对设计模式的理解（入门）

对于开发一个软件来说，最重要的不应该是开发的过程，而是“变”的过程，也就是开发完成后的维护和扩展。为了能够在开发完成后，灵活的应对各种需求的改变，比如功能的改变以及新功能的增加等，我们应该在开发的过程中就考虑到这些，从而开发一个弹性的、可维护的、可扩展的、健壮的系统。

从头开发一个这样完善的系统是非常困难的，但是随着经验的不断积累，前人慢慢的总结出了一些合理且完善的方案，也就是我们所说的设计模式。

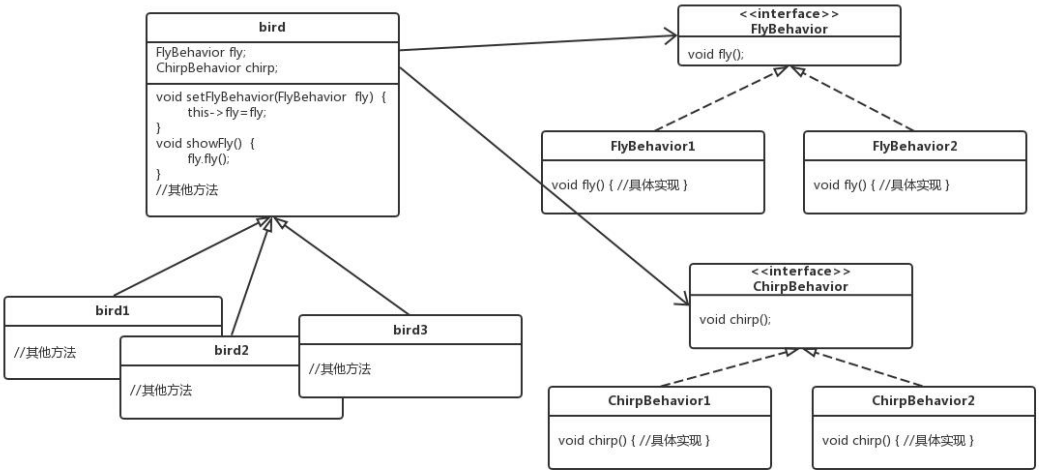
这些设计模式是前人为解决一个个开发过程中的问题而形成的较为完备的经验的总结，我们可以灵活的使用这些设计模式来使我们的系统更加合理，更加容易维护和扩展，这是一种**经验的复用**。

