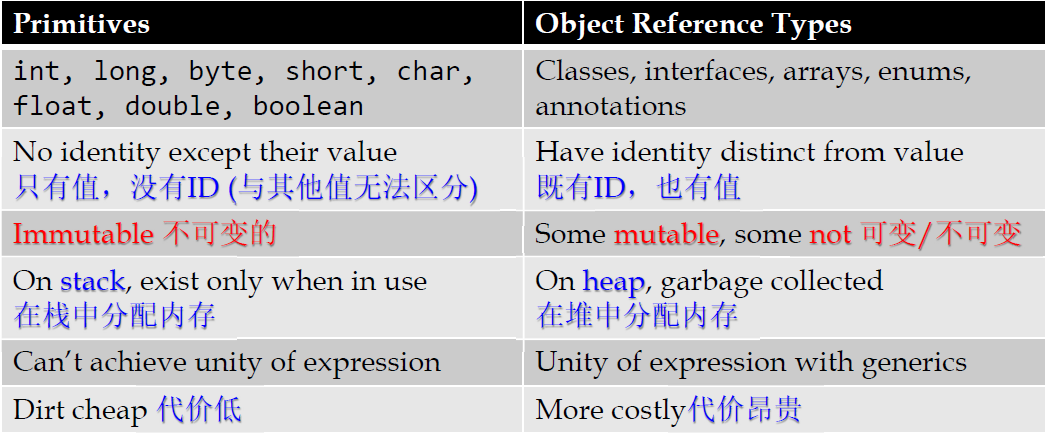
**3.1数据类型与类型检验**



将基本类型包装为对象类型，通常是在定义集合类型的时候使用它们，一般情况下，尽量避免使用，一般可以与基本类型自动转换



Java是静态类型语言，在编译阶段进行类型检查。Python是动态类型语言，在运行阶段进行类型检查。

作用：静态类型检查>> 动态>> 无检查

静态类型检查：可在编译阶段发现错误，避免了将错误带入到运行阶段，可提高程序正确性/健壮性

Static checking包括：

Syntax errors 语法错误,

Wrong names 类名/函数名错误,

Wrong number of arguments参数数目错误,

Wrong argument types参数类型错误,

Wrong return types返回值类型错误

Dynamic checking包括：

Illegal argument values 非法的参数值.

Unrepresentable return values 非法的返回值,

Out-of-range indexes 越界,

Calling a method on a null object reference. 空指针

静态检查：关于“类型”的检查，不考虑值

动态检查：关于“值”的检查

n/0=NaN

Immutability 不变性：重要设计原则

不变数据类型：一旦被创建，其值不能改变

如果是引用类型，也可以是不变的：一旦确定其指向的对象，不能再被改变

如果编译器无法确定final变量不会改变，就提示错误，这也是静态类型检查的一部分。尽量使用final变量作为方法的输入参数、作为局部变量。

final类无法派生子类

final变量无法改变值/引用

final方法无法被子类重写

不变对象：一旦被创建，始终指向同一个值/引用

可变对象：拥有方法可以修改自己的值/引用

使用不可变类型，对其频繁修改会产生大量的临时拷贝(需要垃圾回收)

