软件体系结构
什么是软件体系结构
IEEE-1471中的软件体系结构定义
The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.
<<software Engineering>> Ian Sommerville
大型系统总是由提供一些相关服务的多个子系统构成。设计阶段的初期需要识别这些子系统并且建立这些子系统的控制和通信的一个框架，称为体系结构设计。
这个设计阶段的输出是软件体系结构的一个描述。
体系结构对软件的影响举例（1）
1. 性能（performance）
如果性能是系统的关键需求，应该把关键的操作局部化在少量的几个子系统中，并且在这些子系统之间尽量减少通信。这可能需要使用粒度较大的部件以尽量减少这些子系统之间的通信。
2. security （安全性，也有翻译为保密性的）
安全性要求高的系统往往采用分层结构，最内层是最关键的部分，并且在这些层次应用严格的安全验证。
3. safety（安全性，也有翻译为保险性的）
safety相关的操作应尽量集中在少数几个甚至一个子系统中，这样可以方便safety验证，也方便为其提供相关的保护系统。
4. 可用性（availability）
这时可能需要冗余部件，以便能够在系统不停顿的同时替换和更新部件。
5. 可维护性（maintainability）
系统应该由细粒度、自包含的部件构成，以便更换
体系结构对软件的影响举例（2）
上述这些方面有时会互相冲突
例如用大粒度部件能提高性能，用细粒度部件能提高可维护性，而这两方面都是关键需求时，就会有矛盾。
体系结构的设计与需求密切相关，要识别出变化可能性大的需求，在体系结构设计中应尽量做到当变化发生时，让其影响限制在局部，而不会波及到多个子系统。
需求类似的系统的体系结构也常常是类似的，因此可以利用系统体系结构支持大规模的软件重用。
体现
软件是有结构的
不同的应用程序需要不同的结构
软件体系结构的研究
软件体系结构是软件工程的一个分支
研究软件体系结构的目的在于为软件体系结构的设计和改进提供指导。
基于经验的方法，收集实用有效的体系结构模式及其适用场景，根据实际情况选择应用
面向实用，但不够精确，受主观影响多
基于严格的形式化推导和定量计算，希望能够从形式化的需求说明推导出软件体系结构甚至整个软件的实现
距离实用有相当的距离
也许更适合于对已有的设计进行验证
宏观与微观结构
一般的软件工程师对于宏观的体系结构的选择并没有太大的自由度，这个层次更多取决于具体的问题领域和将要使用的开发工具和技术，例如：
早期是基于主机-终端的结构
然后是单机上的系统
然后是C/S结构、B/S结构
…
很多软件需要在现有框架的基础上开发，框架已经限定了宏观的结构。
前面例子中提到的MFC就是一种框架

在微观结构方面
软件的设计开发过程是一个螺旋上升的过程
根据实际的需要，模仿尽可能相似的已经证明成功的系统的结构进行设计和开发
不断根据需求的变化、对于问题的新的认识、设计和开发中的新的思路，对软件进行重构（refactor）

在这个过程中需要大量借鉴以前的开发经验
软件设计模式
一个好的软件设计解决方案可以在以后的类似应用中被重复使用。
可以把这样的软件设计作为设计模式记录和整理保存，用于其它的类似设计。
特定的应用程序往往有特定的结构模式，例如编译器、操作系统等，但更常用的是一些较微观层次的结构模式，这是本课程的重点
并不是在软件开发中用的设计模式越多越好，有时候可能还要避免使用某些模式。
因为有时候效率、安全性等方面的要求高于灵活性、可复用的要求，而很多设计模式为了提高灵活性和可复用可能会降低系统的运行效率。
软件体系结构与设计模式的关系
一般不在软件体系结构与设计模式之间划等号
但二者之间有密切关系，可以认为设计模式是基础性的微结构
从实用的角度看，设计模式在实际应用中更普遍、也更稳定。

第一章 引言

设计模式的意义

设计模式可以帮助系统的设计者利用以前成功的设计更快更好地建立新的设计。

设计模式可以帮助新的系统设计者和开发人员更快更好地分析理解一个现有的设计。

1.1 什么是设计模式

一个模式的4个基本要素

模式名称：用于代表一个特定的模式。

问题：通过对所要解决的问题及其前因后果的解释，说明应该在什么样的条件下应用特定的模式。

解决方案：描述了构成一个设计的各个元素、这些元素之间的关系、各自的职责和协作关系。

提供的是一个可定制的模板。

效果：应用特定模式的效果、应用中需要权衡的因素。

灵活性、可扩充性、可移植性、时间代价、空间代价、 …

1.2 Smalltalk MVC中的设计模式

MVC结构把系统功能分解为三个部分：

模型（Model）：包括所处理问题域的所有数据和逻辑；

视图（View）：提供模型的视觉界面，必须正确反映模型的当前状态；

控制器（Controller）：定义用户界面对用户输入的响应方式。

动态过程：

用户通过控制器改变模型中的数据，模型通知视图，视图反映模型的改变。

进一步分析

MVC本身是一种设计模式或软件体系结构，但它的实现通常需要组合其它的设计模式

Observer

Composite

Strategy

…

视图和模型

通过一个“定购/通知”协议连接视图和模型。

一个模型可以有多个视图；

模型数据发生变化时通知各个视图，各个视图即时更新状态。

把视图和模型都看作普通的对象，可以得到一个更一般的设计模式—Observer：

被观察对象的改变被通知给观察者对象，观察者对象做出相应的反应。

视图嵌套

在MVC中视图可以嵌套，例如在一个视图中嵌套按钮、调试窗口等，构成一个复合视图。

复合视图的接口与普通视图一致，可以用普通视图的地方就可以用复合视图；

推广复合视图与普通视图的这种关系可以得到一个更一般的设计模式—Composite：

可以定义一些原子对象和由若干个原子对象组成的复合对象，原子对象和复合对象对外提供相同的接口，可以用相同的方式使用。

可以通过这种方式进行功能的组装式实现

视图和控制器

MVC允许在不改变视图外观的情况下改变对用户输入的响应方式：

可以建立多种的控制器，这些控制器实现不同的响应方式，但都提供相同的接口；

一个视图通过替换控制器就可以改变响应方式；

例如在一种模式下点鼠标会改变字体，另一种模式下点鼠标则改变颜色

推广视图与控制器的这种关系可以得到一个更一般的设计模式—Strategy：

可以有多个Strategy对象，每个对象实现一种算法，替换Strategy对象就可以改变算法。

1.3 描述设计模式

设计的结果是一些类和对象以及它们之间的关系

但其它的信息也很重要，因此还需要描述一个设计产生的决策过程、选择过程和权衡过程。

一个设计模式的描述中包含：

模式名称和分类、意图、别名、动机、适用性、结构、参与者、协作、效果、实现、代码示例、已知应用、相关模式

在介绍具体模式时将了解这些术语的含义

1.4 设计模式的编目

列举和简单概括本书中介绍的各种设计模式

（略）

1.5 组织编目

模式的分类

各类模式的特点

模式的分类（表1-1）

根据模式所要完成的工作可以分为：

创建型：对象的创建；

结构型：处理类或对象的组合；

行为型：涉及对象的交互和职责分配；

根据模式主要应用于类还是对象可以分为：

类模式：主要通过类的继承建立，是静态的；

对象模式：通过对象间的关系实现，是动态的，甚至在运行中都可以改变。

图1.1应该在掌握了所有模式的特点后再理解。

各类模式的特点

创建型模式

创建型类模式将对象的部分创建工作延迟到子类中

创建型对象模式将对象的部分创建工作延迟到另一个对象中

结构型模式

结构型类模式使用继承机制来组合类

结构型对象模式通过对象组合方式组装对象

行为型模式
行为型类模式使用继承描述算法和控制流
行为型对象模式则描述一组对象怎样协作完成单个对象无法完成的任务。
1.6 设计模式怎样解决设计问题
利用设计模式可以解决以下一些在面向对象设计中经常遇到的问题：
寻找合适的对象
决定对象粒度
指定对象接口
描述对象的实现
运用复用机制
关联运行时刻和编译时刻的结构
设计应支持变化
1.6.1 寻找合适的对象
面向对象设计最困难的部分是把系统分解成对象的集合，而设计中的抽象对于设计的灵活性至关重要
有时需要抽象出现实世界中并不存在的类、接口以及对象
例如Composite模式中的Component类
设计模式可以辅助确定并不明显的抽象
例如现实世界中一般不把算法或者状态作为对象，但Strategy模式、State模式中则把它们作为对象。
1.6.2 决定对象粒度
对象的大小和数量变化很大
设计模式提供对复杂大对象和大量小对象的支持
用对象表示一个完整子系统的Façade模式
支持大量小粒度对象的Flyweight模式
…
1.6.3 指定对象接口
型构（signature）
对象的每个操作的名称、参数和返回值构成这个操作的型构。
接口（Interface）
对象操作所定义的所有操作型构的集合被称为该对象的接口
对象通过接口与外部交换信息
类型（Type）是用来标识特定接口的一个名字。
动态绑定
一个对象可以有多种类型
子类型继承了它的超类型的接口
不同的对象可以共享同一类型，接口与实现相互独立
运行过程中根据接受请求的对象决定具体的响应方法被称为动态绑定
动态绑定允许在运行时刻替换具有相同接口的对象
动态绑定举例
Circle和Cylinder是两个类
具有相同的接口
但接口的具体实现不同
在程序运行时，请求一个对象提供其面积，如果这个对象是一个圆，则会计算出圆的面积，如果这个对象是个圆柱，则会计算出圆柱的面积。
Circle类

```
class Circle {   //接口
  ...
  // calculate area of Circle
 public double area()  //型构
  {
    return Math.PI * radius * radius;
  }
  ...
}  // end class Circle
```

Cylinder类

```
class Cylinder extends Circle{
  ...
  // calculate area of Cylinder
  public double area()
  {
    return 2 * Math.PI * radius * radius +
          2 * Math.PI * radius * height;
  }
  ...
}  // end class Cylinder
```

设计模式对接口设计的帮助
有些设计模式能够辅助确定接口的定义
确定接口的主要组成部分及接口上的数据类型
例如Memento模式要求定义两个接口
有些设计模式则能够辅助确定接口之间的关系
指定一些类具有相同或相似的接口
对一些类的接口做限制，例如Visitor模式
1.6.4 描述对象的实现
面向对象软件的实现涉及许多基本要素
类、实例（对象）、继承、抽象类、抽象操作、具体类、操作重载(override)
设计模式对于这些基本要素的使用具有指导作用。
例如什么时候采用继承，什么时候采用对象组合，什么时候要实现什么接口
类和类的继承的图形表示
对象的实例化的图形表示
类继承与接口继承的比较
类继承根据一个类的实现定义另一个类的实现
接口继承则描述了一个对象什么时候能被用来替代另一个对象
对接口编程，而不是对实现编程
这是设计模式中的一条重要原则
通常不把变量声明为某个特定的类的对象，而让它遵循某个较高层次的抽象类的接口。
这个抽象类的所有子类对象都能够支持这种接口
优点：

对象之间相互独立，只要能够遵循特定的接口，不必指定对象所属的具体类或这种类的具体实现。
例如可以向一个几何体发出计算面积的请求，但不必指定是Circle还是Cylinder。
1.6.5 运用复用机制
继承和组合的比较
委托
继承和参数化类型的比较
继承和组合的比较
面向对象系统中功能复用的两种最常用技术
类继承：又称白盒复用，超类的内部细节对子类可见。
对象组合：又称黑盒复用，因为对象之间通过严格定义的接口交互，不需要知道对方的内部细节。
优先使用对象组合而不是类继承
继承是静态的，无法在运行时刻动态改变；
继承中子类不能完全独立于超类，超类的改变可能会导致子类的改变。
对象组合则只要求对象之间遵循规定的接口
对象的相互独立性较强，可以在运行中动态改变对象的组合。具有更好的可复用性。
但由于可用的对象构件并不丰富，通过继承创建新的构件要比通过对象组合的方式容易
因此，应该综合使用这两种技术，但优先使用对象组合而不是类继承
缺点是会有较小的类层次和较多的对象
委托（delegation）
有两个对象参与处理一个请求，一个对象对外接受请求，然后转发（委托）给另一个对象做实际的处理。
通过组合实现了与继承同样的复用能力
例如窗口对象可以把一些操作委托给一个矩形对象，或具有相同接口的其它几何形状对象，实现窗口的移动、放大、缩小等操作。
委托方式的缺点（相比较继承方式而言）
不容易理解
效率低
继承和参数化类型的比较
有些语言允许在定义一个类型时把其中所用到的一些类型设定为参数，在使用时再指定这些参数。
例如可以定义一个参数化的列表结构，表项的类型为参数。具体构造一个列表时再指定其类型，如整型、字符串等等。
在复用时可以改变所用到的对象的类型
但不能在运行时刻改变。
参数化类型允许作为参数的类型互相没有继承关系，有时更灵活。
继承和参数化类型可以结合在一起用，实现更好的复用。
1.6.6 关联运行时刻和编译时刻的结构
程序代码的静态结构和程序运行时对象之间的动态结构可能很不相同，理解设计模式将有助于理解二者之间的关系，否则仅从静态结构很难理解其动态意图，例如同样是对象的引用，装饰器和组合模式之间的动态意图是不同的。
例如对象的聚合(aggregation)和相识(acquaintance)关系的表达语法可能是一样的（例如C++中的指针引用）
聚合是对象之间的包含关系，往往能维持较长时间；相识则是相互引用的关系，有时仅维持很短时间。
有时不能从静态的代码识别，而是由设计者主观决定的，而了解设计模式将有助于理解设计者的主观意图。
1.6.7 设计应支持变化
设计一个系统时必须尽可能考虑系统可能的变化，并体现在设计中，以尽量避免重新设计或修改设计。

