


ASSIGNMENT 1 FRONT SHEET

Qualification	BTEC Level 5 HND Diploma in Computing		
Unit number and title	Unit 20: Advanced Programming		
Submission date	9/6/2022	Date Received 1st submission	25/5/2022
Re-submission Date		Date Received 2nd submission	
Student Name	Nguyen Huy Hoang	Student ID	GCH200739
Class	GCH0908	Assessor name	Le Viet Bach
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	

Grading grid

P1	P2	M1	M2	D1	D2

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

Table of Contents

A. INTRODUCTION (Khai).....	5
B. OOP GENERAL CONCEPT	6
a) WHAT IS OOP	6
1. OOP Introduction (Hoang).....	6
2. General Concept of OOP (Hieu).....	7
3. Object and Class in OOP (Huy)	8
4. Pros and Cons of OOP (Khai).....	8
b) APIE	9
1. Inheritance (Hieu).....	9
2. Abstraction (Hoang).....	10
3. Encapsulation (Huy)	10
4. Polymorphism (Huy)	11
c) SOLID	11
1. S - Single-responsibility Principle (Hieu).....	11
2. O - Open-closed Principle (Khai).....	12
3. L - Liskov Substitution Principle (Hoang).....	12
4. I - Interface Segregation Principle (Hoang)	12
5. D - Dependency Inversion Principle (Huy)	12
C. OOP SCENARIO	12
3.1 Scenario (Khai).....	12
3.2 Use case Diagram (Hieu)	14
3.3 Class Diagram (Hoang).....	19
D. Design Patterns (Huy).....	21
Importance of Design Pattern: (Khai)	21
When to use Design Patterns: (Hoang).....	21
4.1 Creational pattern (Hieu)	22
Description of a creational scenario (Khai)	26
4.2 Structural pattern (Hoang)	28
Description of a structural scenario (Huy)	32

4.3 Behavioral pattern (Khai).....	34
Description of a behavioral scenario (Hoang)	43
E. Design Pattern vs OOP (Huy)	45
F. Conclusion (Khai).....	46
References	46

List Of Tables

Table 1: StartGame Usecase	15
Table 2: PlayGame use case	16
Table 3: Change Setting Usecase.....	17
Table 4: View Records usecase	17
Table 5: Exit Usecase.....	18
Table 6: In-game menu usecase.....	19

List Of Figures

Figure 1: OOP Introduction.....	6
Figure 2: OOP Concept.....	7
Figure 3: Inheritance Example	9
Figure 4: Abstraction Example	10
Figure 5: Encapsulation Example	11
Figure 6: Polymorphism Example.....	11
Figure 7: Usecase for Scenario	14
Figure 8: Class Diagram for scenario	19
Figure 9: Abstract Factory.....	23
Figure 10: Factory Method.....	24
Figure 11: Builder Pattern	24
Figure 12: Prototype Pattern	25
Figure 13: Singleton Design Pattern	26
Figure 14: Class diagram of creational scenario.....	27
Figure 15: Adapter Pattern.....	28
Figure 16: Bridge Pattern	29
Figure 17: Composite Design pattern	29
Figure 18: Decorator Design Pattern.....	30
Figure 19: Facade Design pattern	31

Figure 20: Flyweight Design Pattern	31
Figure 21: Proxy Design pattern.....	32
Figure 22: Class diagram of structural Scenario.....	33
Figure 23: Chain of Responsibility.....	34
Figure 24: Command Design pattern.....	35
Figure 25: Interpreter Pattern.....	36
Figure 26: Iterator Design Pattern.....	37
Figure 27: Mediator Pattern.....	38
Figure 28: Memento Design Pattern	38
Figure 29:Observer Design Pattern.....	39
Figure 30: State Design Pattern	40
Figure 31: Strategy Design pattern.....	41
Figure 32: Template Method design pattern	42
Figure 33: Visitor Design Pattern.....	43
Figure 34: Class Diagram of Behavioral Scenario	44

A. INTRODUCTION (Khai)

A design pattern is a reusable solution for common problems in software design with a specific circumstance. A design pattern is a blueprint that cannot be converted into source code. Typically, object-oriented design patterns depict relationships and interactions between classes or objects but do not specify the classes or objects involved in the final application.

This is a report about object-oriented paradigms and design patterns. First, we will learn about OOP and the general concepts of OOP. Next, we will analyze in detail a situation related to OOP. Design patterns will be introduced and analyzed its general concepts. The article will compare and analyze the relationship between object-oriented paradigms and design patterns. Finally, we will design and build class diagrams using a UML tool.

B. OOP GENERAL CONCEPT

a) WHAT IS OOP

1. OOP Introduction (Hoang)



Figure 1: OOP Introduction

With the launch of Simula in the mid-1960s, OOP principles initially appeared, and they expanded further in the 1970s with the introduction of Smalltalk. Object-oriented methods evolved even though software developers did not adopt early breakthroughs in OOP languages. In the mid-1980s, there was a resurgence of interest in object-oriented methods. OOP languages, C++, and Eiffel have grown popular among mainstream computer programmers. Throughout the 1990s, OOP increased in prominence, most notably with the launch of Java and the enormous following it acquired.

Furthermore, in 2002, in conjunction with the release of the .NET Framework, Microsoft launched C (pronounced C-sharp), a new OOP language, and a redesign of their hugely popular existing language, Visual Basic, to make it fully object-oriented. OOP languages are still popular today and play a significant role in contemporary programming (Alexander S. & Sarah Lewis, 2017).

2. General Concept of OOP (Hieu)

OOPs (Object-Oriented Programming System)

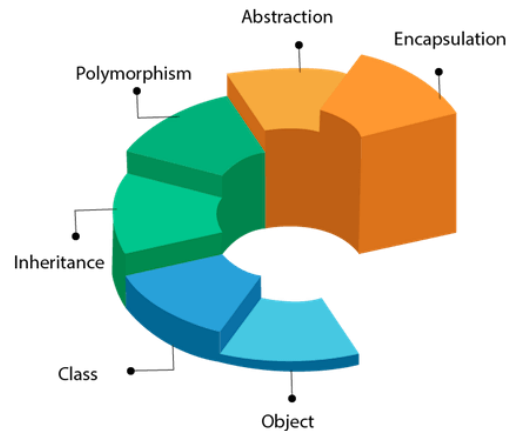


Figure 2: OOP Concept

Object-oriented programming is a way of creating software in which the structure is built on objects interacting with one another to achieve a goal. As part of this interaction, messages are sent between the objects. An object can act in response to a message.

OOP is a programming technique that allows programmers to create objects in code that abstracts real-life objects. This approach is now successful and has become one of the paradigms for software development, especially for enterprise software.

During the development of OOP applications, classes will be defined to model actual objects. These classes will be executed as objects. When users run the application, the methods of the object will be called.

The class defines what the object will look like: what methods and properties will be included. An object is just an instance of the class. Classes interact with each other by the public API: a set of methods, and its public properties.

The goal of OOP is to optimize source code management, increase reusability, and most importantly, help encapsulate procedures with known properties using objects (Alexander S. & Sarah Lewis, 2017).

3. Object and Class in OOP (Huy)

3.1. Class

In OOP, a class is frequently understood as a blueprint for creating objects (a specific data structure), providing initial values and constructors for the nature (instance variables or attributes), and defining and implementing behaviour (member functions or methods).

Classes are used to generate and manage new objects, as well as to facilitate inheritance, which is a crucial component of object-oriented programming and a technique for code reuse (Oracle, 2012).

3.2. Object

A class instance is an object. In OOPS, an object is a self-contained component with methods and attributes that make a specific type of data usable.

An object in OOPS might include a data structure, a variable, or a function from a programming standpoint. It has a memory area set aside for it.

4. Pros and Cons of OOP (Khai)

4.1 Pros

- OOP models complex things as simple structures.
- Reusable OOP code, which saves resources.
- It makes debugging easier. Compared to finding errors in many places in the code, finding errors in (pre-structured) classes are more straightforward and less time-consuming.
- High security, protecting information through packaging.
- Ease of project expansion.

4.2 Cons

- Make data processing separate.
- When the data structure changes, the algorithm must be modified.
- OOP does not auto initialize and release dynamic data, and it does not accurately describe the actual system.

b) APIE

1. Inheritance (Hieu)

In OOP, inheritance is used to classify objects in your programs based on shared characteristics and functions. Working with objects becomes more accessible and more intuitive as a result. It also facilitates programming by allowing you to combine typical characteristics into a parent object and inherit these characteristics in child objects (Alexander S. & Sarah Lewis, 2017).

This is a feature commonly used in software development. Inheritance allows creating a new class (Child class), inheriting, and using properties and methods based on the parent class.

The Child classes inherit all the Parent class components. Subclasses can extend the components or add new ones (Nitendratech, 2018).

Example:

There are many types of vehicles. In this case, the Vehicle can be considered as the base class with properties color, wheel, engine, weight... and actions Start, Stop, Turn. "Car" and "Motorbike" are two derived classes that inherited from the base class. Both classes inherit all attributes and actions from the base class but "Car" also contains action Reverse and "Motorbike" contains action kickstand

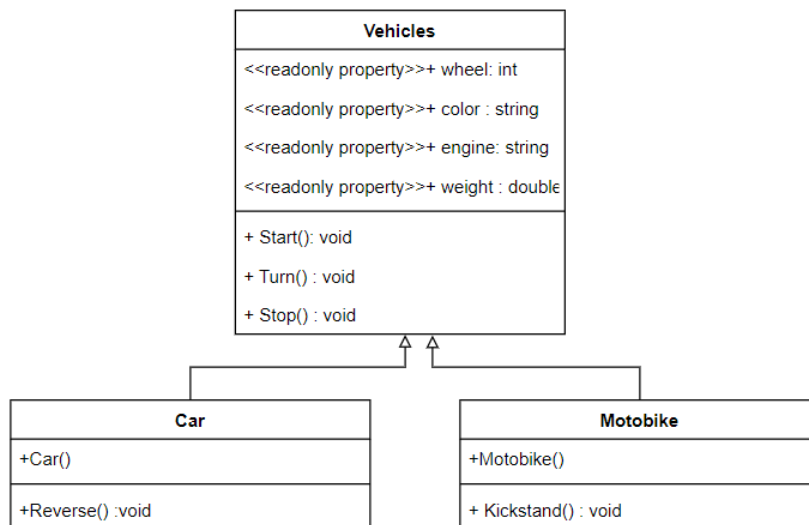


Figure 3: Inheritance Example

2. Abstraction (Hoang)

Abstraction eliminates the unnecessary complexity of the object and focuses only on the core, essential things and preserving information relevant in a given context. Abstraction helps manage complexity

Example: Interface called aircraft and two subclasses – helicopter and airplane. Both subclasses have common actions to perform such as takeoff, fly, land, speedup, speed Down... There actions need to be implemented when it implements aircraft interface (Nitendratech, 2018).

