OOP回顾

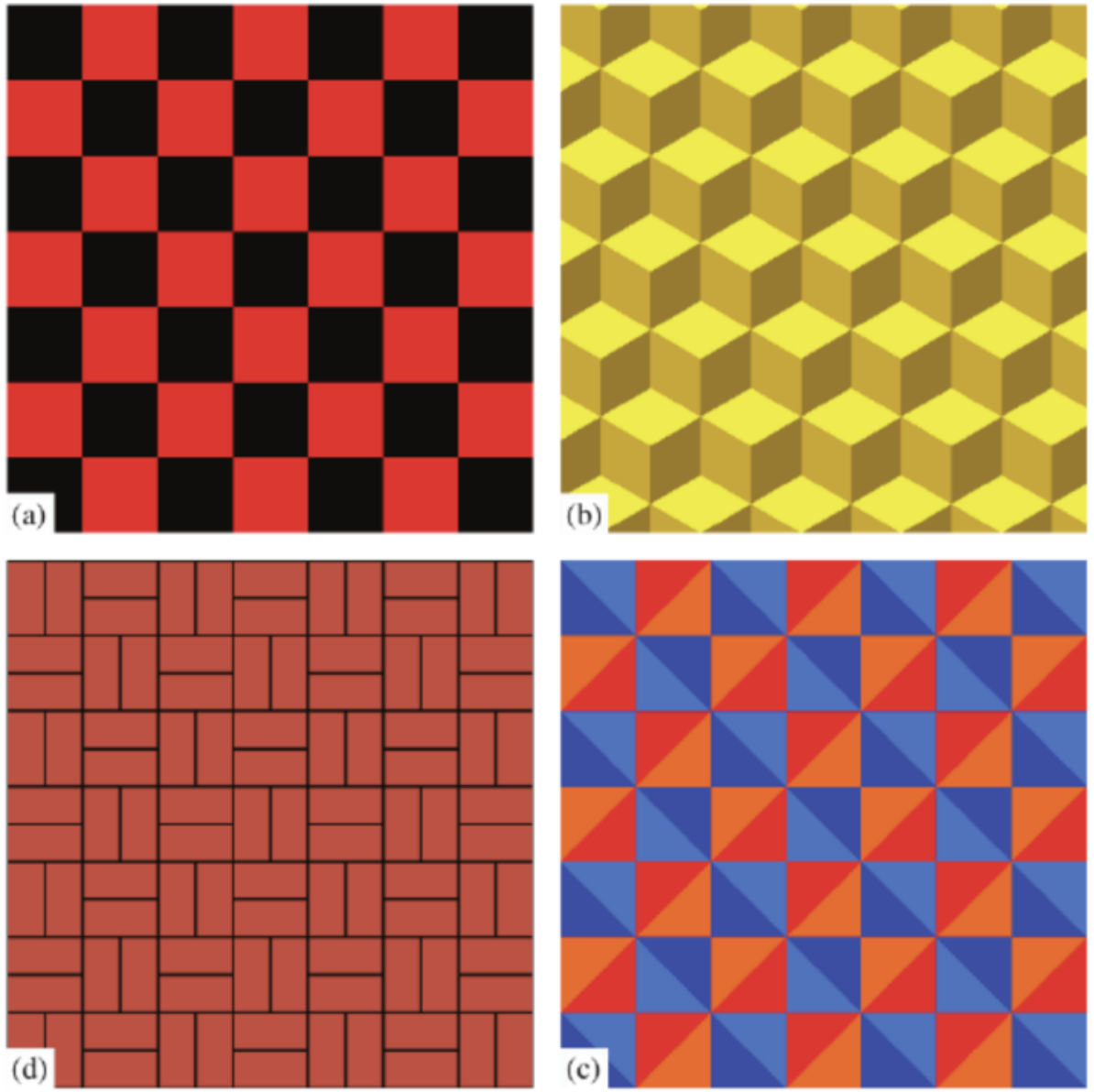
大纲

1. 编程范式(Programming paradigms)
2. OOP 的主要特征

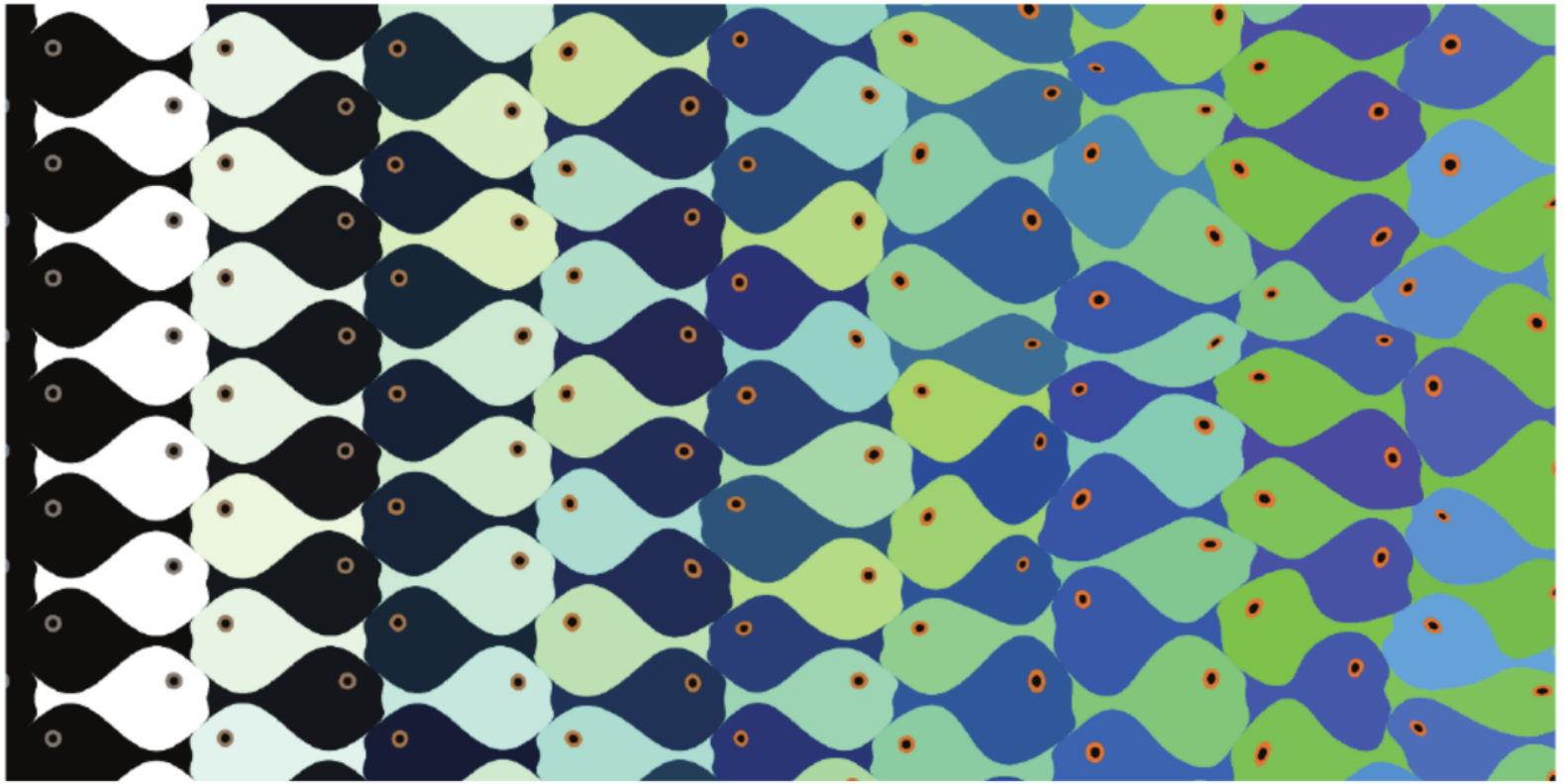
编程范式（Programing Paradigms）

编程范式

Space Filling Pattern



编程范式



编程范式

命令式(Imperative)

- 过程式编程(procedural)
- 面向对象式(object-oriented)

声明式(Declarative)

- 函数式(functional)
- 逻辑式(logic)
- 响应式(reactive)

编程范式

基于类型的分解

如果一个系统在演化过程中需要修改时，基本上可以通过增加或者修改类来实现，那么这样的系统就适合于采用面向对象的方式来实现。

基于功能的分解

如果大部分的演化可以通过增加或者修改功能来实现，那么这样的系统就适合于采用面向过程的方法来实现。

函数式的分解

构造系统的基本单位是类似数学中的函数，支持高阶函数，函数中不能使用变量。

一个反例

面向对象程序设计语言的主要特征

- 抽象 Abstract
- 封装 Information Hiding
- 组合 Composition
- 继承 Inheritance
- 多态 Polymorphism

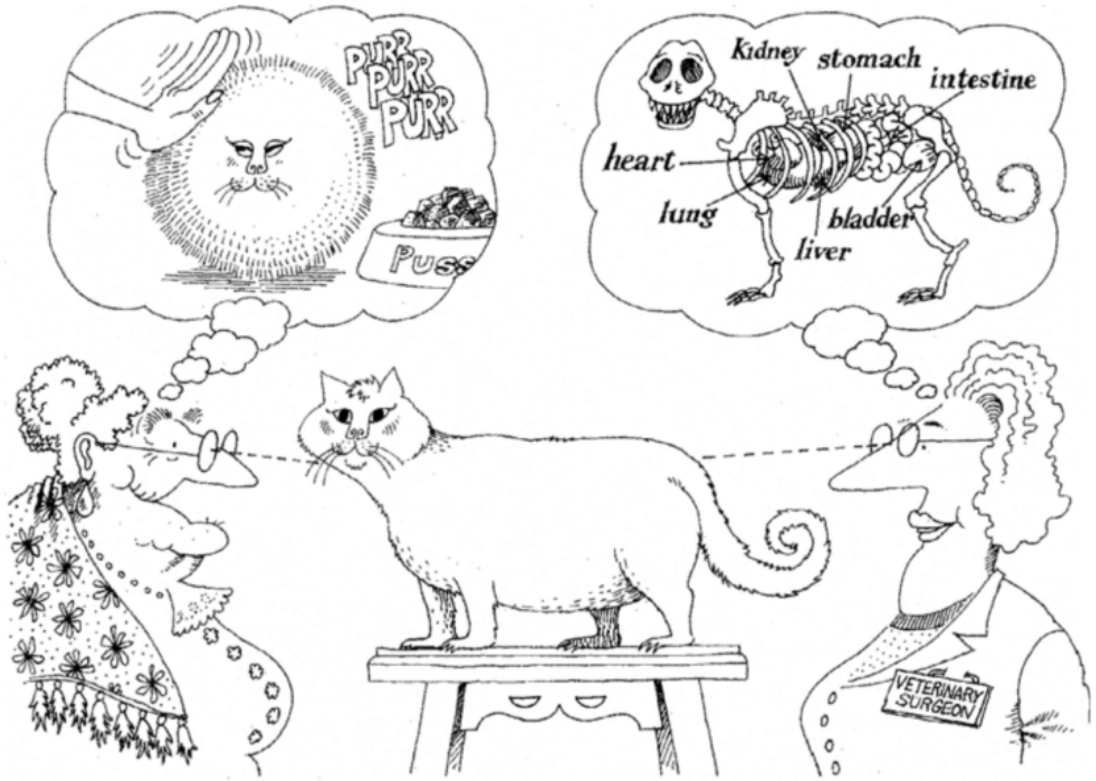
抽象

Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose.

