Name: Trung Kien Nguyen

Student ID: 104053642

SWINBURNE UNIVERSITY OF TECHNOLOGY

COS20007

Object-Oriented Programming

REPORT

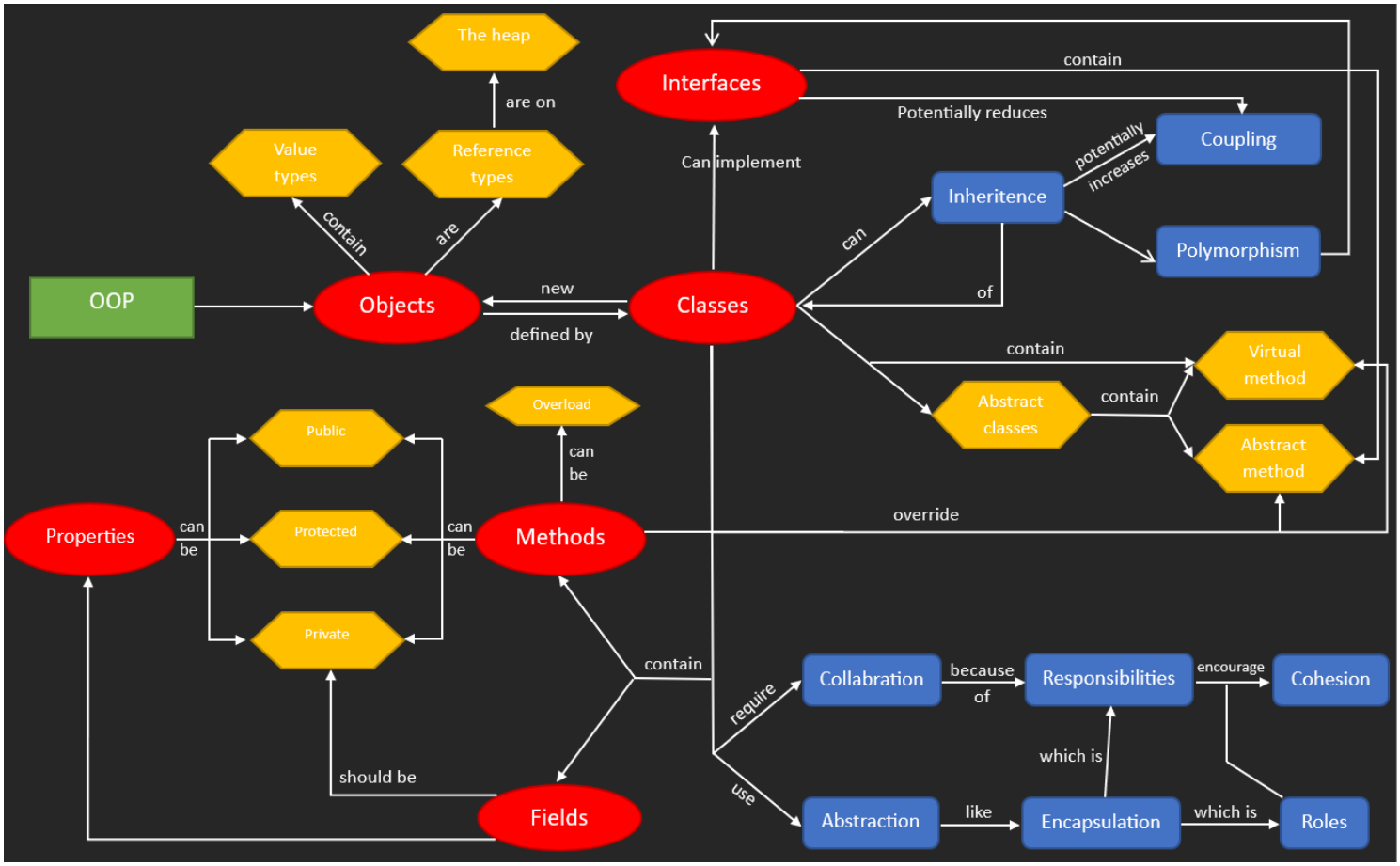
Key Object Oriented Concepts

TABLE OF CONTENTS

DEFINITION AND MAIN CONCEPT

The term of “Object-oriented programming” is based on the concept of "objects", and puts a great deal of emphasis on the objects than the logic and function when creating a program/application (which is usually presented in structured programming). The definition of “object” is defined as a self-contained entity that contains both data (fields and properties) and behavior (methods) that operate on that data.