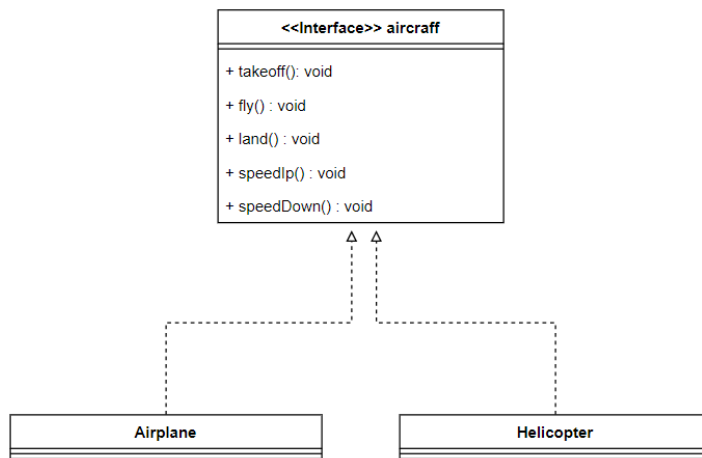


Figure 4: Abstraction Example

3. Encapsulation (Huy)

One of the four primary concepts of OOP, also known as the Protection of data from any accidental corruption. It plays a role as a mechanism that restricts the direct access (which may harm data) to some data members of an object by hiding its properties and methods of a given class. Encapsulation is demonstrated by a class, which bundles data and methods that operate on that data into a single unit.

Example:

Classes that include crucial properties, such as the “user account” class, which has the username and password properties (which always must be prevented from direct access of strangers), need to be implemented encapsulation for this class (W3schools, n.d.).

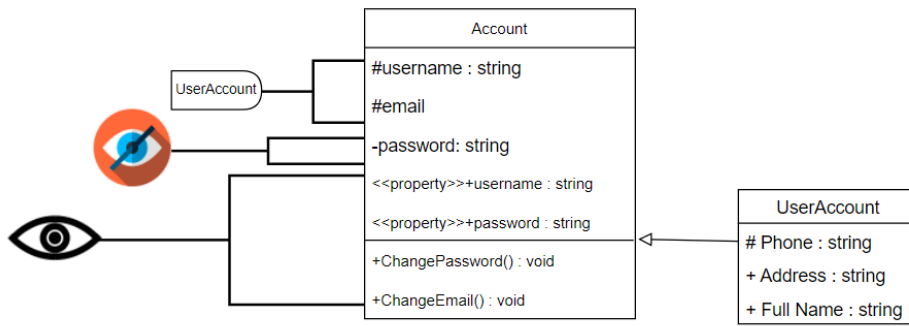


Figure 5: Encapsulation Example

4. Polymorphism (Huy)

In OOP programming, polymorphism permits distinct objects to perform the same function in various ways.

Example:

In the Phone class, each brand inherits the components of the parent class, but iPhone runs on the iOS operating system, and Samsung runs on the Android system.

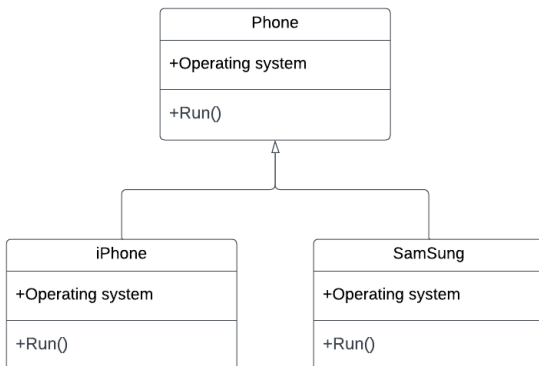


Figure 6: Polymorphism Example

c) SOLID

5 principles of OOP

1. S - Single-responsibility Principle (Hieu)

Each class should have only one function. When a class has more than one functionality, the code becomes extremely hard to read, and prone to errors and bugs. This can overload the class and make

it more challenging to maintain. Having only one function per class makes the code easier to read and fix (TopDev, 2019).

2. O - Open-closed Principle (Khai)

A class can be extended and evolved, but what already exists cannot be modified. We can do this by writing a new class to extend the old class. This makes it easier to test by just testing new classes that contain new functionality

3. L - Liskov Substitution Principle (Hoang)

Objects in a program should be replaceable with instances of their subtypes without affecting the correctness of that program.

4. I - Interface Segregation Principle (Hoang)

Do not use an interface for many purposes. Divide it into many interfaces with specific purposes for better management and implementation. Clients should not be forced to use interfaces they do not want or use functions they do not need.

5. D - Dependency Inversion Principle (Huy)

High-level modules should not rely on low-level modules. Both modules should depend on abstraction. Abstraction should not depend on details but details should depend on abstractions. Classes communicate with each other through the interface (abstraction), not through the implementation.

C. OOP SCENARIO

3.1 Scenario (Khai)

We are assigned to develop an offline game. This game aims to provide an entertaining game when the user's device is not connected to the internet.

Player: When you run the game. The main screen with a menu will appear. You can choose different options from the menu. If you decide to start a new game, first, you will select a character to play. You can name your plane if you want. When you are ready, click start to start playing the game. When encountering enemies' planes and obstacles, players can shoot it down or avoid it to survive. Players will earn points by taking down enemies' planes and surviving as long as possible. The score of the current game and the best score will be displayed after each game.

Characters: there are three different planes for you to choose from: The flash, the Conqueror, and the Sky Doom. Each plane has its unique skill. The flash is very fast, but it has low health. His unique skill is Flashy Flash, which gains five-second invulnerability and attack speed. Conqueror has the highest health bar but has the lowest speed. Its skill is Unstoppable: Conqueror will shoot explosive bullets instead of regular bullets for ten seconds. Last is the Sky Doom. This plane is more balanced in health and speed. Sky Doom's special skill is Making it rain, which is called a rocket rain that deals massive damage in three seconds.

Enemies: Player will face four types of planes: spy plane, light aircraft, heavy aircraft, and the flying fortress. Spy planes are small airplanes that do not shoot back and fly very fast in a zigzag pattern. Light aircraft have more health than a spy plane. It can fire one bullet at your plane every three seconds. Light aircraft fly in a straight line. Heavy aircraft are even hard to take down. It can shoot two bullets and travel at a slow speed at your plane every five seconds. Heavy aircraft stays in one place for ten seconds, then goes straight line toward you. Flying fortress is the most brutal enemy you will face. It has a massive health bar, and his attack will change each time you meet it. Flying fortress has a special attack that changes each time you encounter it.

Special items: some items help you survive in the game: health potion, shield potion, freeze potion, and The Nuke. Health potions help you restore your health. The shield protects you from enemies' projectiles once. The freeze potion will immobilize enemies for 5 seconds.

Obstacles: Big rockets and small rockets are the things you must avoid or shoot down. Big rocket has high health but has a low speed. A small rocket is faster but low on health. A big rocket can deal three damages to your plane, and a small one can deal one damage.

3.2 Use case Diagram (Hieu)

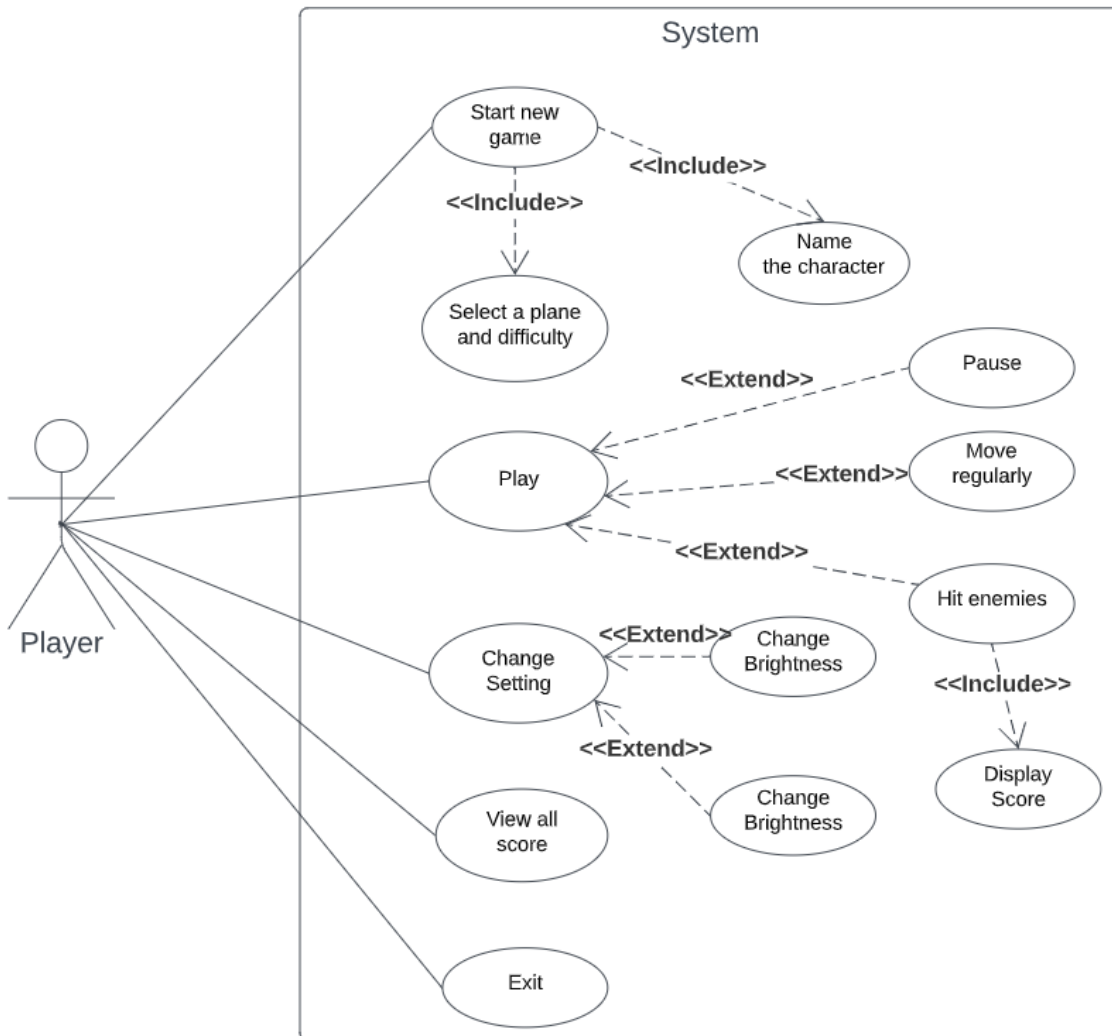


Figure 7: Usecase for Scenario

Explanation:

♥ Start game usecase

Use case Name	Start Game		
Created by:	Group1	Last Updated By:	Mr Hoang
Date Created	28/05/2022	Last Revision Date	28/05/2022
Description:	This use case describes the process of starting the game		
Actors:	♥ Player		
Pre-conditions:	♦ The user had to download and install this game;		
Post Conditions:	♣ The system handles and stores the data of the selections of the user		
Flow:	1) User clicks on the “start game” button on the game menu; 2) User selects Plane and difficulty 3) User name for the chosen plane 4) Finish		
Alternative Course:			
Exception:	♠ Game is no longer approved for use(occurs at step 1); ♠ Crashing;		
Requirements:	The appropriate of the system and user;		

