ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date	3/9/2020	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Phan Ba Hung	**Student ID**	GCH17622
Class	GCH0711	**Assessor name**	Doan Trung Tung
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	Hung.P

Grading grid

P1	P2	M1	M2	D1	D2

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

Table of content

Introduction

Object-oriented programming (OOP) is one of most programming method nowadays. It helps to increase productivity, simplify maintenance when maintaining, easier to learn, easier to learn than previous methods, reduce the amount of code. This document will introduce about basic concepts of OOP and design pattern.

I. OOP

"Object-oriented programming means programming with abstract data types (classes) using inheritance and dynamic binding" (Hanspeter, 1995).

Characteristics of an Object-Oriented Programming language

- Encapsulation: As a way to hide the inner handling properties of an object, other objects cannot be directly manipulated to change state that can only be accessed through the public methods of that object.
- Abstraction: An abstraction method defines the actions and properties of certain types of objects.
- Polymorphism: An object of different classes that can understand the same message in different ways.
- Inheritance: A technique that allows inheriting features that another object already has, helping to avoid redundant code duplication but only when processing the same job.

Scenario

Library

The library has many bookshelves that store different types of books. Borrowers and librarians may search for books by title, category or publisher name of the book. When a book is found, the system will automatically notify whether the book is still available on the shelf or not. Those who want to borrow books can log in and use their account to keep the number of books they want to borrow and come to the library to receive within 3 days. Or the borrower has a direct registration with the librarian at the library. The system will automatically notify the borrower to return the book if it is overdue through the account. After the book is returned to the library, the librarian will immediately check the status of the book and notify the borrower if the book has to be compensated for the broken book and enter the status (paid, broken, ...) of them on the system. Librarians must log in to gain access to the books' data on the system. Admin can decentralize and manage accounts on the system.

