Multiple Inheritance and Design Patterns

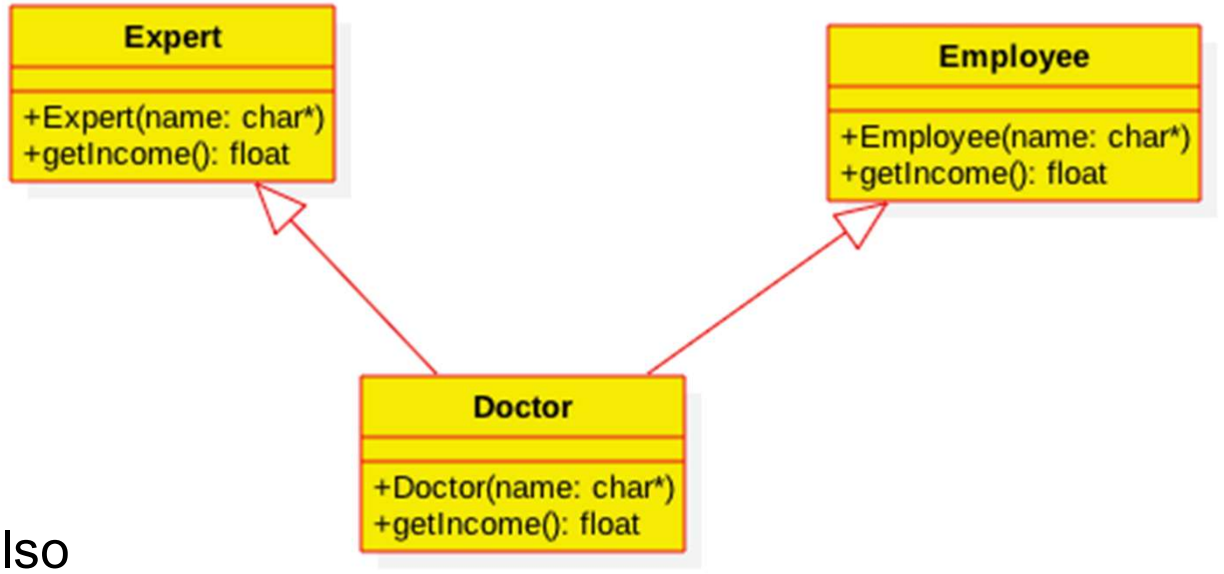
Nguyen Van Vu
nvu@fit.hcmus.edu.vn

Topics

- Multiple Inheritance
- Design Patterns

Multiple inheritance

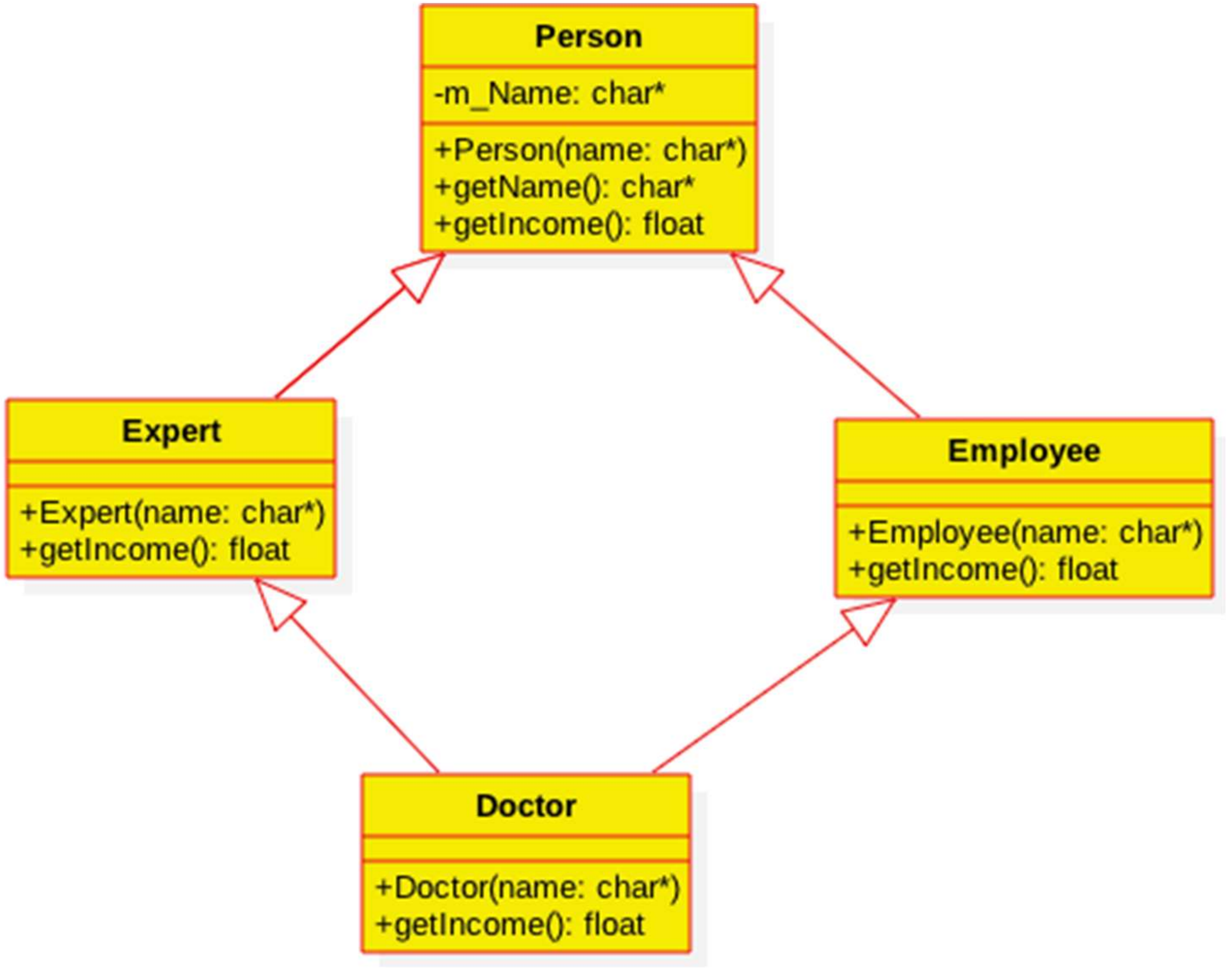
- There are cases when one class inherits multiple parent classes
- C++ supports multiple inheritance



