Polymorphism

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

Topics

- Virtual function
- Abstract class
- Polymorphism

Virtual function

- Static binding problem:

```
class Animal
{
public:
        void talk()  { cout << “Don’t talk”; }
};

class Cat: public Animal
{
public:
        void talk()  { cout << “Meo meo”; }
};

class Dog: public Animal
{
public:
        void talk()  { cout << “Gau gau”; }
};
```

```
void doSomething(Animal  p)
{
        p.talk();
}
void main()
{
        Cat       c;
        Dog       d;
        doSomething(c);
        doSomething(d);

        Animal  *p;
        p = &c;
        p->talk();
        p = &d;
        p->talk();
}
```

Bind to Animal implementation when compile

Bind to Animal implementation when compile

Virtual function

- Virtual function concept
  - Normal function
    - Function call binds to implementation at compile-time.
      - Static binding.
  - Virtual function
    - Function call binds to implementation at run-time .
      - Dynamic binding.
      - Implementation depends on run-time object.
  - C++ usage
    - Declaration: **virtual** <Function signature>;
    - Called through object pointer.

Virtual function

- Dynamic binding

```
class Animal
{
public:
        virtual void talk()  { cout << “Don’t talk”; }
};

class Cat: public Animal
{
public:
        void talk()  { cout << “Meo meo”; }
};

class Dog: public Animal
{
public:
        void talk()  { cout << “Gau gau”; }
};
```

```
void doSomething(Animal  *p)
{
        p->talk();
}
void main()
{
        Cat       c;
        Dog       d;
        doSomething(&c);
        doSomething(&d);

        Animal  *p;
        p = &c;
        p->talk();
        p = &d;
        p->talk();
}
```

implementation depends on run-time object

implementation depends on run-time object

Virtual function

- Pure virtual function
  - Has declaration only, no implementation
  - **virtual** <Function signature> **= 0**
  - Used for dynamic binding
  - Derived class provides implementation

```
class Animal
{
public:
        virtual void talk() = 0;
};
```

Pure virtual function, has no implementation!!

Virtual function/operation

- We sometimes declare a function, but we do not implement it

```
class Shape {
protected:
    int m_Width;
        int m_Height;
public:
    virtual int getArea() = 0;
};
```

For a general shape, we do not know how to calculate area.

A pure virtual function has "= 0"

Virtual function

- Example:

```
class Animal
{
public:
        virtual void talk() = 0;
};

class Cat: public Animal
{
public:
        void talk()  {  cout << “Meo meo”;  }
};

class Dog: public Animal
{
public:
        void talk()  {  cout << “Gau gau”;  }
};
```

Abstract class

```
void doSomething(Animal  *p)
{
        p->talk();
}
void main()
{
        Cat       c;
        Dog       d;
        doSomething(&c);
        doSomething(&d);

        Animal  *p;
        p = new Animal; // Wrong
        p = new Cat;      // Right
        p->talk();
}
```

implementation depends on run-time object

Abstract class

- An abstract class is a class having at least one pure virtual function
  - Pure virtual operation does not have implementation

- An abstract class is called interface (in C++)
- We cannot instantiate an object from an abstract class
- A concrete class is a class that can be instantiated
- A derived concrete class must implement virtual functions from the base class

Abstract class example

```
class Shape {
protected:
        int m_Width;
        int m_Height;
public:
        virtual int getArea() = 0;
};

class Rectangle : public Shape {
public: int getArea() {
        return (m_Width * m_Height);
}
};
class Triangle: public Shape {
public: int getArea() {
        return (m_Width * m_Height)/2;
}
};
```

```
main() {
        Rectangle rect;
        Triangle tri;
        rect.setWidth(5);
        rect.setHeight(7);

        tri.setWidth(5);
        tri.setHeight(7);

        Shape shape; // wrong!
}
```

Why do we need abstract classes?

- An abstract class provides a base class for inheritance
- Detailed implementation of one or many operations is yet to know
- Support polymorphism

Polymorphism

- Polymorphism = having multiple forms
- Function to be bound or called is decided at run-time

```
int getArea( int type )
{
        if ( type == 0 )
        {
                Rectangle  r;
                return r.getArea( );
        }
        else if ( type == 1 )
        {
                Triangle t;
                return t.getArea( );
        }
}
```

What is wrong with this?