Use Case Diagram

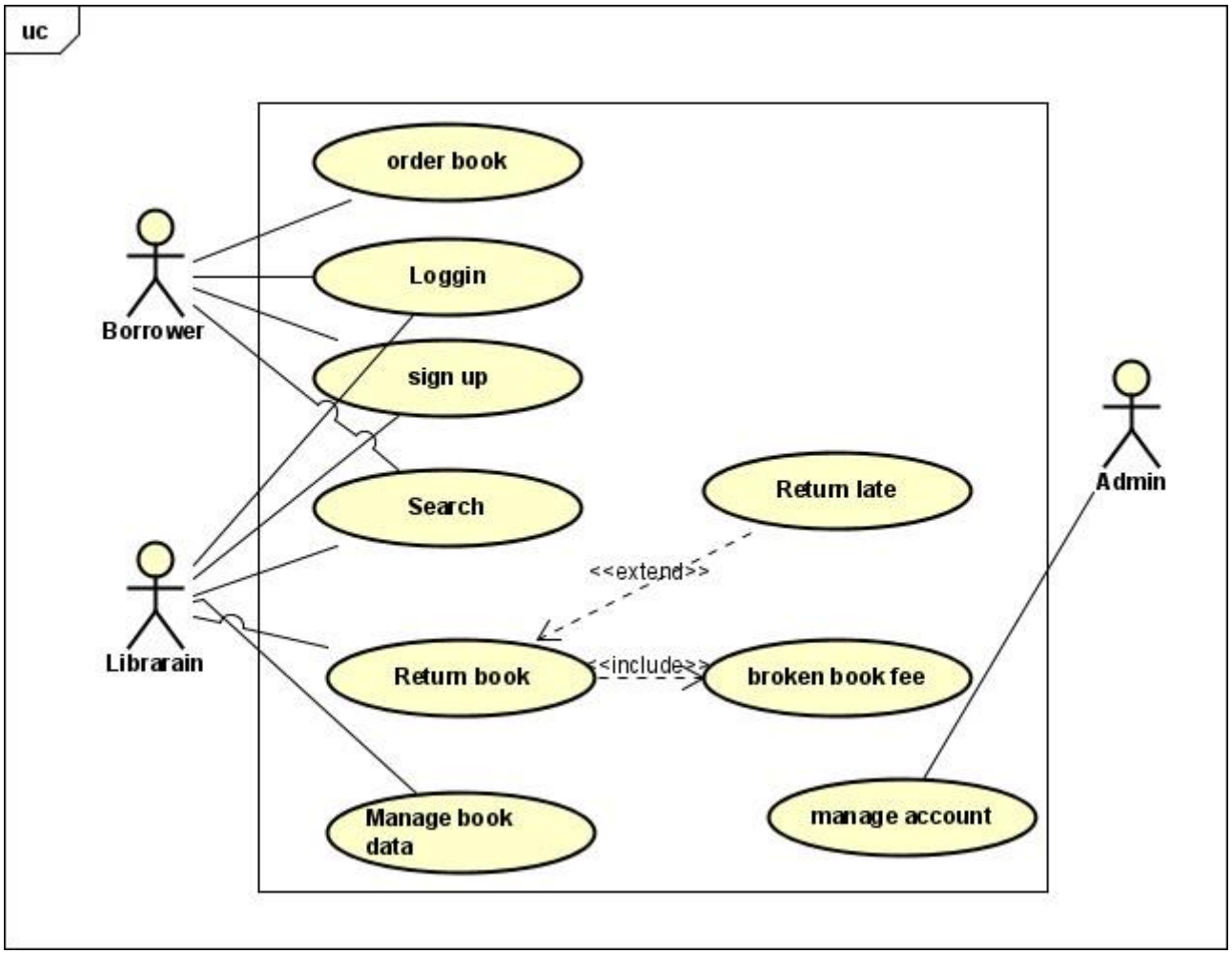


Figure 1: Use case diagram

There are 3 actors: borrower, librarian and admin. Borrower and librarian have some same functions: log in, sign up, search. Besides, there are some other features like: borrower can order book and librarian can return, manage book. “Return late” extend from “Return book” for borrowers who return books late. When executing both of “return late” and “return book”, “broken book fee” will be called.

Admin only have permission to manage borrower’s account and librarian’s account.0