http://en.wikipedia.org/wiki/Abstraction

抽象

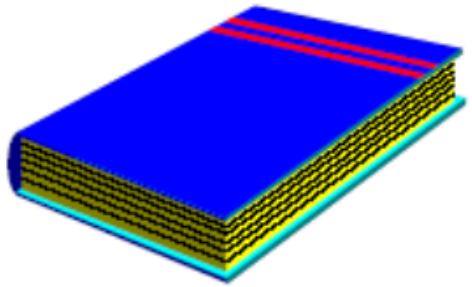
- 抽象是依赖于场景的，不同的场景需要不同的抽象
- Object-oriented analysis and design with applications / Grady



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

面向对象方法中的抽象(1)

domain concept



Book

title

visualization of domain concept

representation in an object-oriented programming language

```
public class Book
{
private String title;

public Chapter getChapter(int) {...}
}
```

抽象

- 所有编程语言都提供抽象。
- 程序作为现实世界的模型。
- 可以说，能够解决的问题的复杂性与抽象的方式和质量直接相关。

面向对象方法中的抽象

以下特点代表了纯粹的面向对象编程方法：

- 一切都是对象;
- 程序是一组对象,通过发送消息来相互协作;
- 每个对象都有自己的存储，可以由其他对象组成;
- 每个对象都有一个类型;
- 同一类型的所有对象都可以接收相同的消息;

纯的面向对象的程序设计语言能够保证写出面向对象风格的程序吗?

