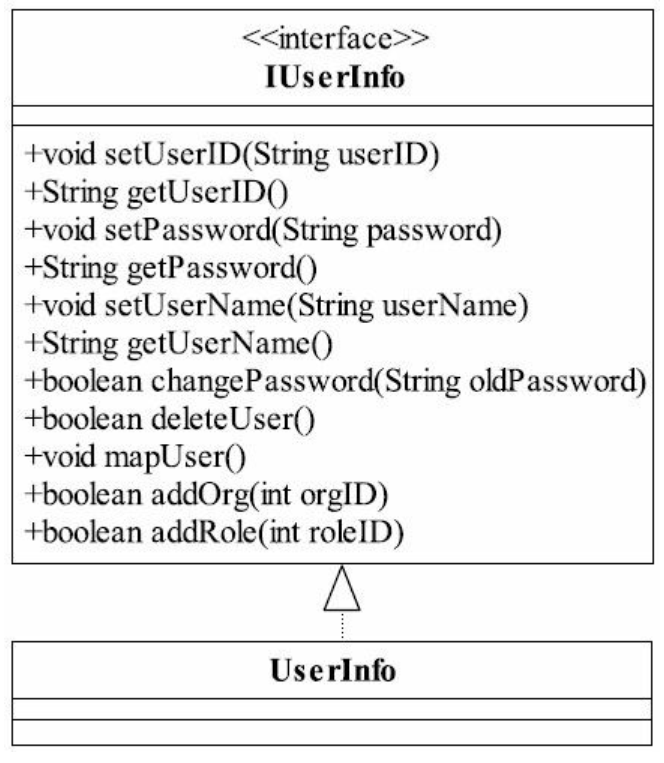
# 六大设计原则

## 单一职责原则

Single Responsibility Principle ，简称SRP**。就一个类而言，应该仅有一个引起它变化的原因。**当然对于职责的定义，什么是类的职责，以及怎么划分类的职责这几点在实际项目开发中很难确定。要灵活应对。

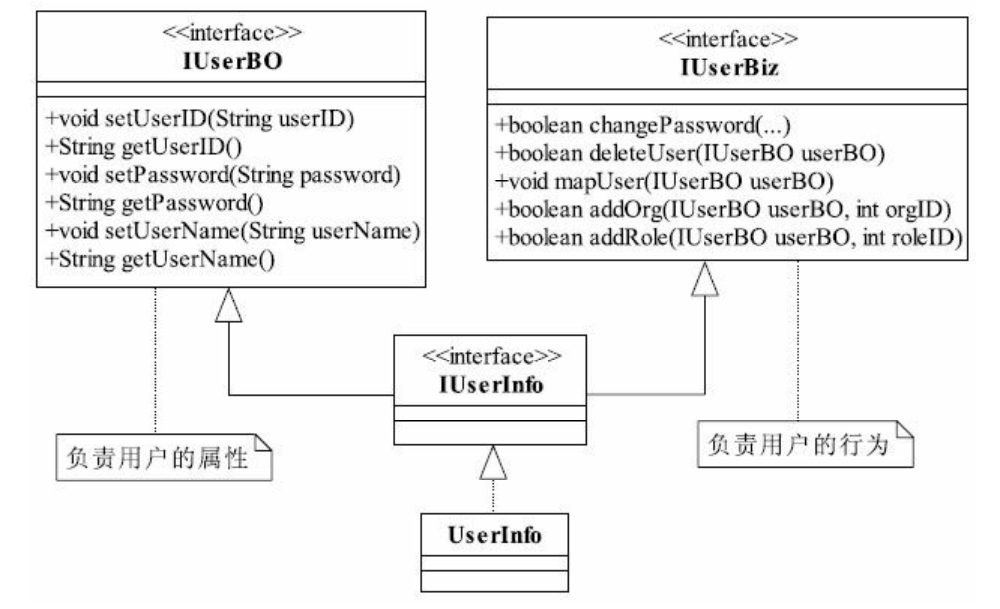
### 举例1：用户角色管理

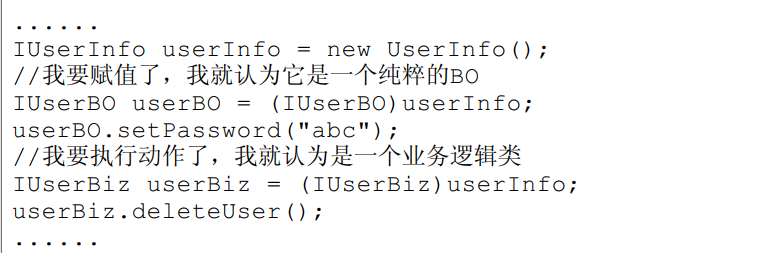


这个类设计的怎么样？

违反单一职责，**用户的属性和用户的行为没有分开，这是个严重的错误。**

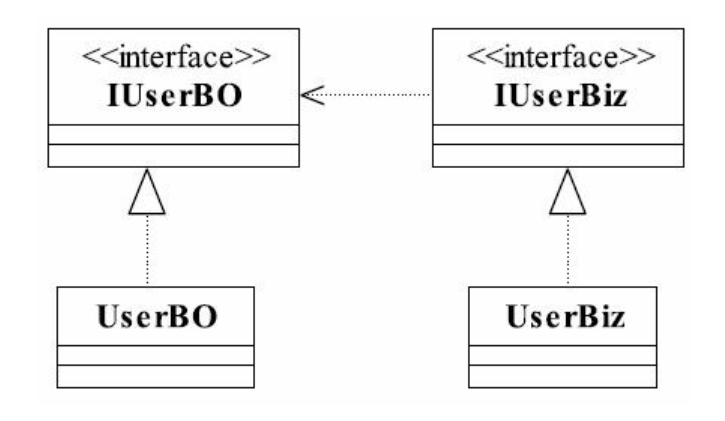
调整一下，把用户熟悉感信息抽成一个业务对象，把行为抽取成一个业务逻辑模块，修正后如下