Class Diagram

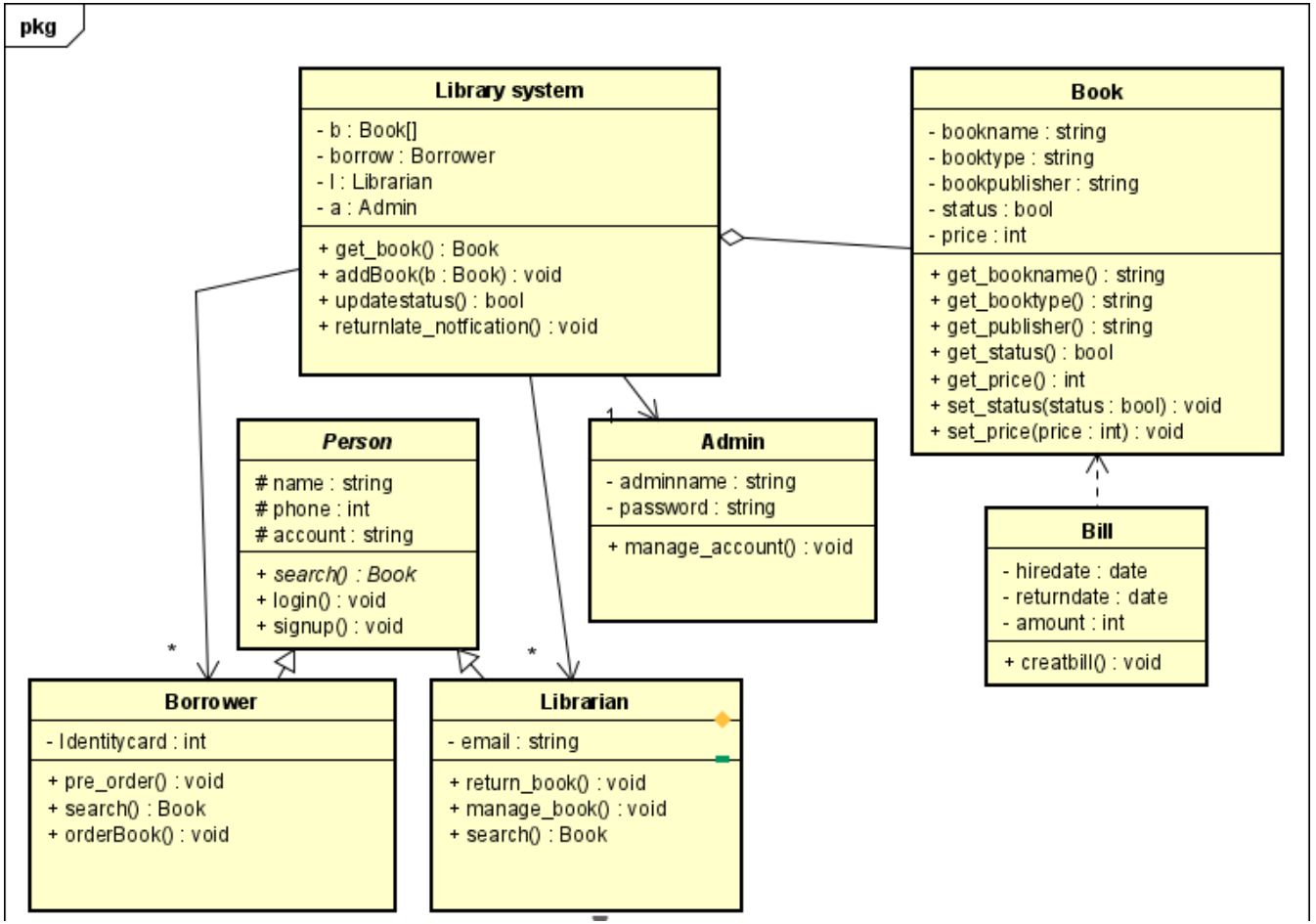


Figure 2: Class diagram

Both of class “Borrower” and class “Librarian” inherited from “Person” class. They have attributes of “Person class” and search method.

Any person inherited from “Person” class will have search and log in method.

Besides, search method will be override.

Class “Book” has attributes need get, set method.

Class “Bill” depend on class “Book”. There are many book, each of them has different price so different books have different bill.

Class “Book” is a part of “Library system”. Library has many books.

Library system has a admin and many borrowers, librarian.