封装

对于一个复杂的系统，我们需要将问题分解为多个类。

系统中定义的大部分类实际上是为了给其他类使用，为其它类提供服务。

这些类是基于对系统的理解和抽象设计和实现的。

这些类应该只对外提供必要的功能，实现的细节应该封装在内部。

封装

- 能够热咖啡的CPU
- 看上去似乎提供了更多的功能，但是有什么问题?

封装

- 基于抽象和封装的设计

封装

获取一年中的某天是否为节假日

```
class HolidayManager{
        public char[365] days;
}
```

封装

获取一年中的某天是否为节假日

```
class HolidayManager{
    private char[365] days;
    public boolean isHoliday(Date date){
            ……..
    }
}
```

封装

获取一年中的某天是否为节假日

```
interface HolidayManager{
    public boolean isHoliday(Date date);
}
```

封装

现实世界中的信息隐藏

- 电视遥控器
- 咖啡加热器

程序中的信息隐藏(封装)

- 提供更加明确的接口(界面)
- 保护易于变化的部分
- 通过访问控制来达到隐藏实现细节的目的

抽象与封装

- 抽象让人能从一个特定的层次来观察对象，而封装则限制你只能从这个层次来观察对象。

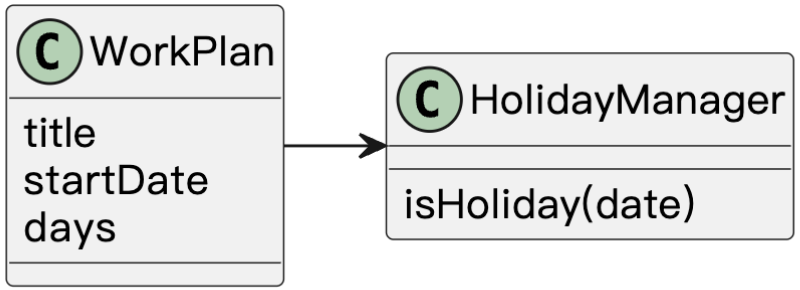
复用

面向对象中一个对象复用另一个对象主要由两种方式：

1. 组合
2. 继承

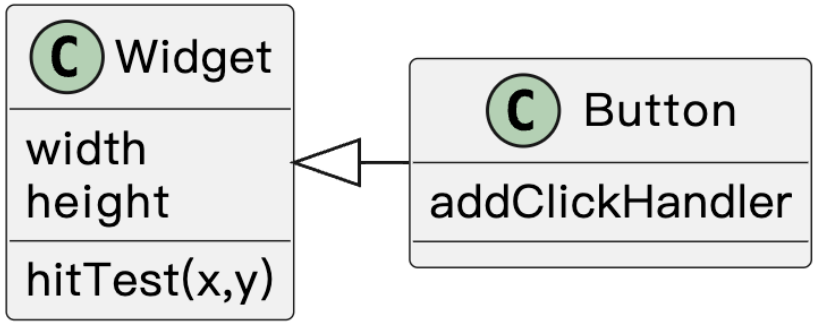
组合

- 一旦一个类被创建并测试通过，它应该（理想情况下）代表一个有用的代码单元。
- 事实证明，实现这种可复用性并不像许多人希望的那样容易。
- 最简单的复用一个类的方法是将该类的对象放置在一个新类中，这个概念被称为组合。