是不是觉得有点别扭，最终的实现类还是拥有了“两类”不同的职责。再来看看使用的代码写法

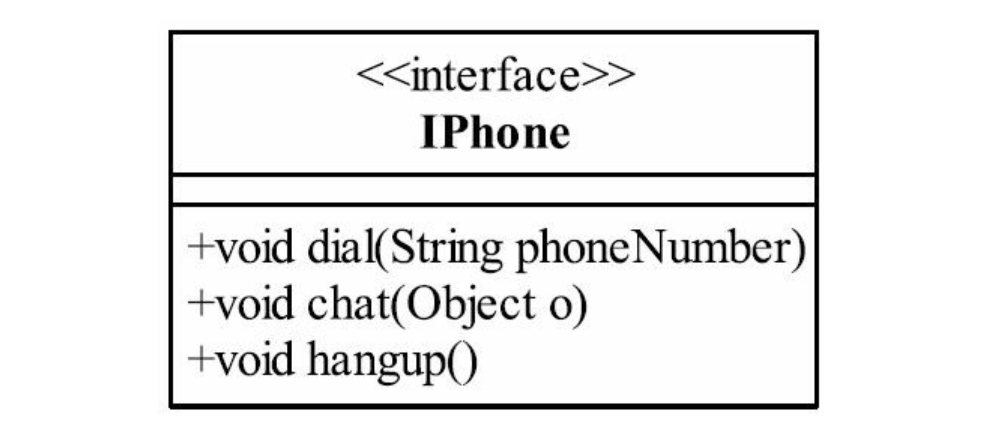
虽然接口实现确实职责分离了，但是实现类和功能写法都有问题。

**为什么要把一个接口拆分成2个，并且实现类又同时拥有了2个不同的职责？更好的处理方式是使用两个不同的类和接口。**

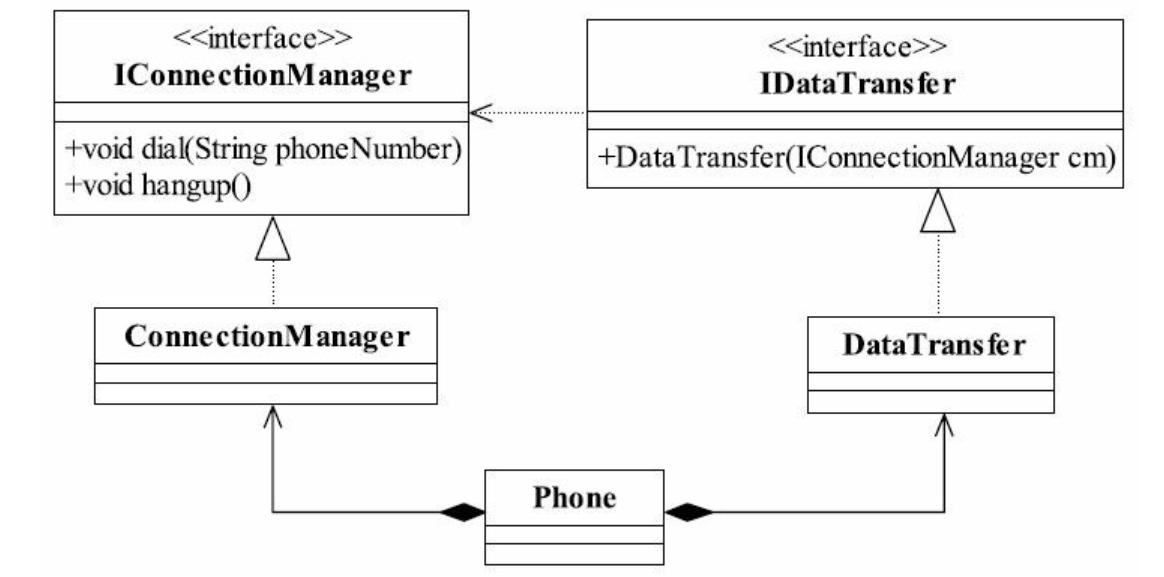


**注意行为和属性的依赖关系。**

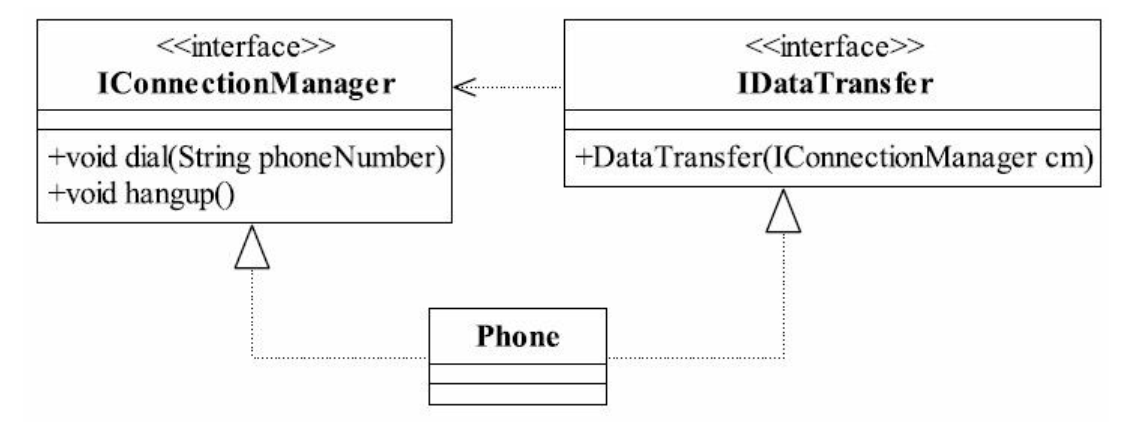
### 举例2：电话



有问题吗？从单一职责角度来讲的话，这个接口包含了拨打，挂断等的协议管理模块和通话过程中的数据传递模块。且协议模块不关心数据的内容（不关心是移动，联通还是电信）。拆分成2个接口：



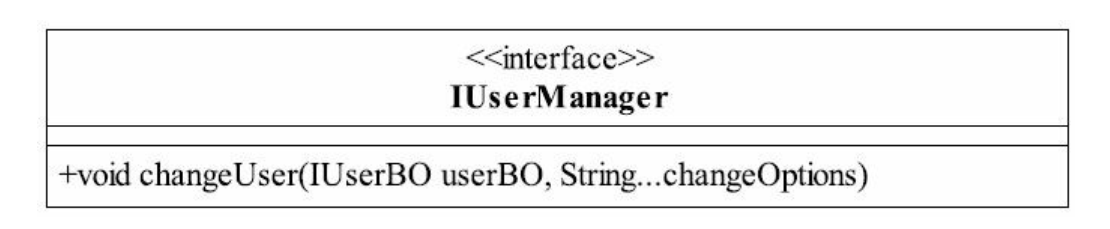
职责是明确了，但是增加了2个**实现类，并且这个两个实现类在phone中是组合到一起用的。组合是一种强耦合关系。**可以直接改成接口



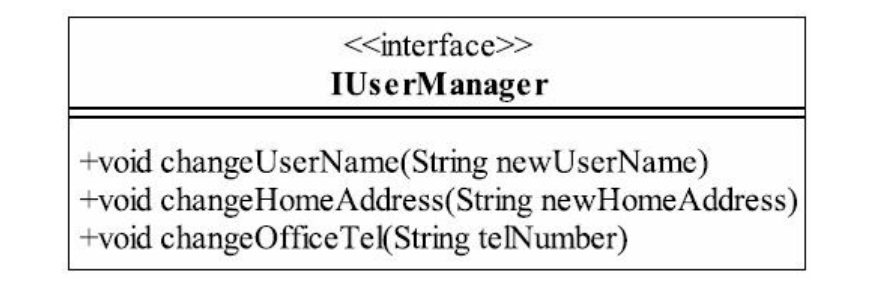
你可能会问Phone也是有两个原因一起变化呀。是的，**别忘了我们面向接口编程，对外公布的是接口而不是实现类。**如果真要实现类的单一职责，就必须用到上面的组合方式，类的数量会增加，并且耦合严重，**人为增加了设计的复杂性。**

**回到现实，单一职责提供了一个编写程序的标准，用“职责”，“变化原因”来衡量接口或类设计是否优良，但这个标准因环境和项目而异。所以最开始IPhone设计其实实际项目中并不需要修改。**

**单一职责适用于接口，类，同时也适用于方法。**一个方法尽可能做一件事情，比如修改用户密码方法，不要把这个方法放到修改用户信息方法中，这个方法颗粒度很粗。



方法职责不清晰，不单一，不要让别人猜测这个额方法可能是用来处理什么逻辑的。



### 总结：

对于单一职责，建议**接口，方法一定要做到单一职责，类的设计尽量做到只有一个原因引起变化**

类的单一职责的好处：  
1，复杂性降低，职责清晰明了

2，可读性提高，复杂性降低，可读性就提高了

3，可维护性提高，可读性降低，维护更容易

4，变更引起的风险降低

举例3：ImageLoader

1. */\*\**
2. \* 图片加载类
3. \*/
4. public class ImageLoader {
5. LruCache<String, Bitmap> mImageCache;
6. ExecutorService mExecutorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
7. Handler mUiHandler = new Handler(Looper.getMainLooper()) {
8. @Override
9. public void handleMessage(@NonNull Message msg) {
10. super.handleMessage(msg);
11. }
12. };
13. public ImageLoader() {
14. initImageCache();
15. }
16. private void initImageCache() {
17. *//j计算可使用的最大内存*
18. final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
19. final int cache = maxMemory / 4;
20. mImageCache = new LruCache<String, Bitmap>(cache) {
21. @Override
22. protected int sizeOf(String key, Bitmap value) {
23. return value.getRowBytes() \* value.getHeight() / 1024;
24. }
25. };
26. }
27. public void disPlayImage(final String url, final ImageView imageView) {
28. imageView.setTag(url);
29. mExecutorService.submit(new Runnable() {
30. @Override
31. public void run() {
32. Bitmap bitmap = downLoadImage(url);
33. if (bitmap == null) {
34. return;
35. }
36. if (imageView.getTag().equals(url)) {
37. updateImageView(imageView, bitmap);
38. }
39. mImageCache.put(url, bitmap);
40. }
41. });
42. }
43. private void updateImageView(final ImageView imageView, final Bitmap bmp) {
44. mUiHandler.post(new Runnable() {
45. @Override
46. public void run() {
47. imageView.setImageBitmap(bmp);
48. }
49. });
50. }
51. public Bitmap downLoadImage(String url) {
52. Bitmap bitmap = null;
53. try {
54. URL bitmapUrl = new URL(url);
55. HttpURLConnection connection = (HttpURLConnection) bitmapUrl.openConnection();
56. bitmap = BitmapFactory.decodeStream(connection.getInputStream());
57. connection.disconnect();
58. } catch (Exception e) {
59. e.printStackTrace();
60. }
61. return bitmap;
62. }
63. }