导致重新设计的原因和解决的方法
1）通过显式地指定一个类来创建对象
2）对特定操作的依赖
3）对硬件或软件平台的依赖
4）对对象表示或实现的依赖
5）算法依赖
6）紧耦合
7）通过生成子类来扩充功能
8）不能方便地对类进行修改
1）通过显式地指定一个类来创建对象
缺点：设计的软件受到具体类的实现的约束，而不仅仅是接口的约束
解决方法：在必要的时候，应该通过间接的方法确定所要建立的对象的类
例如：Abstract Factory, Factory Method, Prototype
2）对特定操作的依赖
缺点：如果在设计中为某个请求指定了一个特定的操作，对这个请求的响应方式也就固定了。
解决方法；需要能够方便地改变对一个请求的响应方法。
例如：Chain of Responsibility, Command
3）对硬件或软件平台的依赖
缺点：降低了软件的可移植性
解决方法：限制软件的平台相关性，例如把与平台有关的部分独立包装起来
例如：Abstract Factory, Bridge
4）对对象表示或实现的依赖
缺点：软件的设计依赖于某些对象的表示、保存、定位和实现，将受到这些对象改变的影响
解决方法：尽量避免对对象的表示和实现的依赖
例如：Abstract Factory, Bridge, Memento, Proxy
5）算法依赖
依赖于特定算法的设计在算法发生变化时也要变化。

解决方法：有可能发生变化的算法应该被独立出来
例如：Builder, Iterator, Strategy, Template Method, Visitor
6）紧耦合
缺点：紧耦合的对象或类互相依赖和关联，不容易修改和扩展
解决方法：尽量采用松散耦合
例如：Abstract Factory, Command, Facade, Mediator, Observer, Chain of Responsibility
7）通过生成子类来扩充功能
缺点：生成子类要求超类的内部实现对于子类是可见的；过多采用子类可能导致“子类爆炸”。
解决方法；可以采用对象组合的方法，但对象组合可能导致设计难于理解，因此可以综合使用继承和对象组合。
例如：Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy.
8）不能方便的对类进行修改
缺点：由于无法得到源码而无法修改一个类，或对一个类的修改将影响许多其它现有的子类。
解决方法：可以用对象组合的方法增加或修改现有对象的特征
例如：Adapter，Decorator, Visitor
设计模式在软件开发的作用
总之，设计模式有助于增强软件的灵活性，这种作用在三种主要的软件中都有所体现。
应用程序
工具箱
框架
三种软件各有特点，对于灵活性（包括复用性、可维护性、可扩充性等等）的要求依次增强，设计难度依次上升
设计模式的应用将有助于提高这些软件的灵活性、可复用性和平台独立性。
关于框架的补充说明
框架是软件开发过程中提取特定领域软件的共性部分形成的，不同领域有不同的框架。
新项目在框架的基础上开发，这样可提高软件的质量，降低成本，缩短开发时间。
框架不是现成可用的应用系统，而是半成品，需要二次开发。
框架是面向应用问题的（例如Junit就是一个框架），而设计模式是面向设计问题的（例如用户界面怎样访问数据模型）；
框架用特定语言实现，而设计模式不依赖于特定语言。
1.7 怎样选择设计模式
理解设计模式：
设计模式怎样解决问题、每个模式的意图、模式之间的相互关联、模式怎样避免重新设计、哪些模式可以允许哪些改变而不需要重新设计。
表1-2：设计模式所支持的设计的可变方面
1.8 怎样使用设计模式
理解每个模式
研究其结构，参与者和协作关系，研究代码实例
把模式应用于具体应用
选择与应用有关的名称，定义有关的类、继承关系和接口，实现具体的操作。
使用设计模式需要进行权衡，特别是灵活性和性能方面。
本章小结
什么是设计模式
设计模式的描述和分类
设计模式的作用
第2章 实例研究
软件测试的框架
软件测试是软件开发的重要工作。
朴素的测试经常在程序中各个位置加入print语句判断和跟踪程序的执行，或者使用Debugger这样的调试工具。
导致的问题：
测试和调试不够系统，带有随意性
不容易保留和重现以前的测试
…
单元测试
软件是由软件模块组成的，每个软件模块被看成一个单元。
在面向对象软件系统中，可以把软件对象看作构成软件的单元。
单元测试
把每个软件对象看成一个单元，为其构造测试用例，测试在各种情况下该对象的执行情况。
回归测试
为了避免修改引入新的错误，在软件修改之后，重新执行与修改部分相关的测试。
方便程序员边开发边测试的工作模式
可以随着开发过程增量建立测试
通过测试评估开发进度、发现新的开发导致的副作用。
JUnit
一个基于java的自动执行回归测试的软件框架
注意并不是一个完整的应用程序
这里以Junit3.8.1为例。
JUnit 4的新特性主要是针对JAVA5中的标记(annotation)来简化测试，而不是利用子类、反射或命名机制。

先从一个简单的Money类开始

```
class Money {
  private int fAmount;
  private String fCurrency;
  public Money(int amount, String currency) {
```

```
    fAmount= amount;
    fCurrency= currency;
  }
  public int amount() {       return fAmount;    }
  public String currency() {  return fCurrency;    }
}
```

在上述基础上进一步开发
增加一个方法实现同币种的加法

```
public Money add(Money m) {
   return new Money(amount()+m.amount(),
currency());
}
```

采取边开发边测试的策略，增加这个方法后，需要验证：
这个方法本身是否正确？在原来的Money类中加入这个方法后，是否会影响原有的其它方法的正确性？

基本的方法就是测试，怎么测试？
临时写一个测试程序

```
public class Test {
   public static void main(String args[]) {
     Money m12CHF= new Money(12, "CHF");
     Money m14CHF= new Money(14, "CHF");

     Money expected= new Money(26, "CHF");
     Money result= m12CHF.add(m14CHF);
     if(expected.amount() == result.amount())
        System.out.println(“ok”);
     else System.out.println(“false”);
   }
}
```

仔细分析，看这个程序可以分成几个逻辑部分？
单元测试程序的一般结构
驱动模块（例如一个main方法）
测试的主体部分
步骤一：设置测试环境
创建和初始化测试中需要用到的对象；
例如：m12CHF， m14CHF
步骤二：执行测试，
执行在步骤一中创建的对象需要测试的方法；
检验并输出测试结果；
步骤三：清除测试造成的影响；
有时候需要释放占用的系统资源，恢复被修改的数据。
问题
考虑测试的主体部分
既然测试的主体部分具有通用的步骤顺序，是否能够按这样的步骤构造一个框架
保证测试人员开发的测试用例遵循测试程序的一般步骤。
简化工作
测试程序的开发者只需要填写每个步骤的具体工作。

想起什么模式？
解决方案
把测试用例抽象成一个类TestCase
采用模板方法模式设定测试的基本过程
在TestCase中定义
模板方法runBare()
三个步骤方法setUp, runTest和tearDown
runBare方法顺序调用setUp, runTest和tearDown方法执行测试的三个步骤。
允许在TestCase的子类中重新定义三个步骤方法的实现。
初步的TestCase类定义

```
public abstract class TestCase {
   public void runBare() throws Throwable {
                  setUp();
                  runTest();
                  tearDown();
   }
   protected void setUp() throws Exception {}
   protected void runTest() throws Throwable {}
   protected void tearDown() throws Exception {}
}
```

基于TestCase类开发测试程序
用户开发的具体测试作为TestCase的子类，覆盖setUp, runTest和tearDown中需要修改的方法。
仍以Money类的测试为例

基于TestCase的Money测试程序—初始化部分

```
public class MoneyTest extends TestCase {
  private Money m12CHF;
  private Money m14CHF;
  protected void setUp() {
     m12CHF= new Money(12, "CHF");
     m14CHF= new Money(14, "CHF");
  }
```

定义具体测试方法

```
public void testSimpleAdd() {
  Money expected= new Money(26, "CHF");
  Money result= m12CHF.add(m14CHF);
  if(expected.amount() == result.amount())
        System.out.println(“ok”);
  else System.out.println(“false”);
 }
```

还可以定义共享同一种初始设置的其它测试方法
例如定义另一个测试方法

```
 public void testEquals() {
    if(m12CHF.amount() ==  (new  Money(12,
“CHF”)))
       System.out.println(“ok”);
```

```
    else System.out.println("false");
}
```
问题
一个TestCase中可能有很多目的不同的测试方法，例如前面的MoneyTest中有testSimpleAdd()和testEquals() 方法
可能有时需要执行testSimpleAdd()，另一些时候需要执行testEquals()
但在runBare()方法中执行的是runTest()方法
runTest()方法怎样适应这种多变的情况?
想到什么模式?
进一步考虑
类模式还是对象模式?
TestCase中的元素和模式中的元素是怎样对应的?
解决方案
定义MoneyTest的子类，在其中重新定义runTest方法
例如
```
 TestCase test= new MoneyTest("simple add")
 {
 public void runTest() {    testSimpleAdd();  }
 };
```
这段代码通过继承建立MoneyTest的一个匿名子类，并构造这个子类的一个对象。
这个匿名子类覆盖了runTest方法
setUp和tearDown仍然采用MoneyTest的实现
子类中的runTest方法
类似的，可以覆盖runTest方法测试其它的方法。
例如
```
public void runTest() {
    testEquals();
}
```
执行TestCase
前面通过在具体TestCase的子类中覆盖runTest方法来实现测试用例的执行，这种方式在Junit中称为静态执行
在Junit中还有一种称为动态执行的方式
问题
静态执行每个测试方法时，都要建立MoneyTest的一个子类
这些子类只有runTest方法的实现不同。
而大部分的runTest都是简单调用测试方法，例如：
```
    public void runTest() {
        testSimpleAdd();
    }
```
是否可以不必每次重写runTest?
问题的关键
由于每次测试都通过runTest，是否能够让runTest自动识别和调用需要测试的方法?
也就是在运行中能够得到对象的类定义
解决方案
TestCase的具体子类不必重新定义runTest方法，则定义在TestCase类中的缺省runTest将在执行runBare时被调用
缺省runTest将自动提取并调用指定TestCase的子类中与TestCase的fName属性同名的方法
fName通过TestCase的构造方法设定
```
TestCase(String name) { fName = name; }
```
runTest缺省实现的简化版
```
  void runTest() {
    assertNotNull(fName);
    Method runMethod =
      getClass().getMethod(fName,null);
    runMethod.invoke(this, new Class[0]);
  }
```
这里利用了Java的Reflection功能实现动态提取和执行方法。
附： Reflection
由java.lang.reflect包实现的允许一个java程序检查其自身或其它java类结构的功能：
确定对象的类
查询类或接口中的方法（名称、修饰符、参数类型、返回值类型…）
构造一个类的新的实例；
调用类或对象的方法；
…
动态执行TestCase
通过缺省runTest执行测试不再需要建立新的子类
```
TestCase test= new MoneyTest("testSimpleAdd");
```
执行这个MoneyTest对象的runTest方法时，testSimpleAdd方法将被执行
```
TestCase test= new MoneyTest("testEquals");
```
执行这个MoneyTest对象的runTest方法时，testEquals方法将被执行
问题
怎样了解测试执行结果以及执行中发生的情况?
主要是了解测试执行过程中发生的错误和测试结果
解决方案
测试执行中发生的情况可以分为两类
可预知并可进行主动探测的错误和其它情况。
例如在MoneyTest中测试加法操作，在返回时检查结果与预期是否一致。
由于预知可能发生的位置，可以在这个位置进行探测
不可预知的错误
例如数组越界，数组越界很可能发生在被测类的内部。
开发测试用例时一般不会在被测类的内部增加代码，而只能在被测类之外进行监控
怎么办?