In OOP, programs are organized around objects and their interactions. The main idea is to encapsulate data and behavior in a single entity for better code abstraction, modularity and reuse. Also, because programmers can change and rearrange elements within a program, OOP is well suited for large, complex, and frequently updated applications. The following diagram shows the main concepts of OOP:



The OOP Concept map

In summery, OOP provides a way to organize and structure your code in a more modular, maintainable, and scalable way. OOP concept, and its key principles as well, is popularly and widely used by high level programming languages such as C#, Java, C++, Python, ... , and is considered a powerful and flexible way to build programs/applications.

KEY PRINCIPLES

There are four key principles of Object-oriented Programming (OOP), including ***Abstraction***, ***Encapsulation***, ***Inheritance***, and ***Polymorphism***.

1. Abstraction: Abstraction in OOP often involves the process of "designing ideas" of related things in a program/application, meaning that its focus is on "what should be done", including what objects should we create, and which aspects of those objects needed to be presented in the code, and also what features are not essential that should be exclude or hide to the clients to reduce complexity. In other words, we use abstraction to "make decisions" and gain information to set up the objects' classifications, roles, responsibilities and collaborations:

- Each object's classifications should be identified: Abstraction principle helps programmer to identify common characteristics and behaviors of a specific object.
- Roles/responsibilities: It is important to determine the roles of classes, as well as their responsibilities based on the roles.
- Collaborations of the objects: Abstraction helps to identify and understand the relationships and interactions between objects in a program/application.

With the abstraction principle, we can decide and manage the level of complexity of the objects in a program/application, as well as the program/application itself from the beginning. This is often used in solving the problem occurring when the program/application is large and complex.

E.g.

- A very simple abstraction in realworld:

Today is the day that car company A releases its latest unique model. Even though we've never seen that specific one in the real life, we can still recognize that it is "a car", because it has the car features:

Four Wheels.

Passenger Compartment, which distinguishes it from open vehicles like motorcycles or scooters.

Steering Wheel (to control the direction of the vehicle)

Seats.

Windows and doors

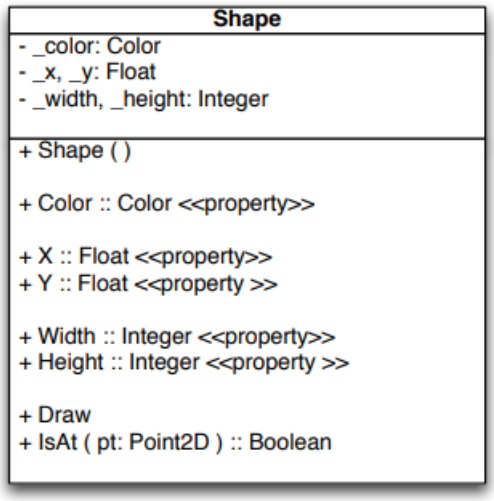
Headlights and taillights

Controls and instrumentation (accelerator pedal, brake pedal, gearshift, speedometer, fuel gauge, ...)

…

They, all made an interface of a typical car, a "model" of a "car" object in my mind, while the hidden technical specifics of the car (engine, …) may not really essential to us – the car drivers. The "abstraction" involves the process of defining that model.