**问题：所有功能都放在一个类，极其不利于修改，扩展！**

1. */\*\**
2. \* 图片加载类
3. \*/
4. public class ImageLoader {
5. private ImageCache imageCache = new ImageCache();
6. private ExecutorService mExecutorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
7. private Handler mUiHandler = new Handler(Looper.getMainLooper()) {
8. @Override
9. public void handleMessage(@NonNull Message msg) {
10. super.handleMessage(msg);
11. }
12. };
13. public ImageLoader() {
14. }
15. public void disPlayImage(final String url, final ImageView imageView) {
16. Bitmap bitmap = imageCache.get(url);
17. if (bitmap != null) {
18. imageView.setImageBitmap(bitmap);
19. return;
20. }
21. imageView.setTag(url);
22. mExecutorService.submit(new Runnable() {
23. @Override
24. public void run() {
25. Bitmap bitmap = downLoadImage(url);
26. if (bitmap == null) {
27. return;
28. }
29. if (imageView.getTag().equals(url)) {
30. updateImageView(imageView, bitmap);
31. }
32. imageCache.put(url, bitmap);
33. }
34. });
35. }
36. private void updateImageView(final ImageView imageView, final Bitmap bmp) {
37. mUiHandler.post(new Runnable() {
38. @Override
39. public void run() {
40. imageView.setImageBitmap(bmp);
41. }
42. });
43. }
44. public Bitmap downLoadImage(String url) {
45. Bitmap bitmap = null;
46. try {
47. URL bitmapUrl = new URL(url);
48. HttpURLConnection connection = (HttpURLConnection) bitmapUrl.openConnection();
49. bitmap = BitmapFactory.decodeStream(connection.getInputStream());
50. connection.disconnect();
51. } catch (Exception e) {
52. e.printStackTrace();
53. }
54. return bitmap;
55. }
56. }
58. public class ImageCache {
59. private LruCache<String, Bitmap> mImageCache;
60. public ImageCache() {
61. initImageCache();
62. }
63. private void initImageCache() {
64. *//j计算可使用的最大内存*
65. final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
66. final int cache = maxMemory / 4;
67. mImageCache = new LruCache<String, Bitmap>(cache) {
68. @Override
69. protected int sizeOf(String key, Bitmap value) {
70. return value.getRowBytes() \* value.getHeight() / 1024;
71. }
72. };
73. }
74. public void put(String url, Bitmap bitmap) {
75. mImageCache.put(url, bitmap);
76. }
77. public Bitmap get(String url) {
78. return mImageCache.get(url);
79. }
80. }

V2：将缓存拆出来，单独成类

## 里氏替换原则

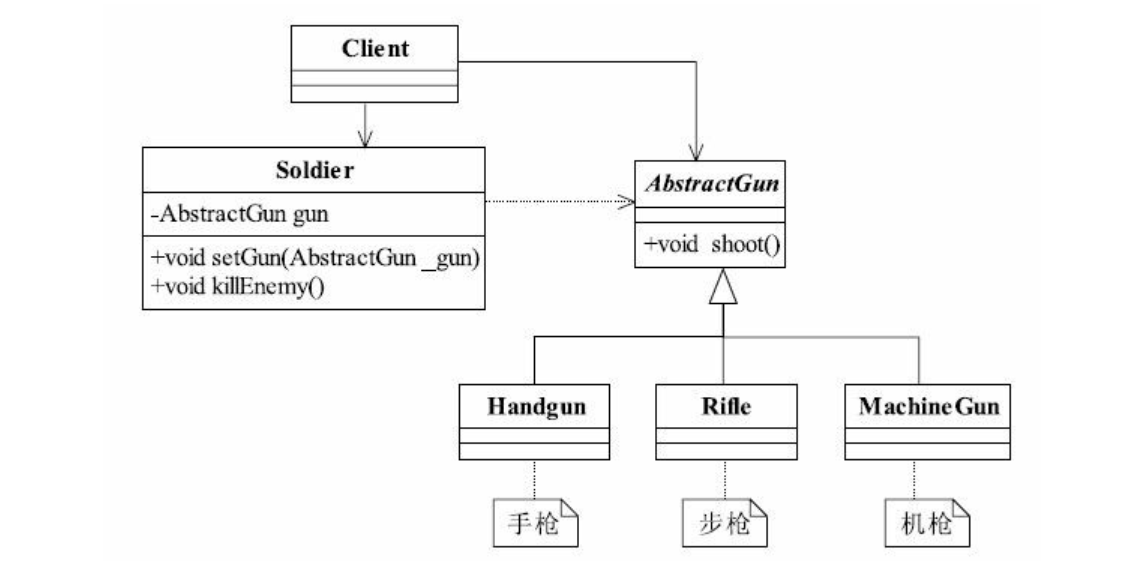
所有引用基类的地方必须能透明的使用其子类的对象。

里氏替换原则为良好的继承定义了一个规范，包含4曾含义：

### （1）子类必须完全实现父类的方法

系统设计时，经常会定义一个接口或抽象类，然后编码实现。其实这里使用了里氏替换。**在类中调用其他类时无比使用父类或接口，如果不能使用父类或接口，则说明设计已经违背了LSP**

举例：设计游戏中的枪



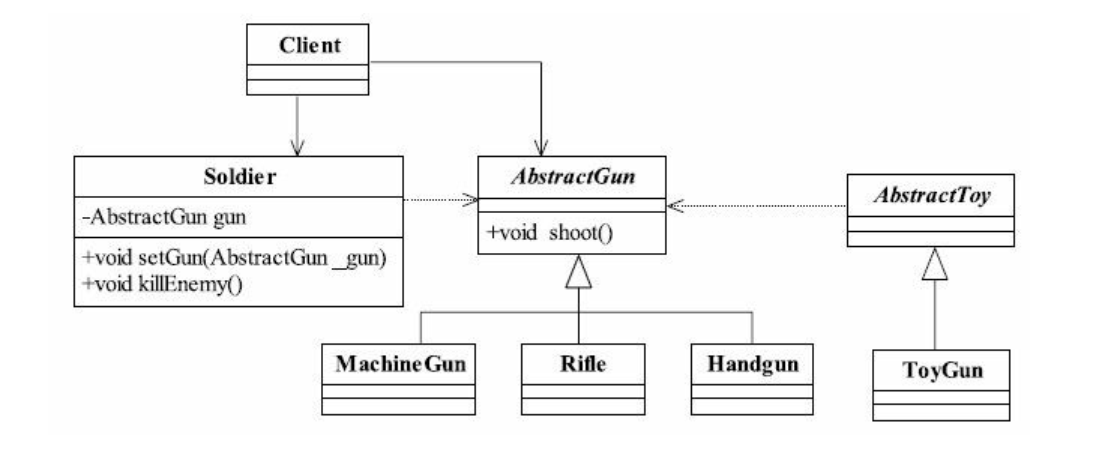
手枪，步枪，机枪都能实现射击操作。这几种枪

需求扩展：添加一个玩具枪

玩具枪不具备射击功能怎么办？有两种解决方法：第一，在Soldier类中增加instanceof的判断，如果时玩具枪则不能用来射击。

但是这种方法每增加一个类就要新增一下判断，非常不可取

第二，玩具枪脱离继承，创建一个独立的父类，为了代码复用，可以与真枪的抽象类建立委托关系



按照继承原则，玩具枪继承AbstractGun是绝对没有问题的，玩具枪也是枪嘛。但是具体应用场景中要考虑：**子类是否能够完整的实现父类的业务。如果不能，或者父类的某些方法在子类中已经发生“畸变”，则建议断开继承关系，采用依赖，聚集，组合灯关系代替继承。**