提示
在Java程序执行过程中如果发生异常，Java虚拟机会处理这个异常并抛出错误。
如果这段程序被包含在一个try/catch结构中，抛出的异常就会被捕获。
探测不可预知的错误
可以把TestCase的执行包含在一个try/catch结构中
在执行过程中发生的异常会被捕获
对捕获的异常做进一步的分类处理
第2章 实例研究
软件测试的框架
软件测试是软件开发的重要工作。
朴素的测试经常在程序中各个位置加入print语句判断和跟踪程序的执行，或者使用Debugger这样的调试工具。
导致的问题：
测试和调试不够系统，带有随意性
不容易保留和重现以前的测试
…
测试用例
根据所要测试程序的特点，设计运行场景（输入数据、环境配置、驱动模块、桩模块等），以便发现错误
需要大量的测试用例。
需要能够成批运行测试用例
例如每次对程序修改后都能够重新运行以前的测试用例以保证没有引入新的错误。
单元测试
软件是由软件模块组成的，每个软件模块被看成一个单元。
在面向对象软件系统中，可以把软件对象看作构成软件的单元。
单元测试
把每个软件对象看成一个单元，为其构造测试用例，测试在各种情况下该对象的执行情况。
回归测试
为了避免修改引入新的错误，在软件修改之后，重新执行与修改部分相关的测试用例。
方便程序员边开发边测试的工作模式
可以随着开发过程增量建立测试用例
通过测试用例评估开发进度、发现新的开发导致的副作用。
JUnit
一个基于java的自动执行回归测试的软件框架
注意并不是一个完整的应用程序
这里以Junit3.8.1为例。
JUnit 4的新特性主要是针对JAVA5中的标记(annotation)来简化测试，而不是利用子类、反射或命名机制。

JUnit基本思路和结构

一个通用的测试驱动模块，执行各种统一接口的测试用例，收集测试用例执行时的各种反馈信息，对这些反馈信息的后期处理，例如显示、分析等等。
因此基本上是三块内容，即驱动部分、测试用例部分、反馈信息的后期处理部分。
恰好可以与控制器、模型、视图对应。
其中的重点在于模型，也就是测试用例
控制器和视图保持与模型的松散耦合，只要保持接口不变，后续的进一步改进不会影响到模型，所以这两部分在一开始可以做得比较简单，这也有利于控制软件开发的成本和进度。
从测试用例开始
一开始大致的思路应该就只有这些，进一步的设计需要进一步的细化研究。
首先研究测试用例的特点，怎样设计测试用例来满足上述需求

设计问题：以开发多币种算术运算为例
先从一个简单的Money类开始

```
class Money {
  private int fAmount;
  private String fCurrency;
  public Money(int amount, String currency) {
     fAmount= amount;
     fCurrency= currency;
  }
  public int amount() {        return fAmount;    }
  public String currency() {   return fCurrency;    }
}
```

在上述基础上进一步开发
增加一个方法实现同币种的加法

```
public Money add(Money m) {
   return new Money(amount()+m.amount(),
currency());
}
```

采取边开发边测试的策略，增加这个方法后，需要验证：
这个方法本身是否正确？在原来的Money类中加入这个方法后，是否会影响原有的其它方法的正确性？

基本的方法就是测试，怎么测试？
临时写一个测试程序

```
public class Test {
   public static void main(String args[]) {
      Money m12CHF= new Money(12, "CHF");
      Money m14CHF= new Money(14, "CHF");

      Money expected= new Money(26, "CHF");
      Money result= m12CHF.add(m14CHF);
```

```
    if(expected.amount() == result.amount())
        System.out.println("ok");
    else System.out.println("false");
  }
}
```

仔细分析，看这个程序可以分成几个逻辑部分?
单元测试程序的一般结构
驱动模块（例如一个main方法）
测试的主体部分
步骤一：设置测试环境
创建和初始化测试中需要用到的对象;
例如：m12CHF， m14CHF
步骤二：执行测试，
执行在步骤一中创建的对象需要测试的方法;
检验并输出测试结果;
步骤三：清除测试造成的影响;
有时候需要释放占用的系统资源，恢复被修改的数据。
问题
考虑测试的主体部分
既然测试的主体部分具有通用的步骤顺序，是否能够按这样的步骤构造一个框架
保证测试人员开发的测试用例遵循测试程序的一般步骤。
简化工作
测试程序的开发者只需要填写每个步骤的具体工作。

想起什么模式?
解决方案
把测试用例抽象成一个类TestCase
采用模板方法模式设定测试的基本过程
在TestCase中定义
模板方法runBare()
三个步骤方法setUp, runTest和tearDown
runBare方法顺序调用setUp, runTest和tearDown方法执行测试的三个步骤。
允许在TestCase的子类中重新定义三个步骤方法的实现。
初步的TestCase类定义

```
public abstract class TestCase {
  public void runBare() throws Throwable {
                setUp();
                runTest();
                tearDown();
  }
  protected void setUp() throws Exception {}
  protected void runTest() throws Throwable {}
  protected void tearDown() throws Exception {}
}
```

基于TestCase类开发测试程序
用户开发的具体测试作为TestCase的子类，覆盖setUp, runTest和tearDown中需要修改的方法。
仍以Money类的测试为例

基于TestCase的Money测试程序—初始化部分

```
public class MoneyTest extends TestCase {
  private Money m12CHF;
  private Money m14CHF;
  protected void setUp() {
     m12CHF= new Money(12, "CHF");
     m14CHF= new Money(14, "CHF");
  }
```

定义具体测试方法

```
public void testSimpleAdd() {
  Money expected= new Money(26, "CHF");
  Money result= m12CHF.add(m14CHF);
  if(expected.amount() == result.amount())
        System.out.println("ok");
  else System.out.println("false");
 }
```

还可以允许多个测试方法共享同一种初始设置
例如定义另一个测试方法

```
 public void testEquals() {
    if(m12CHF.amount() ==  (new  Money(12,
"CHF")))
       System.out.println("ok");
    else System.out.println("false");
 }
```

问题
现在需要定义runtest()方法
但一个TestCase中可能有很多目的不同的测试方法，例如前面的MoneyTest中有testSimpleAdd()和testEquals() 方法
可能有时需要执行testSimpleAdd()，另一些时候需要执行testEquals()
但在runBare()方法中执行的是runTest()方法
runTest()方法怎样适应这种多变的情况?
解决方案
定义MoneyTest的子类，在其中重新定义runTest方法
例如

```
 TestCase test= new MoneyTest("simple add")
 {
 public void runTest() {   testSimpleAdd();  }
 };
```

这段代码通过继承建立MoneyTest的一个匿名子类，并构造这个子类的一个对象。
这个匿名子类覆盖了runTest方法
setUp和tearDown仍然采用MoneyTest的实现
子类中的runTest方法
类似的，可以覆盖runTest方法测试其它的方法。

例如
```
public void runTest() {
    testEquals();
}
```
问题
前面这种实现runTest()采用的是什么模式?
类模式还是对象模式
TestCase中的元素和模式中的元素是怎样对应的?
执行TestCase
前面通过在具体TestCase的子类中覆盖runTest方法来实现测试用例的执行，这种方式在Junit中称为静态执行
在Junit中还有一种称为动态执行的方式
问题
一个具体的TestCase中可能定义了多个测试方法以便共享setUp和tearDown。
例如MoneyTest中的testSimpleAdd和testEquals
静态执行每个测试方法时，都要建立MoneyTest的一个子类
这些子类只有runTest方法的实现不同。
而大部分的runTest都是简单调用测试方法，例如：

```
public void runTest() {        testSimpleAdd();    }
```

是否可以不必每次重写runTest?
问题的关键
由于每次测试都通过runTest，是否能够让runTest自动识别和调用需要测试的方法?
也就是在运行中能够得到对象的类定义
解决方案
TestCase的具体子类不必重新定义runTest方法，则定义在TestCase类中的缺省runTest将在执行runBare时被调用
缺省runTest将自动提取并调用指定TestCase的子类中与TestCase的fName属性同名的方法
fName通过TestCase的构造方法设定
```
TestCase(String name) { fName = name; }
```
runTest缺省实现的简化版
```
  void runTest() {
    assertNotNull(fName);
    Method runMethod =
      getClass().getMethod(fName,null);
    runMethod.invoke(this, new Class[0]);
  }
```

这里利用了Java的Reflection功能实现动态提取和执行方法。
附： Reflection
由java.lang.reflect包实现的允许一个java程序检查其自身或其它java类结构的功能：
确定对象的类
查询类或接口中的方法（名称、修饰符、参数类型、返回值类型…）
构造一个类的新的实例；
调用类或对象的方法；
…
动态执行TestCase
通过缺省runTest执行测试不再需要建立新的子类
```
TestCase test= new MoneyTest("testSimpleAdd");
```
执行这个MoneyTest对象的runTest方法时，testSimpleAdd方法将被执行
```
TestCase test= new MoneyTest("testEquals");
```
执行这个MoneyTest对象的runTest方法时，testEquals方法将被执行
问题
怎样了解测试执行结果以及执行中发生的情况?
主要是了解测试执行过程中发生的错误和测试结果
解决方案
测试执行中发生的情况可以分为两类
可预知并可进行主动探测的错误和其它情况。
例如在MoneyTest中测试加法操作，在返回时检查结果与预期是否一致。
由于预知可能发生的位置，可以在这个位置进行探测
不可预知的错误
例如数组越界，数组越界很可能发生在被测类的内部。
开发测试用例时一般不会在被测类的内部增加代码，而只能在被测类之外进行监控
怎么办?
提示
在Java程序执行过程中如果发生异常，Java虚拟机会处理这个异常并抛出错误。
如果这段程序被包含在一个try/catch结构中，抛出的异常就会被捕获。
探测不可预知的错误
可以把TestCase的执行包含在一个try/catch结构中
在执行过程中发生的异常会被捕获
对捕获的异常做进一步的分类处理
统一处理可预知和不可预知的情况
可预知和不可预知的情况都是在测试中需要处理的
最好能够以统一的方式收集这些情况，以统一的方式处理。
问题
怎么统一?
提示
try/catch结构捕获的异常也可以由程序主动抛出。
解决方案
可预知的情况可以通过在程序中适当的地方插入断言（Assert）进行探测
遇到错误就抛出AssertionFailedError

见Assert.java, AssertionFailedError.java, ComparisonFailure.java
为了方便写断言，Junit提供了Assert类，其中定义了各种简单类型对象的相等判断、真假判断的断言。

TestCase中需要使用这些断言探测错误并产生事件，所以要继承Assert类。
断言方法举例

```
static public void assertEquals(String message, String expected, String actual) {
  if (expected == null && actual == null)
    return;
  if (expected != null && expected.equals(actual))
        return;
  throw new ComparisonFailure(message, expected, actual);
}
```

更完善的TestCase开发过程
继承TestCase；
定义存放测试状态设置的实例变量；
覆盖setUp方法初始化测试状态；
如果有必要，覆盖tearDown清除测试设置；
定义测试方法操纵实例变量并通过断言探测错误；
可以覆盖runTest以静态方式执行测试方法，也可以不覆盖runTest以动态方式执行测试方法。

例如一个完整的MoneyTest如下
MoneyTest

```
public class MoneyTest extends TestCase {
  private Money m12CHF;
  private Money m14CHF;
  protected void setUp() {
     m12CHF= new Money(12, "CHF");
     m14CHF= new Money(14, "CHF");
  }
  public void testSimpleAdd() {
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertEquals(result.fAmount,expected.fAmount);
 }
 public void runTest() {
   testSimpleAdd();
 }
}
```

统一的测试过程监控
把程序执行过程中的各种情况统一作为事件
对可预知的情况做主动探测，发生时抛出事件。
不可预知的异常情况则由java虚拟机处理，一旦发生，java虚拟机将抛出异常

外围的try/catch结构以统一的方式捕获这些事件进行处理
TestResult
TestResult.runProtected方法中的try catch结构实现对监控事件的捕获。
在try部分执行Protectable接口的匿名子类的protect方法，这个方法调用当前具体TestCase的模板方法runBare
在catch部分捕获try部分抛出的所有异常。包括：
AssertionFailedError
ThreadDeath
其它Throwable
Protectable接口的匿名子类

```
Protectable p= new Protectable() {
        public void protect() throws Throwable {
                test.runBare();
        }
}
```

runProtected源代码

```
public void runProtected(final Test test, Protectable p) {
        try {
                p.protect();
        }
        catch (AssertionFailedError e) {
                addFailure(test, e);
        }
        catch ……
}
```

问题
怎样扩充对事件的响应功能
不应该把这些功能全都放在TestResult中，否则TestResult会变得过于复杂
不容易增加新的响应功能
想到什么模式?
注册事件处理对象
需要对执行具体TestCase过程中各种事件做出响应的对象可以在TestResult对象上注册。
TestResult用一个向量保存这些处理器对象。
当各种事件发生时，TestResult先记录这些事件，然后向注册的对象发出信息，由这些对象作出相应的处理。
这些注册对象并不与当前执行的具体TestCase直接联系，而是由TestResult控制，相当于对TestResult的状态变化做出响应。而TestResult的状态变化则由具体TestCase执行过程的事件引起。
TestListener
注册对象必须实现TestListener接口，通过这个接口中的方法接收各种事件：
startTest
处理一个测试方法开始执行的事件

addFailure
处理出现一个预期错误的事件
addError
处理出现一个不可预期错误的事件
endTest
处理一个测试方法执行结束的事件
TestListener源代码

```
public interface TestListener {
        public void addError(Test test, Throwable t);
        public void addFailure(Test test,
AssertionFailedError t);
        public void endTest(Test test);
        public void startTest(Test test);
}
```

TestResult与注册对象的关系
增加或删除收听对象：
TestResult.addListener
TestResult.removeListener
向注册的收听对象发送消息：
TestResult.startTest
开始一个测试方法的执行
TestResult.addFailure
出现一个预期错误
TestResult.addError
出现一个不可预期错误
TestResult.endTest
一个测试方法执行结束
增加或删除收听对象

```
public synchronized void addListener(TestListener
listener) {
        fListeners.addElement(listener);
}

public synchronized void removeListener(TestListener
listener) {
        fListeners.removeElement(listener);
}
```

endTest源代码

```
public void endTest(Test test) {
   for (Enumeration e=
        cloneListeners().elements();
                e.hasMoreElements(); )
   {
      ((TestListener)e.nextElement()). endTest(test);
         }
}
```

一个最简单的TestListener
监控每个TestCase的整个执行过程，并记录执行的情况。要记录的情况包括：
开始一次测试的执行
运行过程中出现预期的错误（例如计算结果不正确）

运行过程中出现不可预期的错误（例如数组越界）
测试用例结束执行
具体代码

```
public class ResultPrinter implements TestListener
{ ……
        public void addError(Test test, Throwable t) {
                getWriter().print("E");             }
        public void addFailure(Test  test,
    AssertionFailedError t) {
        getWriter().print("F");             }
        public void endTest(Test test) { }
        public void startTest(Test test) {
                getWriter().print(".");          }
}
```