A doctor is an employee but also an expert

The diamond problem

- This problem occurs when two parent classes inherit the same class



The diamond problem

```
class Person {
public:
      Person(char* name)  {}
};

class Expert : public Person {
public:
      Expert(char* name) : Person(name)   { }
};

class Employee : public Person {
public:
      Employee(char* name) : Person(name) { }
};

class Doctor : public Expert, public Employee  {
public:
      Doctor(char* name) : Person(name),
      Employee(name), Expert(name)   { }
};
```

```
void main() {
      Doctor d(“Nguyen A”);
}
```

Is there any problem with this code?

Problem
Person() constructor is called TWICE. Thus, two copies of Person’s members.

The diamond problem

- Solution to the diamond problem: use “virtual” keyword for inheritance
- class Expert : virtual public Person {}
- class Employee : virtual public Person {}
- This tells the complier to **dynamically** avoid calling constructors twice

Design patterns

- “Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” (Erich Gamma et al. 1995)

- A pattern is a description of **a repeatable problem** and **its common solution**

Key reason behind design patterns

- Open-closed principle
  - "software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification" (Meyer 1988)

- It is more costly to repaire software than extending it

- Design patterns are a mechanism to implement the open-closed principle

Benefits of design patterns

- Make OOP programs easier to maintain
- Allow OOP programs easier to extend

- Reuse knowledge – known solutions are shared in forms of design patterns
  - Reduce time from understanding code if you are already familiar with design patterns

- Reduce software maintenance and enhancement cost

Four essential elements of a pattern

- Name
  - A meaningful, easy-to-remember pattern identifier
- Problem description
  - When to apply, context, conditions
- Solution description
  - Elements, relationships, responsibilities, collaborations
- Consequences
  - Results and trade-offs of applying the pattern

Practice – continue LN08 Practice 2

- We have the following classes
  - Circle
  - Rectangle
  - Square
- Let's write classes to draw these shapes so that we can have a list of shapes to be drawn on screen