### （2）子类可以有自己的个性

一句话：子类能用的地方，父类不一定能用

### （3）覆盖或实现父类的方法时输入参数可以被放大

1. public class Father {
2. public Collection doSomeThing(HashMap map){
3. System.out.println("父类被执行");
4. return map.values();
5. }
6. }
7. public class Son extends Father {
8. *//放大输入类型*
9. public Collection doSomeThing(Map map){
10. System.out.println("子类被执行");
11. return map.values();
12. }
13. }
14. public class Test {
15. public static void main(String[] args) {
16. HashMap hashMap = new HashMap();
17. Father father = new Father();
18. father.doSomeThing(hashMap);
19. Son son = new Son();
20. son.doSomeThing(hashMap);
21. }
22. }



子类方法不能用override，会抱错。因为方法参数不一样。这其实时重载，因为子类继承了父类的方法。

**两次都时执行了父类的方法，这是对的！因为没有重载的情况下就应该执行父类的方法，这样才是里氏替换，良好的代码规范。**

上面的例子子类的方法参数范围扩大了父类的参数范围，如果反过来呢？

1. public class Father {
2. public Collection doSomeThing(Map map){
3. System.out.println("父类被执行");
4. return map.values();
5. }
6. }
7. public class Son extends Father {
8. *//缩小输入类型*
9. public Collection doSomeThing(HashMap map){
10. System.out.println("子类被执行");
11. return map.values();
12. }
13. }
14. public class Test {
15. public static void main(String[] args) {
16. HashMap hashMap = new HashMap();
17. Father father = new Father();
18. father.doSomeThing(hashMap);
19. Son son = new Son();
20. son.doSomeThing(hashMap);
21. }
22. }

这种情况下override也是会报错的。执行结果为：  


子类缩小了前置条件（方法参数范围），**子类在没有覆盖父类的前提下执行了子类的方法，这会引起业务逻辑混乱，这样的代码设计歪曲了父类的意图，不符合代码设计规范。所以子类中方法的前置条件必须与超类被覆写的方法的前置条件相同或者更宽松**

### （4）覆写或实现父类的方法时输出结果可以相同或者被缩小

### 总结：

采用里氏替换时，尽量避免子类的“个性”，一旦子类有“个性”，这个子类或父类的关系就难调和。把子类当作父类用，子类的个性被抹杀，失去了意义。把子类当作一个业务来使用，则会让代码间的耦合关系变得扑簌迷离，缺乏类替换的标准。

优点：

（1）代码宠用，减少创建类成本，每个子类自动拥有父类属性和方法

（2）子类和父类基本相似，又可以有所区别

（3）提高代码可扩展性

缺点：

（1）继承时侵入式的，只要继承就必须拥有父类属性和方法

（2）可能造成子类代码冗余，灵活性降低

（3）父类修改的时候，所有子类都要修改，成本大

## 依赖倒置原则

1，高层模块不应该依赖低层模块

2，抽象不应该依赖细节

3，细节应该依赖抽象

在Java中，抽象就是接口或抽象类；细节就是实现类。依赖倒置原则在Java语言中表现为

1，模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系时通过接口或抽象类产生的

2，接口或抽象类不依赖于实现类

3，实现类依赖接口或抽象类

**更精简的第一就是面向对象的精髓之一：面向接口编程**

举例：Imageloder 库缓存应该使用接口，而不是直接写实现类，否则想替换缓存类型的时候都要涉及到imageloder

## 接口隔离原则（ISP）

客户端不应该依赖它不需要的接口。另一种定义：类间的依赖关系应该建立在最小的接口上。

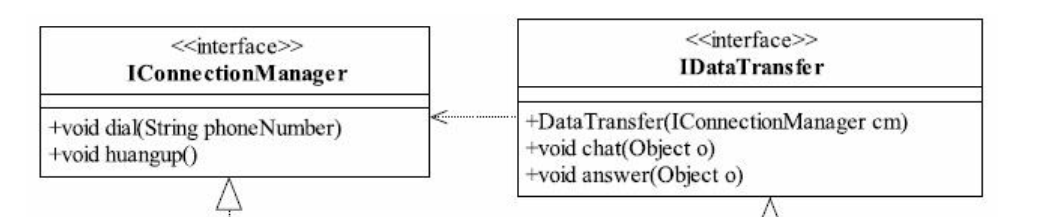
**通过分散定义多个接口，可以预防未来变更的扩散，提供系统的灵活性和可维护性**

接口隔离原色对接口进行规范约束，包含4层含义：

### 1，接口要尽量小

**一个接口只服务于一个子模块或业务逻辑**

不出现臃肿的接口，但是小是有限度的，首**先就是不违反单一职责原则。**



挂断电话接口如果要符合尽量小的原则的话，可以分为正常挂断，异常挂断。如果要这样就得把Iconnection拆分成拨打和挂断两个接口。再细分还可以对业务和协议拆分，是否要考虑3G协议，中继服务器等。拆还是不拆呢？答案是不拆，因为业务逻辑来讲，通信的建立和关闭已经是最小的业务单位了，如果拆分了就不符合单一职责原则了！

### 2，接口要高内聚

提高接口，类，模块的处理能力，减少对外的交互。在接口中尽量少公布public方法，接口时对外的承诺，承诺越少对系统的开发越有利，变更风险越小。

### 3，定制服务

系统内的模块必然存在交互，耦合。之间可能是通过接口调用或者单纯的数据交换。针对不同的调用方要针对性服务，绝不能为了省事暴露给他们不需要的接口。

### 4，接口设计时有限度的

越小越灵活，但是越小也带来了结构的复杂化。所以度要斟酌，要结合实际业务，环境做不同标准

举例：inputstream，outputstream的关闭

1. public class CloseUtil {
2. public static void closeQuietly(Closeable closeable){
3. if(null!=closeable){
4. try {
5. closeable.close();
6. } catch (IOException e) {
7. e.printStackTrace();
8. }
9. }
10. }
11. }

关闭的时候只需要确定支持close接口的就行，不用关心write，read等方法。这也是某种意义上的接口隔离

## 迪米特法则

也称最小知识原则。一个对象应该对其他对象有最少的了解。通俗讲：一个类应该对自己需要耦合或者调用的类知道的最少。包含4层含义：

### 1，只和朋友交流

**只与直接的朋友通信！**

举例：老师要求清点女生

1. public class GroupLeader {
2. *//  清点数量*
3. public void countGirls(List<Girl> girlList){
4. System.out.println("女神数量为："+girlList.size());
5. }
6. }
7. public class Teacher {
8. *//老师发布清点v女生命令*
9. public void commond(GroupLeader leader){
10. List<Girl> girls = new ArrayList<>();
11. for(int i=0;i<20;i++){
12. girls.add(new Girl());
13. }
14. leader.countGirls(girls);
15. }
16. }
17. public class Test {
18. public static void main(String[] args) {
19. Teacher teacher = new Teacher();
20. GroupLeader leader = new GroupLeader();
21. teacher.commond(leader);
22. }
23. }