问题
TestResult和TestListener之间是什么关系？
与相应的模式的元素之间是怎样对应的？
观察者模式在Junit中的实现
Subject（同时也是ConcreteSubject）
由TestResult类实现。
其中的addListener，removeListener对应attach，detach。
startTest，addFailure，addError，endTest对应notify
Observer
对应TestListener接口。
其中的startTest，addFailure，addError，endTest对应update方法。
ConcreteObserver
可以有多种，ResultPrinter类实现的是最简单的一种，以文本方式和特定的格式显示测试运行过程。
问题
到这里，系统的雏形已经有了，进一步看怎样改进。

无论执行被覆盖的还是缺省的runTest方法，TestCase一次只执行一个测试方法：
例如：
testResult.run(new MoneyTest(“testSimpleAdd”))
怎样成批测试TestCase？
什么模式能够把多个对象组合在一起？能否采用？
一次测试多个TestCase
解决方案：
把多个TestCase对象包装成一个TestSuite对象。执行一个TestSuite对象时，依次执行其中的每个TestCase对象。
问题：怎样一次测试多个TestSuite对象？
一次测试多个TestSuite对象
解决方案：

多个TestSuite对象也可以像TestCase对象一样被包装成一个TestSuite对象。
因此TestSuite和TestCase应该实现共同的接口Test，外部对象不必知道所操纵的究竟是一个TestSuite还是TestCase，由具体被操纵的对象实现接口的方法决定具体的操作。
TestSuite中TestCase的组织
TestSuite用一个向量（Vector fTests）保存对多个实现了Test接口的对象的引用。
可以是独立的TestCase
也可以是TestSuite
即TestSuite中可以包含（引用）TestSuite
TestSuite以相同的方式对待保存在其中的TestCase和TestSuite
把它们都看成是实现了Test接口的实例。
Test接口中只有两个方法：countTestCases和run(TestResult result)。
TestSuite中的countTestCases()

```
public int countTestCases() {
        int count= 0;
        for (Enumeration e= tests();
         e.hasMoreElements(); ) {
                Test test= (Test)e.nextElement();
                count= count + test.countTestCases();
    }
        return count;
}
```

TestSuite中的run方法

```
public void run(TestResult result) {
        for (Enumeration e= tests();
        e.hasMoreElements(); ) {
                Test test= (Test)e.nextElement();
                runTest(test, result);
        }
}
```

TestCase中的Test接口

```
public int countTestCases() {
        return 1;
}

public void run(TestResult result) {
        result.run(this);
}
```

问题
TestSuite采用的是什么模式?
与模式中的部件怎样对应?
组合模式在Junit中的实现
Component对应Test接口；
Java在这里倾向于使用接口而不是抽象类。
Leaf对应TestCase；
Composite对应TestSuite
用一个向量fTests引用所包含的TestSuite或TestCase。

addTest方法对应Composite的add方法
新对象只要实现Test接口，包括TestCase或TestSuite

run方法依次找到fTests中每个元素，逐一调用runTest方法执行。
runTest方法调用具体元素本身的run方法执行。
如果这个元素是TestSuite，就成为递归结构。
使用TestSuite
如果当前有一个TestCase类—MoneyTest，其中定义了两个测试方法testSimpleAdd和testEquals：

```
TestSuite suite = new TestSuite();
Suite.addTest(new MoneyTest("testEquals"));
Suite.addTest(new MoneyTest("testSimpleAdd"));
TestResult result = suite.run();
```

问题：如果MoneyTest中有100个测试方法，就要coding同样次数的Suite.addTest(new …)，可不可以有更好的方法?
自动构造一个TestSuite
TestSuite提供了一个构造方法，可以通过参数接受一个TestCase类，自动构造一个TestSuite对象
自动提取其中（包括其实现了Test接口的超类）中所有以"test"开头并且没有输入参数和返回结果的公开方法
以每个这样的方法名作为TestCase名建立一个当前TestCase的实例，用addTest方法把这个实例加入fTests中。
显然这些自动建立的TestCase实例执行缺省runTest方法。
见TestSuite.java的TestSuite(final Class theClass) 方法
代码

```
public TestSuite(final Class theClass) {
   …
        Class superClass= theClass;
        Vector names= new Vector();
        while
(Test.class.isAssignableFrom(superClass)) {
          Method[] methods=
superClass.getDeclaredMethods();
     for (int i= 0; i < methods.length; i++) {
        addTestMethod(methods[i], names, theClass);
     }
     superClass= superClass.getSuperclass();
        }
   …
}

addTestMethod(Method m, Vector names, Class
theClass) {
```

```
    String name= m.getName();
    if (names.contains(name))  return;
    if (! isPublicTestMethod(m)) {
                              … //错误处理
    }
    names.addElement(name);
    addTest(createTest(theClass, name));
}

createTest(Class theClass, String name) {
  Constructor constructor;
          try {
     constructor= getTestConstructor(theClass);
          } catch (NoSuchMethodException e) { … }

  Object test;
          try {
    …
    test=constructor.newInstance(new Object[]
{name});
   } catch (…) {…}
          return (Test) test;
}
```

这样可以更方便地使用TestSuite

如果要测试一个TestCase中的全部test方法：

直接用一个TestCase类实例化

```
TestSuite suite= new TestSuite(MoneyTest.class);
TestResult result = suite.run();
```

问题

参考MVC结构全面考虑整个系统：

M：TestCase，TestSuite等。

V：对测试用例执行时的各种反馈信息的处理部分。

C：驱动程序。

之前主要构造M部分，现在重点转向C和V。

先确定C与M之间的接口：

如果每次把要执行的测试任务包装成一个固定接口的对象提供给C执行，就可以让M部分与C部分相互独立。

想到什么模式?

解决方案

把每次要执行的测试构造成一个TestSuite提交给驱动程序

驱动程序只要调用TestSuite的run方法，所有的测试用例就都可以执行

先看TestCase怎样能够构造出这样一个TestSuite对象

TestCase的suite方法

```
public static Test suite() {
          TestSuite suite= new TestSuite();
             suite.addTest(
          new  MoneyTest(" testSimpleAdd "));
             suite.addTest(
          new MoneyTest("testEquals"));
       return suite;
}
```

测试框架程序与suite方法的关系

通过运行suite方法就能够构造出一个TestSuite对象提供给测试框架执行。

不同的suite方法产生不同的TestSuite对象。

TestRunner

根据每次执行时给定的具体TestCase类的名称，自动装载并执行。

执行前先查找装载的类是否有suite()方法

如果有就执行这个方法，这个方法将返回一个TestSuite对象；

如果没有，就把当前的TestCase类传递给TestSuite的缺省构造方法构造一个TestSuite对象。

对返回的TestSuite对象调用其run 方法执行测试。

具体代码

以junit.textui.TestRunner.java为例：

TestRunner中的具体代码

```
TestResult  start(String args[]) {
      …//从输入args中得到要测试的TestCase
      …//的类名称，保存在字符串testCase中
      Test suite= getTest(testCase);
      return doRun(suite, wait);
      …
}
```

其中，getTest是在TestRunner的超类--BaseTestRunner中定义的方法。

getTest中的有关代码

```
… //根据类名把类定义装载到testClass中
try {   suiteMethod=
testClass.getMethod(SUITE_METHODNAME, new
Class[0]);
} catch(Exception e) {   …
          return new TestSuite(testClass);
}
…
test= (Test)suiteMethod.invoke(null, new Class[0]);
…
return test;
```

Junit中的Command角色

Command：Test

ConcreteCommand

具体构造出的TestSuite对象，执行run方法。

Client

可以把整个Junit看作Client。

Invoker：TestRunner

Receiver

TestSuite中包含的所有TestCase对象。
采用Command模式的效果
责任分解
TestCase的开发人员不必关心TestCase将会被提交给什么程序执行，只要能够构造一个TestSuite对象，在这个TestSuite对象中确保对各个测试的执行。
能够自动执行TestSuite的测试框架程序不需要知道具体TestSuite的内部执行过程，只要调用接口上的run方法就可以了。
完整的测试执行
从命令行执行：
Java TestRunner　TestCaseClass
执行过程（采用了Java的动态类装载技术）：
检查传入的TestCaseClass是否有suite方法
如果有，则调用这个方法，构造出一个TestSuite对象。
否则，用TestCaseClass构造出一个TestSuite对象。
调用TestRunner的run(Test)方法执行。
例如：java -cp . junit.textui.TestRunner junit.samples.SimpleTest1
问题
到这里，控制器和视图也初步构造好了，同样考虑进一步改进
基于文本的界面对用户不够友好，是否可以改成图形界面?
AWT
Swing
…
当然可以，问题是要做多少修改?
解决方案
系统构建时应该尽量降低各部分的耦合度，特别是系统所要处理的数据结构和具体的处理逻辑不要与显示以及用户输入控制有太多关联。
明确各部分的功能和相互间的接口。
Junit中有哪些部分，分别的功能、接口?
Junit中的MVC部件
Model
?
View
?
Controller
?
Junit中的MVC部件
Model
TestCase, TestSuite, TestResult
View
TestRunner中的显示测试结果的部分
实现了TestListener接口的部分对应MVC结构中的V
目前为止是文本方式显示测试结果
通过Observer模式建立与Model的联系
Controller
TestRunner接受和分析用户输入，然后启动测试
使用了Command模式与Model交互
使用Test接口
可用于TestCase、TestSuite，或其它实现了这个接口的对象
改进Junit的用户界面
在通过最简的字符方式实现了视图和控制器后，考虑改进用户界面：
首先提取各种用户界面的共同功能，构造抽象类BaseTestRunner
关键是其中的getTest方法
给定TestCase类后，或者执行指定类的suite方法，或者提取指定的TestCase中的各种测试构造一个TestSuite对象。
有了这个核心功能与模型部分衔接，外围的用户界面就可以比较自由地增加新的功能
例如可以利用AWT、Swing的特点增加各种特色
注意到BaseTestRunner也实现了TestListener接口，这就为控制器和视图的集成提供了条件。
虽然junit.textui.TestRunner中没有用这个特征。
各种用户界面执行的过程
见各种TestRunner的main和start，runSuite方法
从参数中分析出TestCase的类名
建立用户界面窗口及其中部件
调用getTest根据TestCase的类名取得Test对象
执行测试
建立一个TestResult对象
在TestResult对象中注册视图对象
执行Test的run方法，并把TestResult对象作为参数传入，以便收集执行过程中的事件
各种用户界面的测试执行方法
TestRunner中构造好一个listener，一个TestResult和一个Test，把listener注册在TestResult中，然后执行Test的run方法。
junit.textui.TestRunner的doRun方法。
junit.awtui.TestRunner的runSuite方法。
junit. swingui.TestRunner的doRunTest方法。
junit.textui.TestRunner的doRun法

```
public TestResult doRun(Test suite, boolean wait) {
                TestResult result= createTestResult();
                result.addListener(fPrinter);
                …
                suite.run(result);
                …
                return result;
        }
```

参考前面的start(String args[]) 方法
junit.awtui.TestRunner的runSuite方法

```
Test testSuite=getTest(fSuiteField.getText());
fRunner= new Thread() {
```

```
 public void run() {
        fTestResult= createTestResult();
        fTestResult.addListener(TestRunner.this);
        …
        testSuite.run(fTestResult);
        ...         }
 };
 fRunner.start();
```

junit.swingui.TestRunner的doRunTest方法

```
doRunTest(final Test testSuite) {
    fRunner= new Thread("TestRunner-Thread") {
          public void run() {
                ...
                testSuite.run(fTestResult);
           …  }
    }
        fTestResult= createTestResult();
        fTestResult.addListener(TestRunner.this);
        fRunner.start();
```

通过多种用户界面使用Junit

在目录“…\sample\junit3.8.1\junit3.8.1\junit”下，从命令行调用TestRunner

```
c:\jdk1.3.1\bin\java -cp . junit.???ui.TestRunner
junit.samples.SimpleTestX
c:\jdk1.3.1\bin\java -cp . junit.???ui.TestRunner
junit.samples.AllTests
```

思考

为什么在建立多种用户界面的时候不采用创建型模式?

问题：对已有Test的扩展

已经定义好的Test可以通过一些简单的扩展应用于新的情况

例如重复执行或增加一些初始设置。

以MoneyTest为例，如果新的测试要求

重复执行100次

在开始一个或多个测试前需要附加的设置

…

方案1：建子类（以重复执行为例）

建立MoneyTest的子类RepeatMoneyTest，按参数给定重复次数，重复执行MoneyTest指定的测试方法。

缺点：对各个需要重复执行的Test都要建立子类。

通过子类扩展功能示意图

更好的方法？想到什么模式?

使用包装类

把需要附加功能的TestCase用一个包装类包装起来，可以在包装类中增加附加功能。

如果需要重复执行TestCase，就可以建立一个Repeat包装类，各种TestCase对象都可以被包装起来重复执行。

如果采用原来的继承方法，n种TestCase就需要n个子类，而这里只需要建立一个包装类。

TestDecorator

用TestDecorator的子类修改测试过程

用RepeatedTest对某个测试重复执行。

不需要建立新的子类，用具体的TestCase对象和需要重复的次数作为初始化参数实例化RepeatedTest对象。

例如SimpleTest4

用TestSetup在测试前设立附加状态

附加设置在TestSetup子类的SetUp中设置，在tearDown中清除。

与Decorator模式是怎样对应的?