通过防御式拷贝，给客户端返回一个全新的Date对象，大部分时候该拷贝不会被客户端修改，可能造成大量的内存浪费，如果使用不可变类型，则节省了频繁复制的代价

Snapshot diagrams

用于描述程序运行时的内部状态

便于程序员之间的交流

便于刻画各类变量随时间变化

便于解释设计思路

快照图为我们提供了一种可视化区别的方法，在更改变量和更改值之间：

分配给变量或字段时，正在更改变量箭头指向的位置。可以将其指向不同的值。

分配可变值的内容（例如数组或列表）时，正在更改该值内的引用。

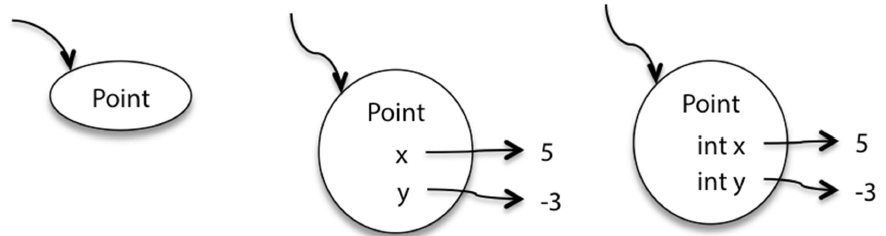
原始值基本类型的值：裸常数

传入的箭头是对变量或对象字段中的值的引用。

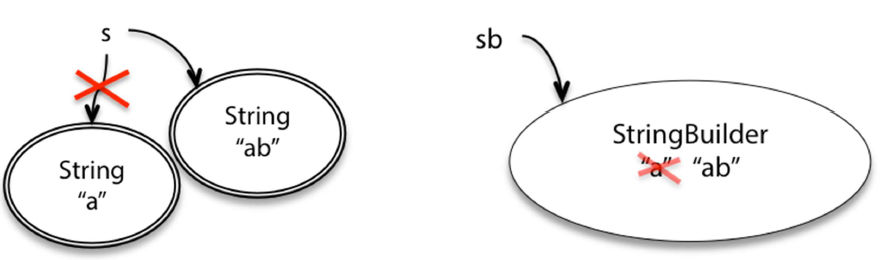
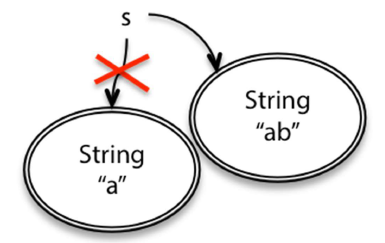


对象值对象类型的值：由其类型标记的圆圈

当想要显示更多细节时，在其中写入字段名称箭头指出他们的价值。对于更多细节，字段可以包括他们声明的类型。



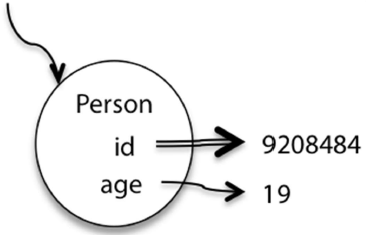
不可变对象：用双线椭圆



不可变的引用：用双线箭头

引用是不可变的，但指向的值却可以是可变的

可变的引用，也可指向不可变的值



**3.2 Designing Specification**

Specification

输入 /输出的数据类型，功能和正确性，性能

只讲 “能做什么”，不讲“怎么实现”

根据规约 判断行为等价

前置条件 ：对客户端的约束，在使用方法时必须满足

后置条件 ：对开发者的约束，方法结时必须满足后置条件

契约：如果前置条件满足了，后必须契约

前置条件不满足，则后置必须满足。前置条件不满足，则方法可做任何事情

静态类型声明是一种规约，可据此进行静态类型检查static checking。

方法前的注释也是一种规约，但需人工判定其否满足

除非在后置条件里声明过，否则方法内部不应该改变输入参数

应尽量遵循此规则，尽量不设计**mutating**的**spec**，否则就容易引发**bugs**。

程序员之间应达成的默契：除非**spec**必须如此，否则不应修改输入参数

尽量避免使用**mutable**的对象

前置条件更弱，后置条件更强

spec变强：更放松的前置条件+更严格的后置条件

欠定的规约：同一个输入可以有多个输出

非确定的规约：同一个输入，多次执行时得到的输出可能不同

欠定的规约通常有确定的实现

操作式规约，例如：伪代码

声明式规约：没有内部实现的描述，只有“初-终”状态

**3.3 Abstract Data Type (ADT)**

Creators创建者创建该类型的新对象。

-创建者可以将对象作为参数，但不是该类型的对象正在建造中。

Producers生产者从该类型的旧对象创建新对象。

-例如，String的concat（）方法是一个生成器：它接受两个字符串并生成一个表示其串联的新字符串。

Observers观察者获取抽象类型的对象并返回不同类型的对象。

-例如，List的size（）方法返回一个int。

Mutators改变对象。

-例如，List的add（）方法通过向末尾添加元素来改变列表。

变值器通常返回void，也可能返回非空类型

表示独立性：客户在使用ADT时无需考虑内部如何实现，ADT内部变化不影响外部spec和客户端。

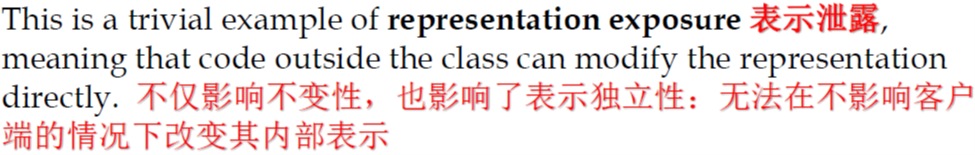


由ADT来负责不变量

-可以指望字符串永远不会改变，在调试使用字符串的代码时排除这种可能性

-或者当尝试为使用字符串的另一个ADT建立不变量时。

-对比一个字符串类型，保证只有当客户承诺不改变它时它才是不可变的。然后必须检查代码中可能使用该字符串的所有位置。



**3.4**

Strict inheritance (严格继承：子类只能添加新方法，无法重写超类中的方法)

-子类只能向父类添加新方法，不能覆盖它们

-如果某个方法无法在Java程序中覆盖，则必须加上前缀用关键字final。

Final

A final field: 防止在初始化后重新分配到字段

A final method: 防止覆盖该方法

A final class: 防止扩展类

方法重写是一种语言功能，它允许子类或子类提供已由其某个超类或父类提供的方法的特定实现。

-相同的名称，相同的参数或签名，以及相同的返回类型。

-执行的方法版本将由用于调用它的对象确定。

-如果使用父类的对象来调用该方法，则将执行父类中的版本;