看出来有什么问题吗？Teacher类有几个朋友类，一个：GroupLeader。为什么girls不是呢，Tearcher也对它产生了依赖。**出现在成员变量，方法输入输出参数的类称为成员朋友，而出现在方法体内部的类不是朋友类。**方法是类的一个行为，类竟然不知道自己的类和其他类产生了依赖关系，这是不允许的，严重违反了迪米特法则。

1. public class GroupLeader {
2. private List<Girl> girls;
3. public GroupLeader(List<Girl> girlList) {
4. girls = girlList;
5. }
6. *//  清点数量*
7. public void countGirls() {
8. System.out.println("女神数量为：" + girls.size());
9. }
10. }
11. public class Teacher {
12. *//老师发布清点v女生命令*
13. public void commond(GroupLeader leader){
14. leader.countGirls();
15. }
16. }
17. public class Test {
18. public static void main(String[] args) {
19. Teacher teacher = new Teacher();
20. List<Girl> girls = new ArrayList<>();
21. for (int i = 0; i < 20; i++) {
22. girls.add(new Girl());
23. }
24. GroupLeader leader = new GroupLeader(girls);
25. teacher.commond(leader);
26. }
27. }

将Teacher中对List的初始化移到场景类中，同时在GroupLeader中增加了对Girl的注入，避开了Teacher类对list girl的耦合，提供了系统的健壮性

### 2，朋友之间也是有距离的

刺猬取暖，太远取不到暖，太近就相互刺伤。

1. public class Wizard {
2. private Random rand = new Random(System.currentTimeMillis());
3. public int first() {
4. System.out.println("执行第一步");
5. return rand.nextInt(100);
6. }
7. public int second() {
8. System.out.println("执行第二步");
9. return rand.nextInt(100);
10. }
11. public int third() {
12. System.out.println("执行第三步");
13. return rand.nextInt(100);
14. }
15. }
16. public class InstallSoftware {
17. public void installWizard(Wizard wizard) {
18. int first = wizard.first();
19. if (first > 50) {
20. int second = wizard.second();
21. if (second > 50) {
22. wizard.third();
23. }
24. }
25. }
26. }
27. public class Test {
28. public static void main(String[] args) {
29. InstallSoftware invoke = new InstallSoftware();
30. invoke.installWizard(new Wizard());
31. }
32. }

有什么问题？

Wizard把太多方法暴露给InstallSoftware类了，如果Wizard中间安装的逻辑改了，那么InstallSoftware也得跟着改。

修改方法：把public方法改为private，具体的安装细节交给Wizard

1. public class Wizard {
2. private Random rand = new Random(System.currentTimeMillis());
3. private int first() {
4. System.out.println("执行第一步");
5. return rand.nextInt(100);
6. }
7. private int second() {
8. System.out.println("执行第二步");
9. return rand.nextInt(100);
10. }
11. private int third() {
12. System.out.println("执行第三步");
13. return rand.nextInt(100);
14. }
15. public void installWizard() {
16. int first = first();
17. if (first > 50) {
18. int second = second();
19. if (second > 50) {
20. third();
21. }
22. }
23. }
24. }
25. public class InstallSoftware {
26. public void installWizard(Wizard wizard) {
27. wizard.installWizard();
28. }
29. }

**迪米特法则要求尽量不要对外公布太多的public方法和静态public变量，尽量内敛，多使用private，protected，final关键字。**

### 3，是自己的就是自己的

如何决定方法放在这个类还是哪个类？**如果一个方法放在本类中，既不增加类间关系，也对本类不产生负面影响，那就放在本类中。**

### 4，谨慎使用Serializable

Private 改为public 容易导致客户端和服务端反序列化异常

### 总结

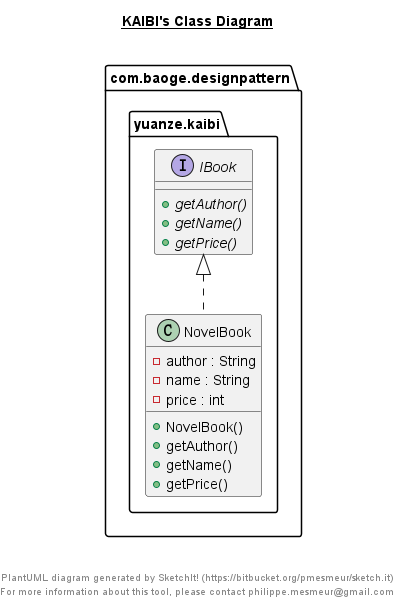
迪米特法则核心观念就是类间解耦，弱耦合，只有弱耦合类的复用率才可以提高。其要求的结果就是产生了大量的中转和跳转类，导致复杂性提高，维护困难。所以要反复权衡，及做到结构清晰，又做到高内聚低耦合。纪要减少暴露又要避免“路径”过长。

## 开闭原则

**对扩展开放，对修改关闭！**

**面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。**

举例：买书



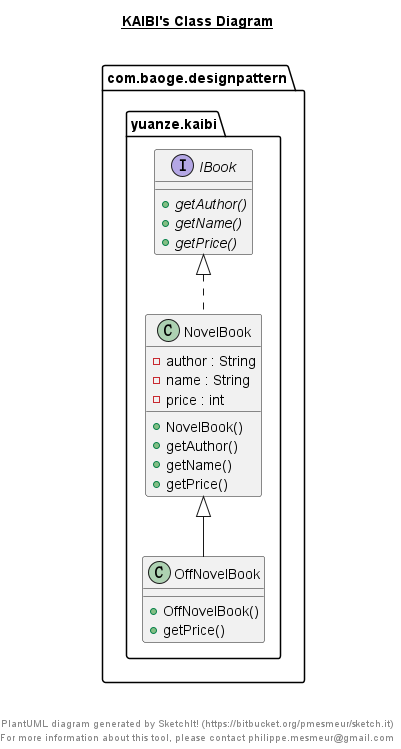
现在发生需求变更，大于50的书打8折，其他打9折

方法一：在IBook接口上增加方法getOffPrice()。这种方案的后果是所有实现类都要变，调用方也得改。同时IBook作为接口应该是稳定且可靠的，不应该经常变动，否则就失去了**契约的效能**

方法二：修改实现类，在实现类getprice方法中直接做价格打折逻辑。这是常见的方法，某些情况下这种方案是不错的方法，但是这里某些管理员是需要知道原价的，所以不能直接改

方案三：**通过扩展实现变化**

增加OffNovelBook打折类继承NovelBook，然后覆盖getprice方法，不同的角色调用不同的类



### 为什么要采用开闭原则

**1，开闭原则对测试的影响**

保持之前的代码不动，保证代码质量

**2，开闭原则可以提高复用性**

**3，可以提高可维护性**

**4，面向对象开发的要求**

### 如何使用开闭原则

**1，抽象约束**

通过接口或抽象类可以约束一组可能变化的行为，并且能够实现对扩展开放。包含3层含义：

（1）通过接口或者抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的public方法；

（2）类型参数，引用对象尽量使用接口或抽象类，而不是实现类；

（3）抽象层尽量保持稳定，一旦确定即不允许修改

**2，元数据控制模块行为**

元数据：用来描述环境和数据的数据。通俗的说就是配置参数，通过参数的修改实现业务的调整

**3，执行项目章程**

规则统一，风格统一

**4，封装变化**

将相同的变化封装到一个接口或抽象类中；将不同的变化封装到不同的接口或抽象类中

# 设计模式

单例模式：某个类只能有一个实例，提供一个全局的访问点。

简单工厂：一个工厂类根据传入的参量决定创建出那一种产品类的实例。

工厂方法：定义一个创建对象的接口，让子类决定实例化那个类。

抽象工厂：创建相关或依赖对象的家族，而无需明确指定具体类。

建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造。

原型模式：通过复制现有的实例来创建新的实例。

适配器模式：将一个类的方法接口转换成客户希望的另外一个接口。

组合模式：将对象组合成树形结构以表示“”部分-整体“”的层次结构。

装饰模式：动态的给对象添加新的功能。

代理模式：为其他对象提供一个代理以便控制这个对象的访问。

亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。

外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。

桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。

解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。

策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

观察者模式：对象间的一对多的依赖关系。

备忘录模式：在不破坏封装的前提下，保持对象的内部状态。

中介者模式：用一个中介对象来封装一系列的对象交互。

命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

访问者模式：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。

责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

## 创建型设计模式

单例模式，原型模式，工厂方法模式，抽象工厂模式，建造者模式

### 单例模式

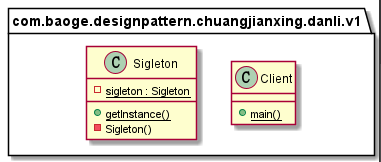
#### 定义：

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

#### 使用场景

要求只有一个对象的场景

#### UML类图



#### V1：恶汉模式

1. public class Sigleton {
2. private static Sigleton instance = new Sigleton();
3. private Sigleton() {
4. }
5. public static Sigleton getInstance() {
6. return instance;
7. }
8. }