Junit中Decorate模式的实现

Component

Test

ConcreteComponent

TestCase, TestSuite

Decorator

TestDecorator

ConcreteDecorator

RepeatTest, TestSetup, …（可自行增加）

见TestDecorator.java, TestSetup.java, RepeatTest.java

Junit Overview

设计模式对软件设计的指导作用

以Junit为例

起点：初步的MVC结构

C：一个执行测试程序的框架

M：测试用例

V：显示测试结果

接下来主要考虑测试用例（M）的设计。

TestCase

测试过程是程序化的

需要有测试前的状态设置（setUp）

执行测试的过程（runTest）

测试后的状态清理（tearDown）

因此为所有的测试用例定义一个顶层超类TestCase

模板方法模式的应用

为了确保所有测试都能够按setUp，runTest，tearDown这个顺序执行，在TestCase中定义了模板方法runBare，顺序调用setUp，runTest，tearDown。

允许子类覆盖setUp，runTest，tearDown

但不覆盖runBare

每次测试总是调用runBare，确保了测试程序的执行顺序和测试用例相互的独立性。

设计模式在这里为定义类继承层次结构和属性、方法的确定提供了指导。

适配器模式的使用
TestCase通过runTest实现了一个适配器
一个具体TestCase内部可以有多个testXXX方法，分别执行不同的测试。
但TestCase每次执行的都是runTest方法
设计模式在这里提供了接口之间衔接的设计指导

接下来考虑结果的显示
V：观察者模式的应用
观察者需要观察测试中发生的情况
由于这些情况已经被TestResult收集，所以可以通过观察TestResult实现。
由在TestCase中有意识地加入Assert以及TestCase执行过程中发生的例外产生事件
使用java的try-catch机制捕获事件
观察者需要实现收听接口（TestListener），并在所观察的对象（TestResult对象）上注册
这里，设计模式提供了M和V之间衔接的设计方案
M与C的衔接
测试用例可以被包装成一个对象，由测试框架调用执行。
采用Command模式衔接测试框架和被测试程序。
构造出基本的MVC结构
至此已经构造出了基本的MVC结构
Controller：TestRunner
Model：TestCase
View：实现TestListener接口
由于这三个子系统互相独立，只通过固定的接口衔接，可以分别独立改进。
先看对Model的改进
需要能够同时测试多个TestCase
采用复合结构建立TestSuite类。
TestSuite和TestCase共享相同的接口Test。
测试执行框架不必区分TestSuite和TestCase，它只通过Test接口访问提供给它的对象。
TestSuite在问题域中并没有对应物，是由于应用了Composite设计模式而引入的。
装饰TestCase
总是可以通过继承建立具有更多功能的TestCase子类，但是
子类数量的急剧增加导致管理的困难
并不是所有对象都需要这些增加的功能
可以用装饰模式实现功能的增加
装饰类在问题域中也没有对应物。
对V和C部分的改进
先是基于文本的
然后是基于AWT的
再进一步是基于Swing的
结论1：设计模式和OO设计的关系
从对Junit的分析可以发现
OO设计提供了一个一般的过程
但应该定义哪些类、属性、方法则并不明确，有时候一些关键部件，例如对改善设计起关键作用的类、属性或方法等甚至在问题域中没有直接的对应物。
设计模式则在具体怎样设计类、属性、方法、继承关系，对象的复合关系，对象互相之间的访问方式上提供了指导和可被复用的已证明是有效的模式
结论2：注意
应该以解决问题为目的，逐渐发现可用的模式，逐渐把软件设计成模式密集（Pattern Dense）的结构。

确保适合所要处理的问题，并提供灵活性。
但不应该在一开始就大量使用模式
因为一开始对于问题本身的理解还不够透彻，可能导致采用不适当的模式，会使系统混乱和失去灵活性。
不熟悉模式的人不易理解大量使用模式的设计。
很多时候，是在软件改进的过程中逐渐采用合适的模式的，例如：
Observer模式加入Junit
在Junit中，原来定义TestResult的两个子类TextTestResult和UITestResult，分别用于文本的TestRunner和图形界面的TestRunner
两种TestResult与各自对应的TestRunner之间的联系是硬编码实现的。
当新的需求要求多个对象能同时观察TestResult中的变化的时候，这种设计就不能适应了。
这时候采用了observer模式。
建立一个观察者接口TestListener；
最后发现两个子类TextTestResult和UITestResult是不必要的，只要TestResult就可以了。
小结：Junit中的设计模式
模板方法
适配器
观察者
复合
命令
装饰
MVC
第3章 创建型设计模式（Creational design patterns）
创建型模式抽象了对象的实例化过程。
确保系统尽可能独立于创建、组合和表示它的那些对象。
类创建型模式使用继承改变被实例化的类。
对象创建型模式将实例化委托给另一个对象。
这些模式隐藏了使用的具体的类，以及这些类的实例对象被创建和组合的情况。
允许使用不同的对象构造一个系统，这些对象可以是静态（编译时）指定，也可以动态（运行时）指定。
单件模式则限制某个类的实例对象只能有一个。

贯穿本章的例子—创建迷宫
迷宫由房间组成，房间由墙和门组成。
P56是一个硬编码的创建迷宫程序。
创建型模式将提高灵活性，特别是能方便地修改定义一个迷宫构件的类，方法包括：
通过一个工厂对象创建各种构件（3.1）
通过一个生成器对象创建和组装各种构件（3.2）
覆盖虚拟的工厂方法（3.3）
通过复制创建各种构件（3.4）
3.1抽象工厂
意图
提供一个创建一系列相关或相互依赖对象的接口，但不指定这个接口的实现类
别名
Kit
问题
建立一个系统能够支持多种用户界面风格（MS-Window, X-Window, Motif, …）。
用户界面的部件包括：
菜单、滚动条、按钮、…
解决方案：
（1）为每种用户界面风格开发一个完整的系统
程序量太大
（2）只开发一个系统，但可以指定不同的用户界面风格
困难在哪里?
只开发一个系统的困难
但要指定不同的用户界面风格
用什么类构造用户界面上的菜单、滚动条、按钮、…等等对象?
例如滚动条，如果指定MS-Window风格的滚动条的实现类MSScrollBar
要改成X-Window风格的滚动条怎么办?
假设X-Window风格的滚动条是由类XScrollBar实现的
什么是变化的? 什么是不变的?
程序的运行逻辑是不变的
只需要开发一次
程序的界面对象的视觉效果是随着要模拟的平台的不同而不同的
可以有多种解决方案
方案1：在每个用户界面对象中设置某些标志信息，这些对象根据标志信息显示出不同的风格
例如scrollbar的内部标志为1时显示windows风格，为2时显示xwindows风格
缺点：scrollbar的类定义会非常庞大，带来巨大的代码复杂性。
方案2：用不同的界面对象，把创建这些界面对象的工作委托给另一个对象来完成
解决方案
采用抽象工厂模式
定义一个抽象的窗口工厂类，这个类声明了一个接口，接口中定义了用于创建各种窗口部件（例如滚动条，按钮等等）的方法的型构。
为每种风格定义具体的窗口工厂类，作为抽象窗口工厂的子类，其中定义创建各种窗口部件的方法。
在具体工厂类的这些方法中再指定创建哪个类的实例对象。
模拟实现
建立一个抽象工厂和两个具体工厂类
AbsWindowFactory
MSWindowFactory
XWindowFactory
建立一个抽象产品和两个具体产品类
JTextComponent
JTextArea
JTextField
要产生不同的窗口风格只要改变具体工厂对象
例TheWindow.java
使用抽象工厂
按需要传递给客户代码不同的具体工厂对象。客户代码按照抽象工厂类定义的接口访问具体工厂对象。

每一种窗口部件（例如Scrollbar）都有一个顶层的抽象类，并且在每一窗口风格中都有对应的具体子类。
同一种部件的所有具体子类都遵循相同的顶层抽象类的接口
客户代码通过抽象类所定义的部件接口以相同的方式访问用不同子类定义的特定风格的部件。
P58 图示
适用性
允许一个系统的构造独立于它所使用的部件的创建、组合和表示
允许一个系统可以由多个部件系列之一来配置
确保部件的风格一致（同一系列）
例如MSWindow对象只应该与MSScrollBar的对象同时使用。
部件开发方可以开发遵循特定接口的部件系列，而不公开具体的实现。
使用者只知道需要调用的具体工厂，不知道工厂产生的对象的具体类
参与者
AbstractFactory
提供创建一系列部件对象的接口
ConcreteFactory
实现具体部件对象的创建
AbstractProduct
同一类型部件的接口
ConcreteProduct
具体部件对象的类，实现AbstractProduct接口
Client

客户代码，仅使用AbstractFactory和AbstractProduct接口
抽象工厂模式图示
协作关系
在运行时刻创建一个ConcreteFactory对象，由这个对象负责创建特定类型的同一系列的部件。
为了使用不同系列的部件，客户应创建和使用不同的ConcreteFactory对象。
例如创建Product2系列与创建Product1系列用的是不同的ConcreteFactory对象。
效果
客户代码和实现具体部件的一系列类相互独立
容易更换部件系列
只需要改变ConcreteFactory
容易增加新的部件系列
只要增加新的ConcreteFactory
容易保持部件的一致性
一个ConcreteFactory只创建同一系列的部件
缺点
不容易增加新的部件（例如原来没有Toolbar，现在要增加Toolbar），否则就要改变AbstractFacory的接口
怎么办?
可扩展的工厂
可以通过参数指定所要创建的对象的类
这样，一个Factory实际只需要一个创建方法
这个创建方法返回的是所有部件类都能够支持的类型
例如Java中的Object，或C++中的无类型的指针
客户程序负责对得到的对象做强制类型转换
缺点:
容易产生类型错误
问题
是否可以在两个窗口系统之间移植（而不是同一窗口系统的不同风格）？即建立一个系统能够在多种视窗平台（MS-Window, X-Window, Motif, …）中执行。
抽象工厂用于建立迷宫
一个迷宫包含一系列部件:
房间、墙壁、房间之间的门
可以有多种类型的迷宫:
由需要通过咒语开门的房间构成的迷宫
由可能包含炸弹的房间构成的迷宫
不同类型的迷宫由不同类型的具体部件实现
用于创建迷宫的抽象工厂
为不同类型的迷宫定义具体的工厂，负责创建相应迷宫的各种部件
创建迷宫的方法接受工厂对象作为参数，让工厂对象负责创建各种部件。
比较P61和P56的CreateMaze方法
P61的代码只要更换不同的factory就可以产生不同的迷宫。
抽象工厂应用举例

```
public interface Person
{
  public String sayHello(String name);
  public String sayGoodBye(String name);
}
```

American类

```
public class American implements Person
{
  public String sayHello(String name)
  {  return " Hello, " + name;  }
  public String sayGoodBye(String name)
  {  return "Good Bye, " + name; }
}
```

Chinese类

```
public class Chinese implements Person
{
  public String sayHello(String name)
  {  return name + ", 您好";  }
  public String sayGoodBye(String name)
  {  return name + ", 下次再见"; }
}
```

配置文件psnCfg.xml

```
<? xml version="1.0" encoding="gb2312">
<! DOCTYPE …>
<beans>
  <bean id="chinese" class="Chinese"/>
  <bean id="american" class="American"/>
</beans>
```

主程序

```
public class Test {
  public static void main(String[] args) {
    ApplicationContext ctx = new
FileSystemXmlApplicationContext("psnCfg.xml");
    Person p = null;
    p = (Person)ctx.getBean("chinese");
    p.sayHello("tom");
    p = (Person)ctx.getBean("american");
    p.sayHello("tom");
  }
}
```

与抽象工厂模式的比较
抽象工厂、具体工厂?
抽象产品、具体产品?
3.2 生成器设计模式（Builder）
意图
让构造一个复杂对象的过程独立于产生的对象的表达，用相同的构造过程可以创建不同的对象表达。
问题

一个RTF阅读器需要把一个RTF转换成多种文档格式：
文本文档、Doc文档、…
要求在不改变RTF阅读器的前提下能够方便地增加新的转换功能。

怎样设计这样一个阅读器?
解决方案
解决方案包含RTF阅读器和转换器两部分（p64图）

RTF阅读器实现一个固定的对RTF文档中的符号的分析过程
每当识别一个RTF标记，就发送给转换器的对应方法作处理
转换器负责转换各种标记并把转换的结果组装成一个文档或其它对象。
通过一个抽象的接口衔接RTF阅读器和各种具体的转换器。
不同的转换器将会产生不同的结果对象
与抽象工厂模式的区别
抽象工厂要创建各种部件，对这些部件的组合则交给Client
Builder则是在客户程序控制下既完成部件的创建又进行部件的组合。
不同Builder产生的对象之间可能差异很大，因此这些对象往往没有公共的超类
用于创建迷宫的生成器
为不同类型的迷宫定义具体的生成器，负责创建相应迷宫的各种部件并组装它们。
P67更加简单的客户代码
创建迷宫的方法接受生成器对象作为参数，让生成器对象负责创建各种部件并组合成完整的结果。
迷宫的各种部件之间的关系被隐藏在生成器之内。
生成器在构造结果对象的同时遍历了所有的部件，因此可以附加各种与遍历有关的功能，例如计数
3.3 工厂方法设计模式（Factory Method）
意图
（在一个相对固定的框架中）定义一个用于创建对象的接口，把实例化延迟到子类中，让子类决定实例化哪个类。
别名
虚构造器（Virtual Constructor）
类模式
问题
经常要开发这样的应用程序，其中需要管理和操作多个文档
部分任务已经固定，例如“增加新文档”的操作：
总是要创建一个文档，
然后把这个文档加入到应用程序当前管理的文档集合中
然后打开这个文档。
部分任务还不确定，特别是不知道要创建的文档的具体类型。
解决方案
建立一个框架，其中有两个主要的抽象类Application和Document。
Application中实现了固定部分的任务。
同时为可变部分留下了接口
在具体应用中需要定义抽象类Application和Document的具体子类。
例如创建一个绘图应用时要定义两个具体类DrawingApplication和DrawingDocument。
在子类DrawingApplication中才确定要创建的是DrawingDocument对象。
不变的部分仍然从抽象类Application中继承
见P71图
代码示例（p75）
不用抽象工厂或生成器，直接把工厂方法定义在MazeGame类中
3.4 原型设计模式（Prototype）
意图
用原型实例指定创建对象的种类，通过对原型对象的拷贝创建新的对象。
问题
要构造一个通用的图形编辑器框架
Tool的子类GraphicTool中引用并且操作Graphics接口的对象在编辑区绘制图形
要能够通过定制这个图形编辑器框架构造各种图形编辑器，例如word中的每种形状都可以看作是一种Tool
解决方案
在设计这个框架时，不能预测到各种定制情况下可能有哪些Graphics子类。
可以用抽象工厂模式，为每一种定制建立一个具体工厂来创建其中各种图形元素。
框架需要能访问factory接口，如果不同定制的图形元素的种类数不同，还要用到可扩展的factory的接口。
更好的方法是让每个图形元素有一个clone方法，Application通过调用图形元素对象的clone方法，可以创建任意多个这种类型的对象
而Application并不需要知道这个对象的具体类是什么。
代码示例
结合抽象工厂模式和原型模式实现的迷宫创建(P81)