Table 1: StartGame Usecase

♥ PlayGame usecase

Use case Name	Play Game		
Created by:	Group1	Last Updated By:	Mr Hoang
Date Created	28/05/2022	Last Revision Date	28/05/2022
Description:	This use case describes the process of Playing the game		
Actors:	♥ Player		
Pre-conditions:	♦ The user had to download and install this game; ♦ The user has already done the “start game” use case;		
Post Conditions:	♣ The system handles and stores the data of the selections of the user ♣ The system stores the score of the user		
Flow:	1. User plays this game; 2. System displays the score of the user;		

	3. System updates the time. System and user repeat steps 1 to 2 until user loses; 4. Finish;
Alternative Course:	In step 1 of the Flow, if the user doesn't want to stand all the time, the user can: <ol style="list-style-type: none"> 1) Constantly moving regularly around; 2) Hitting the enemies appearing in the game; 3) Click on the in-game menu; The use case continues at step 1 of the flow
Exception:	♠ Game is no longer approved for use(occurs at step 1); ♠ Crashing;
Requirements:	The appropriate of the system and user;

Table 2: PlayGame usecase

♥ Change Setting Usecase

Use case Name	Change setting		
Created by:	Group1	Last Updated By:	Mr Hoang
Date Created	28/05/2022	Last Revision Date	28/05/2022
Description:	This use case describes the activities sequence of changing the setting		
Actors:	♥ Player		
Pre-conditions:	♦ The user had to download and install this game;		
Post Conditions:	♣ The system handles and stores the data of the selections of the user		
Flow:	<ol style="list-style-type: none"> 1. User clicks on the "setting" button on the menu; 2. System displays the selections in the menu; 3. User makes change; 4. The system team stores the changes; 5. System and user repeat steps 1 to 4; 6. Finish; 		
Alternative Course:	In step 3 of the Flow, the user can make alternative selections: <ol style="list-style-type: none"> 1. Change brightness; 		

	2. Change volume; 3. Change nothing; The use case continues at step 4 of the Normal Course
Exception:	♠ Game is no longer approved for use(occurs at step 1); ♠ Crashing;
Requirements:	The appropriate of the system and user;

Table 3: Change Setting Usecase

♥ View Records Usecase

Use case Name	View Records		
Created by:	Group1	Last Updated By:	Mr Hieu
Date Created	28/05/2022	Last Revision Date	28/05/2022
Description:	This use case describes the moment when the user views records		
Actors:	♥ Player		
Pre-conditions:	♦ The user had to download and install this game;		
Post Conditions:	♣ The system handles the data of the selections of the user		
Flow:	1. User clicks on the “Records” button on the menu; 2. System handles and gets the data from the database; 3. System displays the records of the user in the menu; 4. Finish;		
Alternative Course:			
Exception:	♠ Game is no longer approved for use(occurs at step 1); ♠ Crashing;		
Requirements:	The appropriate of the system and user;		

Table 4: View Records usecase

♥ Exit Usecase

Use case Name	Exit		
Created by:	Group1	Last Updated By:	Mr Hieu
Date Created	28/05/2022	Last Revision Date	28/05/2022

Description:	This use case describes the moment when the user clicks exit this game
Actors:	♥ Player
Pre-conditions:	♦ The user had to download and install this game;
Post Conditions:	♣ The system handles the data of the selections of the user
Flow:	1. User clicks on the “Exit” button on the menu; 2. System handles the selection of the user; 3. System terminate all program of this game; 4. Finish;
Alternative Course:	
Exception:	♠ Lag;
Requirements:	The appropriate of the system and user;

Table 5: Exit Usecase

♥ Select in-game Menu usecase

Use case Name	Select in-game menu		
Created by:	Group1	Last Updated By:	Mr Huy
Date Created	28/05/2022	Last Revision Date	28/05/2022
Description:	This use case describes the moment when the user selects the in-game menu		
Actors:	♥ Player		
Pre-conditions:	♦ The user had to download and install this game; ♦ The user had already done the “start game” use-case		
Post Conditions:	♣ The system handles the data of the selections of the user		
Flow:	1. User clicks on the “Menu” button on the top-left of the game screen; 2. System displays the selection of the user; 3. User makes selections; 4. User click on the resume button; 5. Finish;		
Alternative Course:	In step 3 of the Flow, the user can make alternative selections: 4. Click on the “setting” button to change the game’s setting; 5. Exit this game; 6. Change nothing;		

	The use case continues at step 3 of the Normal Course
Exception:	<ul style="list-style-type: none"> ♠ Game is no longer approved for use(occurs at step 1); ♠ Crashing;
Requirements:	The appropriate of the system and user;

Table 6: In-game menu usecase

3.3 Class Diagram (Hoang)

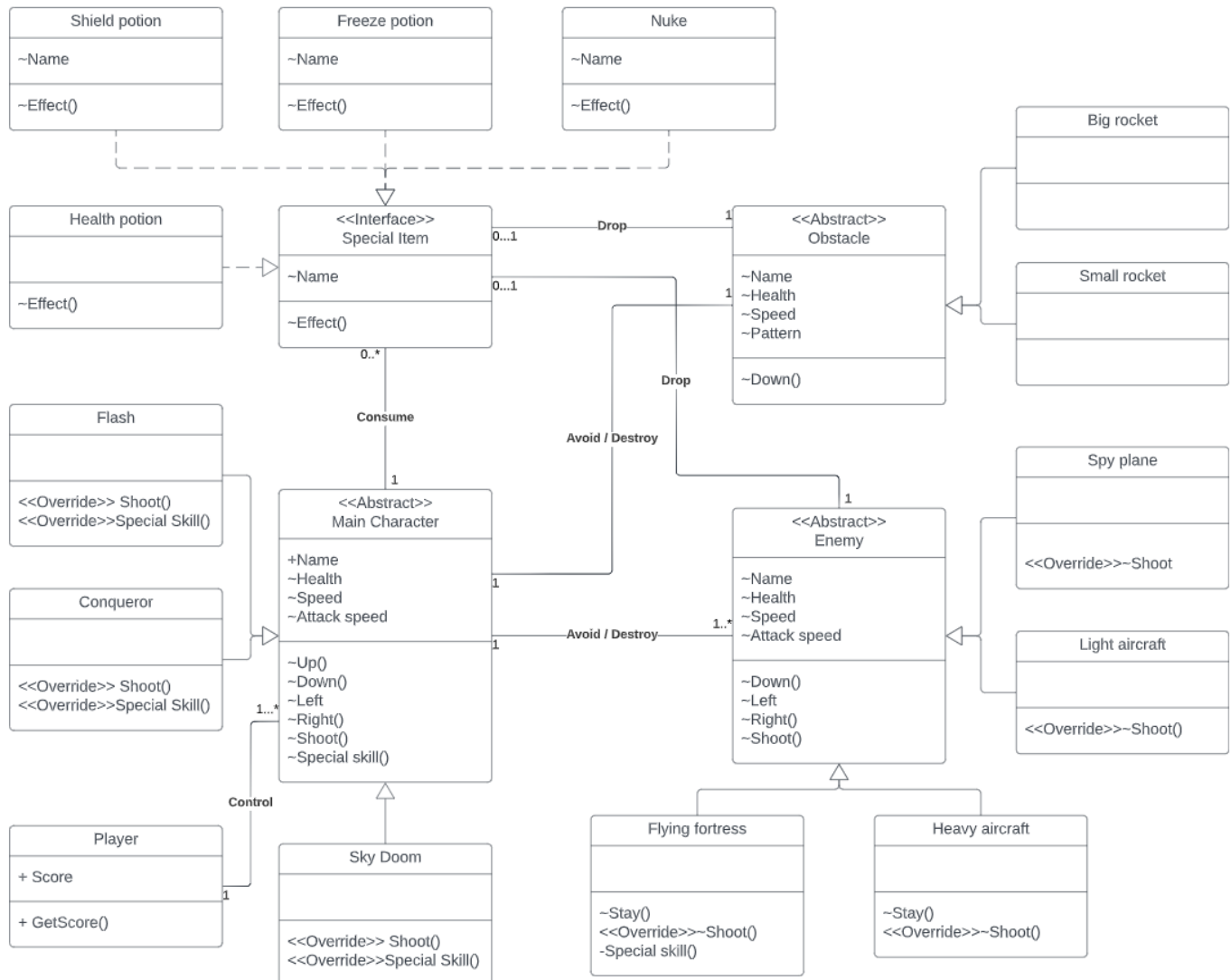


Figure 8: Class Diagram for scenario

Explanation:

The abstract main character (Character that the player will control) class: Main Character is an Abstract class which contains the general properties of a combat aircraft. These concrete classes: Flash, Conqueror, and Sky Doom will inherit the Main character abstract class. Flash, Conqueror, and Sky Doom will override the Shoot action and special skill action of the Main character class.

The enemy is an Abstract class which contains the general properties of these subclasses which will inherit the Enemy class: Spy Plane, Light aircraft, heavy aircraft, and Flying Fortress. Spy Plane, Light aircraft, heavy aircraft, and Flying Fortress concrete class override Shoot action of the Enemy class. Flying fortress will have its own new action is Special skill.

The obstacle is an abstract class which represent the general obstacles in this game and it is inherited by two concrete class: big rocket and small rocket(the primary obstacles appear in-game).

The special item is an interface. All items will have an effect and will be different to each other. Moreover, Special Item interface is implemented by these four concrete classes: Health Poition, Shield Poition, Freeze Poition and Nuke

Player, a concrete class, which performs the one task is storing the score data and the program can get these data.

Among these classes in the above class diagram, there additionally are some multiplicity relationships and it defines a specific range of allowable instances of attributes. Between main character and enemy, the multiplicity is 1 and 1...* it shows that only one instance of character performs avoid or destroy the amount of instances of enemy. Between main character and obstacle also is the same, the multiplicity is 1 and 1...* it shows that only one instance of main character performs avoid or destroy the amount of instances of obstacle. Between main character and special item, the multiplicity is 1 and 0...* it vaguely shows that there is only one instance of main character which could performs consume some special item or not.

Between special item and special item as well as special item and obstacles, both of them also has the multiplicity is 0...1 and 1. Thereby, it vaguely shows that one instance of enemy could drop nothing or one instance of speacial item. Eventually, the multiplicity between player class and main character is 1 and 1...*. It simply shows that one instance of player class could controll one or more instance of character based on the number of times played.

D. Design Patterns (Huy)

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations (Design Patterns and Refactoring, 2022).

Design patterns are recycled, optimized overall solutions to everyday software design challenges. This is a collection of solutions that have been considered and solved in a specific context. You must understand that it is not a distinct language. Most language configurations support design patterns. It assists you in solving the problem in the most efficient manner by delivering solutions in object instruction set (OOP).

The design pattern system is divided into three groups: Creational, Structural, and Behavioral.

Importance of Design Pattern: (Khai)

- Making our goods more adaptable, changeable, and easy to maintain.
- The requirement changes are something that constantly happens in software development. At this point, the system is expanding, new features are being introduced, and performance must be enhanced.
- In software engineering, design patterns give optimal, tried-and-true solutions to challenges. The solutions are generic, which aids in the acceleration of software development by offering test models and tested development models.
- When faced with a problem that has previously been addressed, design patterns might assist you in solving it rather than searching for a time-consuming solution yourself.
- Help programmers easily grasp other people's code (can be understood as relationships between modules, for example). All team members can readily interact with one another in order to complete the project quickly.