优点：在类加载的时候就实例化了对象，类加载只会有一次，所以线程安全

缺点：没有用的时候就实例化了，浪费。

#### 懒汉模式

1. public class Sigleton {
2. private static Sigleton instance;
3. private Sigleton(){}
4. public static Sigleton getInstance(){
5. if(instance == null){
6. instance = new Sigleton();
7. }
8. return instance;
9. }
10. }

优点：使用的时候才会创建  
缺点：线程不安全

#### V2，加锁

1. public class Sigleton {
2. private static Sigleton instance;
3. private Sigleton(){}
4. public static synchronized Sigleton getInstance(){
5. if(instance == null){
6. instance = new Sigleton();
7. }
8. return instance;
9. }
10. }

getInstance方法上加锁，可以保证单例安全性。  
优点：第一次使用的时候才会初始化。

缺点：每次调用这个方法都会进行同步开销，造成不必要的开销。

#### 再优化->Double Check Lock (DCL) 双重检查锁定-双重锁懒汉模式

1. public class Sigleton {
2. private static Sigleton instance;
3. private Sigleton(){}
4. public static  Sigleton getInstance(){
5. if(instance == null){
6. synchronized (Sigleton.class) {
7. if(instance == null) {
8. instance = new Sigleton();
9. }
10. }
11. }
12. return instance;
13. }
14. }

进行了两次判空，**第一次判空避免了不必要的同步，第二次判断则是为了在null情况下创建实例。**这就万无一失了吗？

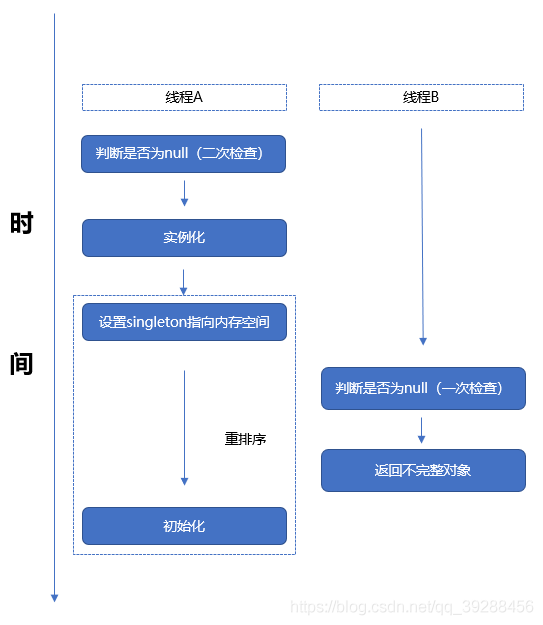
Instance = new Sigleton();最终会被编译成多条汇编指令，大概做3件事情：

1. 给Sigleton的实例分配内存；
2. 调用Sigleton构造函数
3. 将instance对象指向分配的内存空间（此时instance就不是null了）

对象初始化初略分为2个部分：实例化和初始化。实例化指系统为对象分配内存；初始化指实例化后的赋值操作。

**对象的初始化不是一个原子操作，复合操作，有发生重排序的可能。同步块能够保证多个线程有序的（即同步）执行同步块中的代码，但并不能避免块中代码发生重排序。**

上面执行环节有可能是 1-2-3 ，也有可能是1-3-2。所以当A线程按照1-3-2执行到第二次检查步骤3的时候，线程B执行第一次检查的时候发现不为null，返回。但是这个对象地址不为空，**但是没有经过构造函数，是一个不完整的对象。**



Volatile变量是java语言提供的一种稍弱的同步机制。有两个重要的特点：不会被重排序；不会被缓存到工作线程的本地内存。基于这两点，变量增加volatile关键字，就能避免DCL失效问题

1. public class Sigleton {
2. private volatile static Sigleton instance;
3. private Sigleton(){}
4. public static Sigleton getInstance(){
5. if(instance == null){
6. synchronized (Sigleton.class) {
7. if(instance == null) {
8. instance = new Sigleton();
9. }
10. }
11. }
12. return instance;
13. }
14. }

优点：资源利用率高，第一次执行才会初始化

缺点：第一次加载反应稍慢，volatile关键字稍微影响一些性能

#### 静态内部类

1. public class Sigleton {
2. private Sigleton() {
3. }
4. public static Sigleton getInstance() {
5. return SingleonHolder.instance;
6. }
7. private static class SingleonHolder {
8. private static final Sigleton instance = new Sigleton();
9. }
10. }

**静态内部类避讳随着外部类的加载而加载。所以：**

第一次加载Singleton类时并不会初始化instance，只有调用getInstance方法虚拟机才会加载SingleonHolder 类，执行构造函数初始化。线程安全&实例化延迟，推荐。

#### 枚举类型

1. public enum SingletonEnum {
2. INSTANCE;
3. public void doSomting(){}
4. }
5. *//调用方式*
6. SingletonEnum.INSTANCE.doSomting();

默认枚举实例的创建是线程安全的，并且任何情况下都是一个单例。实际上：  
**枚举类隐藏了构造器**

https://blog.csdn.net/ningmengshuxiawo/article/details/115594979

<https://blog.csdn.net/qq_43778308/article/details/110873569>

枚举：Y隐藏构造，怎么防止放射，克隆，序列化？

#### 破坏单例模式的三种方式

1，反射

解决方案：定义一个全局变量，当第二次创建时抛出异常

2，克隆

重写clone，直接返回单例对象

3，序列化

添加readRFesolve()，返回object对象

#### 枚举单例是怎么解决的？

**枚举单例不仅能避免多线程问题，也可以防止放序列化和反射的破坏。**

通过反编译枚举单例，发现  
1，枚举中各个枚举项都是通过static代码块来定义和初始化的，他们会在类加载时完成初始化，所以线程安全

2，防止反序列化的破坏

Java的序列化对枚举做了规定，序列化时，只是将枚举对象的name属性输出到结果中；反序列化时通过java.lang.Enum的valueOf方法根据名字查找对象，而不是新建一个新的对象，所以防止了序列化对单例的破坏。

1. private Enum<?> readEnum(boolean unshared) throws IOException {
2. if (this.bin.readByte() != 126) {
3. throw new InternalError();
4. } else {
5. ObjectStreamClass desc = this.readClassDesc(false);
6. if (!desc.isEnum()) {
7. throw new InvalidClassException("non-enum class: " + desc);
8. } else {
9. int enumHandle = this.handles.assign(unshared ? unsharedMarker : null);
10. ClassNotFoundException resolveEx = desc.getResolveException();
11. if (resolveEx != null) {
12. this.handles.markException(enumHandle, resolveEx);
13. }
14. String name = this.readString(false);
15. Enum<?> result = null;
16. Class<?> cl = desc.forClass();
17. if (cl != null) {
18. try {
19. Enum<?> en = Enum.valueOf(cl, name);
20. result = en;
21. } catch (IllegalArgumentException var9) {
22. throw (IOException)(new InvalidObjectException("enum constant " + name + " does not exist in " + cl)).initCause(var9);
23. }
24. if (!unshared) {
25. this.handles.setObject(enumHandle, result);
26. }
27. }
28. this.handles.finish(enumHandle);
29. this.passHandle = enumHandle;
30. return result;
31. }
32. }
33. }