- In the task 2.3P - 2.3P - Drawing Program - A Basic Shape, when I define the “Shape” class to represent the rectangle objects in my program, I need to decide some features of it, including



  - Its color
  - Its position in the screen
  - Its size (width and height)
  - Its responsibility of “Draw” when asked, or return the boolean value to answer the question whether the mouse is on the rectangle (“IsAt”)

  They are the interfaces of a rectangle in my program that the clients/users want to know, while some specific details may be hidden as they are not essential to them, or the external class within the program, such as we used SplashKit.FillRectangle(…) in the Draw() method, or how we checked in the “IsAt” property.

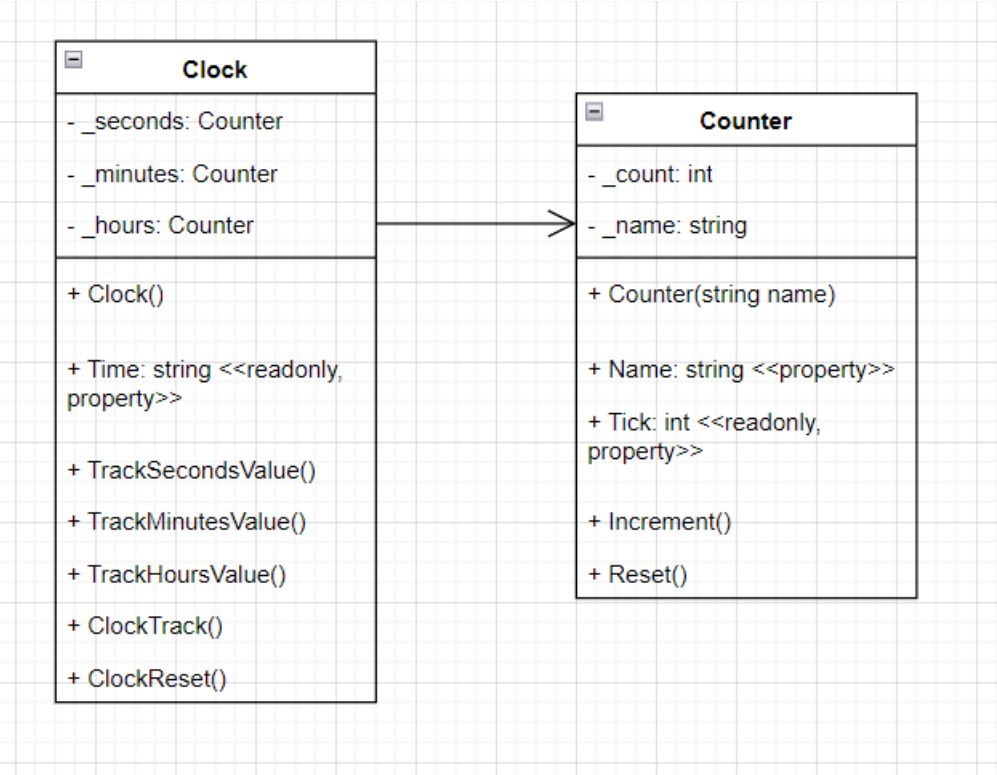
```
1 reference
public void Draw()
{
    SplashKit.FillRectangle(_color, _x, _y, _width, _height);
}

1 reference
public bool IsAt(Point2D point2D)
{
    if (point2D.X >= _x && point2D.X < _x + _width && point2D.Y >= _y && point2D.Y < _y + _height)
        return true;
    else
        return false;
}
```

2. Encapsulation: This OOP’s principle simply means is to encapsulate of data and code within an object. Encapsulation allows an object to control its own data (e.g, fields), and to provide access to only properties and methods necessary to interact with that data. This keeps the object's internal state remains consistent and protected from external interference of the program/application. In addition, in this principle, programmers also bundle data fields and methods that are directly associated with each other to present a specific functionality of behaviour of the class

