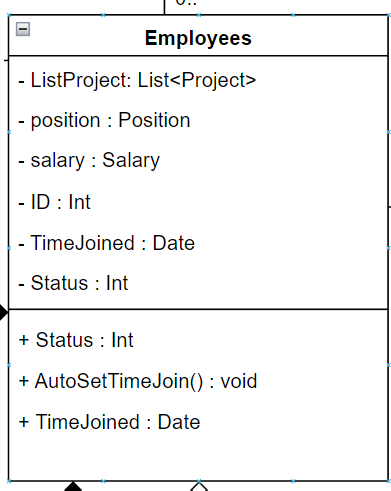


Figure 10: Employees Class

* 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.

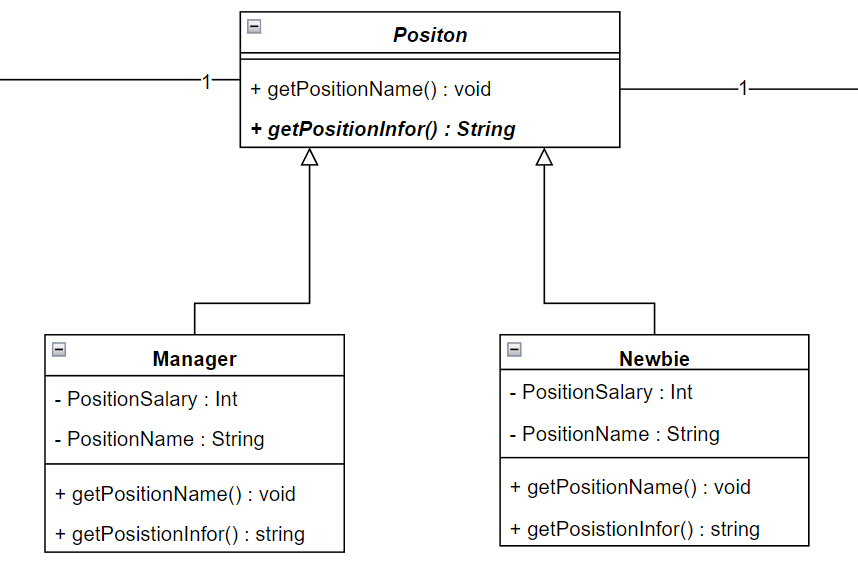


Figure 11: Position Class

* 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.

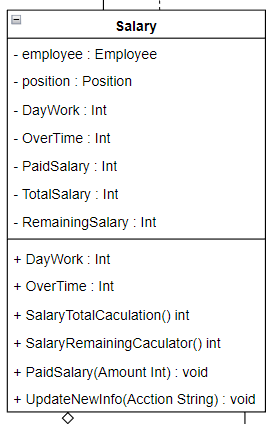


Figure 12: Salary Class

* 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.

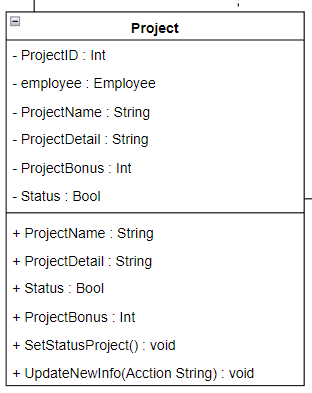


Figure 13: Project Class

* 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

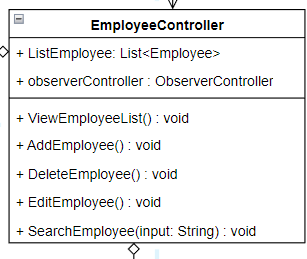


Figure 14: Employee Controller

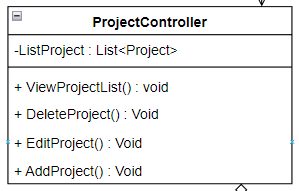


Figure 15: Project Controller

### 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.

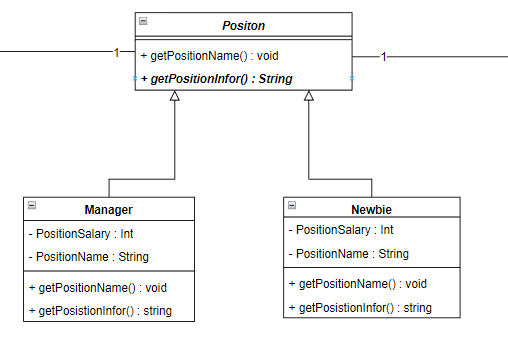
****

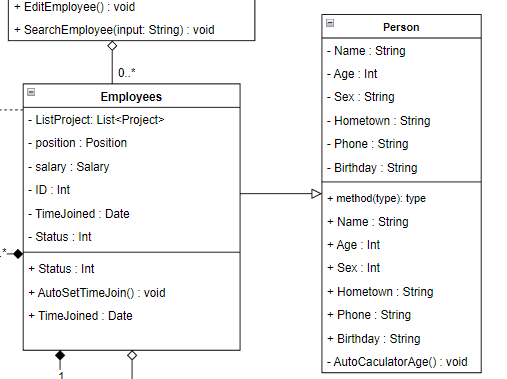
Figure 16: Abtraction Explain

* **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.