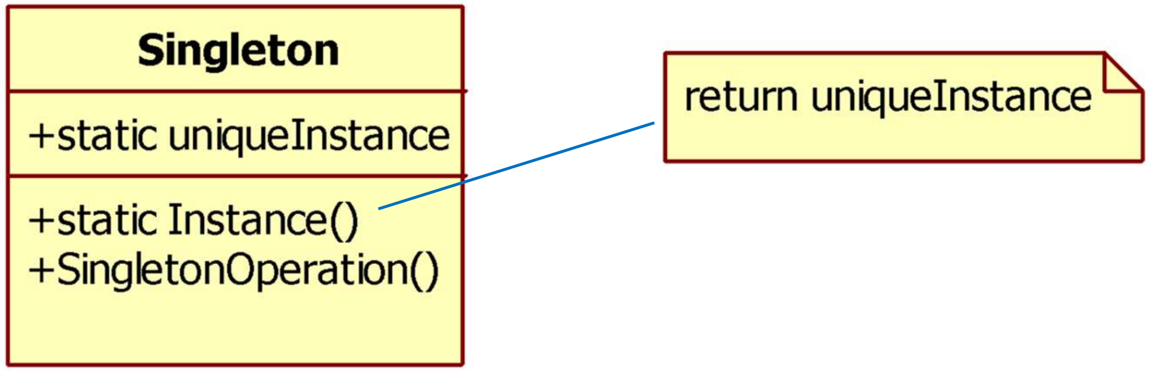
- What would you do if we want to draw a circle in two ways, thick circle and filled circle?

Popular design patterns

- Singleton
- Builder
- Factory Method
- Prototype
- Bridge
- Iterator
- Observer
- State
- Strategy
- Command

Singleton

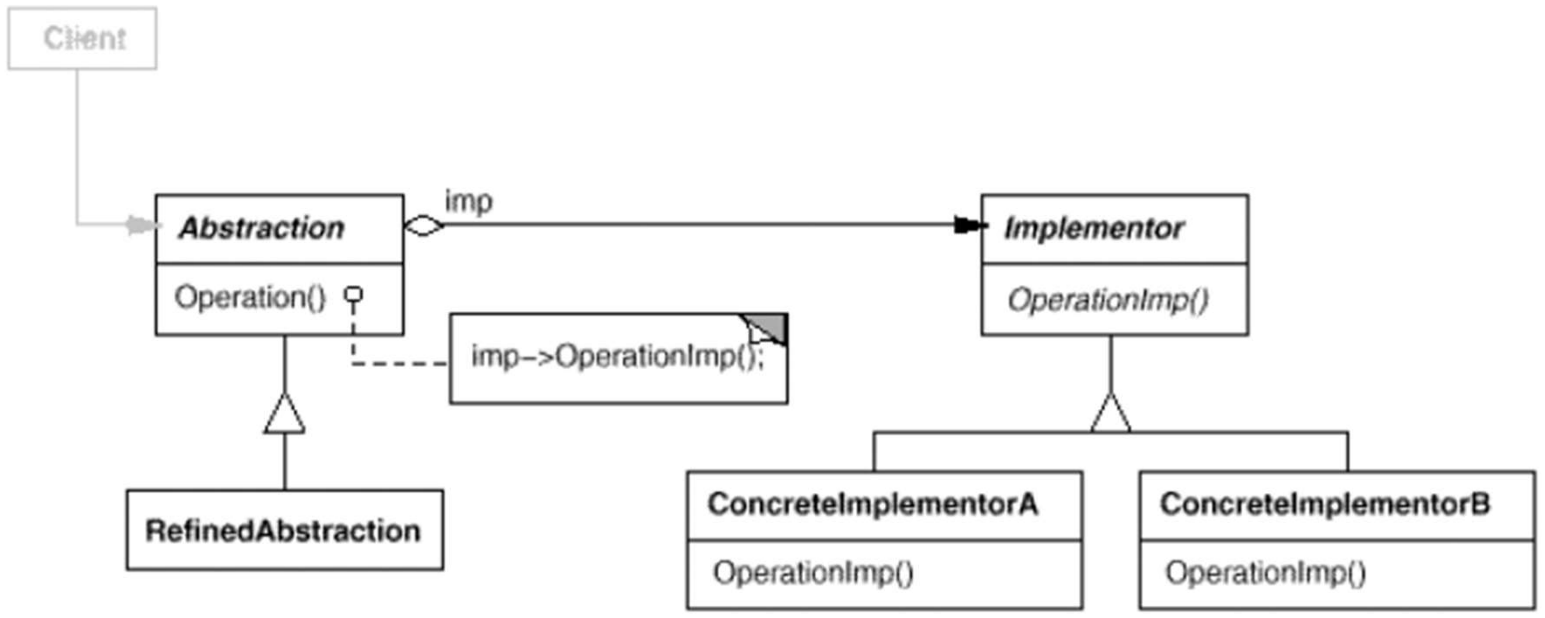
- Problem
  - Make sure one class to have only one instance. Provide a point of access to obtain the instance
- Consequences
  - Controlled access to sole instance
  - Allows extension to have a number of instances
- Solution



- Can you implement this?