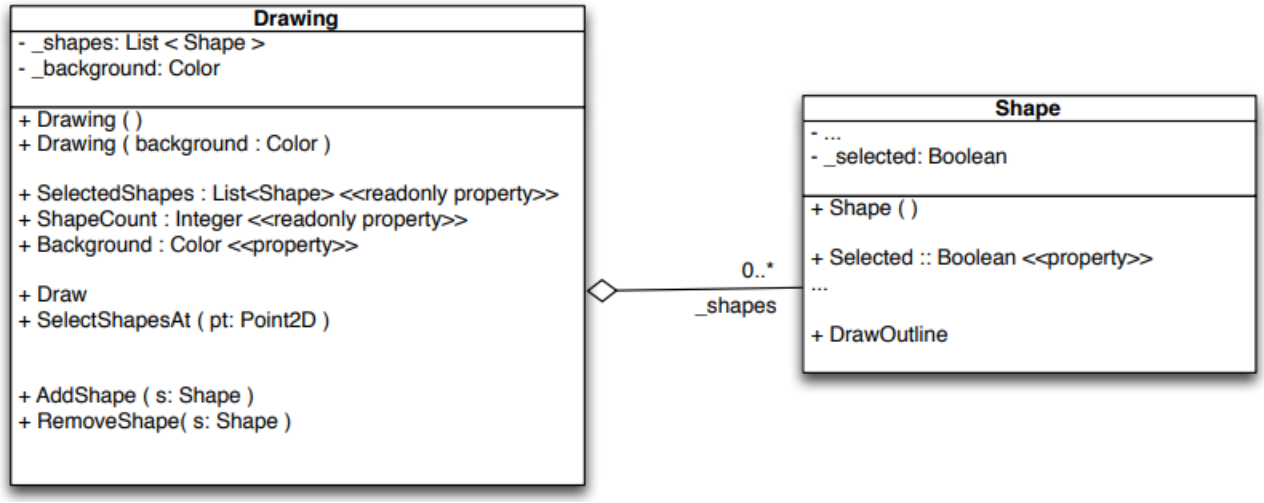
E.g.

- A simple example of using encapsulation is presented in the task “3.1P - Clock Class”:



In each “Counter” object, the state is defined by the private fields“_count” and “_name”. The external object like a “Clock” can not change the value of the private fields “_count” **directly**, but can modify the “Counter” object’s state of Count by calling the method “Increment()” or “Reset()”. Also, by encapsulating the “_name” and “_count” fields within public properties of “Name” and “Tick” the class can control how the field is accessed and modified from the external class like “Counter”.

- Taking the 3.3P - Drawing Program - A Drawing Class task as the more complicated example, in the “Shape” class there are:



+ Private fields: “_color”, “_x”, “_y”, “_width”, “_height”, and “_selected”. These are encapsulated within the class and are not directly accessible from the outside objects. They are used to store the internal data of a Shape object.