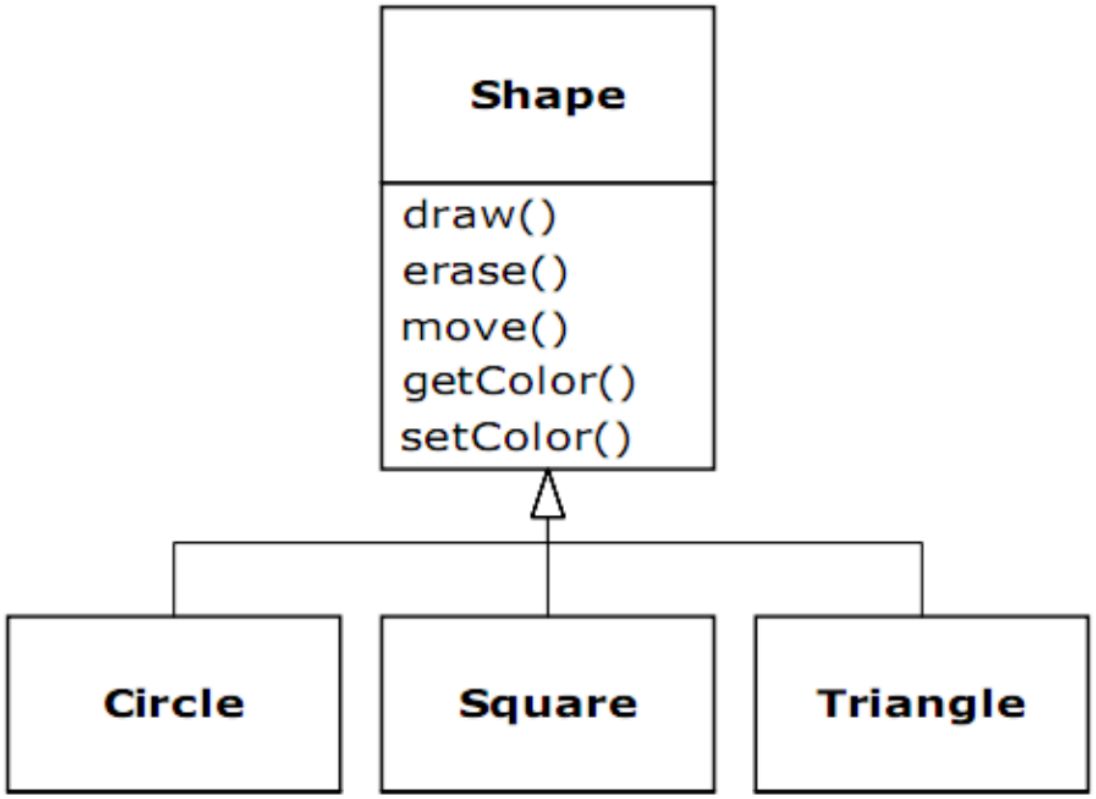
继承

- 当你从一个现有类型继承时，它会复制基类的接口及其实现代码。也就是说，你可以将所有可以发送给基类对象的消息也发送给派生类对象。



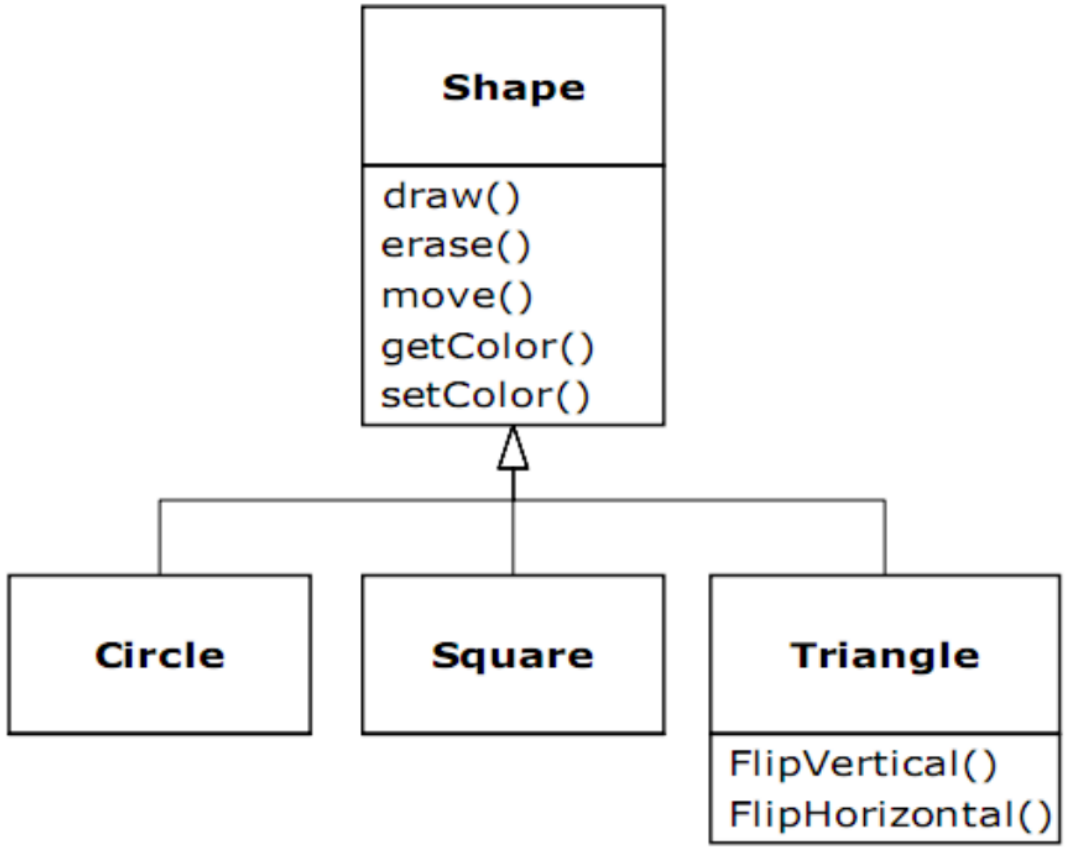
继承

- 继承反映了类型之间的关系
- 简单地继承一个类，而不做其他任何事情，派生出的类于其父类具有相同的行为，这一点并不是特别有意义。



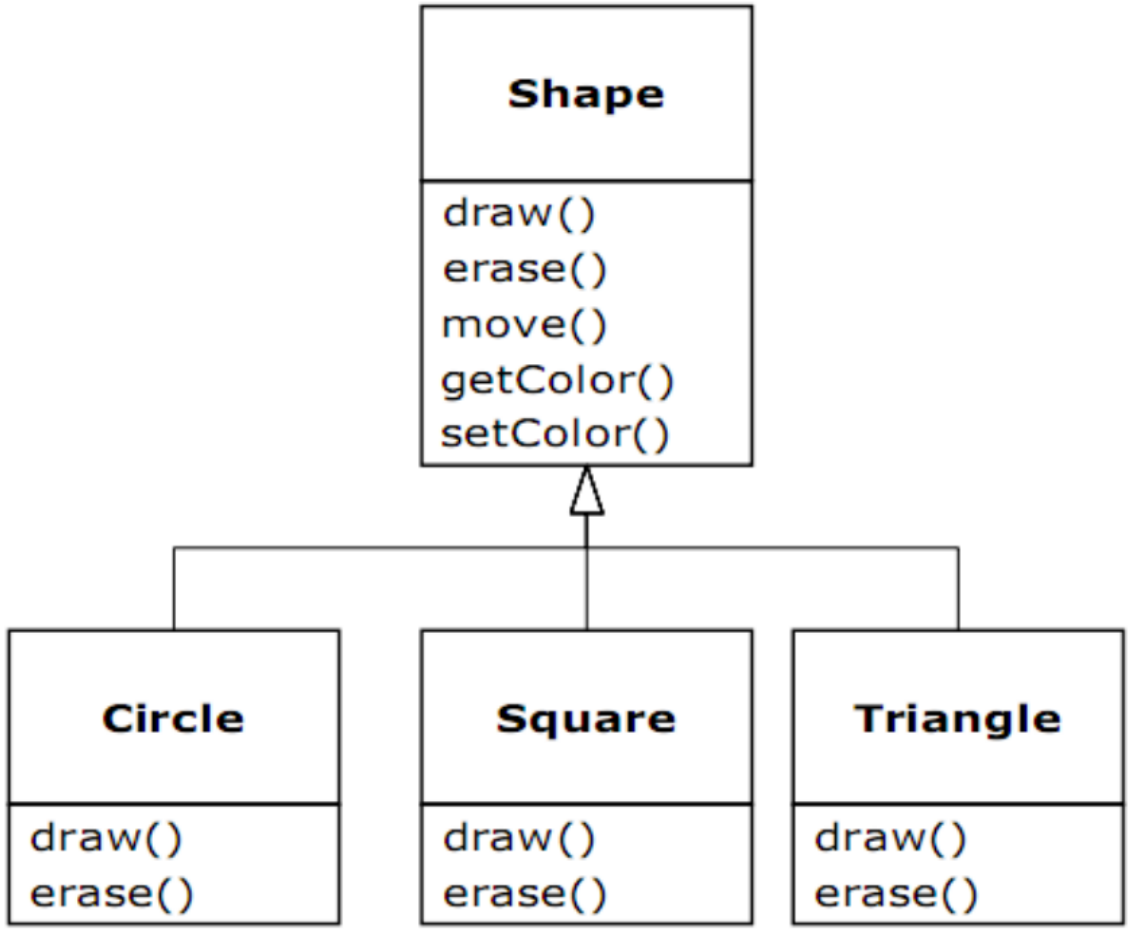
继承

- 改变接口
- 由于基类没有我们期望这个子类提供的接口，因此需要向子类中增加全新的函数。
- 在这种情况下，Triangle作为Shape的行为没有改变。

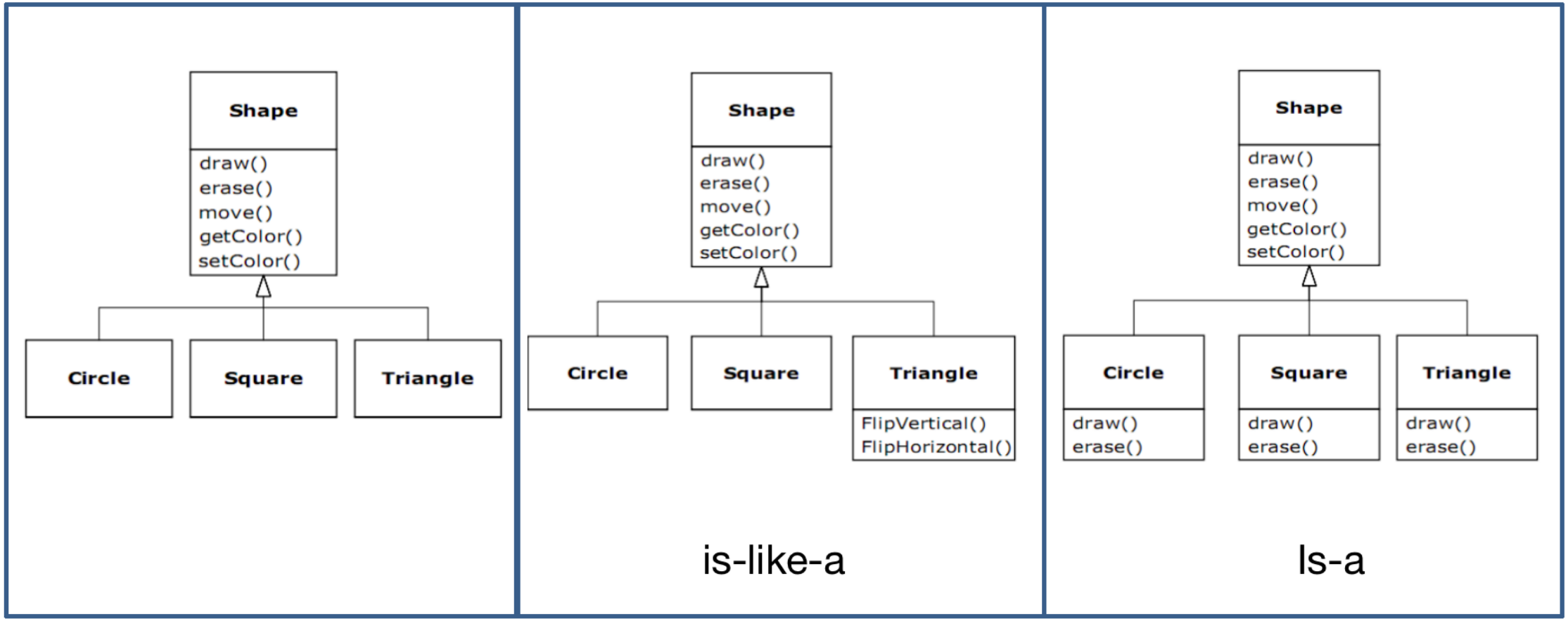


继承

- 覆盖(overriding)
- 使用同样的接口，但子类与基类将具有不同的行为。



继承


Shape
draw()
erase()
move()
getColor()
setColor()
Circle
Square
Triangle
Shape
draw()
erase()
move()
getColor()
setColor()
Circle
Square
Triangle
FlipVertical()
FlipHorizontal()
is-like-a
Shape
draw()
erase()
move()
getColor()
setColor()
Circle
draw()
erase()
Square
draw()
erase()
Triangle
draw()
erase()
Is-a