****

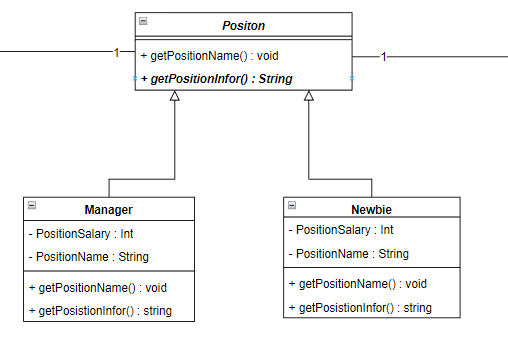
****

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:

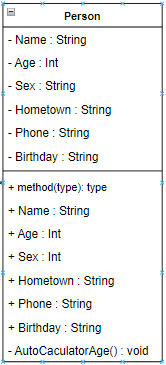
****

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:

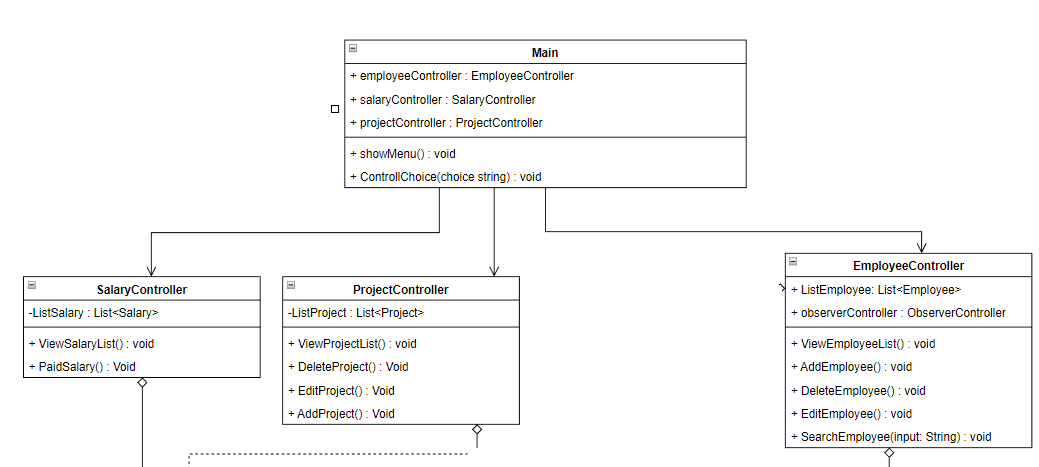


Figure 19: Association Unary

* + 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:

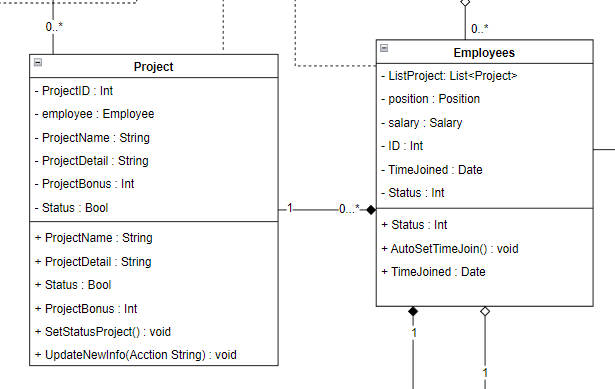


Figure 20: ASsociation Binary

### D. Multiplicity:

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

Example:

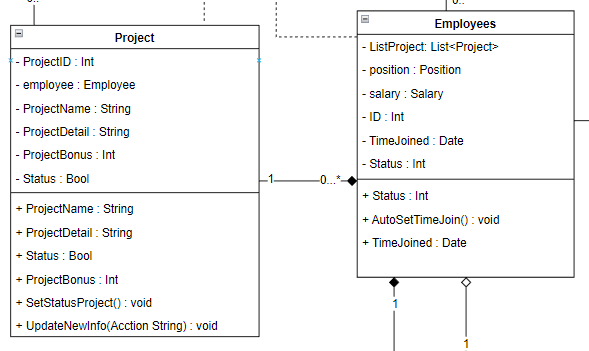


Figure 21: Example For multiplicity

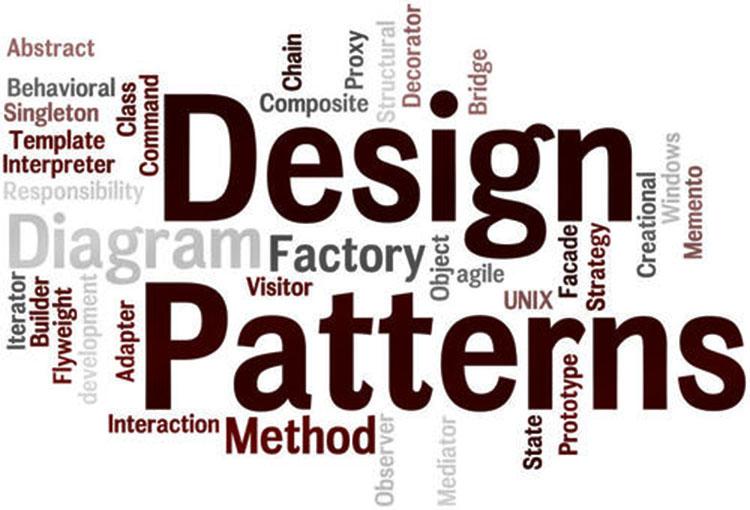
As shown in the picture, the relationship from Project to Employee is 0...\* because an employee can have many Projects and also no Project. For 1 Project, only 1 Employee should be allowed to do it.

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.

* + - Abstract Factory Pattern says that just define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.That means Abstract Factory lets a class returns a factory of classes.

Example UML:

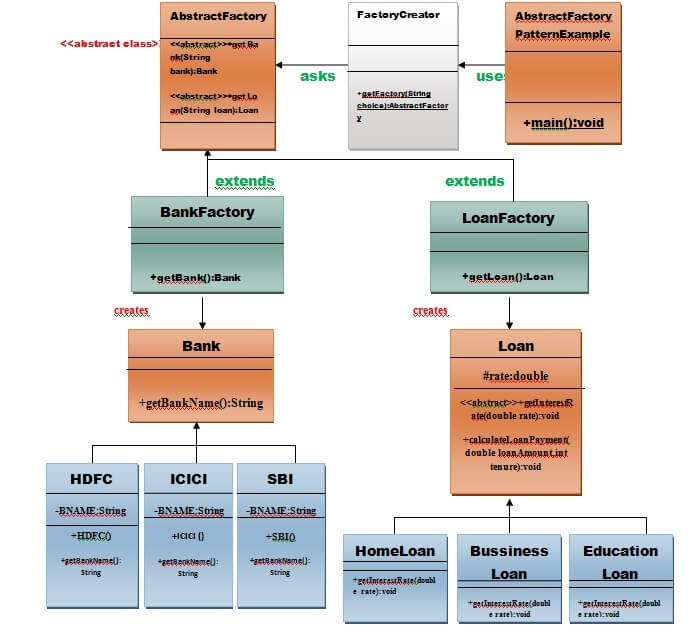


Figure 22: Example Creaional - Abstract Factory Pattern UML

Example Code:

import java.io.\*;

class AbstractFactoryPatternExample {

      public static void main(String args[])throws IOException {

      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

      System.out.print("Enter the name of Bank from where you want to take loan amount: ");

      String bankName=br.readLine();

System.out.print("\n");

System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");

String loanName=br.readLine();

AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");

Bank b=bankFactory.getBank(bankName);

System.out.print("\n");

System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

double rate=Double.parseDouble(br.readLine());

System.out.print("\n");

System.out.print("Enter the loan amount you want to take: ");

double loanAmount=Double.parseDouble(br.readLine());

System.out.print("\n");

System.out.print("Enter the number of years to pay your entire loan amount: ");

int years=Integer.parseInt(br.readLine());

System.out.print("\n");

System.out.println("you are taking the loan from "+ b.getBankName());

AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");

           Loan l=loanFactory.getLoan(loanName);

           l.getInterestRate(rate);

           l.calculateLoanPayment(loanAmount,years);