When to use Design Patterns: (Hoang)

- The adoption of design patterns will help us save time and effort in thinking of solutions to previously addressed problems. The advantage of utilizing Design Patterns in software is that it makes the program operate more smoothly, makes the operation process easier to manage, is easy to upgrade and maintain, and so on.

- The problem of design patterns is that they are usually a complex and sometimes abstract area. When building fresh code from the start, it's simple to understand the value of a design pattern. Applying the design pattern to outdated code, on the other hand, is more complex.
- When we use the available design patterns, we will run across another issue: product performance (code will run slowly, for example). Before touching the code, make sure you understand how it all works. Depending on the intricacy of the code, this might be simple or difficult.
- We are now using a lot of design patterns in our programming work. If you often download and install specific libraries, packages, or modules, it's time to include a design pattern into the system.
- All web application frameworks, such as Laravel and CodeIgniter, employ the available design pattern architecture, and each framework has its own design pattern.
- Design Pattern uses the foundation of object-oriented programming, so it applies 4 characteristics of OOP: Inheritance, Polymorphism, Abstraction, Encapsulation (Techopedia, 2011).
- Understand and apply the two concepts interface and abstract because it is very necessary.

4.1 Creational pattern (Hieu)

These design patterns focus on class instantiation. This pattern can be divided into class-creation patterns and object-creational patterns. In the instantiation process, class-creation patterns make effective use of inheritance, whereas object-creation patterns make effective use of delegation.

Creational Pattern include:

- ♥ **Abstract Factory:** Creates an instance of multiple class families.

According to the Abstract Factory Pattern, create families of linked (or dependent) items by simply defining an interface or abstract class without identifying their particular sub-classes. This means that a class can return a factory of classes using Abstract Factory. As a result, the Abstract Factory Pattern is a level higher than the Factory Pattern (Javatpoint, n.d.).

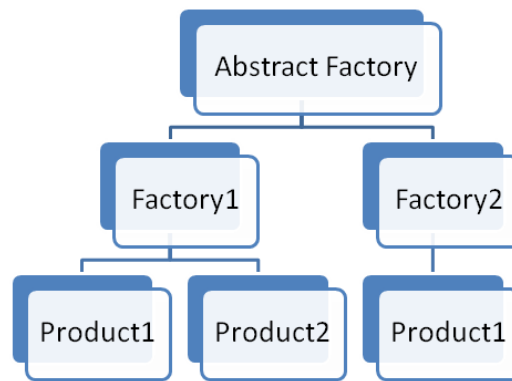


Figure 9: Abstract Factory

ImageCre: C# corner

Usage of Abstract Factory Pattern:

- When the system must be independent of the making, putting together, and showing of its objects.
- This constraint must be applied when a group of related objects must be used concurrently.
- When it's necessary to give a library of objects that only shows interfaces and hides implementations.
- When the system must be set up using one of several object families.

Advantage of Abstract Factory Pattern:

- With the Abstract Factory Pattern, client code and concrete (implementation) classes are kept separate.
- It makes it easier to move object families.
- It helps object consistency.

♥ **Factory Method:** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. You can override the factory method in a subclass and change the class of products being created by the method. When products have a common base class or interface, subclasses could return different types of products (Javatpoint, n.d.).

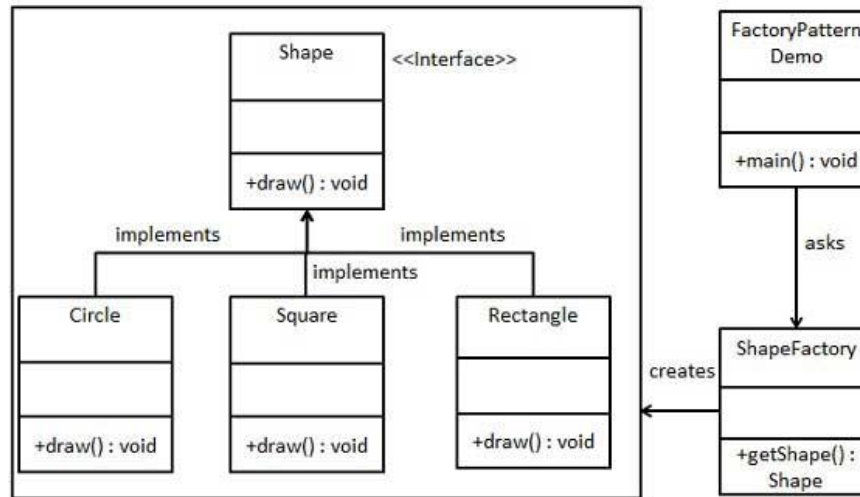


Figure 10: Factory Method

ImageCre: Tutorialpoint

♥ **Builder:** is a creational design pattern that lets you build complex objects step by step. The pattern lets you use the same construction code to make different types and representations of an object. The Builder pattern suggests that you should transfer the code for building an object out of its class and into a separate object called builders. To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object (Javatpoint, n.d.).

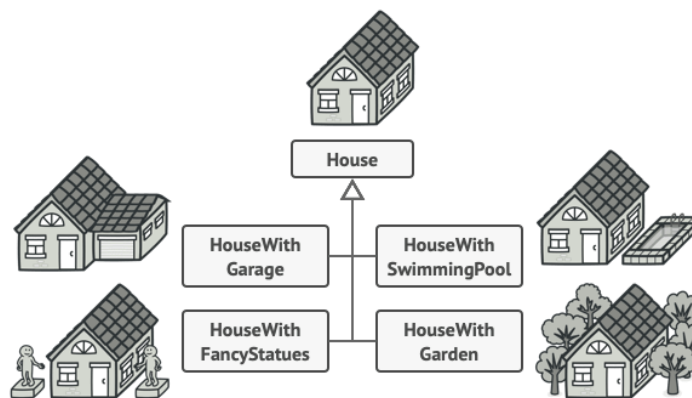


Figure 11: Builder Pattern

ImageCre: Builder Pattern RefactoringGuru

- ♥ **Prototype:** Prototype: is a creational design pattern that lets you copy existing objects without making your code rely on their classes. The Prototype pattern gives the cloning process to the actual objects that are being cloned. The pattern declares that all objects that can be copied have the same interface. With this interface, you can copy an object without tying your code to its class.

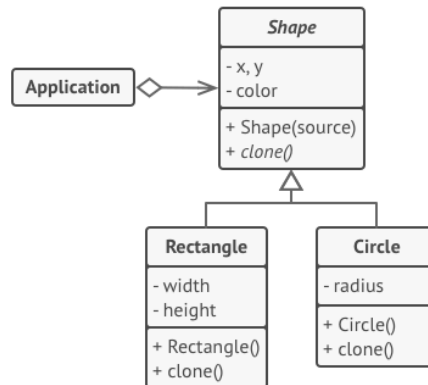


Figure 12: Prototype Pattern

ImageCre: Prototype Pattern medium.com

- ♥ **Singleton:** The singleton design pattern ensures that a class has only one instance while offering a global access point to this instance. These two steps are shared by all Singleton implementations:
- Make the default constructor private, to prevent other objects from being used the new operator with the Singleton class.
 - Create a static creation method that acts as a constructor. Below, this method calls the private constructor to make an object and saves it in a static field. When this method is called again, the cached object is returned (Anon., 2022).

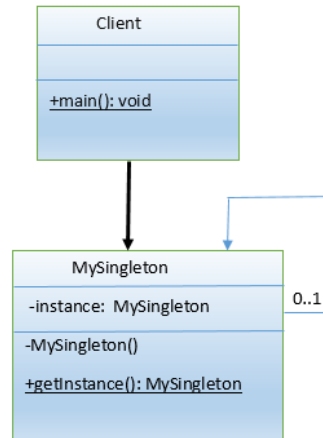


Figure 13: Singleton Design Pattern

ImageCre: Singleton Pattern Dzone

Description of a creational scenario (Khai)

There are three types of base planes for players to choose from. Each plane has its differences:

Flash: health = 75, speed = 150. Special skill: ignore enemies' projectiles damage and gain double attack speed for 5 seconds

Conqueror; health = 150, speed = 75, Special Skill: Heal five health per second normal attack become super attack that shot an exploding grenade for 5 seconds.

Sky Doom health = 125, speed = 100, Special Skill: Make all enemies' planes and enemies' projectiles slower by 50% for 15 seconds, and call a random special item.

After selecting character, players can name the plane or using the default name.

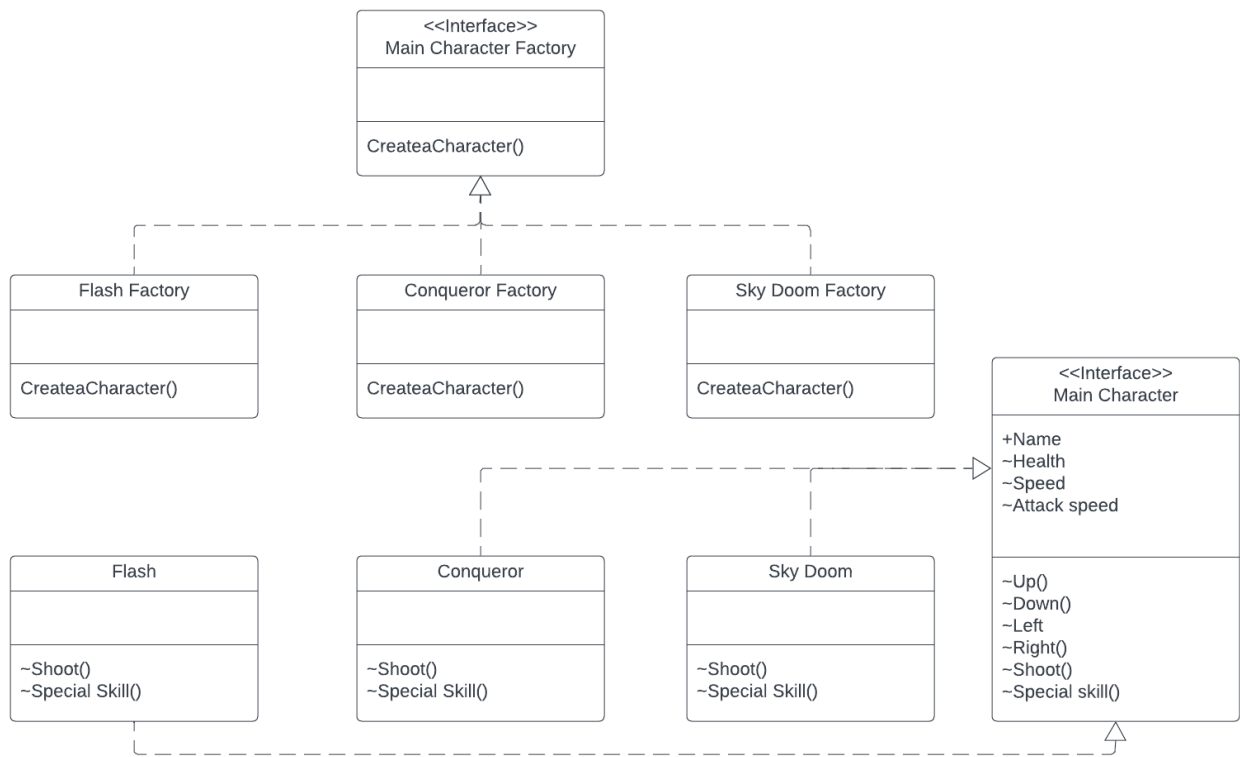


Figure 14: Class diagram of creational scenario

Explain:

The Main Charactor Factory interface defines a set of methods that return different Planes. These Charaters could be seen as a high-level conception relation. These Charaters have several differences, but the Charater of one variant are incompatible with Charaters of another.