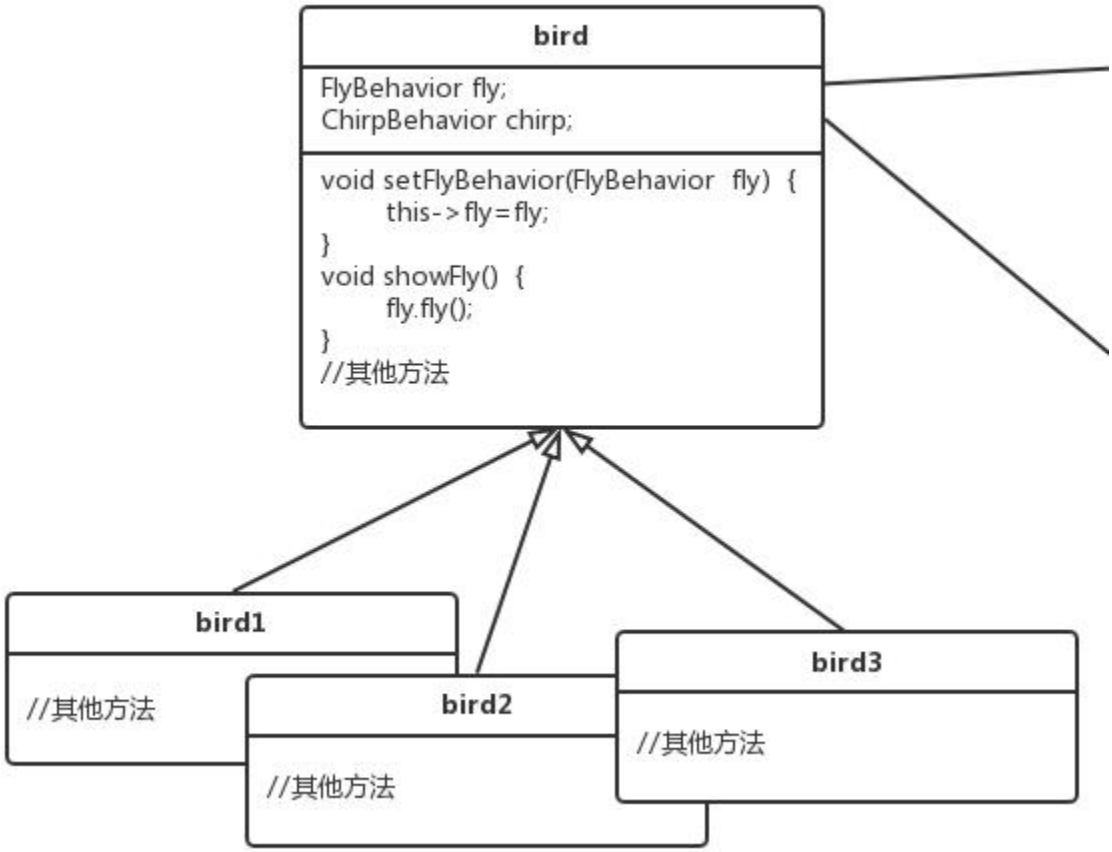
首先，我们来看下面一个例子：

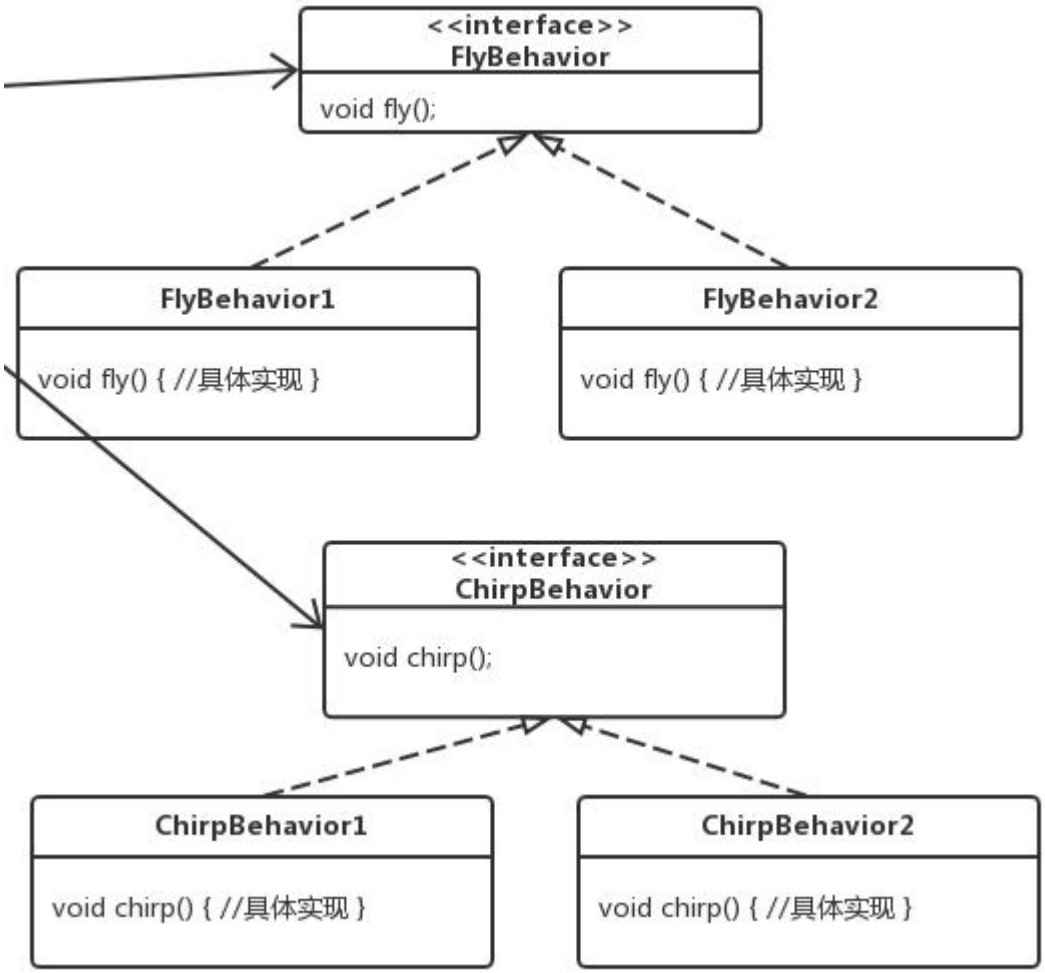
我们要做一个关于鸟类世界的小游戏，根据需要我们设计了一个鸟（bird）基类，它拥有叫（chirp()）和吃（eat()）两个成员方法，同时让鸽子、鹦鹉等类继承自它。现在，我们要让鸟能够飞行，所以在基类中加入了飞（fly()）这个成员方法，可是对于一些不能都飞行的鸟类来说，便必须覆盖此方法，而且对于不同的鸟类来说，它的叫的方式也各不相同，所以我们每增加一个子类都不得不重写基类中的很多方法，这明显不是一个合理的设计。此时我们又想，能否将飞行和叫设计为接口，只有真正需要时才让子类实现这些接口，虽然这样能够解决一部分问题，比如不能飞行的鸟类便不可能拥有 fly()这个成员方法，可是这样的话又造成代码无法复用，每个实现接口的类都必须重写该方法，即使对于一些可能实现方法相同的类来说也必须多写一遍。我们可以看出，继承和接口对于这个问题来说都不是很好的解决方案。