继承/多态

Is-a

派生类型与基类完全相同
可以将其视为纯粹的替代
这时我们称之为多态

is-like-a

当必须向派生类型添加新的接口时，使用该方式创建新的类型。
新类型仍然可以替代基类型，但替代并不完美，因为新接口不能从基类型访问。

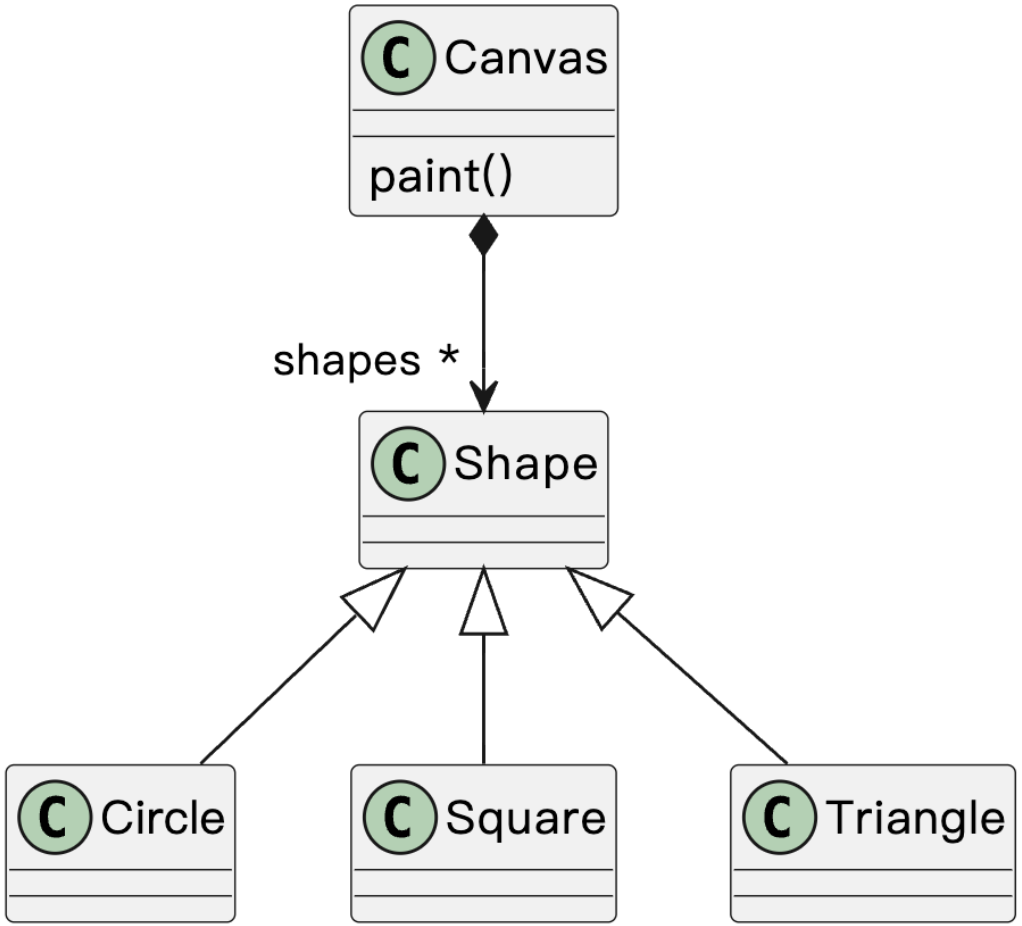
多态

在处理类型层次结构时，通常希望将对象视为其基类类型，而不是特定的类型。

这样可以编写不依赖于特定类型的代码。

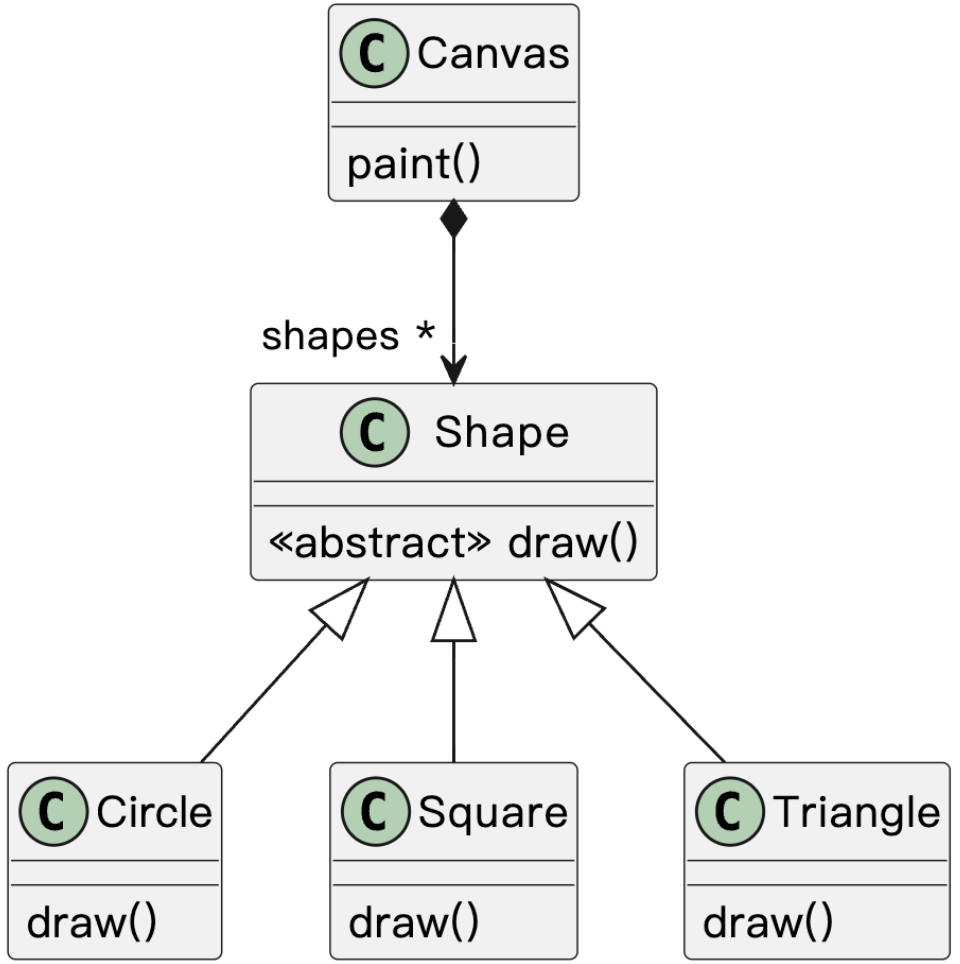
多态(示例一)

```
public class Canvas{
  public void paint(){
    for(Shape shape : shapes){
      if(shape instanceof Circle)
        ...
      else if(shape instanceof Square)
        ...
      ...
    }
  }
}
```



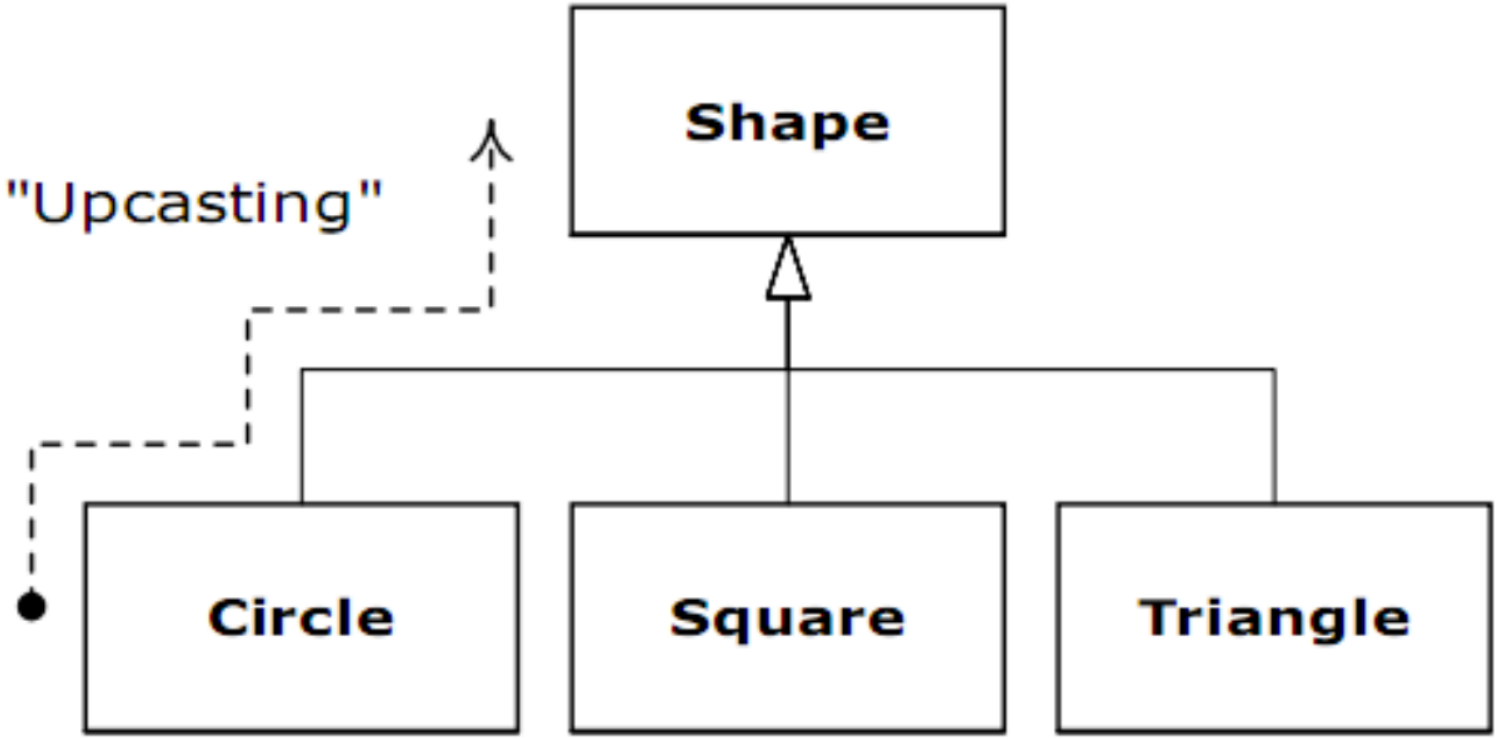
多态(示例一)

```
public class Canvas{
  public void paint(){
    for(Shape shape : shapes){
      shape.draw()
    }
  }
}
```