3，防止反射的破坏

反射在通过newInstance创建对象时会检查这个类是否是枚举类，如果是枚举类，就会抛出异常：

1. public T newInstance(Object... initargs) throws InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {
2. if (!this.override) {
3. Class<?> caller = Reflection.getCallerClass();
4. this.checkAccess(caller, this.clazz, this.clazz, this.modifiers);
5. }
6. if ((this.clazz.getModifiers() & 16384) != 0) {
7. throw new IllegalArgumentException("Cannot reflectively create enum objects");
8. } else {
9. ConstructorAccessor ca = this.constructorAccessor;
10. if (ca == null) {
11. ca = this.acquireConstructorAccessor();
12. }
13. T inst = ca.newInstance(initargs);
14. return inst;
15. }
16. }

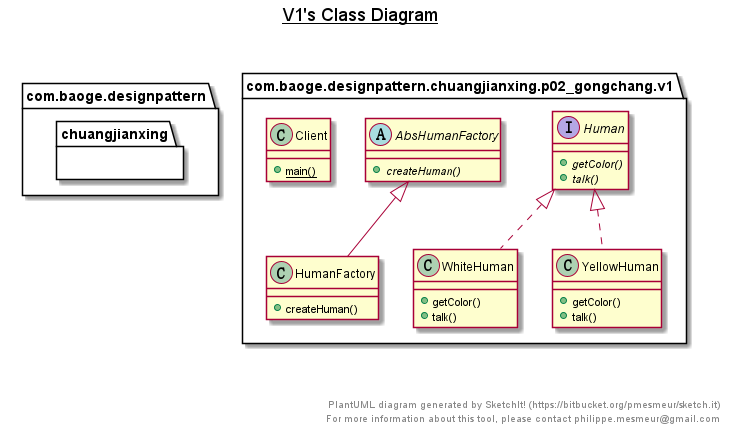
### 工厂方法模式

#### 定义

定义一个用于创建对象的接口，让子类决定实例化哪个类。工厂方法时一个类的实例化延迟到其子类。

#### UML类图

1. public interface Human {
2. public void getColor();
3. public void talk();
4. }
5. public class WhiteHuman implements Human {
6. @Override
7. public void getColor() {
8. System.out.println("白色");
9. }
10. @Override
11. public void talk() {
12. System.out.println("白说");
13. }
14. }
15. public class YellowHuman implements Human {
16. @Override
17. public void getColor() {
18. System.out.println("黄色");
19. }
20. @Override
21. public void talk() {
22. System.out.println("黄说");
23. }
24. }
25. public abstract class AbsHumanFactory {
26. public abstract <T extends Human> T createHuman(Class<T> tClass);
27. }
28. public class HumanFactory extends AbsHumanFactory {
29. @Override
30. public <T extends Human> T createHuman(Class<T> tClass) {
31. Human human = null;
32. try {
33. human = (T) Class.forName(tClass.getName()).newInstance();
34. } catch (Exception e) {
35. e.printStackTrace();
36. }
37. return (T) human;
38. }
39. }
40. public class Client {
41. public static void main(String[] args) {
42. AbsHumanFactory factory = new HumanFactory();
43. Human yellowHuman = factory.createHuman(YellowHuman.class);
44. yellowHuman.getColor();
45. yellowHuman.talk();
46. WhiteHuman whiteHuman = factory.createHuman(WhiteHuman.class);
47. whiteHuman.getColor();
48. whiteHuman.talk();
49. }
50. }



#### 工厂方法扩展

##### 缩小为简单工厂模式（静态方法模式）

1. public class HumanFactory   {
2. public static <T extends Human> T createHuman(Class<T> tClass) {
3. Human human = null;
4. try {
5. human = (T) Class.forName(tClass.getName()).newInstance();
6. } catch (Exception e) {
7. e.printStackTrace();
8. }
9. return (T) human;
10. }
11. }
12. public class Client {
13. public static void main(String[] args) {
14. Human yellowHuman = HumanFactory.createHuman(YellowHuman.class);
15. yellowHuman.getColor();
16. yellowHuman.talk();
17. WhiteHuman whiteHuman = HumanFactory.createHuman(WhiteHuman.class);
18. whiteHuman.getColor();
19. whiteHuman.talk();
20. }
21. }

HumanFactory类有个地方发生变化：去掉抽象类，并将创建方法改为static关键字；调用房直接调用静态方法即可。

##### 升级多个工厂类

1. public abstract class AbsHumanFactory {
2. public abstract Human createHuman();
3. }
4. public class WhiteHumanFactory extends AbsHumanFactory {
5. @Override
6. public Human createHuman() {
7. Human human = new WhiteHuman();
8. return  human;
9. }
10. }
11. public class YellowHumanFactory extends AbsHumanFactory {
12. @Override
13. public Human createHuman() {
14. Human human = new YellowHuman();
15. return  human;
16. }
17. }
18. public class Client {
19. public static void main(String[] args) {
20. Human yellowHuman = new YellowHumanFactory().createHuman();
21. yellowHuman.getColor();
22. yellowHuman.talk();
23. Human whiteHuman = new WhiteHumanFactory().createHuman();
24. whiteHuman.getColor();
25. whiteHuman.talk();
26. }
27. }

变化点：AbsHumanFactory 抽象类不在是泛型；增加了工厂模式的实现类；调用方式改变，不再需要传入类参数。

#### 使用场景

工厂方法是new一个对象的替代，所以在需要生成对象的地方都可以使用，但是要慎重考虑是否要增加一个工厂类来进行管理，增加代码复杂度。**一般复杂对象适合使用工厂模式，用new就可以完成创建的对象无需使用工厂模式**

map的迭代器，Activity的oncreate等生命周期回调。

### 抽象工厂模式

#### 定义

为创建一组相关或者相互依赖的对象提供一个接口，而不需要指定他们的具体类

#### UML图

1. public interface IBrake {
2. void brake();*//制动系统*
3. }
4. public class NormalBrake implements IBrake {
5. @Override
6. public void brake() {
7. System.out.println("普通制动");
8. }
9. }
10. public class SeniorBrake implements IBrake {
11. @Override
12. public void brake() {
13. System.out.println("高级制动");
14. }
15. }
16. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
17. public interface IEngine {
18. void engine();*//发动机*
19. }
20. public class DomesticEngine implements IEngine {
21. @Override
22. public void engine() {
23. System.out.println("国产发动机");
24. }
25. }
26. public class ImportEngine implements IEngine {
27. @Override
28. public void engine() {
29. System.out.println("进口发动机");
30. }
31. }
32. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
33. public interface ITire {
34. void tire();*//轮胎*
35. }
36. public class NormalTire implements ITire {
37. @Override
38. public void tire() {
39. System.out.println("普通轮胎");
40. }
41. }
42. public class SuvTire implements ITire {
43. @Override
44. public void tire() {
45. System.out.println("越野轮胎");
46. }
47. }
48. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
49. public abstract class CarFactory {
50. public abstract ITire createTire();
51. public abstract IEngine createEngine();
52. public abstract IBrake createBrake();
53. }
54. public class AudiQ3Factory extends CarFactory {
55. @Override
56. public ITire createTire() {
57. return new NormalTire();
58. }
59. @Override
60. public IEngine createEngine() {
61. return new DomesticEngine();
62. }
63. @Override
64. public IBrake createBrake() {
65. return new NormalBrake();
66. }
67. }
68. public class AudiQ7Factory extends CarFactory {
69. @Override
70. public ITire createTire() {
71. return new SuvTire();
72. }
73. @Override
74. public IEngine createEngine() {
75. return new ImportEngine();
76. }
77. @Override
78. public IBrake createBrake() {
79. return new SeniorBrake();
80. }
81. }
82. public class Client {
83. public static void main(String[] args) {
84. CarFactory audiQ3 = new AudiQ3Factory();
85. audiQ3.createBrake().brake();
86. audiQ3.createEngine().engine();
87. audiQ3.createTire().tire();
88. CarFactory audiQ7 = new AudiQ7Factory();
89. audiQ7.createBrake().brake();
90. audiQ7.createEngine().engine();
91. audiQ7.createTire().tire();
92. }
93. }