Activity diagram

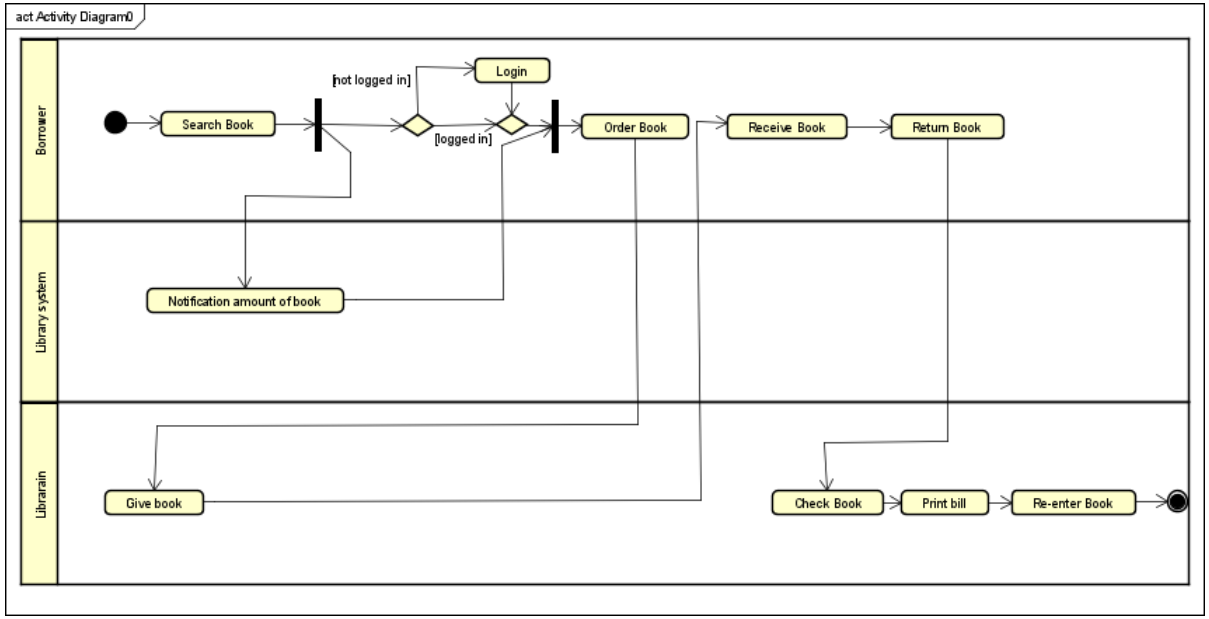


Figure 3: Activity diagram

When borrower want to hire book, they must search book firstly. Then, there are 2 process is performed in parallel: display amount of book and ask borrower log in if they have not. After that they order book, receive book from librarian and return. The librarian will check status of book when they have returned. Then librarian will print bill for borrower and re-enter book into system.

II. Design Pattern

1. Creation pattern

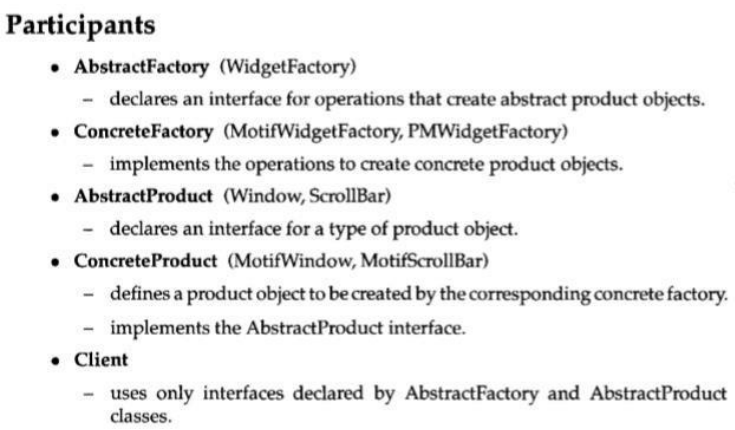
According to (Erich, John, Richard & Ralph, 1994), "*Creational design patterns abstract the instantiation process. They help make a system independent of how its object s are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object*." In other word, creative design patterns are design patterns that deal with processes of object formation, trying to create objects in a way that suits the situation. They reduce complexities and uncertainty through managed object formation.

Some types of creation pattern:

- Singleton
- Factory method
- Builder
- Prototype
- Abstract Factory:

According to (Erich, John, Richard & Ralph, 1994), "*Provide an interface for creating families of related or dependent objects without specifying their concrete classes*." That allows us to implement a variety of factories that produce products meant for different contexts.

An Abstract Factory includes: Abstract Factory, Concrete Factory, Abstract Product, Concrete Product, and Client.



Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Figure 4: Participants of Abstract Factory (Erich, John, Richard & Ralph, 1994)

When we use Abstract Factory:

When you want to build an independent system with the products it creates. One of the multiple families of products should be configured to a system. A family of related product objects is intended to be used together, and that constraint must be implemented by you. Finally, when you want to provide a product, you just want to present its interface and not want to disclose how it is done.

Scenario

U is a software development company and you are an employee of that company. The Board of Directors is tasked with the knowledge of sample design to build an application building system that can optimize the interface for applications for iPhone, Samsung and BlackBerry mobile. In which the user can make declarations of interface properties without knowing how it works.