Each Concrete Factory class has a corresponding character variant.

CreateaCharacter() method in Main Charater Factory interface is returning an instance of Main Character. Then these factory classes(FlashFactory, ConquerorFactory, SkyDoomFactory) will implement this Main Charater Factory interface and return their respective sub-class.

Each concrete character class (Flash, Conqueror, Sky Doom) also implement the MainCharater interface. Concrete character object are declared by corresponding Concrete Factory class.

4.2 Structural pattern (Hoang)

Class and Object composition is central to these design patterns. Inheritance is used to construct interfaces in structural class-creation patterns. Structural object patterns outline how objects may be combined to create new functionality.

Structural Pattern include:

♥ **Adapter:** Match interfaces of different classes

The adapter pattern acts as a link between two interfaces that are incompatible. This design pattern is classified as a structural pattern since it integrates the capabilities of two separate interfaces.

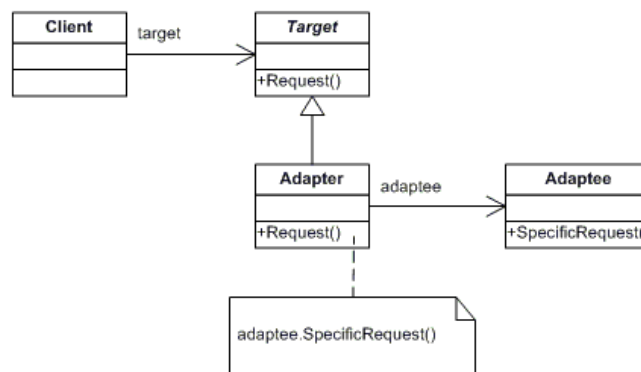


Figure 15: Adapter Pattern

Image Cre: Adapter Design Pattern - Stack Overflow

A single class is responsible for joining the capabilities of separate or incompatible interfaces in this design. A card reader, which functions as an adaptor between a memory card and a laptop, is a real-life example (sourcemaking, n.d.).

♥ **Bridge:** Separates an object's interface from its implementation

When we need to separate an abstraction from its implementation so that both may change independently, we utilize Bridge. This design pattern is classified as a structural pattern because it provides a bridge structure that connects implementation and abstract classes.

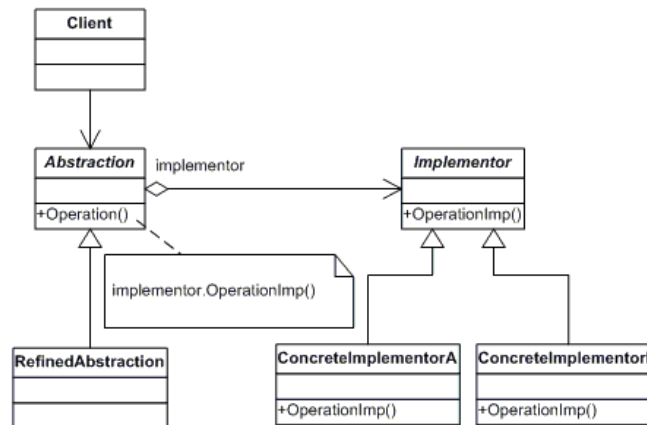


Figure 16: Bridge Pattern

ImageCre: C# bridge Design Pattern

Dofactory

This design uses an interface as a bridge, separating the functionality of concrete classes from the functionality of interface implementer classes. Structures of both sorts of classes can be changed without impacting the other.

♥ **Composite:** A composite object is a tree structure built from of simple and composite items.

The composite pattern is used when we need to manage a group of objects as if they were a single object. The composite pattern arranges objects into a tree structure to depict both the pieces and the whole hierarchy. Because it produces a tree structure for a group of items, this design pattern is characterized as a structural pattern.

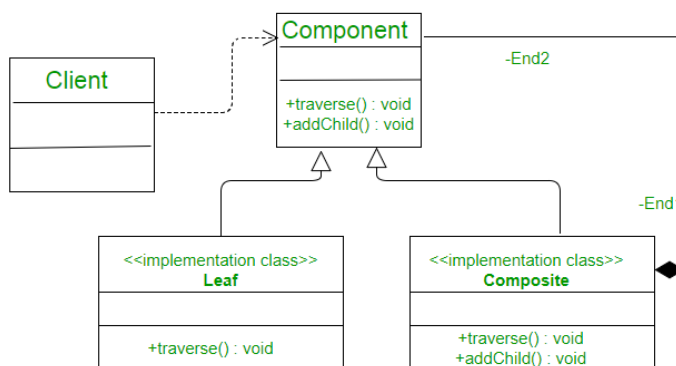


Figure 17: Composite Design pattern

Cre: Composite Design Pattern in C++ - GeeksforGeeks

This pattern generates a class with a collection of its own objects. This class allows you to change the properties of a set of similar objects.

We'll illustrate how to utilize the composite pattern in the following example, which depicts an organization's staff structure.

♥ **Decorator:** Add responsibilities to objects dynamically

A user can use the Decorator pattern to add additional functionality to an existing object without changing its structure. This design pattern is classified as a structural pattern since it functions as a wrapper for an existing class.

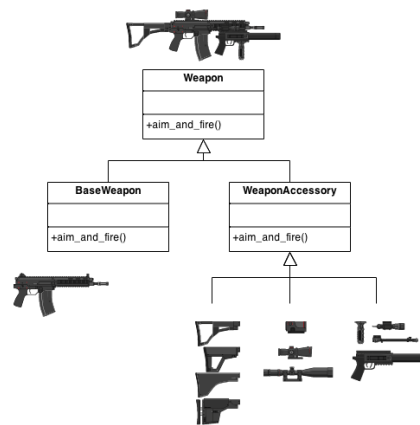


Figure 18: Decorator Design Pattern

ImageCre: SourceMaking Decorator Design Pattern

This approach produces a decorator class that encapsulates the original class and adds functionality while leaving the signature of the class methods intact.

We'll show how to utilize the decorator pattern in the following example, in which we'll colorize a shape without changing its class.

♥ **Facade:** A single class that represents an entire subsystem

The facade design hides the system's intricacies and offers a client interface through which the client may access the system. This design pattern is classified as a structural pattern since it provides an interface to an existing system in order to mask its intricacies.

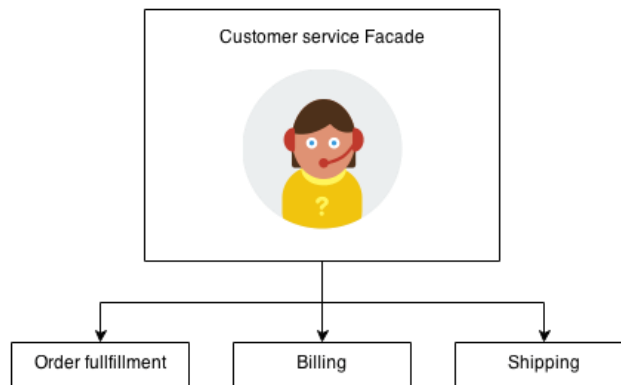


Figure 19: Facade Design pattern

ImageCre: SourceMaking Facade Design

Pattern

This design uses a single class to provide client-side simplified methods while delegating calls to existing system classes' functions.

♥ **Flyweight:** A fine-grained instance used for efficient sharing

Flyweight is referred to as a structural pattern since it is used to create huge object constructions from a variety of components. Flyweight is defined as follows in the original Gang of Four book on Design Patterns: Facilitates the reuse of multiple fine-grained objects, allowing for more efficient usage of huge numbers of objects (sourcemaking, n.d.).

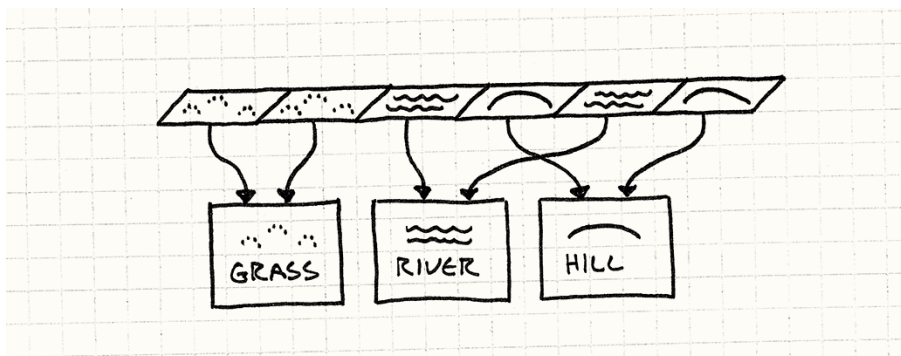


Figure 20: Flyweight Design Pattern

ImageCre: Game Programming Pat Flyweight · Design Patterns Revisited · Game Programming Patterns

You'll need to examine both intrinsic and extrinsic data while thinking about this pattern. Intrinsic data is the information that distinguishes this object instance from others. Extrinsic data, on the other hand, is information that may be handed in by arguments.

♥ **Proxy:** An object representing another object

To manage access to another item, create a proxy or placeholder for it.

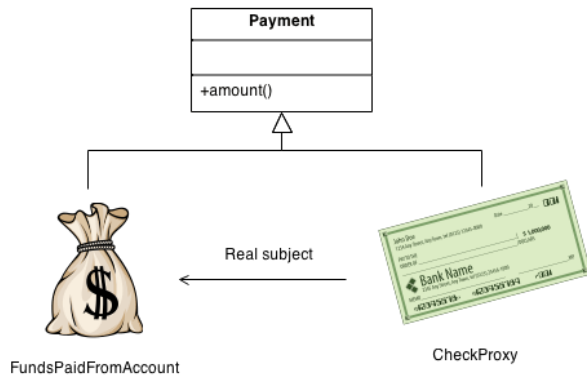


Figure 21: Proxy Design pattern

Cre: SourceMaking Proxy Design Pattern

To provide dispersed, regulated, or intelligent access, add an extra layer of indirection.

To shield the real component from needless complexity, use a wrapper and delegation (sourcemaking, n.d.).

Description of a structural scenario (Huy)

In our plane shooting game, players now can use points to buy upgrades in the store. There are four types of upgrades players can purchase armors, guns, engines, and support planes. Each plane will start with light armor, a light gun, a standard engine, and no support plane. When players have enough points, they can buy upgrades.

Each plane can only have one armor, one gun, one engine, and one support plane.