Polymorphism

```
class Shape {
protected:
        int m_Width;
        int m_Height;
public:
        virtual int getArea() = 0;
};

class Rectangle : public Shape {
public: int getArea() {
        return (m_Width * m_Height);
}
};
class Triangle: public Shape {
public: int getArea() {
        return (m_Width * m_Height)/2;
}
};
```

```
main() {
        Rectangle rect;
        Triangle tri;
        rect.setWidth(5);
        rect.setHeight(7);

        tri.setWidth(5);
        tri.setHeight(7);

        Shape *shape;
        shape = &rect;
        cout << shape->getArea() << endl;

        shape = &tri;
        cout << shape->getArea() << endl;
}
```

Virtual destructor: problem

```
class Employee
{
private:
        char        *m_Name;
public:
        ~Employee()  {
                delete m_Name;
        }
};
class Doctor : public Employee
{
private:
        char        *m_Specialty;
public:
        ~Doctor()  {
                delete m_Specialty;
        }
};
```

```
void main()
{
        Doctor  *doc = new Doctor;
        delete  doc;

        Employee  *e = new Doctor;
        delete  e;
}
```

Call order:
~Doctor()
~Employee()

Call order:
~Employee()

Is there any problem with this?

Virtual destructor: solution

```
class Employee
{
private:
        char        *m_Name;
public:
        ~virtual Employee()  {
                delete m_Name;
        }
};
class Doctor : public Employee
{
private:
        char        *m_Specialty;
public:
        ~Doctor()  {
                delete m_Specialty;
        }
};
```

```
void main()
{
        Doctor  *doc = new Doctor;
        delete  doc;

        Employee  *e = new Doctor;
        delete  e;
}
```

Call order:
~Doctor()
~Employee()

Call order:
~Doctor()
~Employee()

Practice 1

The table below tells the average speeds of some animals:

Animal	Speed
Cheetah	100km/h
Antelope	80km/h
Lion	70km/h
Dog	60km/h
Human	30km/h

Write a function to take 2 animals from the above table as arguments then tells which animal wins the race.

In case we add the horse (60km/h) to the table, what changes can be made?

- Open/closed principle

- Open to extension
- Closed from modification

```
Races(Animal *a, Animal *b)
{
}
```

Practice 2

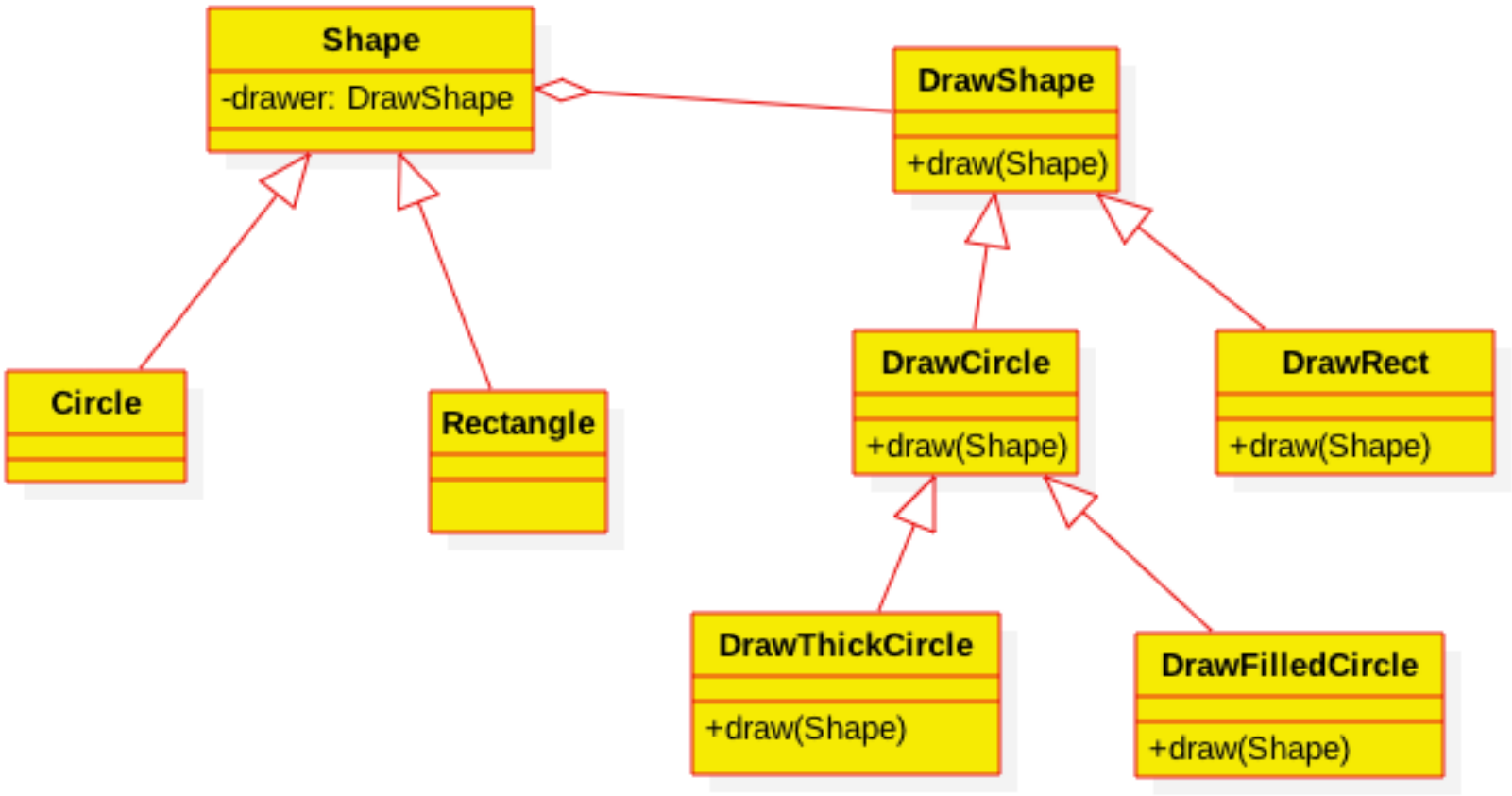
- We have the following classes
  - Circle
  - Rectangle
  - Square
- Let's write classes to draw these shapes so that we can have a list of shapes to be drawn on screen