Class diagram

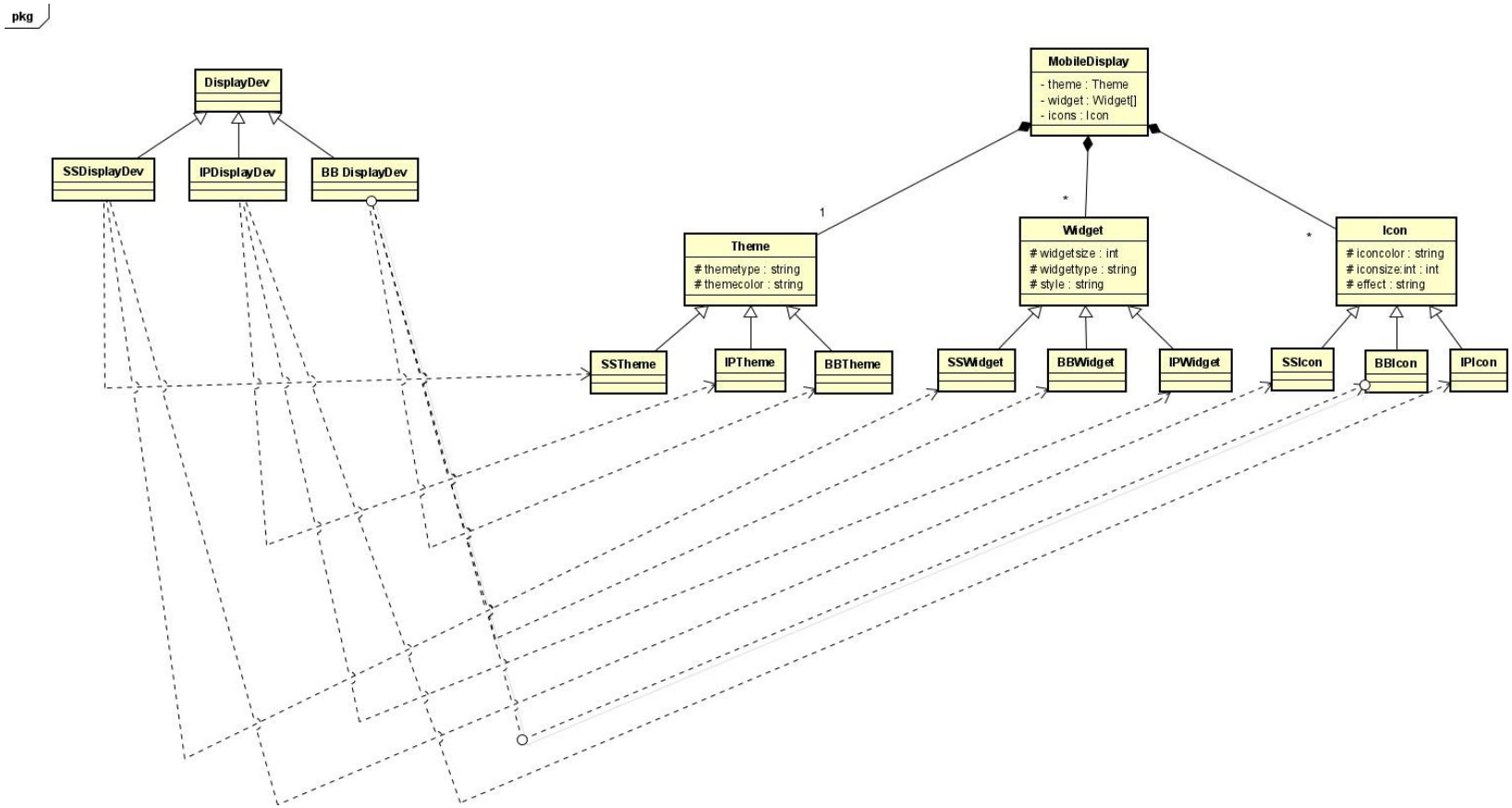


Figure 5: Abstract factory

Diagram Description:

In the Class Diagram (Figure 5), we can see that the Application class represents AbstractFactory, declaring an interface to create abstract products (Theme, Icon, Widget, MobileDisplay). The ConcreteFactory classes are represented by SSApp, BBApp, IPApps classes, which are inherited from the abstract layer (Application layer) and they perform operations to create specific applications for the equipment of each brand and they depends on the concret products.

Next, the MobileDisplay, Theme, Widget, Icon classes, it is understood as an abstractproduct in this system in which the Theme class, Widget class, and Icon class all have a composition relationship with MobileDisplay class which means that Theme, Widget, Icon are all one part of MobileDisplay. they provide an interface for a type of product object which in this case is MobileDisplay.