Armor will increase plane health:

- Light armor: + 1 health (free)
- Standard armor: + 2 health (50000 points)
- Heavy armor: + 3 health (200000 points)
- Legendary armor: + 5 health (1000000 points)

Guns will increase plane damage

- Light gun: + 1 damage (free)
- Standard gun: + 2 damage (50000 points)
- Heavy gun: + 3 damage (200000 points)
- Legendary gun: + 5 damage (1000000 points)

Engine will increase plane speed

- Standard engine: + 10 speed (free)
- Double engine: + 20 speed (100000 points)
- Space engine: + 30 speed (500000 points)

Support plane: Depending on the type of aircraft supporting it, it will have different functions

- Trinity: increase 3 health, 3 damage and 30 speed for player main plane when active. (5000000 points)
- Blood thirster: heal 1 health after taking down an enemy heavy aircraft when active. (5000000 points)

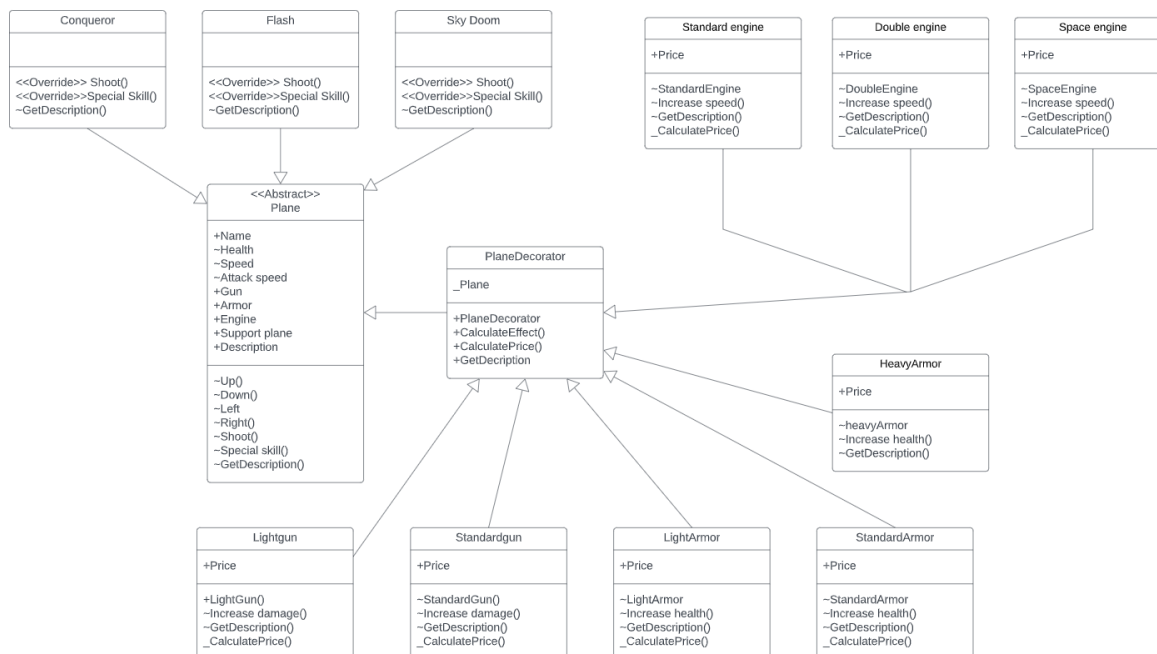


Figure 22: Class diagram of structural Scenario

Explanation

The base Plane Abstract class defines properties, methods for the subclass as well as some methods (GetDecription,..) that these concrete decorators can alter. These concrete classes (Conqueror, Flash, Sky Doom) provide these default implementations of the properties and methods.

The PlaneDecorator class extends the Plane class as the other components. The primary intent of this class is to clarify the wrapping for all concrete decorators (Lightgun, HeavyGun, Light Amor). The default implementation of the wrapping code includes the `_plane` (DataType: Plane) property for storing a wrapped plane and the means to declare it. The `_plane` variable should be accessible to the sub decorator classes, so it's necessary to make this variable with the "protected" access modifier. This class also delegates work to concrete decorator classes. These concrete decorator classes (Lightgun, HeavyGun, Light Amor...) extend the base decorator functionality and modifying the component behavior accordingly of the plane Decorator class.

4.3 Behavioral pattern (Khai)

These design patterns are all about Class's object communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Behavioral Pattern Include:

♥ **Chain of responsibility:** A way of passing a request between a chain of objects

Sender transmits a request to a chain of objects in chain of responsibility. Any item in the chain can handle the request.

"Avoid tying the sender of a request to its recipient by giving different objects a chance to handle the request," according to the Chain of Responsibility Pattern. In the process of dispensing money, an ATM, for example, employs the Chain of Responsibility design pattern (sourcemaking, n.d.).

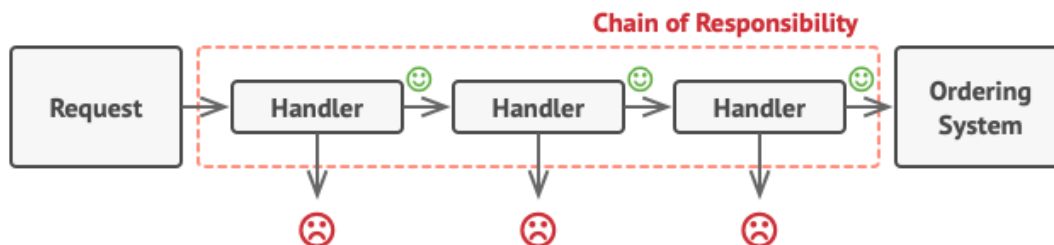


Figure 23: Chain of Responsibility

ImageCre: Chain of responsibility design pattern RefactoringGuru

To put it another way, each receiver generally contains a reference to another receiver. If one object is unable to process the request, it is sent to the next recipient, and so on.

Advantage of Chain of Responsibility Pattern

- It decreases coupling.
- While assigning duties to objects, it offers flexibility.
- It lets a group of classes function as if they were one; events generated by one class may be passed to other handler classes via composition.

Usage of Chain of Responsibility Pattern:

- When a request can be handled by more than one object and the handler is uncertain.
- When a group of objects capable of handling a request must be dynamically provided.

♥ **Command:** Encapsulate a command request as an object

According to the Command Pattern, "Encapsulate a request in an object and deliver it to the invoker object as a command. The invoker object searches for a relevant object that can handle this command and passes the command to that object, which then performs the command".

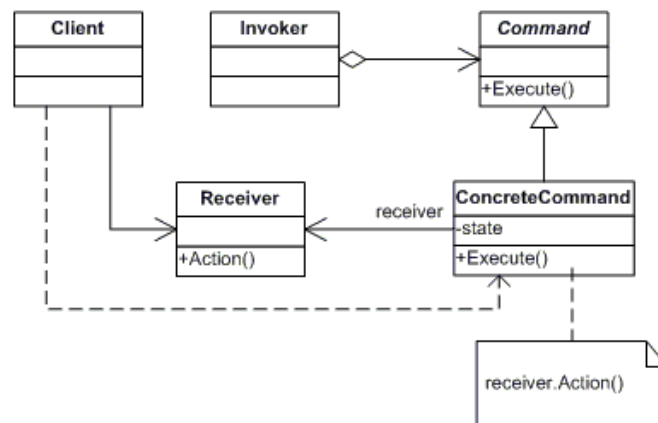


Figure 24: Command Design pattern

ImageCre: Command Design Pattern Stackoverflow

Advantage of command pattern:

- It distinguishes between the object that executes the action and the object that invokes it.
- Because current classes aren't modified, it's simple to add new commands.

Usage of command pattern:

- When you need to parameterize, objects based on an action.
- When many requests must be created and executed at separate times.
- When rollback, logging, or transaction functionality is required.

♥ **Interpreter:** A way to include language elements in a program

"To define a representation of grammar of a particular language, as well as an interpreter that uses this representation to interpret sentences in the language," according to an Interpreter Pattern.

In general, the Interpreter pattern can only be used in a restricted number of situations. The Interpreter pattern can only be discussed in terms of formal grammar, although there are superior solutions in this field, which is why it is not often utilized.

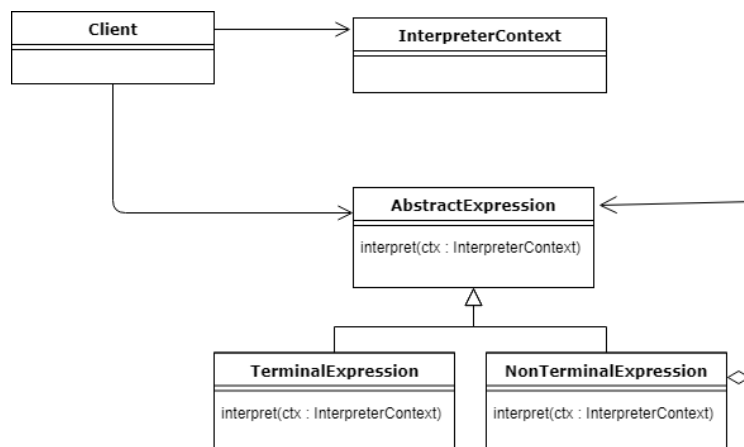


Figure 25: Interpreter Pattern

ImageCre: Interpreter Pattern Baeldung

Advantage of Interpreter Pattern

- It is less difficult to modify and expand the grammar.
- It's simple to put the grammar into practice (sourcemaking, n.d.).

♥ **Iterator:** Sequentially access the elements of a collection

Provide a method for progressively accessing the constituents of an aggregate object without revealing its underlying representation.

The traversal of a collection should be elevated to "full object status." traverse with polymorphism

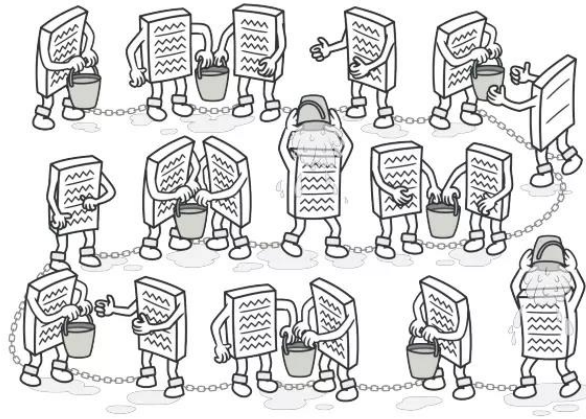


Figure 26: Iterator Design Pattern

ImageCre: Iterator Pattern RefactoringGuru

Advantage of Iterator Pattern

- It allows you to traverse a collection in different ways.
- It streamlines the collection's user interface.

Usage of Iterator Pattern:

- When you need to get a hold of a group of things without revealing their internal representation.
- When many traversals of items are required, the collection must be able to accommodate them.

♥ **Mediator:** Defines simplified communication between classes

"To define an entity that captures how a group of objects communicate," states the Mediator Pattern.