简化了抽向工厂的开发。
3.5 单件设计模式（Singleton）
意图
确保系统中只实例化某个类的至多一个对象
适用性

某个类只需要实例化一个对象，并且允许通过一个全局访问点访问它。
可以用子类进行扩展，并且客户代码可以不需要更改就能用子类的实例化对象。
参与者
Singleton
定义一个Instance操作，允许客户访问这个类的唯一实例。
Instance是一个类操作（即C++和Java中的静态方法）
负责找到，也可能需要创建唯一的实例对象
结构图
P84
协作
客户程序只能通过Singleton的Instance操作得到这个唯一的Singleton实例对象
效果
因为Singleton类包装了它的唯一实例对象，所以可以严格控制对这个对象的访问。
缩小名空间：避免了太多的全局变量
允许通过子类扩展功能，同时又不影响客户程序。
可以用相同的方式控制一个类允许的实例对象的数目
比类操作灵活：可以通过类操作代替Singleton的功能，但类操作无法实现对实例对象数目的指定。
实现
保证一个唯一的实例
创建Singleton的子类
保证一个唯一的实例
常用的方法是把创建实例的操作隐藏在一个类操作中，由这个类操作控制只有一个实例被创建。
见P85的代码，注意：
构造方法不能被外部程序访问。
为什么不作为private方法?
私有的实例引用是静态的，在全局范围同类对象中唯一。
与全局或静态对象的比较
可以通过声明一个全局（或静态）对象的方式来达到Singleton的效果，但与Singleton模式相比存在这样的缺点：
责任被转移给客户程序，需要确保只声明了一次；
静态对象在程序运行的一开始就会被初始化，而这时可能还没有足够的信息用于初始化。
Singleton模式则允许在程序运行中的任意时刻创建实例对象。
程序启动时静态对象的初始化顺序是不确定的，如果这些对象之间存在依赖关系，就可能发生错误。
创建Singleton的子类
可在Singleton中指定使用的Singleton实例的类型。
必须是Singleton的某个子类，具体例子见P88的MazeFactory::Instance()。
增加或减少Singleton子类都需要修改Instance()的代码。
也可以让子类覆盖Instance方法，这样客户程序中就要指定用哪个子类，不能在程序运行中改变。
还可以使用注册表，把各种可能的Singleton子类的实例对象都创建好，用名字做索引，每次根据名字查注册表找到需要的Singleton实例对象
见p87的Instance方法
代码示例
每次创建一个迷宫时，只需要一个相应的工厂对象，可以用Singleton实现。
见MazeFactory::Instance()
可以有多种迷宫，每次可以选择所用的工厂对象。
见MazeFactory::Instance()
把所有的工厂对象类型硬编码，灵活性不如前面提到的注册表方式。
小结　创建型设计模式
纯粹用于对象创建
抽象工厂设计模式
工厂方法设计模式
原型设计模式
对象创建+组装
构造器设计模式
限定对象数目
单件设计模式
作业
见elearning的“第3章作业”的说明。
注意其中已经指定了作业截止日期。
第4章　结构型模式
组合现有的类和对象成为更大的结构
结构型类模式
通过继承来组合接口和实现，进行静态的结构组合。

例如通过多继承将多个类组合成一个类
结构型对象模式
通过对象组合来实现新的功能。
可以在运行时改变对象的组合关系，更加灵活。
结构型模式分类（1）
适配器设计模式（Adapter）
为一个对象提供一个接口，与另一个对象要求的接口一致，以便两个对象能够协作。
桥接设计模式（Bridge）
将对象的抽象和具体实现分离，从而可以独立地改变它们。
组合模式（Composite）
组合对象成为层次结构，每个对象实现相同的接口。

包括基元对象和组合对象
结构型模式分类（2）
装饰设计模式（Decorator）

可以动态地为每个对象增加新的功能，而不需要建立新的类。
外观设计模式（Facade）
用一个外观对象为一个复杂的子系统提供一个简单的接口，子系统则可能由多个对象以各种方式组成。

结构型模式分类（3）
享元设计模式（Flyweight）
用于对象的共享，目的在于节省空间
被共享的对象不应该有上下文相关的状态
代理设计模式（Proxy）
通过一个对象（Proxy）访问其它对象
可以限制、增强或修改这些访问。
4.1 适配器
意图
将一个类的接口转换成另一个接口，使原来接口不兼容的类可以一起工作。
别名
Wrapper
动机
工具箱中现成定义好的类不能够被复用在当前应用程序中的原因仅仅是因为它们之间的接口不匹配
例如
现有一个图形编辑器可以通过Shape接口访问和控制用户创建的各种图形对象。
现在要扩展这个图形编辑器的功能，使之能够访问和控制一个文本编辑窗口
给定条件如下：
现有条件
在一个现有的软件工具箱中已经有了一个类TextView用于文本的编辑和显示。
为了减少开发的工作量，希望在图形编辑器中复用TextView的功能实现文本的编辑和显示。
但是，TextView不支持Shape接口
解决方案
定义一个类TextShape，用来适配TextView和Shape接口。
方案一：采用类适配器模式
TextShape继承TextView和Shape
代码示例：p97
方案二：采用对象适配器模式
把一个TextView对象作为一个TextShape对象的一部分。
P92的图
Shape的BoundingBox请求被转换成TextView的GetExtent请求。
代码示例：p98
适用性
想要使用一个已经存在的类，而它的接口不符合要求
参与者
Client, Target, Adapter, Adaptee
Adapter将Adaptee的接口转换成Target的接口，以便Client可以访问。
有两种实现方法
类适配器
对象适配器
类适配器
Adapter 继承Adaptee并实现Target接口。
在Target接口中的方法中调用从Adaptee继承的方法实现具体功能。
例如图中的request()方法
Client仍按Target接口调用Adapter的方法。
类适配器模式结构
对象适配器
Adapter 继承Target并覆盖其接口中的方法。
Adapter同时把一个Adaptee对象作为Adapter的成员（属性）
对Target中的方法的调用被转发给Adaptee的对应方法实现具体功能。
这里采用了委托的方法
Client仍按Target接口调用Adapter的方法。
对象适配器模式结构
协作
Client向Adapter实例发出一个请求
接着适配器调用Adaptee的操作实际处理这个请求
效果
类适配器和对象适配器的比较
类适配器：
用一个具体的Adapter类适配Adaptee和Target的接口，但无法用这个类为Adaptee的子类提供适配。
Adapter可以重定义Adaptee中的操作。
对象适配器
只需要一个具体的Adapter类就可以为Adaptee及其子类提供适配。
不容易重定义Adaptee中的操作。
需要通过对象指针联系Adaptee
问题
怎样在软件设计中应用适配器模式?
举例说明什么时候会用到。
怎么用的?
使用适配器模式时需要考虑的因素
Adapter的匹配程度
设计Adapter的工作量取决于Taget与Adaptee接口的相似程度。
可插入的adapter
即设计软件时为可变的对象实现留下接口，从而增加所设计软件的灵活性。
使用双向适配器提供透明操作
同一个对象可以连接两个客户，每个客户通过不同的接口访问这个对象。
实现

可插入的适配器，实现方式包括：
先要为Adaptee找到一个“窄”接口（即需要适配的最小操作集），窄接口的3种实现方式：
在子类中实现，例如子类DirectoryTreeDisplay中实现最小接口（仅2个方法）
使用代理对象，代理对象的类继承自上述抽象接口
把p96的图与p93对象适配器的标准结构作比较
参数化的适配器：把适配器接口的功能分解到多个模块，每个模块完成一个功能，每次根据需要选择合适的模块组合成新的适配器。
例如与上面同样的例子中，可以一个模块完成CreateGraphicNode，另一个模块完成GetChildren
代码示例
TextShape的实现
类适配器（p97）
对象适配器（p98）
略
已知应用
相关模式
4.2 桥接模式
意图
将抽象部分与它的实现部分分离，使它们都可以独立地变化。
别名
Handle/Body
问题
考虑一个GUI工具箱，允许这个GUI工具箱在不同的平台间移植，使用这个GUI工具箱的程序在不同的平台间移植时不需要修改。
例如可以在X-Window和PM之间移植。
解决方案
将GUI工具箱中会用到的与平台有关的部分（例如书上例子中的Window）分解为抽象和实现两个部分。
工具箱只与抽象接口（例如各种Window）有关系
抽象接口通过实现接口（例如WindowImp）连接具体实现部分
具体平台（例如PM）上的类定义这些具体实现
图示
桥接模式分解了不同的类层次关系
例如在一个应用中会用到各种类型的窗口，这些类型可能构成一个类层次
另一方面，同一种窗口在各种不同的平台上又会有不同的具体实现类，这构成另一个类层次
把两个类层次合并，会导致大量的子类和类层次管理的混乱。
如果有m种窗口，n个平台，共需要m*n种窗口类
如果能够为所有这些窗口与具体平台有关的部分抽象出一个共同的接口（即Implementor），采用桥接模式，则：
只需要m种窗口类和n种Implementor，即总共m+n个类
两个清晰的相互独立的类层次，也容易管理。
思考
与适配器模式的比较?
4.3 组合模式（Composite）
意图：
将对象组合成树型结构以表示“整体/部分”关系，使得单个的基本对象和组合对象对外具有相同的特征。
动机
在图形编辑器中操纵的图形对象包括：
最基本的图形元素
例如直线、圆、…
组合图形元素，由基本图形元素和其它组合图形元素复合而成
例如圆柱由圆和直线组成，某个零件又由圆柱和直线组成
如果组合对象和基本对象的接口不同，则图形编辑器以及组合对象内部都需要区分操纵的对象的类型，导致程序复杂化
解决方案
为组合对象和基本对象建立共同的超类
这样客户程序以及组合对象内部都按这个超类定义的接口操纵这些对象
不必区别是组合对象还是基本对象
图形编辑器中的解决方案--P107图
组合对象内部结构形成递归的引用（或者包含）关系
图形对象的内部结构--P108图
适用性
需要表示对象的整体-部分的层次结构
需要在外部接口上忽略组合对象和基本对象的不同结构
参与者
Component
为参加组合的对象定义一个接口。
Composite
实现Component接口，并维护一个由实现了Component接口的对象组成的集合。
Composite更像一个容器，但提供与Component相同的接口。
Leaf
参与组合模式的基本对象，没有子节点。
实现了Component接口。
协作
客户程序使用Component接口与组合对象和基本对象以相同的方式交互
基本对象将直接做出响应
组合对象则把收到的请求转发给内部的每个子部件
效果

定义了包含基本对象和组合对象的组合层次结构，基本对象可以被组合成组合对象，组合对象也可以被组合。
形成递归结构
客户代码通常不必明确知道访问的是基本对象还是组合对象，因此简化了客户代码。
可以灵活地增减组合对象或基本对象，而不影响客户代码。
例子程序：Composite\Test.java
实现
一些实现中的技巧，概括为：
从子部件到父部件的遍历
最大化Component接口
声明管理子部件的操作
引用或保存子部件
从子部件到父部件的遍历
如果增加从子部件到父部件的引用能简化组合结构的遍历和管理。
如果一个子部件可以被多个父部件引用，则可以在被共享的子部件中存储对多个父部件的引用
这样可以共享子部件，但这时从子部件向上查找父部件时会有歧义
可以把与向上查找父部件有关的状态信息保存在外部，以确定当前的父部件
例如两次遍历从不同的父部件进入同一子部件，为每次遍历用一个对象保存本次遍历的路径就可以了。

最大化Component接口
为了客户代码能够以相同的方式访问基本部件和组合部件，应该在Component接口中包含尽量多的操作。
由于有些操作只对组合类有效，有些只对基本类有效，这种方式可能带来矛盾，例如组合类中的访问子结点的方法，如果定义在Component中，就要被Leaf实现，但显然这个操作对于Leaf是没有意义的。