在某些应用中，每次修改某个行为时，我们都必须往下追踪并在每一个定义此行为的类中修改它，这样一不小心便会造成新的错误。有一个设计原则适用于此状况：**找出应用中可能需要变化之处，把它们独立出来，并加以封装，以便以后可以轻易的改动或扩充此部分，而不影响其他部分。**对于上面这个例子，bird 类中的 fly()和 chirp()应该独立出来，那么对于独立出来的行为应该如何设计呢？我们希望系统是有弹性的，对于这些行为最好能够动态改变，换句话说，对于不同的鸟类，应该在运行时动态的指定它的飞行行为。还有一个设计原则为：**针对接口编程，而不是针对实现编程。**无论我们是使用继承还是接口，其实都是在针对实现编程，我们将行为的具体实现放到基类或者子类中，这种做法使我们无法动态的改变行为。而如果我们将 fly()和 chirp()设计成接口（FlyBehavior 和 ChirpBehavior），但并不是由 bird 的子类来实现这些接口，而是提供一些其他的类来实现接口（这样的类成为“行为类”），并在 bird 类中加入 FlyBehavior 和 ChirpBehavior 类型的两个成员变量，这样我们就能动态的指定这两个变量的具体实现类，“实现”也就不会绑死在 bird 类中。这样的设计使得飞行和叫的行为可以被不同的鸟类复用，同样也可以增加一些新的飞行或叫的方式而不影响到鸟类，因为这些行为已经和鸟类无关了。这里我们用到了另一个设计原则：**多用组合，少用继承。**“有一个”要比“是一个”更好，我们让 bird 类拥有 FlyBehavior 和 ChirpBehavior 类型的两个成员变量要比 bird 类实现 FlyBehavior 和 ChirpBehavior 接口更好，因为这样的话就能够动态的指定行为。

该例子最终的设计如下：


bird
FlyBehavior fly;
ChirpBehavior chirp;
void setFlyBehavior(FlyBehavior fly) {
this->fly=fly;
}
void showFly() {
fly.fly();
}
//其他方法
<<interface>>
FlyBehavior
void fly();
FlyBehavior1
void fly() { //具体实现 }
FlyBehavior2
void fly() { //具体实现 }
<<interface>>
ChirpBehavior
void chirp();
ChirpBehavior1
void chirp() { //具体实现 }
ChirpBehavior2
void chirp() { //具体实现 }
bird1
//其他方法
bird2
//其他方法
bird3
//其他方法


bird
FlyBehavior fly;
ChirpBehavior chirp;
void setFlyBehavior(FlyBehavior fly) {
this->fly=fly;
}
void showFly() {
fly.fly();
}
//其他方法
bird1
//其他方法
bird2
//其他方法
bird3
//其他方法



在上述例子中，我们使用了**策略模式（Strategy Pattern）：定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。**策略模式让算法独立于使用它的客户而独立变化。“我们使用策略模式实现鸟的各种行为”表示：我们将鸟的各种行为封装到一组类中，并且可以轻易的扩充和改变，甚至可以在运行时改变行为。

设计模式不是库和框架，不是说让我们直接调用就好，而是解决问题的方案或经验，当然库和框架本身可能会用到设计模式。

设计模式的分类：

1、根据其目的可分为创建型(Creational)，结构型(Structural)和行为型(Behavioral)三种：

创建型模式主要用于创建对象。

结构型模式主要用于处理类或对象的组合。

行为型模式主要用于描述对类或对象怎样交互和怎样分配职责。

2、根据范围，即模式主要是用于处理类之间关系还是处理对象之间的关系，可分为类模式和对象模式两种：

类模式处理类和子类之间的关系，这些关系通过继承建立，在编译时刻就被确定下来，是属于静态的。

对象模式处理对象间的关系，这些关系在运行时刻变化，更具动态性。

3、分类图如下：

范围\目的	创建型模式	结构型模式	行为型模式
类模式	工厂方法模式	（类）适配器模式	解释器模式 模板方法模式
对象模式	抽象工厂模式 建造者模式 原型模式 单例模式	（对象）适配器模式 桥接模式 组合模式 装饰模式 外观模式 享元模式 代理模式	职责链模式 命令模式 迭代器模式 中介者模式 备忘录模式 观察者模式 状态模式 策略模式 访问者模式

OO 设计包含：OO 基础（抽象、封装、多态、继承）、OO 原则（单一职责原则、里氏替换原则、封装变化、多用组合少用继承……）、OO 模式（策略模式、单例模式……），只有合理的使用这些 OO 设计方法，我们才能设计出可维护、可扩充、可复用的系统。