I'll demonstrate the Mediator pattern by examining an issue. When we first start developing, we have a few classes that interact with one another to produce outcomes. Consider how, as functionality develops, the reasoning becomes more complicated. So, what happens next? We added additional classes, and they still communicate, but maintaining this code is becoming increasingly tough. As a result, the Mediator pattern solves this issue (sourcemaking, n.d.).



Figure 27: Mediator Pattern

ImageCre: Mediator Pattern DotNetTutorials

The mediator pattern is intended to simplify communication across various objects or classes. This technique creates a mediator class that generally handles all communication between distinct classes and makes the code easier to maintain.

Memento: Capture and restore an object's internal state

"To restore the state of an item to its prior state," according to a Memento Pattern. However, it must do so without breaking Encapsulation. In the event of a mistake or failure, such a scenario is beneficial.

Token is another name for the Memento pattern.

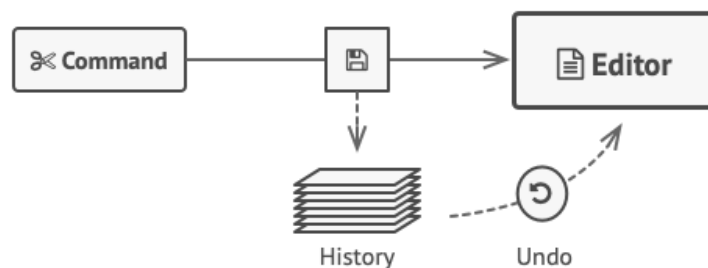


Figure 28: Memento Design Pattern

ImageCre: Memento Desine Pattern RefactoringGUru

One of the most often used operations in an editor is undo, sometimes known as backspace or ctrl+z. The undo action is implemented using the Memento design pattern. This is accomplished by storing the object's current state as it changes.

♥ **Observer:** A way of notifying change to a number of classes

"Just construct a one-to-one dependence so that when one object changes state, all its dependents are alerted and changed immediately," according to the Observer Pattern.

Dependents or Publish-Subscribe are other names for the observer design.

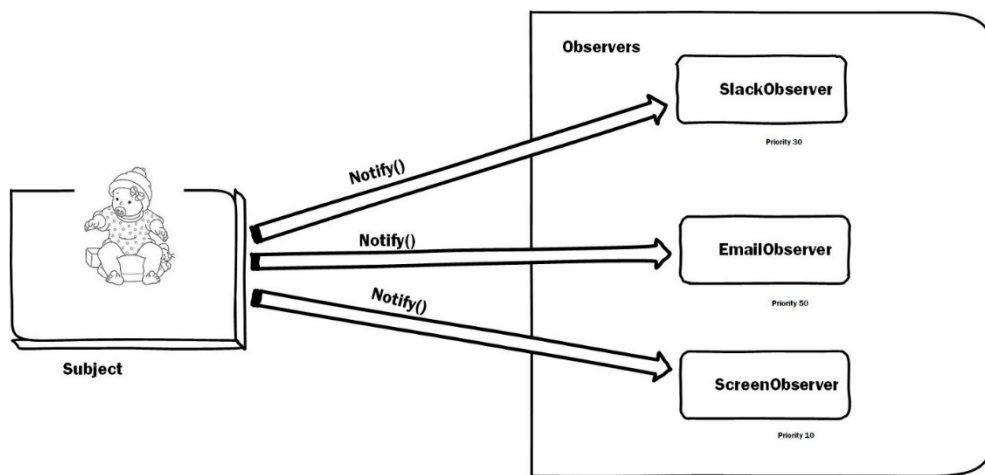


Figure 29:Observer Design Pattern

ImageCre: Observer DesignPattern SourceMaking

Benefits:

- It explains the interaction between the observer and the objects.
- It allows for broadcast-style communication to be supported.

Usage:

- When a state changes in one item must be mirrored in another without the objects being tightly connected.
- When we design a framework that will be updated in the future with additional observers with few changes (sourcemaking, n.d.).

♥ **State:** Alter an object's behavior when its state changes

"The class behavior changes dependent on its state," according to a State Pattern. We generate objects that represent distinct states and a context object whose behavior changes as its state object changes in State Pattern.

Objects for States is another name for the State Pattern.

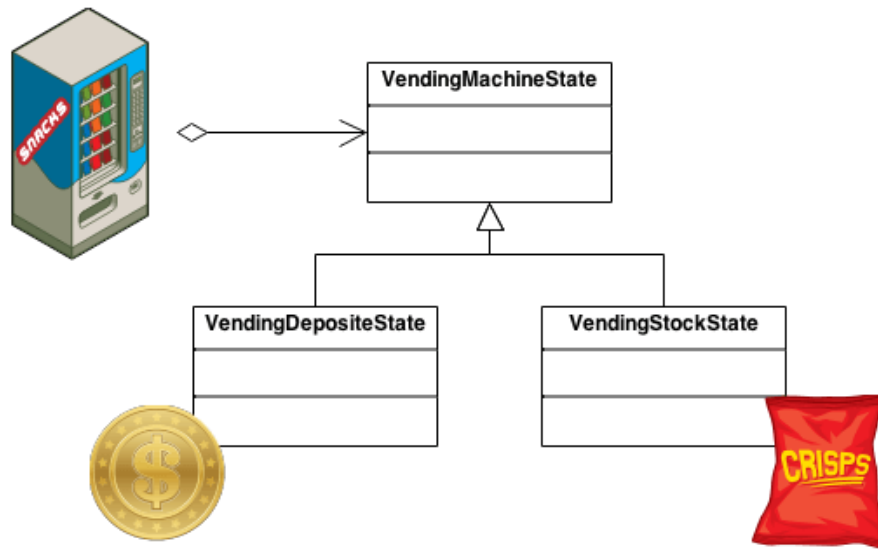


Figure 30: State Design Pattern

ImageCre: State Design Pattern

Advantages:

- It preserves state-specific behavior.
- Any state transitions are made explicit.

Usage:

- When an object's behavior is dependent on its state and it has to be able to adjust its behavior at runtime to match the new state.
- It's utilized when there are a lot of multipart conditional statements in the actions that are dependent on the state of an object.

♥ **Strategy:** Encapsulates an algorithm inside a class

"Defines a family of functions, encapsulates each one, and makes them interchangeable," according to a Strategy Pattern. Policy is another name for the Strategy Pattern.

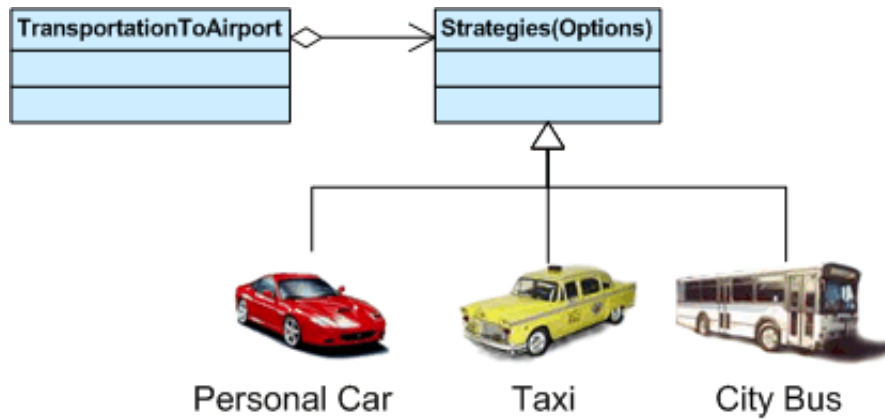


Figure 31: Strategy Design pattern

ImageCre: Strategy Design Pattern Sourcemaking

Benefits:

- It may be used instead of subclassing.
- Each behavior is defined within its own class, removing the need for conditional expressions.
- It makes it easy to add new functionality without having to rewrite the program.

Usage:

- When many classes differ simply in their behaviors, this is used.
- When multiple variants of an algorithm are required, it is utilized.

♥ **Template method:** Defer the exact steps of an algorithm to a subclass

In an operation, define the skeleton of an algorithm while deferring some stages to client subclasses. Subclasses can rewrite some phases of an algorithm using the Template Method without affecting the algorithm's structure.

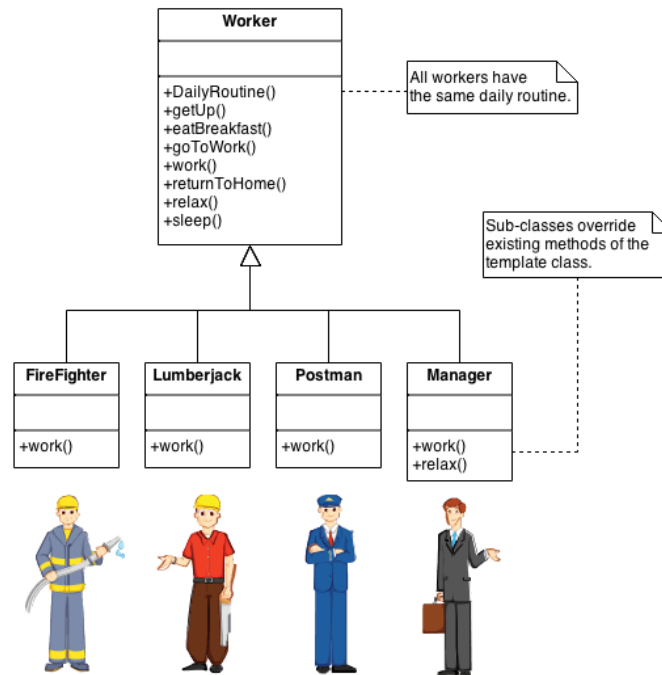


Figure 32: Template Method design pattern

ImageCre: Template Method Source Making

Algorithm 'placeholders' are declared in the base class, and the placeholders are implemented in the descendant classes.

Strategy is similar to Template Method in terms of granularity.

Inheritance is used in the Template Method to alter parts of an algorithm. Delegation is used in strategy to change the whole algorithm.

Individual objects' logic is altered by strategy. The Template Method affects the entire logic of a class. Template Method is a subset of Factory Method (Anon., 2022).

♥ **Visitor:** Defines a new operation to a class without change.

Represent an operation that will be executed on an object structure's components. Without modifying the classes of the components on which it acts, Visitor allows you to specify a new operation.

The traditional method for restoring type information that has been lost,

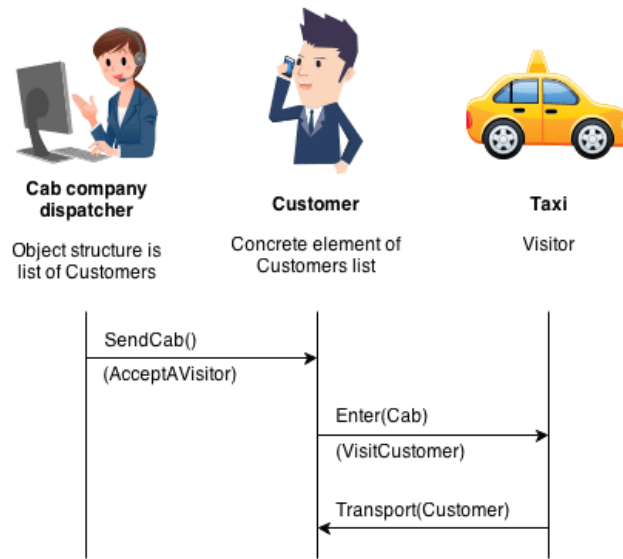


Figure 33: Visitor Design Pattern

ImageCre: Visitor Pattern SourceMaking

Based on the two types of items, take the appropriate action. Interpreter's abstract syntax tree is a Composite (therefore Iterator and Visitor are also applicable).