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

Practice 3

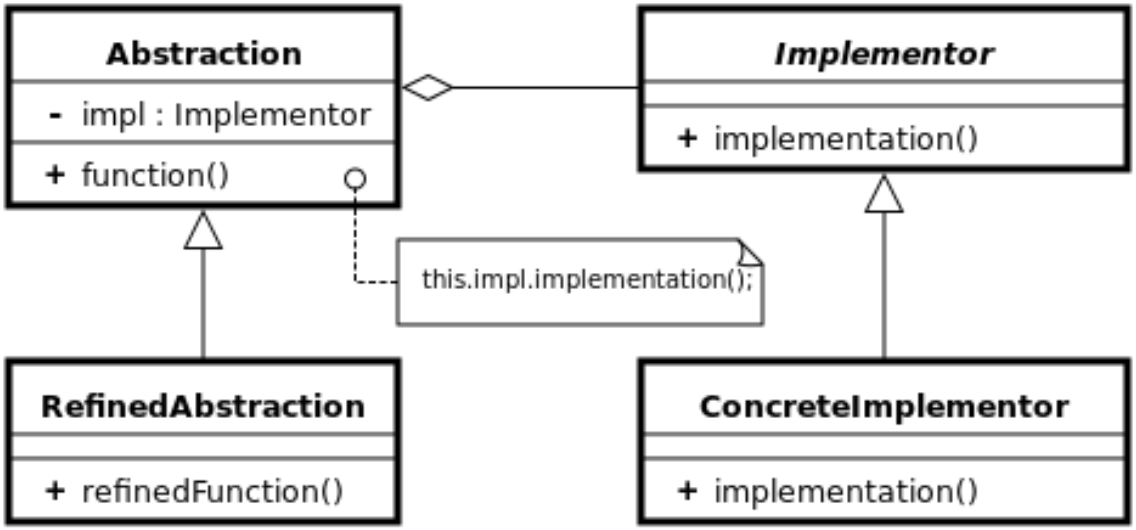
- Continue Practice 2
  - We have shapes of different types, including circles and rectanges.
  - We need to draw these shapes in different ways, such as thick circles, filled circles
  - In the future we may have other ways to draw these shapes, like a circle with a center dot
  - How do we construct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

Practice 3: solution



Bridge

- The solution to Practice 2 is called Bridge design pattern



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

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.