-如果使用子类的对象来调用该方法，则将执行子类中的版本

父类型中的被重写函数体不为空：意味着对其大多数子类型来说，该方法是可以被直接复用的。

对某些子类型来说，有特殊性，故重写父类型中的函数，实现自己的特殊要求

如果父类型中的某个函数实现体为空，意味着其所有子类型都需要这个功能，

但各有差异，没有共性，在每个子类中均需要重写。

重写之后，利用super()复用了父类型中函数的功能，并对其进行了扩展

抽象方法：

-具有签名但没有实现的方法（也称为抽象操作）

-由关键字abstract定义

抽象类：

-包含至少一个抽象方法的类称为抽象类

接口：只有抽象方法的抽象类

-接口主要用于系统或子系统的规范。实现由子类或其他机制提供。

Concrete class→Abstract Class→Interface

如果某些操作是所有子类型都共有，但彼此有差别，可以在父类型中设计抽象方法，在各子类型中重写。

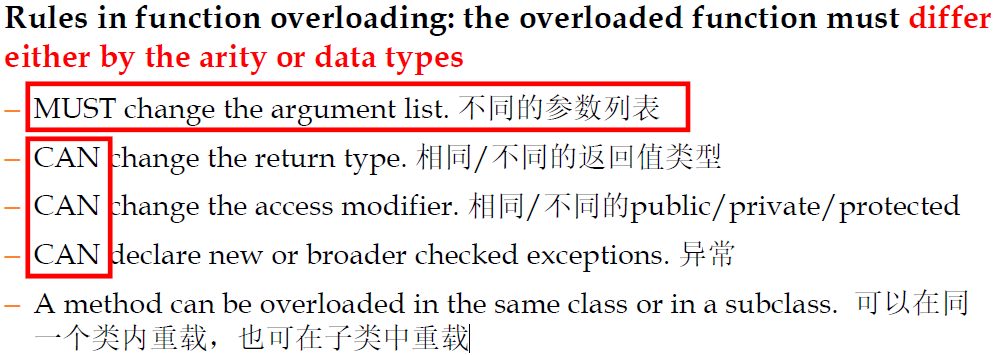
所有子类型完全相同的操作，放在父类型中实现，子类型中无需重写。有些子类型有而其他子类型无的操作，不要在父类型中定义和实现，而应在特定子类型中实现。

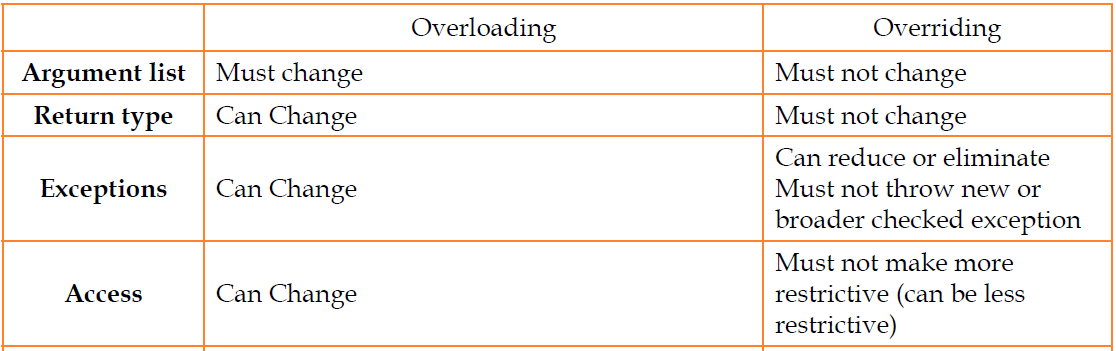
重载：多个方法具有同样的名字，但有不同的参数列表或返回值类型

价值：方便client调用，client可用不同的参数列表，调用同样的函数

静态多态：根据参数列表进行最佳匹配

静态类型检查：在编译阶段时决定要具体执行哪个方法(static type checking)





泛型接口

如果接口声明类型变量，则接口是通用的

-这些类型变量称为接口的类型参数。

-它定义了一个或多个充当参数的类型变量。

-通用接口声明定义了一组类型，每种类型都可以调用类型参数部分。

-所有参数化类型在运行时共享相同的接口。

泛型方法

如果方法声明类型变量，则该方法是通用的。

-这些类型变量称为方法的形式类型参数。

-形式类型参数列表的形式与类或接口的类型参数列表相同。

站在外部观察者角度：对两个对象调用任何相同的操作，都会得到相同的结果，则认为这两个对象是等价的。反之亦然！

在自定义ADT时，需要重写Object的equals()

==比较引用的等价性

.equals()比较对象等价性

== 对基本数据类型，使用==判定相等

对对象类型，使用equals()