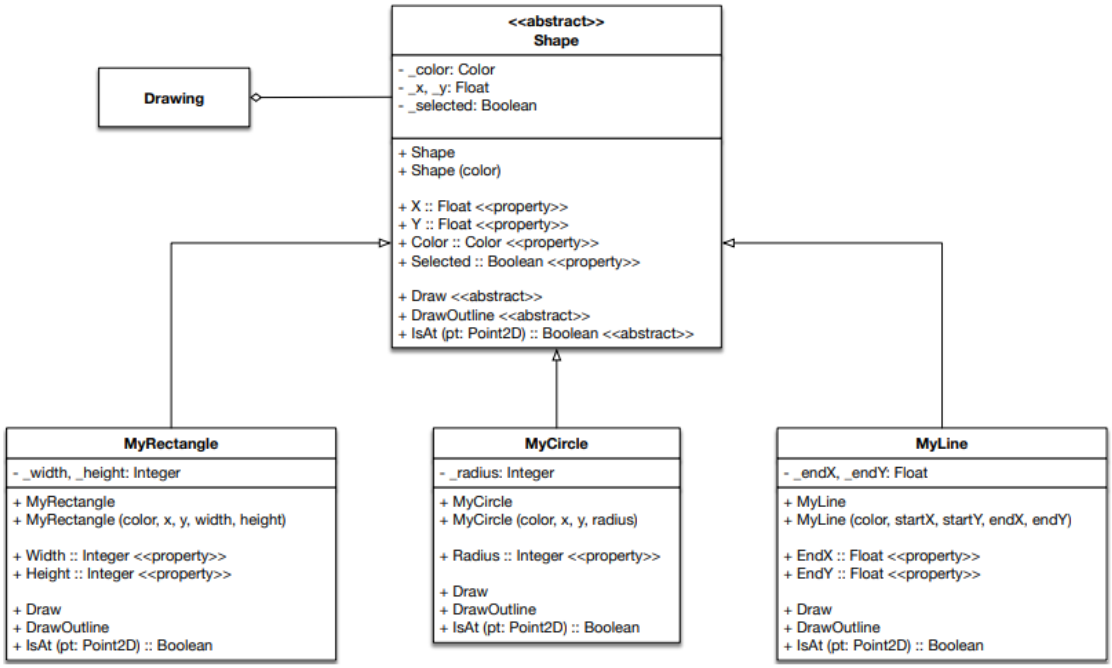
+ Public properties like “Color”, “X”, “Y”, “Width”, “Height”, and “Selected” provide controlled access to the private fields. These properties encapsulate the internal fields and

expose an interface for interacting with the Shape objects, allowing the values of private fields to be “get” and “set”, thus, enabling external objects to interact with the class while maintaining encapsulation.

+ Public methods, including “Draw()”, “IsAt()”, and “DrawOutline()”, encapsulate the functionality related to drawing and interacting with the shape. They utilize the private fields and provide a way to interact with the shape objects while abstracting away the implementation details.

3. Inheritance: This principle allows a class (child class/subclass) to inherit “protected” and “public” members from another class (parent class). Inheritance is often used to promotes code reuse and helps simplify the design of the programs/applications. Inheritance relationships are usually expressed as "is-a" relationships, with subclasses being a more specialized type of parent class, and is closely related to the principle Polymorphsim.

E.g. Let take an example of the work I have done in the task 4.1P - Drawing Program - Multiple Shape Kinds



The "MyRectangle", "MyCircle", and "MyLine" classes inherit from the "Shape" abstract class, which is a clearly example of inheritance in action. By inheriting from the abstract class "Shape", these subclasses inherit its public properties ("Color", "X", "Y", and "Selected"), and abstract methods ("Draw()", "IsAt()", and "DrawOutline()") that is overidden by those three with their own implementations specific to rectangles, circles, and lines respectively.

4. Polymorphism: This concept refers to the ability of objects to take on different forms or behaviors, depending on the context in which they are used. In other words, this means that the child class has both its type and its parent’s type, by implementing interface(s) or inheriting from a parent class. Thus, polymorphism promotes code reusability, readability, flexibility, and extensibility in OOP program/application designs.

E.g. Also take the above example of 4.1P task, the "MyRectangle", "MyCircle", and "MyLine" classes demonstrate the fourth principle by overriding the "Draw()", "IsAt()", and "DrawOutline()" methods inherited from "Shape". This means that any code that expects a "Shape" object can also use a "MyRectangle", "MyCircle", or "MyLine" object in its place, since those are the types of Shape. In other word, if there is a method that takes a "Shape" object as a parameter, it can be passed a "MyRectangle" object instead as "MyRectangle" is a child class of "Shape", as presented in the “Main()” method.

REFERENCES

[1] ChatGPT. Retrieved from: https://chat.openai.com/

[2] What is encapsulation?. Retrieved from: https://scoutapm.com/blog/what-is-encapsulation

[3] What is “Abstraction” in Programming?. Retrieved from: https://levelup.gitconnected.com/what-is-abstraction-in-programming-2f35c8c72e15