A Composite can be traversed by an Iterator. Over a Composite, a visitor can perform an operation.

Because the visitor can begin whatever is suitable for the type of item it meets, the Visitor pattern is similar to a more powerful Command pattern (Javatpoint, n.d.).

Description of a behavioral scenario (Hoang)

In Greenwich, while the current semester isn't finished yet, students will immediately receive notification of paying tuition fees for the next semester. Tuition Fees could be paid online via various e-money transfer platforms: ViettelPay, DNG bank-Debit card, and TPbank. Students can check the transaction cost of the payment via the payment tuition fee online.

With paying via Viettel Pay, go to Viettel Pay App, verify the Viettel phone number and Select FPT tuition payment. Enter student code and perform pay the tuition fee. The transaction fee of Viettel pay is free. With DNGbank, the necessary info is the information on the bank card (Name, card number, Expired date). The DNGBank's transaction fee is 3300 vnd. The other is the DNGDebit payment method, its necessary info also the information on the Debit Card (Name, Card number, CVV and expired date). The transaction of this payment method is 1,4% of the tuition fee + 3300 vnd.

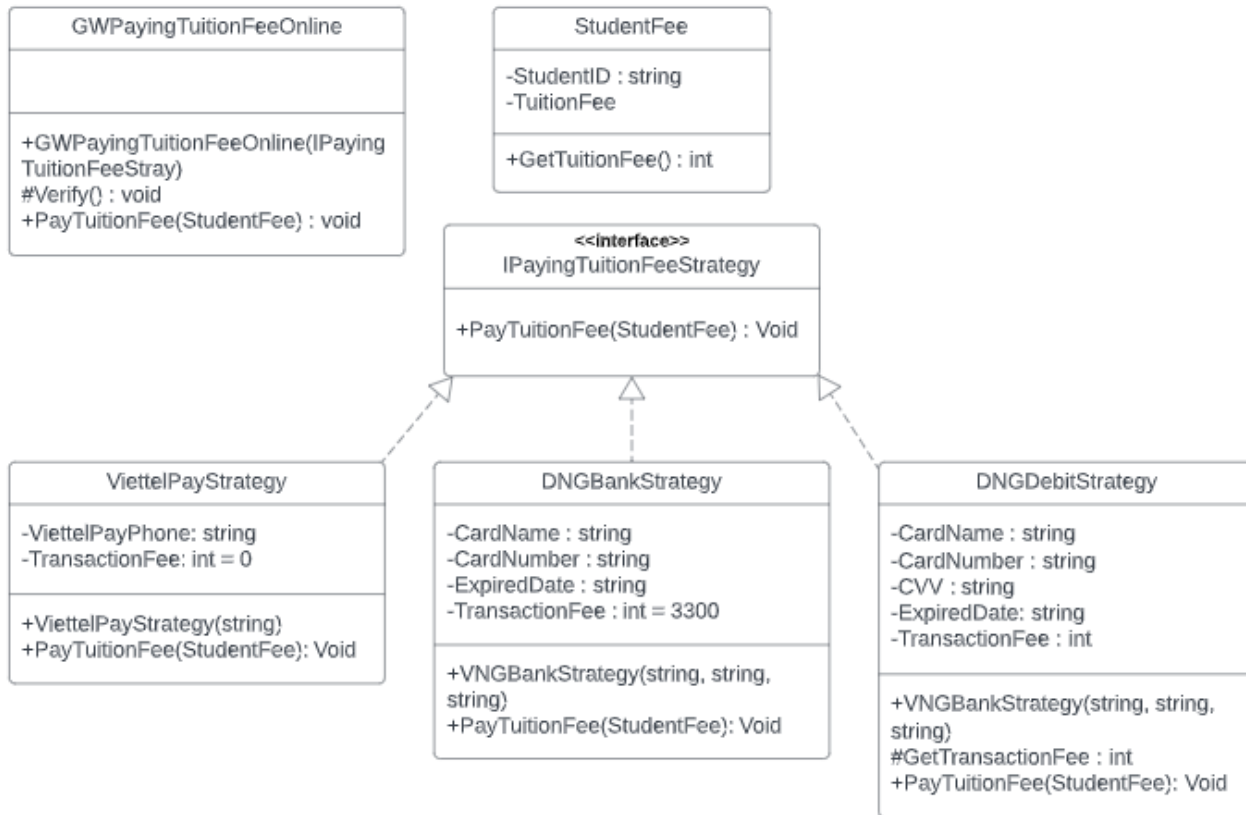


Figure 34: Class Diagram of Behavioral Scenario

Explanation: (Hieu)

The `GWPayingTuitionFeeOnline` accepts `IPayingTuitionFeeStrategy` object through the constructor. The `GWPayingTuitionFeeonline` does not know the concrete classes of this strategy. But It could work with all strategy classes through the `IPayingTuitionFeeStrategy` interface.

`IPayingTuitionFeeStrategy` Interface for strategy pattern in this case to pay tuition fee, the student object passed as an argument. These concrete classes implement the `IPayingTuitionFeeStrategy` Interface to implement algorithms for paying tuition fees using DNGdebit, DNGbank or via Viettelpay.

Concrete Strategies (`ViettelPayStrategy`, `DNGBankStrategy`, `DNGDebitStrategy`) implement the algorithm while implementing the `IPayingTuitionFeeStrategy` interface. This interface helps these concrete strategy classes interchangeably.

This solution delegates the selection of payment method to the Strategy object instead of implementing multiple versions of the algorithm on its own. The selection of the algorithm defined by these concrete strategy classes will be called via the IPayingTuitionFeeStrategy interface.

E. Design Pattern vs OOP (Huy)

The concepts of OOP provide design patterns, which assist to build OOP more effectively and adapt to scenarios. It will take much longer to run the program or address difficulties if you don't use a design pattern. Design patterns can make your code more reusable and adaptable, allowing you to include any function that fulfills the criterion (Techopedia, 2011).

For example, when a car is created, there are only 4 doors, 1 engine, 4 wheels. But what will happen when owners take their car to a modification garage to add more features (body kit, spoiler, sticker). The simplest way to solve that problem is extend the base class and create a bunch of subclasses or create a massive constructor with lots of unused parameters. In this case, the builder design pattern helps the user to execute these steps of the builder. You can create various builder classes with the same set of builder steps set but in a different manner.

Another example that can be applied is vehicle painting. The simplest way is creating a Vehicle class and pairs of subclasses such as Car and Motorbike. To create colored vehicles in red and green, users need to create a bunch of subclasses such as GreenCar, RedCar, GreenMotorbike, RedMotorbike. Problems will occur if users decide to add a few more colors. By switching from inheritance to object composition, the Bridge pattern tries to tackle this problem. The color-related code can be extracted into its own class with two subclasses: Red and Green. The Vehicle class then receives a reference field that points to one of the color objects. Any color-related tasks can now be delegated to the connected color object by the shape. This reference will serve as a bridge.

Design pattern is not a design that can be transferred directly into code, it is only a description of way to solve many problems in many situations. Even without the specification of class and object, design pattern illustrates the relationship of classes on objects and how they interact. There's always occurs problem when programming and various ways to solve it, but it maybe not be optimal choice. Design patterns are created to handle an issue in the most efficient way possible, with solutions available in OOP programming and most modern programming languages.

F. Conclusion (Khai)

The report covered ideas and features of Object-Oriented Programming (OOP), as well as how to apply OOP in practice with considerable scripting. Explain the system's activities and behavior for the scenario described in the research using two use case diagrams (use case diagram, class diagram). We may learn the foundations of OOP and its concept through this report. We've also detailed the general definitions and features of three different types of design patterns: structural, creational, and behavioral patterns. Each one includes a class diagram, activity, and explanation to show how design patterns and OOP are related. This can assist us improve our programming language approach and make our project run more smoothly. This report can also serve as a foundation for future reports.

References

- Alexander S. & Sarah Lewis, 2017. *object-oriented programming (OOP)*. [Online]
Available at: <https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP>
- Anon., 2022. *Refactoring.Guru*. [Online]
Available at: <https://refactoring.guru/design-patterns>
- Anon., n.d. [Online]
Available at: https://topdev.vn/blog/solid-la-gi/?fbclid=IwAR2UzzITTxPd3eXG7b4kpG40kJYj3pMs_kSnYo30wjPIMlo61F8A4pveS80#:~:text=L%E1%BA%ADp%20tr%C3%ACnh%20h%C6%B0%E1%BB%9Bng%20%C4%91%E1%BB%91i%20t%C6%B0%E1%BB%A3ng,t%C6%B0%E1%BB%A3ng%20trong%20th%E1%BA%BF%20gi%E1%BB%9Bi%20
- Anon., n.d. *FULL STACK DEVELOPER*. [Online]
Available at: <https://devfull.me/>
- HOÀNG, P. H., 2015. [Online]
Available at: https://toidicodedao.com/2015/03/24/solid-la-gi-ap-dung-cac-nguyen-ly-solid-de-tro-thanh-lap-trinh-vien-code-cung/?fbclid=IwAR2hOXf9Dg_V_H8TfeyD3A5md2fri6nQZVMqTA3QE-M6zJBKsdHfIZnylBo

Javatpoint, n.d. *Design Pattern*. [Online]

Available at: <https://www.javatpoint.com/design-patterns-in-java>

Mui, Nguyen Van, 2018. [Online]

Available at: <https://viblo.asia/p/solid-trong-android-ORNZqPmnK0n>

Nitendratech, 2018. *Object-Oriented Programming concepts*. [Online]

Available at: <https://www.nitendratech.com/programming/object-oriented-programming/>

Oloruntoba, S., 2020. [Online]

Available at: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#liskov-substitution-principle

Oracle, 2012. *Object Class Definitions*. [Online]

Available at: <https://docs.oracle.com/javase/jndi/tutorial/ldap/schema/object.html>

Shvets, A., 2021. *Dive Into DESIGN PATTERNS*. s.l.:s.n.

sourcemaking, n.d. *behavioral patterns*. [Online]

Available at: https://sourcemaking.com/design_patterns/behavioral_patterns

sourcemaking, n.d. *structural patterns*. [Online]

Available at: https://sourcemaking.com/design_patterns/structural_patterns

Techopedia, 2011. *Design Pattern*. [Online]

Available at: <https://www.techopedia.com/definition/18822/design-pattern>

TopDev, 2019. *SOLID là gì? Áp dụng SOLID để trở thành lập trình viên giỏi*. [Online]

Available at: <https://topdev.vn/blog/solid-la-gi/>

Tutorial, O., n.d. *Object Class Definitions*. [Online]

Available at: <https://docs.oracle.com/javase/jndi/tutorial/ldap/schema/object.html>

W3schools, n.d. *C++ OOP*. [Online]

Available at: https://www.w3schools.com/cpp/cpp_oop.asp