  }

}//End of the  AbstractFactoryPatternExample

Figure 23: Example Creaional - Abstract Factory Pattern Code

## 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:

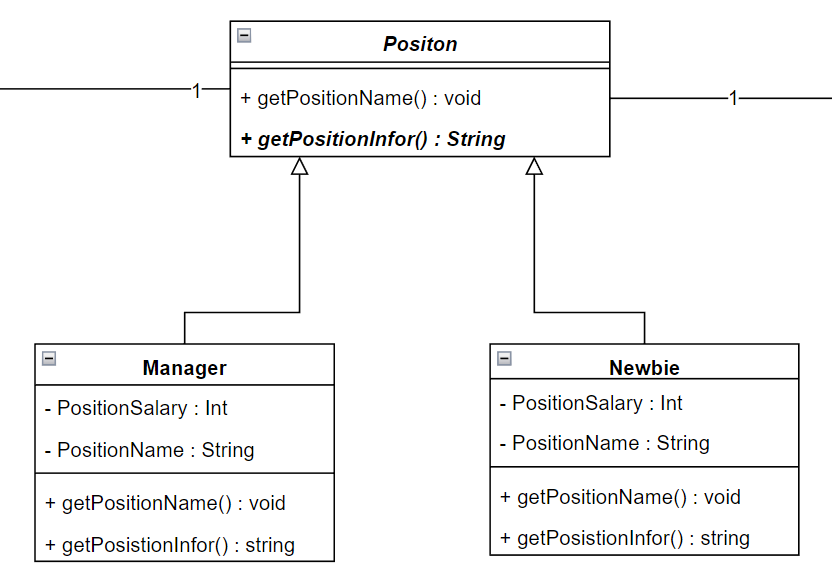


Figure 24: Template Method - Structural Pattern UML

I use Abstract Position along with getPositionInfor() function as a sample function and Manager and Newbie will use it to rewrite it properly.

## 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:

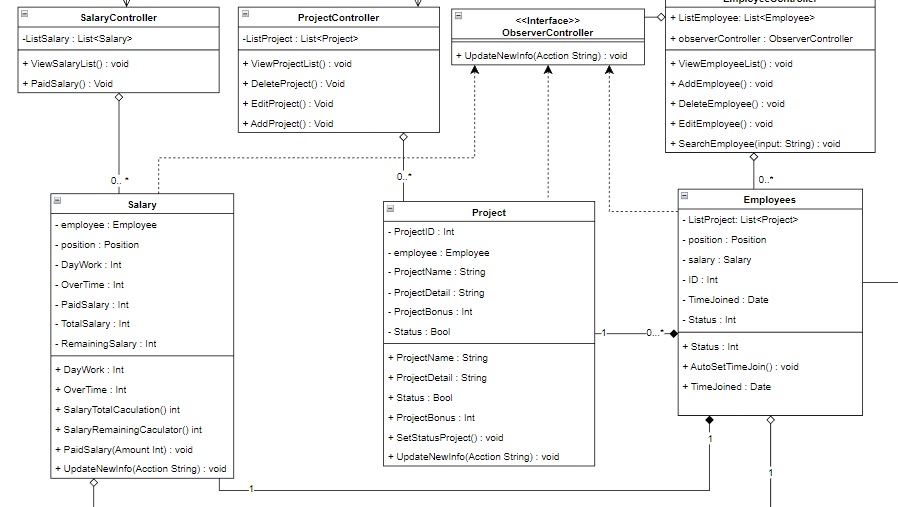


Figure 25: Observer Behavioral Pattern

I use Observer to monitor EmployeeController, if there is an employee-related change, it will require 3 classes: Salary, Project and Employee.

# V. Design Pattern vs OOP

* Design pattern Template Method:

Design pattern Template Method is used in Position class to handle common but possibly different behavior of child objects. In this case, you have 3 child objects of Position: Manager, Newbie and Project. Each child object will provide its own implementation of the methods in the Position class to perform its specific behaviors. This helps you reuse the source code and increase the modularity of the system.

* Design pattern Observer:

The Observer design pattern is used in the ObserverController class to monitor and notify about changes in the data of other objects. The ObserverController can subscribe to notifications from other objects, such as Project or Salary, and when a change occurs in them, the ObserverController will be notified and take corresponding actions.

* Main objects:
  + Project: This object represents a project in your employee management system.
  + Salary: This object represents the salary of an employee.
  + Position: This object has the role of implementing the design pattern Template Method. It includes child objects such as Manager, Newbie, and Project, and provides methods to handle the individual behavior of each child object.
  + Person: Parent class of Employee, representing a person in the employee management system. Maybe you have other subclasses that inherit from Person class like Employee, Manager etc.

In a nutshell, your project uses the design pattern Template Method to manage the behavior of Position objects and the Observer design pattern to manage changes and interactions between objects in the employee management system. The main objects include Project, Salary, Position and Person. ObserverController acts as an observer class and handles notifications from other objects.

# 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>.