多态(示例一)

向上类型转换



组合与继承：如何选择

组合具有更好的灵活性。

- 组合的对象通常是私有的，具有更好的封装性。
- 成员可以动态更改，可以在运行时改变程序的行为。

作为一个示例，考虑人分为男性和女性这个需求如何用OO建模。

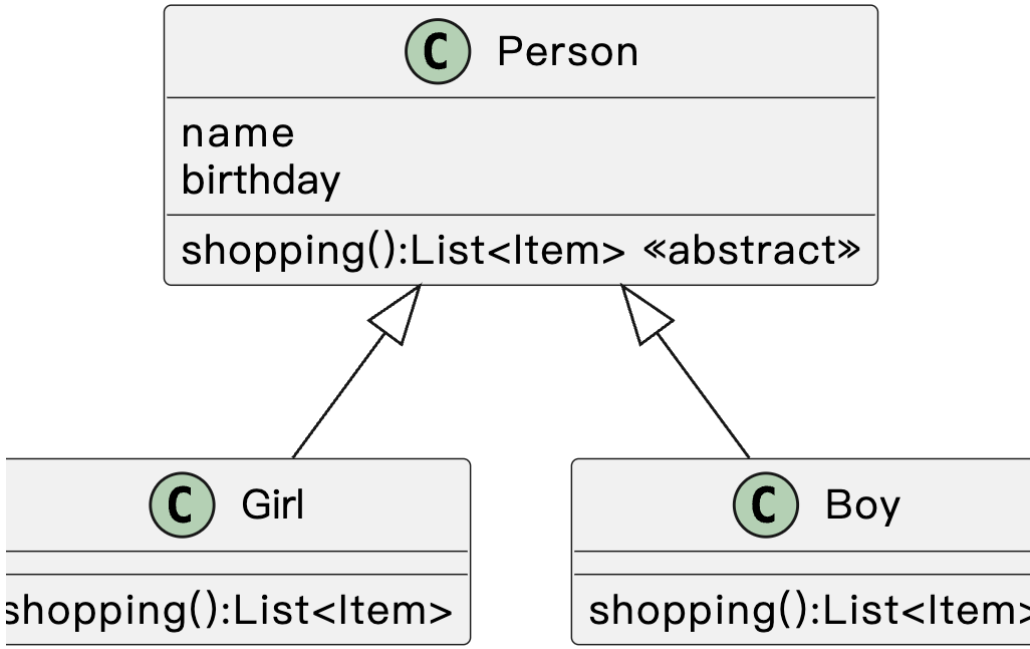
组合与多态(示例二)

购物行为模拟器

```
public abstract class Person{
        public String getName(){...}
        public Date getBirthday(){...}
        public abstract Collection<Item> shopping();
}

public class Girl extends Person{
        public Collection<Item> shopping(){...}
}

public class Boy extends Person{
        public Collection<Item> shopping(){...}
}
```



组合与多态(示例二)

注意：通常表示人的性别这种分类是不应该通过继承的方式的。
继承被滥用很危险。
在可能的情况下，用属性来组合其他对象以达到复用的效果。

```
enum Gender{
        male, female
}

public class Person{
        public String getName(){...}
        public Date getBirthday(){...}
        public Gender getGender(){...}
}
```

组合与多态(示例二)

然而，就我们的应用而言，由于Girl和Boy在Shopping时的行为有所不同，如果不采用继承的方式会导致下面的代码：

```
public class Person{
  public Gender getGender(){...}
  public Collection shopping(){
    if(getGender()==Gender.male){...}
    else{...}
  }
}
```

组合与多态(示例二)

使用多态的方式，我们的模拟器的框架代码为:

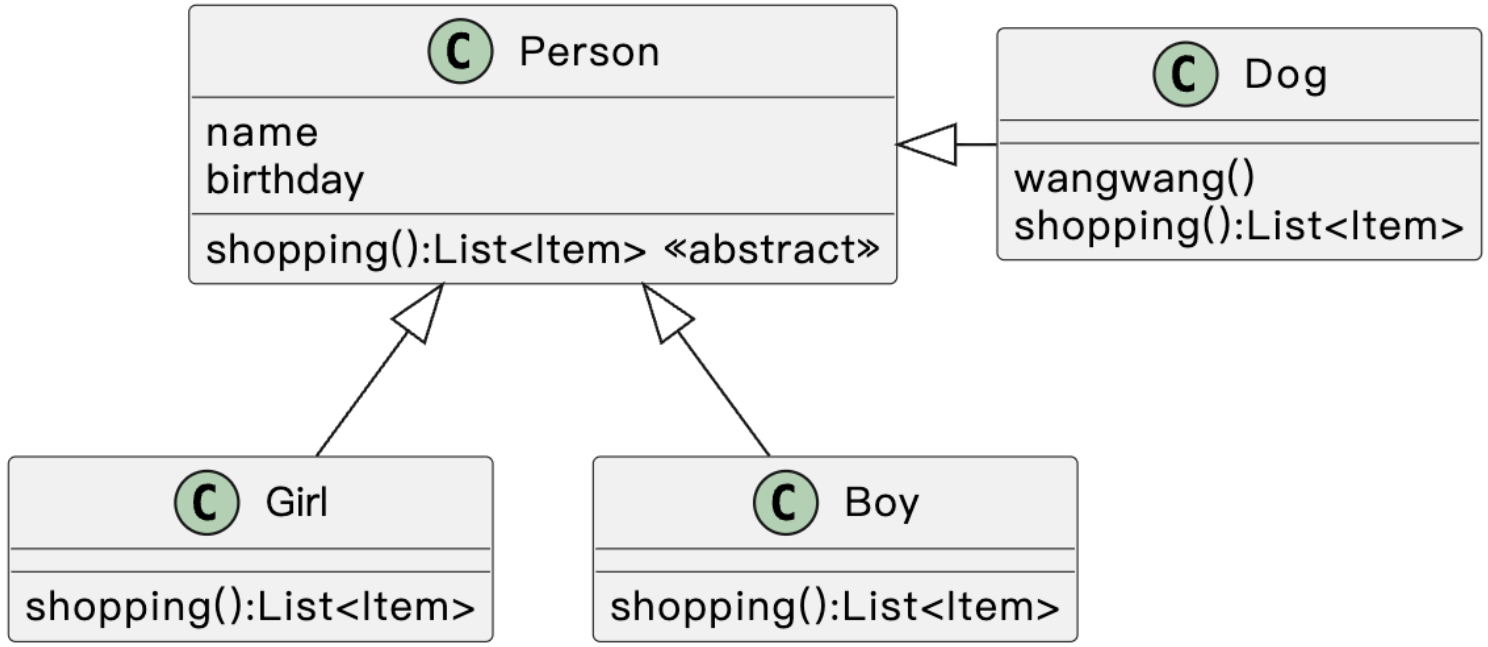
```
public class ShoppingSimulator{
        public Collection simulate(Collection<Person> persons){
                Collection<Item> items = new ArrayList();
                for(Person person : persons){
                        items.addAll( person.shopping() );
                }
                //analysis items
        }
}
```