Bridge

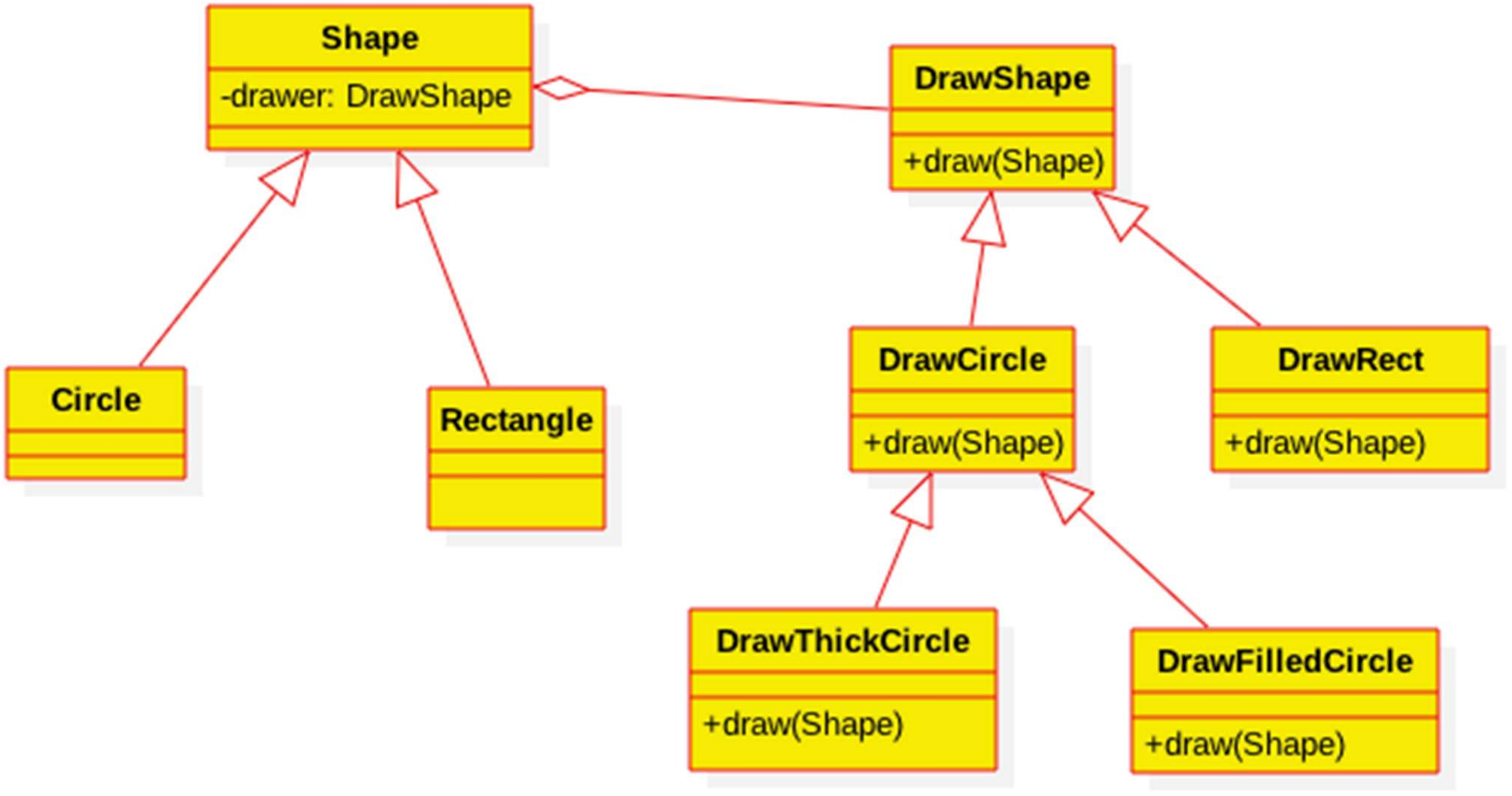
- Purpose
  - Separate an abstraction from its implementation so they can vary independently
- Structure



Practice 3

- Continue Practice 2
  - We have shapes of different types, including circles and rectanges.
  - We need to draw these shapes in different ways, such as thick circles, filled circles
  - In the future we may have other ways to draw these shapes, like a circle with a center dot
  - How do we construct classes so that when adding new ways to draw circles or rectangles, we DO NOT change the circle and rectangle class?