如果用==，是在判断两个对象身份标识ID是否相等（指向内存里的同一段空间）

在Object中实现的缺省equals()是在判断引用等价性

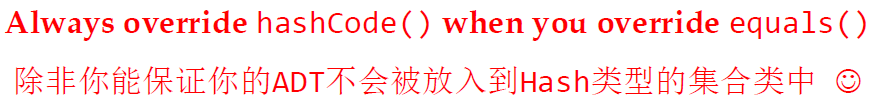
@Override。方法签名中容易出错，并在意图覆盖方法时重载方法。

只要您的目的是覆盖超类中的方法，就应该使用Java的注释@Override。

使用此注释，Java编译器将检查超类中实际存在具有相同签名的方法，如果您在签名中出错，则会给出编译器错误。

严格来说，在没有AF的情况下直接在equals()中判断每个域的等价性，是不正确的

等价的对象必须有相同的hashCode



观察等价性：在不改变状态的情况下，两个mutable对象是否看起来一致

行为等价性：调用对象的任何方法都展示出一致的结果

如果某个mutable的对象包含在Set集合类中，当其发生改变后，集合类的行为不确定，务必小心

在JDK中，不同的mutable类使用不同的等价性标准

对可变类型，实现行为等价性即可。只有指向同样内存空间的objects，才是相等的。所以对可变类型来说，无需重写equals() hashCode()，直接继承Object的两个方法即可。如果一定要判断两个可变对象看起来是否一致，最好定义一个新的方法。

**5.1可复用性的度量、形态与外部表现**

白盒复用：源代码可见，可修改和扩展

–复制已有代码当正在开发的系统，进行修改

–可定制化程度高

–对其修改增加了软件的复杂度，且需要对其内部充分的了解

黑盒复用：源代码不可见，不能修改

–只能通过API接口来使用，无法修改代码

–简单，清晰

–适应性差些

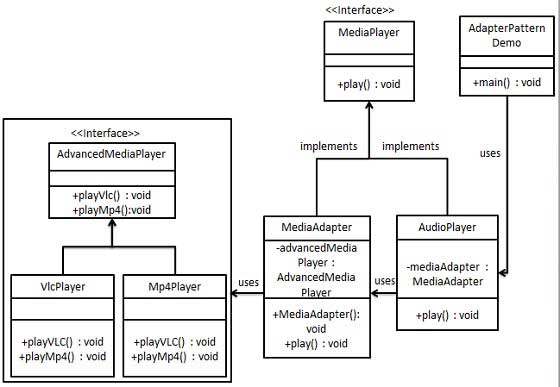
**5.2**

**5.3设计模式**

**适配器模式**

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。



# 装饰器模式

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

**介绍**

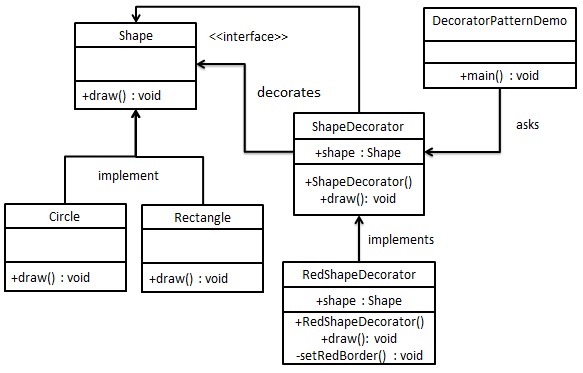
**意图：**动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

**主要解决：**一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

**何时使用：**在不想增加很多子类的情况下扩展类。

**如何解决：**将具体功能职责划分，同时继承装饰者模式。

**关键代码：** 1、Component 类充当抽象角色，不应该具体实现。 2、修饰类引用和继承 Component 类，具体扩展类重写父类方法。

**外观模式**

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。

这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

**介绍**

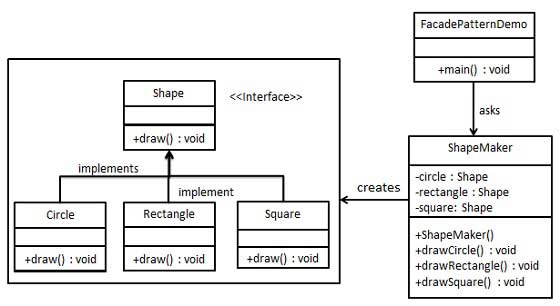
**意图：**为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