可以有变通的方法，例如在Leaf中实现的访问子节点的方法返回一个特殊的空节点，表示没有子节点存在。
声明管理子部件的操作
Add和Remove只对组合对象有意义
如果声明在Component中，所有的Component和Composite具有相同的接口，提高了部件的透明性
但可能不安全，例如在叶子结点上执行这两个操作可能没有意义。
如果声明在Composite中，则提高了安全性，但降低了透明性。
解决方案
定义在Composite中，但在Component中声明一个操作Composite* GetComposite()。
用于检验当前部件是一个Composite还是Component。
如果不是Composite，就不必调用Add和Remove
引用或保存子部件的方法
引用或保存子部件可以采用各种数据结构，包括List，Tree, Table等等。
有时候还要保存子节点的顺序。
在没有自动内存回收机制的语言中，销毁一个组合对象时应该同时销毁所包含的对象。
代码示例
表示计算机设备的组合结构，用一致的方法计算不同配置的设备的成本
本节末尾的代码部分
P112-114
4.4 装饰（Decorate）模式
意图
动态地给一个对象（不是类）添加一些额外的职责，从增加功能的角度， Decorate模式比用子类更为灵活。
Decorator和所装饰的对象具有相同的接口，使用它们的程序不必区别它们。
别名
Wrapper
动机
有一个窗口对象可以显示文本图形等，窗口没有滚动条。
假定需要增加滚动条，当然可以定义这个窗口类的一个子类
但这样会需要有这个窗口的类定义
而且会有不灵活性，因为有时窗口对象需要滚动条，有时不需要
解决方案
可以用一个滚动条装饰对象装饰这个窗口对象（本节第1个图， P115）
滚动条装饰对象必须和这个窗口对象有相同的接口，这样客户程序不需要修改
需要滚动条时加上装饰对象，不需要时不加。
也可以加上多重装饰对象
（ P115-116，本节第2，3个图）
适用性
在不影响其它对象的情况下，以动态、透明的方式给每个对象增加功能
这些功能可以动态撤销
尤其可以用在不能通过继承扩充当前类的情况，或者需要建立大量孤立扩展子类（带边框子类、带横向滚动条子类、带纵向滚动条子类、带边框并且带横向滚动条子类、带边框并且带纵向滚动条子类、 …）的情况
结构
参与者
Component

定义一个接口
ConcreteComponent
定义一个实现了Component接口的类
Decorator
维持一个指向实现了Component接口的对象的指针，并定义一个与Component一致的接口。
ConcreteDecorator
具体添加职责，将请求转发给其中的component对象，并可能在转发请求前后执行一些附加的动作。
协作
Decorator接受请求，把请求转发给所包装的对象，收到响应后把响应转发出去。
在转发前后可以有附加的操作
例
Decorator\Test.java
在Composite中用Decorator
效果
比继承方式灵活
可以动态增减和组合附加的功能
可以避免在设计初期把太多的功能加入高层的类中
高层类可以设计得较为简洁，在具体应用中需要时通过装饰模式增加功能
可能会产生很多小对象
这是对象组合的特点。
实现
装饰与被装饰的对象必须具有相同的接口
要保持Component类的简单性
代码示例
为窗口对象增加边框的例子
“10. 代码示例”
P119-120

Decorate模式的另一个例子（略）
为简单的字节流增加数据压缩和编码转换功能
p121
讨论
Decorate模式和Adapter模式的比较。
有相同的别名
4.5 外观模式
意图
为某个子系统中的一组接口提供一个一致的高层接口，方便对这个子系统的使用。
问题
一个编程环境允许应用程序访问它的编译子系统。这个编译子系统由若干个类构成。一般的用户只需要这个子系统的普通的编译功能，不必了解子系统内部的类和接口。
解决方案：定义一个Compiler类，为这个子系统定义一个简单并且统一的接口，对普通用户屏蔽这个子系统内部的实现类。
这里的Compiler就是编译器的外观类
为完成客户的请求对构成子系统的对象进行调用和协调
适用性
要为一个复杂的子系统提供一个简单接口。
将一个系统划分成若干个子系统以降低系统复杂性，同时为每个较为复杂的子系统提供一个外观类，从而最小化子系统之间的通信和相互依赖关系。
应用于层次结构的系统中，为较为复杂的层次定义入口点。
参与者
Facade
知道构成子系统的各个对象
为完成客户的请求对构成子系统的对象进行调用和协调
Subsystem classes
实现子系统的功能
处理由Facade对象请求的任务
模式图示
P123
协作
客户程序通过发送请求给Facade的方式与子系统通讯，Facade将这些消息转发给适当的子系统对象。
效果
对客户屏蔽构成子系统的部件，简化了子系统的使用。
实现了子系统与客户之间的松耦合关系。
有助于建立层次结构的系统，也有助于为对象之间的依赖关系分层。
层间通过Facade联系
还可以降低编译依赖性。
并不妨碍客户程序使用子系统内部的部件，确保子系统具有易用性的同时还具有通用性。
实现
降低客户/子系统之间的耦合度
Facade接口可以由不同的子类实现，也可以由不同的对象配置
代码示例
一个编译器子系统由Scanner，Parser，ProgramNode，CodeGenerator，BytecodeStream等类实现。
定义一个Compiler类，为编译器定义一个简单并且统一的接口，对普通用户屏蔽编译器的实现类。
Web应用中的Facade
4.6 享元模式
意图
运用共享技术有效地支持大量细粒度的对象。
问题
在一个文档编辑器中可以把文档中的每个字符描述成一个对象：
太消耗内存，一个小文档都可能包含成千上万个字符对象。

本节的第二个图（ P129 ）
Column(栏)由多个row(行)组成，row中包含多个字符
解决方案：共享对象，每个字符用一个对象表示，文档中出现的所有这个字符都共享这个对象。
本节的第三个图（ P129 ）
字符全部引用flyweight pool，避免重复
状态信息怎么办？例如字符的位置、颜色…
外部状态
把每个具体位置上的字符的其它与上下文有关的状态信息（字体、颜色等）用一个外部对象表示
这些状态也可以共享，例如整个段落的文字采用的是相同的字体和颜色。
适用性
由于使用了大量的对象而造成很大的存储开销
对象的大多数状态都可放到对象外部，而剩余部分的对象就可以用数量相对较少的一组对象代表。
如果不同种类的外部状态和共享前对象的数目差别不大，使用这种模式就没有很大意义。
上述例子中可以只使用一个对象存储整篇文章的排版布局信息（即后面的GlyphContext），只要提供字符所在的位置，就可以通过这个对象计算出该字符的具体排版布局属性。（“8.实现”中的第一小点）

应用程序不需要依赖于对象标识区别每个对象。
代码示例
内部状态和外部状态
本节7、8、9图
结合组合模式的使用
还配合了抽象工厂模式
思考：现在的实现方式
内存和文件操作：字符编码
显示：显示字库
打印：打印字库
不同字体一般都有字库
有些应用还有专门的字库，例如pdf

4.7 代理模式
意图
为其它对象提供一种代理以控制对这个对象的访问。

别名
Surrogate
问题
在一个文档中可以嵌入图形对象，创建一个图像的开销很大，为了快速打开文档，文档编辑器应该避免在打开文档时一次性创建所有开销很大的对象，只有在需要时才创建这些对象（例如在这些对象变为可见时）
在文档中用什么来代表这些暂时没有创建的对象？
怎样让用户感觉不到这些对象是在需要时才创建的?

解决方案
使用另一个对象，即Proxy对象，替代真正的对象。

本节的第一个图（p137）

图像的Proxy
当编辑器激活图像代理的Draw操作显示这个图像时，图像代理才创建真正的图像。
图像代理把随后的请求转发给这个图像对象。
图像代理中记录着保存这个图像的文件的名称，以及图像的长和宽，这样，图像代理不需要创建出真正的对象就能响应文档的格式化请求。
本节的第二个图（P138 图）
适用性（1）
使用代理模式的常见情况：
远程代理：为一个对象在不同的地址空间提供局部代表。
例如用一个代理对象代表在另一个进程空间或另一台机器上的对象。
虚代理：根据需要创建开销很大的对象。
例如这里的ImageProxy
保护代理：控制对原始对象的访问，用于限制对对象的不同访问权限。
例如在打开下载文件时先进行警告
适用性（2）
智能指引：取代了简单的指针，在访问对象时执行一些附加操作，包括：
对指向实际对象的引用计数，引用数为0时释放这个对象。
当第一次引用一个持久对象时，将它装入内存
在访问一个对象时检查是否已经锁定这个对象以确保其它对象不能改变它。
4.8 结构型模式的讨论
Adapter， Decorator ， Proxy
Adapter用于适配已经设计好的类或对象之间不同的接口
Decorator则在保持接口一致的同时增加功能
Proxy通常不为被代理对象增加功能。
代理模式中对象的引用可以有各种形式
Bridge
提供了一种类层次的组织方式
Decorator对类层次的组织也有指导意义。
Composite， Facade
提供的是对象的组织方式
作业
见elearning的“第4章作业”的说明。
注意其中已经指定了作业截止日期。

第5章 行为模式

构造适用于多种应用的、由对象的协作提供的特殊行为。除规定对象和类的组合方式外，重点转向描述对象之间的通信和协作。
行为类模式使用继承机制在类间分配功能
例如Template
行为对象模式通过对象之间的协作来实现新的功能。

例如Observer
行为设计模式分类（1）
职责链设计模式（Chain-of-Responsibility）
把一系列对象组织成一个链，每个对象或者处理一个到达的消息或者把消息转发给下一个对象。
命令设计模式（Command）
可以把一个功能包装成一个命令，复用于各种环境。

解释器设计模式（Interpreter）
设计一个文法解释器的一般模式，用类的继承层次和对象复合结构表达文法规则。
行为设计模式分类（2）
迭代器设计模式（Iterator）
可以访问一个结构中的每个对象，而不必知道结构的内部情况。
中介器设计模式（Mediator）
用一个中介对象包装一组对象之间的协作关系，这些对象通过中介对象以间接的方式相互联系，实现一种松散的耦合。
备忘录设计模式（Memento）
允许对象保持历史状态，必要时可以恢复到先前状态。
行为设计模式分类（3）
观察者设计模式（Observer）
在主体对象和观察者对象之间建立松散耦合关系，当状态改变时，主体对象向观察者对象发送通知，观察者收到通知后，做出相应改变。
状态设计模式（State）
把对象可能的各种状态描述成若干个类，这些类构成继承层次。一个对象可以包含不同的状态对象，包含不同状态对象时具有不同的行为特征。
行为设计模式分类（4）
策略设计模式（Strategy）
把不同的算法封装成不同的策略对象，其它对象需要不同的算法时可以引用不同的策略对象。
与状态设计模式类似，不同在于状态设计模式中是把状态封装成对象。
模板方法设计模式（Template Method）
允许多个类共享相同的算法“骨架”。
行为设计模式分类（5）
访问者设计模式（Visitor）
访问一个数据结构中的对象需要不同的操作，可以把这些操作封装在不同的Visitor中，根据需要采用不同的Visitor。
新的功能只要增加新的Visitor，不必修改原有的数据结构和其中元素的类。
5.1 职责链模式
1. 意图
使多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系。
将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。
问题
在一个GUI中实现的上下文有关的帮助机制，用户在界面的任一部分上点击都可以得到与所点击部分最接近的相关帮助信息
问题在于帮助信息并不一定是由接受点击的对象提供的。
解决方案
通常一个GUI对象只能有一个上层对象（窗口），GUI对象之间一般构成一个树形结构，提供帮助信息的对象必然处在接受点击的对象和树根之间。
构造从叶子结点到树根的路径，用户请求将沿着这条路径自下而上传递。
本节第1、2个图（ p148 ）
传递请求的链接通道
从直接接收请求的对象开始，沿着链接通道传递，直到有一个对象能够处理这个请求。
链上的每个对象都要有一致的处理请求和访问链上后继者的接口。
例如本节第3个图（P149）的HelpHandler就定义了这样的接口。
适用性
使用职责链模式的常见情况
有多个对象可以处理一个请求，但每个请求只需要一个具体对象来处理，需要在运行时动态确定每个具体请求由哪个对象处理。
如果采用类似于switch的方式同时在所有可能处理这个请求的对象之间选择，会怎样?
是否存在树形层次关系
灵活性
效率
参与者
Handler
定义一个处理请求的接口
（可选）实现后继链
ConcreteHandler
处理所负责的请求
可访问它的后继者
如果可处理该请求，就处理它，否则将这个请求转发给它的后继者
Client
向链上的具体处理者（ ConcreteHandler ）对象提交请求
模式图示