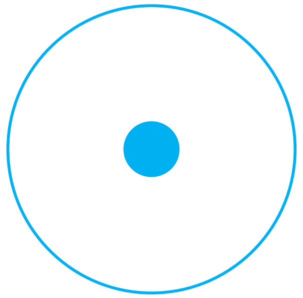
Practice 3: solution



Practice 3: solution

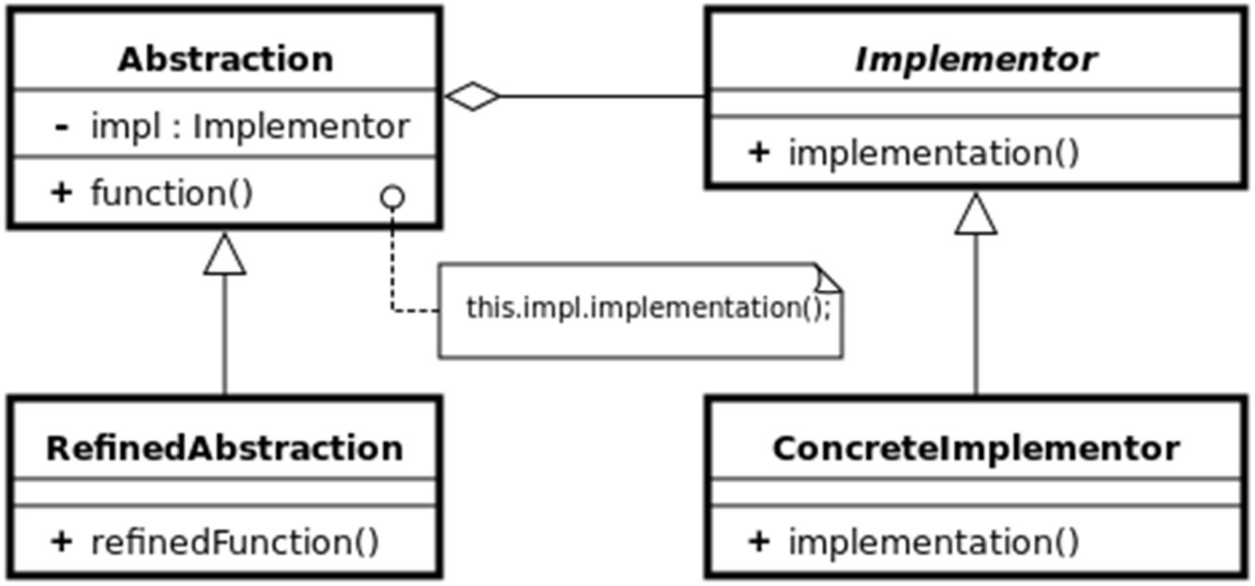
- Implement the above classes using C++
- Write the main method to draw both thicked circle and filled circle

- Add a class to draw a centered circle (a circle having a center like below)



Practice 3

- The solution is called Bridge design pattern



Source: https://en.wikipedia.org/wiki/Bridge_pattern

Bridge design pattern

- Separate an abstraction (declaration) from its implementation to allow multiple implementation for a function
  - E.g., different ways to draw a circle
- Hide implemetation details from clients
  - E.g., the main function does not need to know details on how to draw a circle

Practice 4

A motor-bike consumes 2 litters of fuel for 100 km and consumes more 0.1 litters for every additional 10 kg of goods.

A truck consumes 20 litters of fuel for 100 km, consumes more 1 litters for every additional 1000 kg of goods.

Construct classes **Vehicle**, **MotorBike** and **Truck** that can do the followings:

- Add a weight of goods to the vehicle.
- Remove a weight goods from the vehicle.
- Add an amount of fuel to the vehicle.
- Run the vehicle a length of km.
- Get the current fuel left in the vehicle.

Practice 5

- Van felt sick one day and she went to Thong Nhat hospital. There she was first checked by Minh who is a nurse. Minh recorded Van’s patient information including Name, ID, Date of birth, Weight, and Height. Van was then examined by Mai who a doctor at the hospital. After examining, Mai consulted and provided Van a prescription (đơn thuốc). Van used the prescription to by medicine (thuốc) needed to treat her illness. Minh and Mai are employees working in the same department at Thong Nhat hospital, and they have Name, ID, Date of birth.

- Write neccessary code (classes, operations, attributes, main function, etc.) to implement the situation

References

- Meyer, Bertrand (1988). Object-Oriented Software Construction. Prentice Hall. ISBN 0-13-629049-3.
- Gamma, Helm, Johnson & Vlissides (1994). Design Patterns. Addison-Wesley. ISBN 0-201-63361-2.