**主要解决：**降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。

**何时使用：** 1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。 2、定义系统的入口。

**如何解决：**客户端不与系统耦合，外观类与系统耦合。

**关键代码：**在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。



**模板模式**

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

**介绍**

**意图：**定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法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。

**主要解决：**一些方法通用，却在每一个子类都重新写了这一方法。

**何时使用：**有一些通用的方法。

**如何解决：**将这些通用算法抽象出来。

**关键代码：**在抽象类实现，其他步骤在子类实现。



**迭代器模式**

迭代器模式（Iterator Pattern）是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。

迭代器模式属于行为型模式。

**介绍**

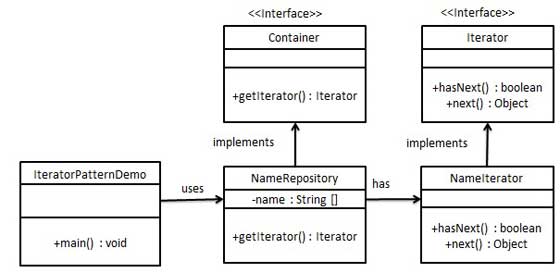
**意图：**提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

**主要解决：**不同的方式来遍历整个整合对象。

**何时使用：**遍历一个聚合对象。

**如何解决：**把在元素之间游走的责任交给迭代器，而不是聚合对象。

**关键代码：**定义接口：hasNext, next。



**6.1 可维护性的度量与构造原则**

可维护性指标：

Maintainability可维护性：-可以轻松修改软件系统或组件以纠正故障，提高性能或其他属性，或适应变化的环境。

Extensibility可扩展性：-软件设计/实施考虑了未来的增长，被视为扩展系统能力和实施扩展所需工作量的系统性衡量标准。

Flexibility灵活性：-软件根据用户要求，外部技术和社交环境等轻松更改的能力

Adaptability可适应性：-交互式系统（自适应系统）的能力，可以根据获取的有关其用户及其环境的信息，使其行为适应个人用户。

Manageability可管理性：-如何高效，轻松地监控和维护软件系统，以保持系统的顺利运行，安全和运行。

Supportability支持性：-基于包括质量文档，诊断信息以及知识渊博且可用的技术人员在内的资源，如何有效地在部署后保持软件运行。

软件维护的类型：

Corrective maintenance 25% 纠错性：-交付后对软件产品进行反应性修改以纠正发现的问题;

Adaptive maintenance 21% 适应性：-在交付后对软件产品进行修改，以使软件产品在变化或变化的环境中可用;

Perfective maintenance 50% 完善性：-交付后增强软件以提高性能或可维护性;

Preventive maintenance 4% 预防性：-交付后修改软件产品，以便在软件产品成为有效故障之前检测并纠正软件产品中的潜在故障

聚合度耦合度：

耦合是模块之间依赖性的度量。如果一个模块中的更改可能需要更改另一个模块，则两个模块之间存在依赖关系。

模块之间的耦合程度由以下因素决定：

- 模块之间的接口数量（数量）和

- 每个接口的复杂性（由通信类决定）（质量）

精心设计的网络应用程序模块化：

-指定数据和语义的HTML文件

- CSS规则，用于指定HTML数据的外观和格式

-定义页面行为/交互性的JavaScript

聚合度衡量模块的功能或职责的相关程度。

-如果一个模块的所有元素都朝着同一个目标努力，那么它就具有很高的聚合度。

-最佳设计在模块内具有高聚合度（也称为强聚合度），在模块之间具有低耦合（也称为弱耦合）。

-当耦合高时，内聚力往往较低，反之亦然。

SOLID: 5类设计原则：

(SRP) The Single Responsibility Principle 单一责任原则

-不应该有多于1个原因让你的ADT发生变化，否则就拆分开。一个类，一个责任

(OCP) The Open-Closed Principle 开放-封闭原则

模块的行为应是可扩展的，从而该模块可表现出新的行为以满足需求的变化。但模块自身的代码是不应被修改的。扩展模块行为的一般途径是修改模块的内部实现。如果一个模块不能被修改，那么它通常被认为是具有固定的行为

(LSP) The Liskov Substitution Principle Liskov替换原则

子类型必须能够替换其基类型。派生类必须能够通过其基类的接口使用，客户端无需了解二者之间的差异

(DIP) The Dependency Inversion Principle 依赖转置原则

不能强迫客户端依赖于它们不需要的接口：只提供必需的接口

(ISP) The Interface Segregation Principle 接口聚合原则

抽象的模块不应依赖于具体的模块。具体应依赖于抽象

**6.2可维护性的度量与构造原则**

**抽象工厂模式**

抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

介绍

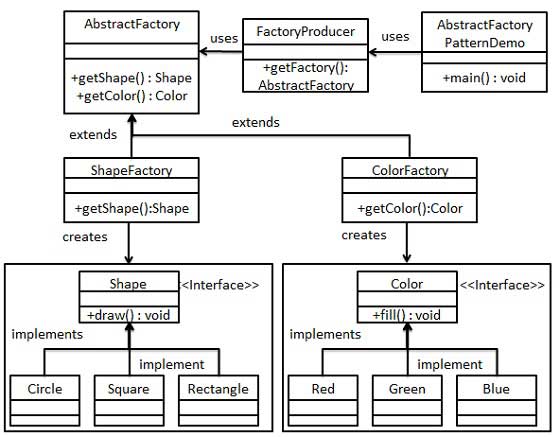
意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