Next, the concreteproduct is shown in this diagram by SSTheme, SSIcon, SSWidget, IPTheme, IPIcon, IPWidget, BBTheme, BBIcon, BBWidget. They define the product object created by the concretefactory (BBApp, IPApp, BBApp) and they are inherited from abstractproduct (Theme, Icon, Widget). And finally, on behalf of the client is Developer can declare with abstactfactory (App) and abstractproduct (MobileDisplay) to create applications that have optimized interface for each devices.

2. Structural pattern

According to (Erich, John, Richard & Ralph, 1994), "*Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations*." In other word, structure design is a design pattern that simplifies the design by defining a simple way to recognize relationships between entities.

Some types of structural pattern:

- Bridge
- Composite
- Decorator
- Facade
- Flyweight

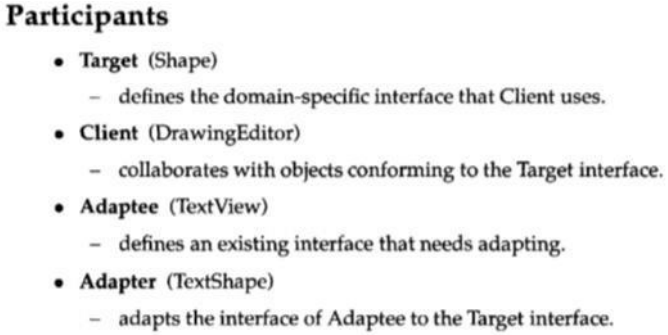
- Proxy
- Adapter:

According to (Erich, John, Richard & Ralph, 1994), "*Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces*". The adapter describes the creation of an abstraction that mediates the conversion of old components into the new system. On the client-side, calling methods on the object, the adapter redirects to the inheritance component.

When we use Adapter:

When you want to use an existing class and it doesn't suit the interface you need or you want to create a reusable class that cooperates with unrelated or unforeseen classes.

An adapter includes: Target, Client, Adapter and Adaptee



Participants

- **Target** (Shape)
  - defines the domain-specific interface that Client uses.
- **Client** (DrawingEditor)
  - collaborates with objects conforming to the Target interface.
- **Adaptee** (TextView)
  - defines an existing interface that needs adapting.
- **Adapter** (TextShape)
  - adapts the interface of Adaptee to the Target interface.

Figure 6: Participants of Adapter (Erich, John, Richard & Ralph, 1994)

Scenario

Company A is a game company specializing in producing games. The directors give you the task of building an Adapter to use the functions of the wizard character from previous games on the new game with a different interface

Class diagram

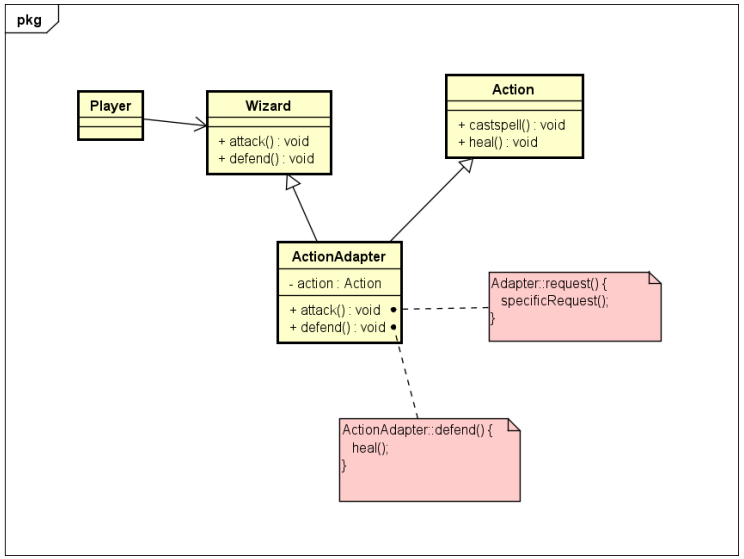


Figure 7: Adapter Class Diagram

Diagram Description:

In the diagram in Figure 7, the Wizard is understood as the target in the adapter. It defines the specific interface that the Client uses (The Client in this case is the player). Action represents adaptee, it is the interface that needs adapting in this system, and finally apdater is represented by ActionAdapter, which implements multiple inheritance with adaptee(Action) and target(Wizard) to adjust Adaptee's interface with target interface.