组合与多态(示例二)

假如狗也可以购物

????

组合与多态(示例二)




```
public class Dog extends Person{
  public void wangwang()
  public void shopping(){...}
}
```

组合与多态(示例二)

较好的解决方案：引入接口

```
public interface ICanShopping{   //有购物能力的类型
  Collection shopping();
}

public class Person{
  public String getName(){...}
  public Date getBirthday(){...}
  public Gender getGender(){...}
}
```

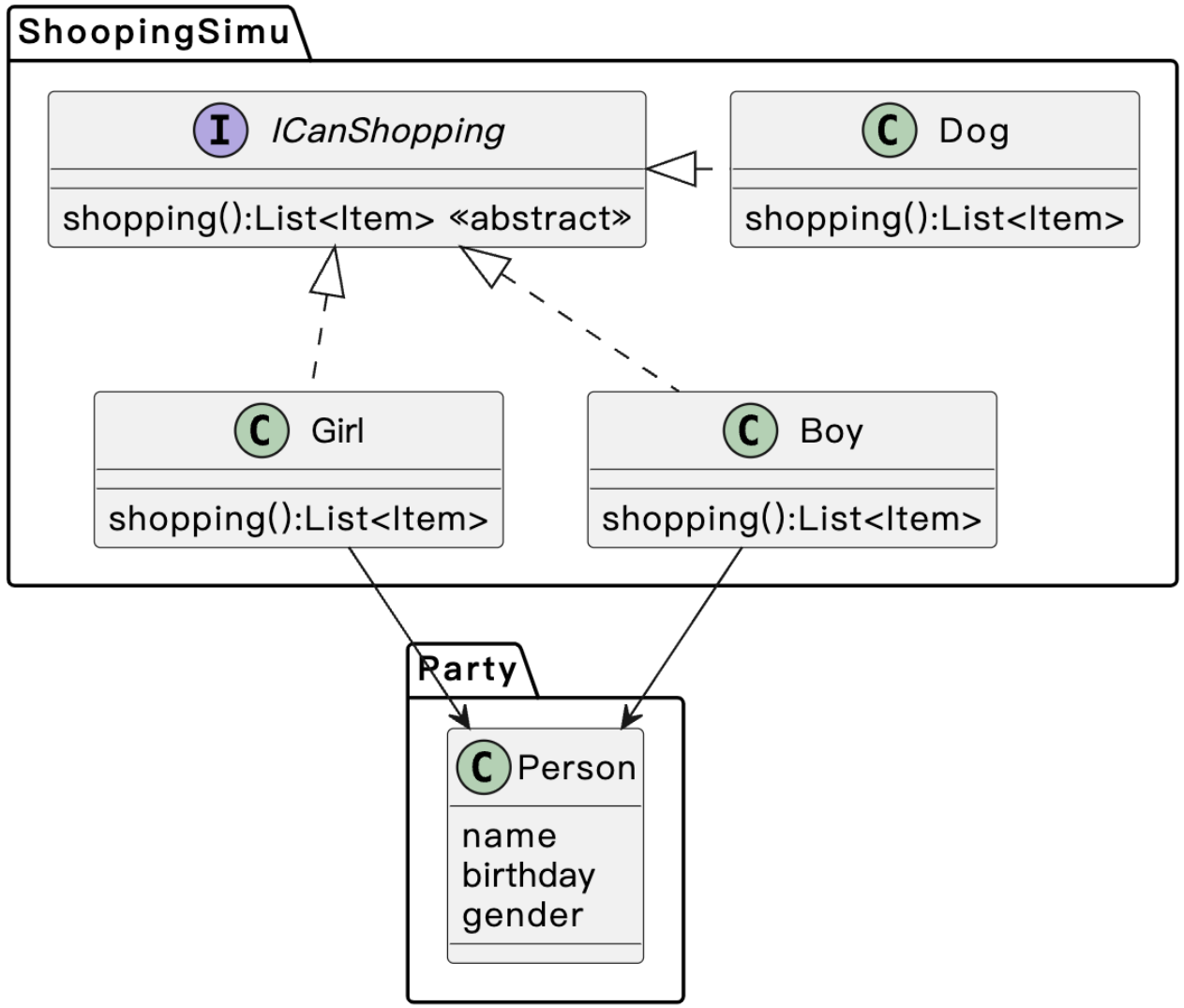
组合与多态(示例二)

```
public class Girl extends Person implements ICanShopping{
        public Gender getGender(){return Gender.femal;}
        public Collection shopping(){...}
}

public class Boy extends Person  implements ICanShopping{
         public Gender getGender(){return Gender.male;}
         public Collection shopping(){...}
}

public class Dog implements ICanShopping{
        public void wangwang(){...}
        public Collection shopping(){...}
}
```

组合与多态(示例二)


ShoopingSimu
ICanShopping
shopping():List<Item> «abstract»
Dog
shopping():List<Item>
Girl
shopping():List<Item>
Boy
shopping():List<Item>
Party
Person
name
birthday
gender