主要解决：主要解决接口选择的问题。

何时使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

如何解决：在一个产品族里面，定义多个产品。

关键代码：在一个工厂里聚合多个同类产品



**工厂模式**

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

介绍

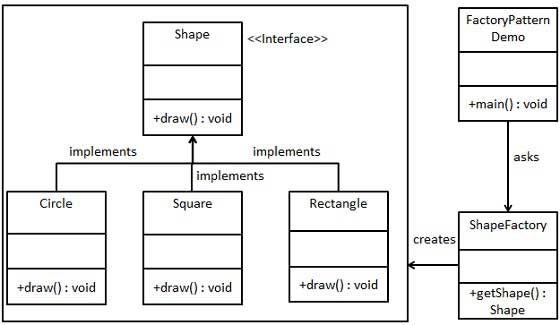
意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决接口选择的问题。

何时使用：我们明确地计划不同条件下创建不同实例时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码：创建过程在其子类执行。



**建造者模式**

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

介绍

意图：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

主要解决：主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

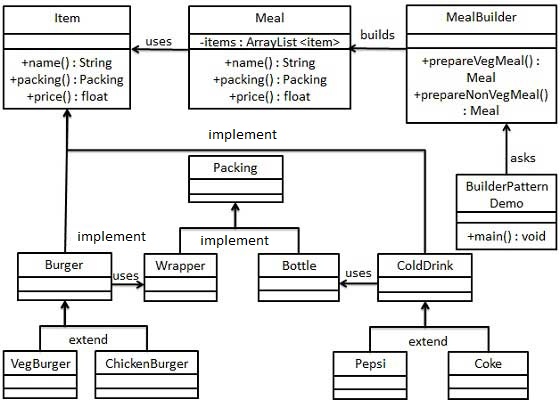
何时使用：一些基本部件不会变，而其组合经常变化的时候。

如何解决：将变与不变分离开。

关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

优点： 1、建造者独立，易扩展。 2、便于控制细节风险。

缺点： 1、产品必须有共同点，范围有限制。 2、如内部变化复杂，会有很多的建造类。



**桥接模式**

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

介绍

意图：将抽象部分与实现部分分离，使它们都可以独立的变化。

主要解决：在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。

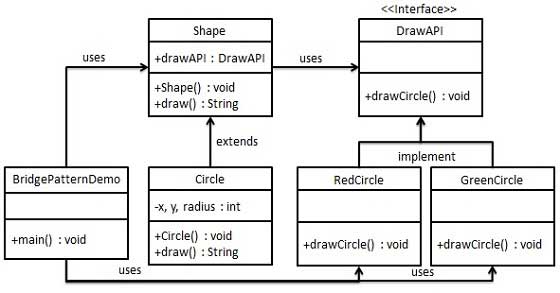
何时使用：实现系统可能有多个角度分类，每一种角度都可能变化。

如何解决：把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。

关键代码：抽象类依赖实现类。

优点： 1、抽象和实现的分离。 2、优秀的扩展能力。 3、实现细节对客户透明。

缺点：桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。



**代理模式**

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。

在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。

介绍

意图：为其他对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

何时使用：想在访问一个类时做一些控制。

如何解决：增加中间层。

关键代码：实现与被代理类组合。

优点： 1、职责清晰。 2、高扩展性。 3、智能化。

缺点： 1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。 2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

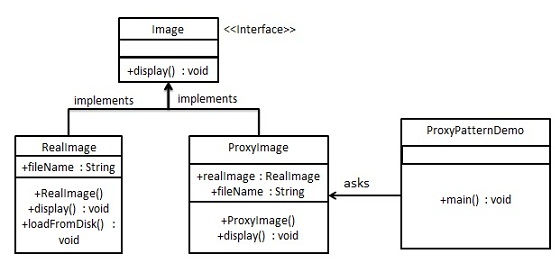
使用场景：按职责来划分，通常有以下使用场景： 1、远程代理。 2、虚拟代理。 3、Copy-on-Write 代理。 4、保护（Protect or Access）代理。 5、Cache代理。 6、防火墙（Firewall）代理。 7、同步化（Synchronization）代理。 8、智能引用（Smart Reference）代理。

注意事项： 1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。 2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

实现

我们将创建一个 Image 接口和实现了 Image 接口的实体类。ProxyImage 是一个代理类，减少 RealImage 对象加载的内存占用。

ProxyPatternDemo，我们的演示类使用 ProxyImage 来获取要加载的 Image 对象，并按照需求进行显示。



**组合模式**

组合模式（Composite Pattern），又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。

介绍

意图：将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

主要解决：它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

何时使用： 1、您想表示对象的部分-整体层次结构（树形结构）。 2、您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