3. Behavior pattern

According to (Erich, John, Richard & Ralph, 1994), "*Behavior pattern is the design pattern that defines and defines common communication patterns between objects*."

Some types of structural pattern: Chain of responsibility, Iterator, State. Strategy. Template method.

Chain of responsibility is chosen to do example for behavior pattern. According to (Erich, John, Richard & Ralph, 1994), chain of responsibility describes a big task which has many small tasks, the small task is given to many handlers and the small task will be propagated along the handlers until a handler handles it.

Scenario

Process of getting parcels in a company.

Class diagram

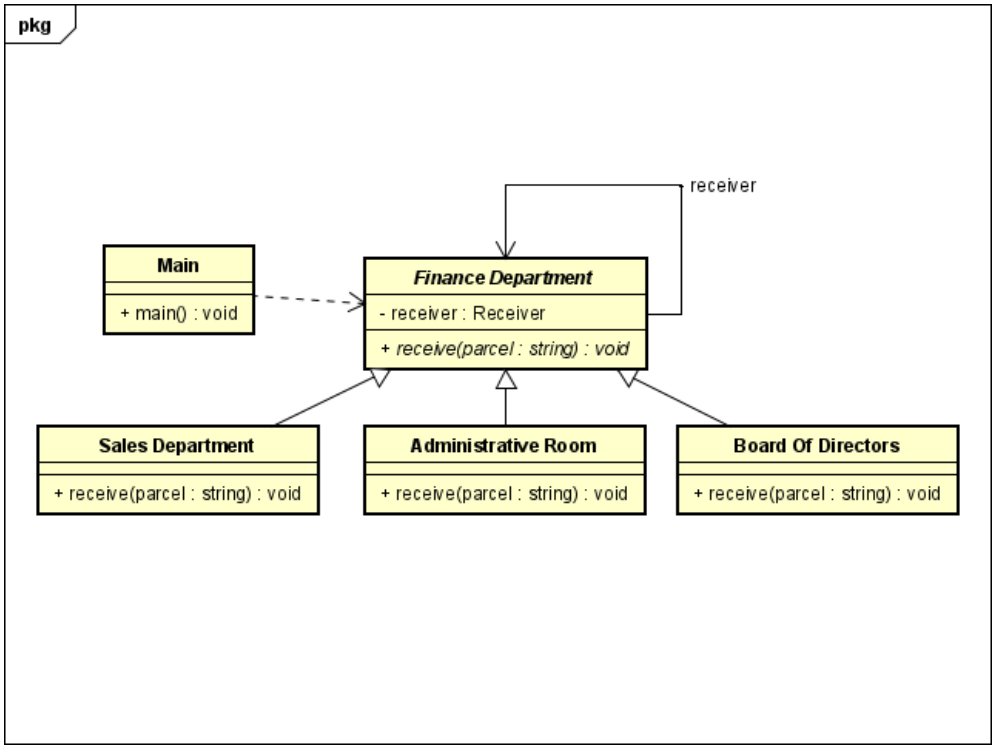


Figure 8: Chain of responsibility

Diagram Description:

The parcel will be delivered to the company's finance department. Finance department will bring them to every room in the company (sales department, administrative room, board of directors). to which department the parcel is delivered, that department will take its parcel.

Summary

In addition to defining OOP and design patterns, a number of scenarios are also built to practice UML diagrams.

References

Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson (1994) Design Patterns: Elements of Reusable Object-Oriented Software. USA , Addison-Wesley Publishing .

Hanspeter Mössenböck (1995) Object-Oriented Programming in Oberon-2. German, Springer Berlin Heidelberg Publishing. Available from: http://norayr.am/papers/oop_in_oberon-2_book.pdf [Accessed 5th Mar 2020].