#### 优缺点

优点：分离接口和实现，调用方只是面向产品接口编程，从具体的产品实现中解耦，抽象该工厂方法模式在切换产品类时更加灵活容易

缺点：类文件爆炸性增加；不太容易扩展新的产品类，因为增加一个产品类就需要修改抽象工厂，那么所有具体工厂类均被修改

### 原型模式

#### 定义

用圆形实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象；

1，性能优良。原型模式是在内存二进制流的拷贝，大多数情况下要比new一个对象性能要好，特别是一个循环体内产生大量对象时，效果更明显

2，逃避构造函数约束；**内存中拷贝，不执行拷贝函数。**既是优点也是缺点。

#### 使用场景

1，资源优化场景

类初始化需要消化非常多的资源，这个资源包括数据，硬件资源等

2，性能和安全要求的场景

通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式

3，一个对象多个修改者场景

一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象共调用者使用

#### 注意事项

1，不会执行构造函数

2，浅拷贝。数组，对象都指向同一个地址

3，clone和final 冲突，会报错。

### 建造者（build）模式

#### 定义

**将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示**

Build模式是一步步创建要给复杂对象的创建型模式，它允许用户在**不知道内部构件细节**的情况下，可以更精细的控制对象的构造流程。该模式是为了将构建复杂对象的过程和他的部件解耦，是的构建过程和部件的表示隔离开来。**为了在构建过程中对外隐藏细节，就可以使用build模式将部件和组装过程分离，使得构建过程和部件都可以自由扩展，两者之间的耦合也降到最低。**

#### 使用场景

1，相同的方法，不同的执行顺序，产生不同的事件结果

2，多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同

3，产品类非常复杂，或者产品类中的调用顺序不同产生了不同的作用，这个时候使用建造者模式非常合适

4，当初始化一个对象特别复杂，如参数多，且很多参数都具有默认值时。

#### UML图

Product产品类：产品的抽象类

Builder ：抽象Build类，规范产品的组件，一般是由子类实现具体的组件过程；

ConcreateBuilder：具体的Build类；

Director：统一组装过程

1. */\*\**
2. \* 抽象产品类
3. \*/
4. public abstract class Computer {
5. protected String board;
6. protected String display;
7. protected String os;
8. protected Computer(){}
9. public String getBoard() {
10. return board;
11. }
12. public void setBoard(String board) {
13. this.board = board;
14. }
15. public String getDisplay() {
16. return display;
17. }
18. public void setDisplay(String display) {
19. this.display = display;
20. }
21. public abstract void setOs() ;
22. }
23. */\*\**
24. \* 具体产品类
25. \*/
26. public class MacBook extends Computer {
27. protected MacBook(){}
28. @Override
29. public void setOs( ) {
30. os = "Mac OS X 10.10";
31. }
32. }
33. public abstract class Builder {
34. public abstract void buildBoard(String board);
35. public abstract void buildDisplay(String display);
36. public abstract void buildOs();
37. public abstract Computer create();
38. }
39. public class MacBookBuilder extends Builder {
40. private Computer computer = new MacBook();
41. @Override
42. public void buildBoard(String board) {
43. computer.setBoard(board);
44. }
45. @Override
46. public void buildDisplay(String display) {
47. computer.setDisplay(display);
48. }
49. @Override
50. public void buildOs() {
51. computer.setOs();
52. }
53. @Override
54. public Computer create() {
55. return computer;
56. }
57. }
58. public class Director {
59. Builder builder = null;
60. public Director(Builder builder){
61. this.builder = builder;
62. }
63. public void construct(String board,String display){
64. builder.buildBoard(board);
65. builder.buildDisplay(display);
66. builder.buildOs();
67. }
68. }
69. public class Client {
70. public static void main(String[] args) {
71. Builder builder = new MacBookBuilder();
72. Director director = new Director(builder);
73. director.construct("英特尔主板","三星显示器");
74. builder.create();
75. }
76. }

**通过具体的MacBuilder（持有macbook对象）来构建Macbook对象。而Diretor封装了调用过程，隐藏了创建细节。Builder和Director一起将复杂对象的创建与它的表示分离，使得同样的构建过程可以创建不同的对象。**

实际开发中Director常被省略。而直接使用一个Builder来进行组装，Builder通常为链式调用，set返回自身。

Android典型用法就是AlertDialog

### 工厂方法VS建造者模式

工厂方法模式注重的是整体对象的创建方法，而建造者模式注重的是部件构建的过程。比如要制造一个超人，如果使用工厂方法模式，直接产生出来的就是一个力大无穷，能够飞翔，内裤外穿的超人；如果使用建造者模式，则需要组装手，头，脚，躯干等部分。

意图不同：

工厂方法模式关注的是一个产品整体，无须关心产品各个部分是如何创建出来的；但在建造者模式中，一个具体产品的产生是依赖各个部件的产生以及封装顺序。简单来说，工厂模式是一个对象创建的粗线条应用，建造者模式则是通过细线条勾勒出一个复杂对象，关注的是产品组成部分创建过程

产品复杂度不同：

工厂方法模式创建的产品一般都是单一性质产品，而建造者模式创建的则是一个复合产品，它由各个部件复合而成，部件不同产品对象不同。一般来说，工厂方法模式对象颗粒度比较粗，建造者模式颗粒度比较细。

如果需要详细关注一个产品的生产，安装步骤，则选择建造者模式，否则工厂方法模式。

## 行为型设计模式

### 模板方法

#### 定义

定义一个操作中的算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

知道算法所需的关键步骤，并且确定了步骤的执行顺序，但是步骤的具体实现未知。

#### 使用场景

1，多个子类有公有的方法，并且逻辑基本相同

2，重要，复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。

3，重构时把相同的代码抽取到父类中，通过钩子函数约束行为

#### UML图

1. public abstract class AbsComputer {
2. protected void powerOn() {
3. System.out.println("开启电源");
4. }
5. protected void checkHardware() {
6. System.out.println("硬盘检查");
7. }
8. protected void loadOs() {
9. System.out.println("载入操作系统");
10. }
11. protected void login() {
12. System.out.println("登录成功后，进入系统");
13. }
14. protected boolean isNeedLogin() {
15. return true;
16. }
17. public final void startUp() {
18. powerOn();
19. checkHardware();
20. loadOs();
21. if (isNeedLogin()) {
22. login();
23. }
24. }
25. }
26. public class CoderComputer extends AbsComputer {
27. @Override
28. protected void login() {
29. System.out.println("程序员登录，权限最大");
30. }
31. }
32. public class NormalComputer extends AbsComputer {
33. @Override
34. protected void checkHardware() {
35. System.out.println("不需要检测");
36. }
37. @Override
38. protected boolean isNeedLogin() {
39. return false;
40. }
41. }
42. public class Client {
43. public static void main(String[] args) {
44. AbsComputer corderComputer = new CoderComputer();
45. corderComputer.startUp();
46. AbsComputer normalComputer = new NormalComputer();
47. normalComputer.startUp();
48. }
49. }

1，抽象类中方法的限定符，被调用的时public，其他的时protect

2，钩子函数可以改变行为

3，被调用的public方法用final修饰，防止被子类覆写