如何解决：树枝和叶子实现统一接口，树枝内部组合该接口。

关键代码：树枝内部组合该接口，并且含有内部属性 List，里面放 Component。

应用实例： 1、算术表达式包括操作数、操作符和另一个操作数，其中，另一个操作符也可以是操作数、操作符和另一个操作数。 2、在 JAVA AWT 和 SWING 中，对于 Button 和 Checkbox 是树叶，Container 是树枝。

优点： 1、高层模块调用简单。 2、节点自由增加。

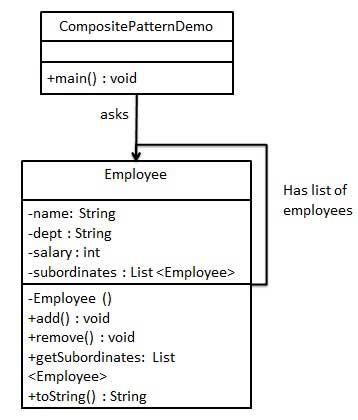
缺点：在使用组合模式时，其叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则。

使用场景：部分、整体场景，如树形菜单，文件、文件夹的管理。

注意事项：定义时为具体类。

实现

我们有一个类 Employee，该类被当作组合模型类。CompositePatternDemo，我们的演示类使用 Employee 类来添加部门层次结构，并打印所有员工。



**观察者模式**

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

介绍

意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

如何解决：使用面向对象技术，可以将这种依赖关系弱化。

关键代码：在抽象类里有一个 ArrayList 存放观察者们。

使用场景：

（1）一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。

（2）一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。

（3）一个对象必须通知其他对象，而并不知道这些对象是谁。

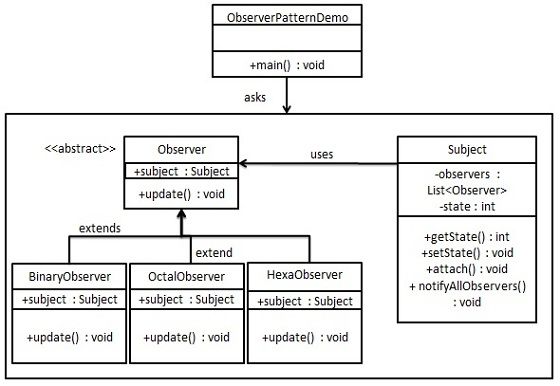
（4）需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

注意事项： 1、JAVA 中已经有了对观察者模式的支持类。 2、避免循环引用。 3、如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。

实现

观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法。我们创建 Subject 类、Observer 抽象类和扩展了抽象类 Observer 的实体类。

ObserverPatternDemo，我们的演示类使用 Subject 和实体类对象来演示观察者模式。



**访问者模式**

在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

介绍

意图：主要将数据结构与数据操作分离。

主要解决：稳定的数据结构和易变的操作耦合问题。

何时使用：需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。

如何解决：在被访问的类里面加一个对外提供接待访问者的接口。

关键代码：在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。

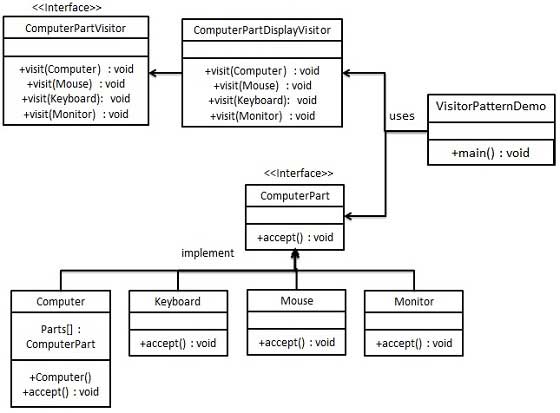
优点： 1、符合单一职责原则。 2、优秀的扩展性。 3、灵活性。

缺点： 1、具体元素对访问者公布细节，违反了迪米特原则。 2、具体元素变更比较困难。 3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象。

使用场景： 1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

注意事项：访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

实现



**状态模式**

在状态模式（State Pattern）中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。

在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。

介绍

意图：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

主要解决：对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

何时使用：代码中包含大量与对象状态有关的条件语句。

如何解决：将各种具体的状态类抽象出来。

关键代码：通常命令模式的接口中只有一个方法。而状态模式的接口中有一个或者多个方法。而且，状态模式的实现类的方法，一般返回值，或者是改变实例变量的值。也就是说，状态模式一般和对象的状态有关。实现类的方法有不同的功能，覆盖接口中的方法。状态模式和命令模式一样，也可以用于消除 if...else 等条件选择语句。

优点：

1、封装了转换规则。

2、枚举可能的状态，在枚举状态之前需要确定状态种类。

3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。

4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。

5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

缺点：

1、状态模式的使用必然会增加系统类和对象的个数。

2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。

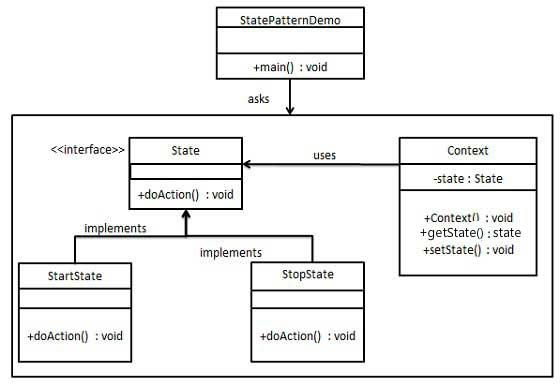
3、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

使用场景： 1、行为随状态改变而改变的场景。 2、条件、分支语句的代替者。

注意事项：在行为受状态约束的时候使用状态模式，而且状态不超过 5 个。

实现

我们将创建一个 State 接口和实现了 State 接口的实体状态类。Context 是一个带有某个状态的类。StatePatternDemo，我们的演示类使用 Context 和状态对象来演示 Context 在状态改变时的行为变化。



**备忘录模式**

备忘录模式（Memento Pattern）保存一个对象的某个状态，以便在适当的时候恢复对象。备忘录模式属于行为型模式。

介绍

意图：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

主要解决：所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以以后将对象恢复到原先保存的状态。

何时使用：很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有"后悔药"可吃。

如何解决：通过一个备忘录类专门存储对象状态。

关键代码：客户不与备忘录类耦合，与备忘录管理类耦合。

优点： 1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。 2、实现了信息的封装，使得用户不需要关心状态的保存细节。

缺点：消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

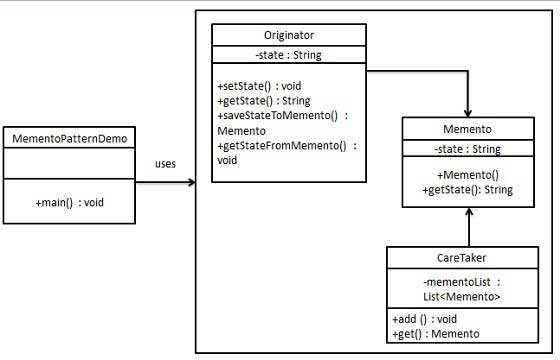
使用场景： 1、需要保存/恢复数据的相关状态场景。 2、提供一个可回滚的操作。

注意事项： 1、为了符合迪米特原则，还要增加一个管理备忘录的类。 2、为了节约内存，可使用原型模式+备忘录模式。

实现

备忘录模式使用三个类 Memento、Originator 和 CareTaker。Memento 包含了要被恢复的对象的状态。Originator 创建并在 Memento 对象中存储状态。Caretaker 对象负责从 Memento 中恢复对象的状态。

MementoPatternDemo，我们的演示类使用 CareTaker 和 Originator 对象来显示对象的状态恢复。



**6.3面向可维护性的构造技术**

**8.2 jvm**

Static mode在将程序load进内存的时候或开始执行时，确定所有对象的分配，运行时无法改变, 不支持递归，不支持动态创建的可变长的复杂数据类型

代码中的一个变量ref可在不同时间被指向到不同的内存对象上，无法在编译阶段确定。内存对象也可以进一步指向其他对象

所有局部的基本数据类型都在栈上创建

每个线程有自己的栈，管理其局部数据，各栈之间彼此不可见

多线程之间传递数据，是通过复制而非引用

在静态内存分配模式下，无需进行内存回收：所有都是已确定的

从root对象开始进行有向图的搜索，将图分为root可达部分和root不可达部分，后者将被进行内存回收

引用计数的基本思想：为每个object存储一个计数RC，当有其他reference指向它时，RC++；当其他reference与其断开时，RC--；如果RC==0，则回收它(及其所有指向的object)。

引用计数方法的优点：简单、计算代价分散，“幽灵时间”短→0

引用计数方法的缺点：不全面（容易漏掉循环引用的对象）、并发支持较弱、占用额外内存空间、等

标记-清除：为每个object设定状态位(live/dead)并记录，即mark阶段；将标记为dead的对象进行清理，即sweep阶段。

缺点：需要停止程序执行来Mark和Sweep，导致幽灵时间过长，也影响程序本身的性能；内存碎片化

**单例模式**

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意：

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

介绍

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

何时使用：当您想控制实例数目，节省系统资源的时候。

如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

关键代码：构造函数是私有的。

应用实例：

1、一个班级只有一个班主任。

2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。

3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点：

1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。

2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

使用场景：

1、要求生产唯一序列号。

2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。

3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

实现

我们将创建一个 SingleObject 类。SingleObject 类有它的私有构造函数和本身的一个静态实例。SingleObject 类提供了一个静态方法，供外界获取它的静态实例。SingletonPatternDemo，我们的演示类使用 SingleObject 类来获取 SingleObject 对象。