本节第4、5个图（P149）
协作
当客户提交一个请求时，请求沿链传递直到有一个具体处理者负责处理它。
效果
降低耦合度
一个对象无需知道其它哪个对象处理其请求，接收者和发送者都没有对方的明确信息。
增强了给对象指派职责的灵活性
运行时刻可以动态修改职责链改变对请求的响应功能。
不保证被接受
不保证一个请求一定能被处理
实现（1）
实现后继者链
如果现有的链接可用，就不必定义新的链接
例如如果Composite中有对于父部件的引用
连接后继者
如果没有现成的链接可用，就需要由Handler自己维护这个链，并且提供HandleRequest的缺省实现
参考：例如HelpHandler的实现代码
实现（2）
表示请求
每种请求用一个硬编码方法调用表示
方便，但只能处理固定的一组请求
用一个方法处理所有请求，在方法的参数中指定请求的类型。
需要处理参数，不正确的参数不能在编译时发现，却可能在整个职责链上无法处理。
用一个对象包装一个请求，不同类型的请求用不同类型的对象表示
参考代码(P151)：Handler::HandleRequest和ExtendedHandler
以下详细分析
实现（3）
所有请求对象的类有一个共同的超类（Request）
请求分派方法根据每个对象所属类做不同的处理（例如HandleRequest方法）
每个类提供一个返回该类标识符的方法
Request的GetKind方法
或者有的语言可以提供运行时类型信息
子类重定义请求分派方法，改变对某些请求的处理，不需要改变的仍然由超类的HandleRequest方法处理。
超类的HandleRequest方法向后继者转发请求
开发子类的程序员不必担心，因为即使最坏情况下，每个合法的Request对象至少会被链上最后的（继承层次中最顶层的）HandleRequest处理，而不正确的对象会在编译的时候被发现。
代码示例
用职责链模式实现一个在线帮助系统
HelpHandler类定义了处理帮助请求的接口
Widget是HelpHandler的子类，有一个Widget类型的对象作为其后继者
所有的窗口部件都由Widget的各种子类实现
Button有一个Widget类型的对象作为其后继者
Dialog的后继者可以是任一个HandleHelp类型的对象
Application实现了HelpHandler接口，但没有后继者，处于链的末端。
本节最后的代码（P154）形成了怎样的职责链?
Button->HandleHelp()是怎样处理的?
如果button=new Button(dialog,
NO_HELP_TOPIC)，又会怎样?
讨论
为什么现在很少见到这种类型的帮助系统?
哪些应用中见到这种职责链模型?
职责链模式的应用实例—例如基于java的Web应用中的Filter
当web容器收到一个请求，将判断是否有过滤器与这个请求相关联，如果有，容器将把这个请求交给过滤器进行处理。
过滤器可改变请求的内容，然后再将请求发送给目标资源。当目标资源对请求响应时，容器同样会将响应先发给过滤器，过滤器同样可修改响应内容，然后再将响应发给客户。
客户端和目标资源并不需要知道过滤器的存在，也就是说，在web应用中部署过滤器，对客户端和目标资源是透明的。
可以部署多个过滤器组成过滤器链。链中的过滤器依次处理请求，直到目标资源，在发送响应时，则按照相反的顺序对响应进行处理，直到客户端。
过滤器并不是必须要将请求传递给下一个过滤器（或者目标资源），它也可以自行处理请求，然后响应给客户端，或者将请求发送给另外一个目标资源。

基于filter链的单点登录(Single Sign On,简称SSO)
SSO是企业业务整合的解决方案之一，使得在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。
CAS（Central Authentication Service）是一款针对Web应用的单点登录框架。
CAS的基本机制
当Web容器收到对设定的URL的请求时，会跳转到指定的CAS Server登录页，登录成功后，带着Ticket，访问一开始请求的URL。然后带着这个Ticket，就可以不必再次登录而访问所有相互信任的应用系统
CAS包含两个部分：
CAS Server 要独立部署，负责认证用户
CAS Client收到访问请求时，如需要登录，就把用户的请求重定向到CAS Server.

CAS Client
与受保护客户端应用部署在一起，保护 Web 应用的受保护资源。CAS相关的Filter被加入到受保护资源的Filter Chain的最前端，如果请求中不包含Ticket，CAS Client会重定向用户请求到CAS Server。
接下来，如果用户正确登录，CAS Server 会产生一个Ticket ，然后重定向用户到CAS Client（带上Ticket）。然后，CAS Client 和 CAS Server之间完成对用户的身份核实，用Ticket查用户的身份信息。
相关模式
Composite
5.2 命令模式（ Command ）
别名
动作（Action）
事务（Transaction）
意图：把一个请求包装为一个Command对象，客户程序的功能因为不同的Command对象而得到调整，也可以对请求做日志或排队。
Command对象需要提供一个执行入口，最简单的情况是一个execute方法。
动机
例如要在一个应用程序中采用一个预先开发好的用户界面工具箱
对用户界面工具箱的要求
对用户界面工具箱的要求
用户界面工具箱中包括按钮、菜单等用户界面对象，它们响应用户的请求，执行相应的动作。
这些对象可以被重复应用在不同的应用程序中。
对用户的各种请求究竟应该做什么响应是由所在的具体应用程序及当时的上下文决定的。
用户界面对象和应用程序应该互不依赖于对方的内部实现。
解决方案
为衔接应用程序和用户界面对象，当用户界面对象被触发时，应用程序向用户界面对象提供一个特定的Command对象，用户界面对象调用Command对象的execute方法执行。
以一个图形编辑应用程序为例
在一次图形编辑过程中，如果当前一个图形对象被选中时，用户界面上的Copy菜单被触发，则图形编辑应用程序应该包装并提供给Copy菜单一个Command对象，其中包含对这个图形对象的Copy方法的调用。
优点
菜单不必知道命令的接收者是谁，也不必知道它怎样执行命令
当前被选中的可以是一个圆，或一个矩形，它们将以不同的方式执行Copy命令。
命令的接收者不必知道谁以怎样的方式请求这个命令的执行
可以是通过菜单，也可以是通过快捷键，或其它方式。
适用性
可以通过参数化的方式设置对象需要执行的任务
是过程语言中的callback函数的面向对象替代物
命令对象可以被作为一般对象进行排列和执行
甚至可以在不同的进程之间传递
执行的命令可以被存放在一个列表中，需要的时候可以按顺序遍历这个列表实现Redo和Undo
需要有配合的恢复数据的保存方式，简单的数据可以直接保存在Command对象中
这个列表记录在日志中就可以用于系统恢复
结构
参与者
Command：定义执行操作的接口
ConcreteCommand
把一个接收者对象和一个具体操作联系在一起
在execute方法中调用接收者相应的操作
Client
建立一个具体的命令对象并设定它的接收者
Invoker：负责发出请求
Receiver：负责具体实现一个请求所对应的操作。
协作
Client创建一个ConcreteCommand对象并指定它的Receiver；
Client把这个ConcreteCommand对象提供给一个Invoker；
Invoker调用这个ConcreteCommand的Execute方法；

在这个ConcreteCommand的Execute方法中对应的Receiver的一些操作被调用，完成实际的工作。
效果
保持触发操作的对象和知道怎样实现这个操作的对象之间的松耦合。
每个Command都是一般意义上的一个对象，可以把它看作普通对象进行各种处理（例如用队列或者栈对其进行管理）。
可以把多个Command装配成一个复合Command，复合Command通常采用Composite模式组合各个Command。
可以很方便地实例化Command，建立不同的命令。

图形编辑软件的Command实现
Command： 一个接口或抽象类
ConcreteCommand：例如一个Copy命令
Client：当前的图形编辑应用程序，负责创建ConcreteCommand对象。
Invoker：菜单、按钮、…
Receiver：任何可以实现具体操作的对象

例如GraphObject, Document, Application, Circle,…
在ConcreteCommand对象中被引用
实现
命令对象的功能可强可弱
可以委托接收者实现所有功能，也可以由命令对象自己实现所有功能
支持取消和重做
使用C++的template机制以减少Command子类
见代码示例中的SimpleCommand
代码示例
抽象类Command
子类：OpenCommand，PasteCommand
子类： SimpleCommand
简单命令，不需要向Action转发参数、不需要Undo、…
可以用Template方式实现，以避免建立Command的太多的子类
子类： MacroCommand
应用了组合模式和迭代器模式
命令模式的另一个例子

```
public class CatalogApp... {
  private HandlerResponse executeAction (String
actionName, Map parameters)...{
    if(actionName.equals(NEW_WORKSHOP)) {
      ... //lots of code to create a new workshop
    }
    else if (actionName.equals(ALL_WORKSHOPS)) {
      ... //lots of code to display information
     }
       ... //many more "else if" statements
  }
    ... //other methods definitions
}
```

问题
（1）需要灵活增加处理的分支，包括在运行中动态增加；
（2）有些分支的工作需要重复（包含）另一些分支的代码；
解决方案
先把每个分支提取出来做成一个方法；
然后把提取出来的每个方法包装成一个命令；
然后建立命令的接口，也就是定义一个接口或者抽象类，其中声明了一个执行方法，每个具体命令都必须实现这个方法。

```
public class NewWorkshopHandler... {
    public HandlerResponse execute(Map parameters)
        throws Exception;
}
```

然后让原来对应每个分支的命令成为这个类的子类，原来的程序变成：

```
if(actionName.equals(NEW_WORKSHOP)) {
  return new
NewWorkshopHandler(this).execute(parameters);
} else if(actionName.equals(ALL_WORKSHOPS)) {
  return new
AllWorkshopsHandler(this).execute(parameters);
} else if …
```

但现在还不够灵活，还不能动态增加新的命令
因此定义一个映射表Map，其中保存各种handler的实例，并且用handler的具体类名称作为索引关键字，得到：

```
public class CatalogApp ... {
    private Map handlers;
    public CatalogApp(...) {
          ...
       createHandlers();
          ...
    }

    public void createHandlers() {
       handlers = new Map();
       handlers.put(NEW_WORKSHOP,
            new NewWorkshopHandler(this));
       handlers.put(ALL_WORKSHOP,
            new AllWorkshopHandler(this));
            ...
    }

    public HandlerResponse
executeActionAndGetResponse (
       String HandlerName, Map parameters) throw
Exception {
          Handler handler =
lookupHandlerBy(handlerName);
          return Handler.execute(parameters);
    }

    private Handler lookupHandlerBy(String
handlerName) {
      return (Handler)handlers.get(handlerName);
    }
      ...
 }
```

5.3 解释器模式
（略）
5.4 迭代器模式
1. 意图
提供一种方法顺序访问一个聚合对象中各个元素，而又不需要暴露该对象的内部表示。
2. 别名

游标（Cursor）
动机
一个聚合对象需要能够被遍历其中的元素，同时又不暴露它的内部结构。
解决方案
用一个迭代器（iterator）实现对于聚合对象中元素的访问和遍历功能。
迭代器对象同时负责记录哪些元素已经遍历过了。
例如
为一个列表（List）类定义一个列表迭代器(ListIterator)
ListIterator的操作包括：
CurrentItem返回List的当前元素
First初始化ListIterator，使List中的第一个元素成为当前元素
Next使下一个元素成为当前元素
IsDone判断是否越过最后一个元素，完成一次遍历。

本节第一个图（p171）
实现不同的遍历策略
List和ListIterator是相互独立的，可以对同一个List用不同的ListIterator实现不同的遍历策略。
例如改为FilteringListIterator
只访问满足特定过滤约束条件的元素
问题
应用程序需要为List创建ListIterator对象
如果用其它的聚合类对象（例如树）替换List对象，就需要改变所创建的ListIterator的类
也就要改变应用程序的代码
怎样可以不需要改变应用程序的代码?
聚合类和迭代器类都可以改变
多态迭代
为遍历不同的聚合结构提供一个统一的接口
为所有的列表定义一个公共的操作接口AbstractList
为所有的迭代器类定义一个公共的接口Iterator
为每种具体的列表定义具体的迭代器，由列表对象负责创建相应的迭代器
可以在列表类中定义相应的工厂方法创建对应的迭代器
通过工厂方法协调聚合对象和迭代器的类层次结构的对应关系
应用程序中不需要知道列表对象和迭代器对象的具体类，这样就不需要改变应用程序代码适应不同的列表和迭代器。
本节第二个图（p172）
适用性
访问一个聚合对象的内容而无需暴露它的内部表示
支持对同一聚合对象的多种遍历
为遍历不同的聚合结构提供一个统一的接口（多态迭代）
参与者
Iterator（迭代器）
定义用于访问和遍历聚合对象的接口
ConcreteIterator （具体迭代器）
实现迭代器接口，记录遍历时的当前位置
Aggregate（聚合）
定义用于创建相应迭代器对象的接口
ConcreteAggregate（具体聚合）
实现Aggregate接口
模式图示
本节第三个图（P172）
协作
ConcreteIterator跟踪聚合对象中的当前元素的位置，并能够找到待遍历的后继对象。
效果
支持以不同的方式遍历一个聚合对象
例如按前序、中序或后序遍历一棵树。
只需要改变迭代器就可以了
迭代器简化了聚合的接口
聚合类不需要定义遍历接口了
在同一个聚合上可以同时有多个遍历
每个迭代器保持其自身的状态，多个迭代器可以同时用于对一个聚合对象的不同遍历。
实现
谁控制迭代过程
谁定义遍历算法
迭代器的健壮程度
附加的迭代器操作
在C++中使用多态迭代器
迭代器可以有访问特权
用于复合对象的迭代器
空迭代器
谁控制迭代过程
外部迭代器：由使用迭代器的客户控制迭代过程，客户需要主动推进遍历的步骤，显式地向迭代器请求下一元素。
内部迭代器：客户向迭代器提交一个待执行的操作，迭代器将自动完成对聚合中元素的遍历并对每个元素实施这个操作。
谁定义遍历算法
在迭代器中定义
易于在相同的聚合上使用不同的迭代算法
改变迭代器即可
也易于在不同的聚合上重用相同的算法
同样的迭代器（例如同样的过滤标准）用在不同的聚合上
但如果遍历算法需要访问聚合对象的私有变量，这种方式会破坏聚合的封装性。
在聚合中定义
在遍历过程中迭代器存储当前迭代的状态，这种迭代器称为游标，客户程序以游标为参数调用聚合的

Next操作得到下一元素， Next操作同时修改游标的状态。
注意这里Next是聚合对象的方法
迭代器的健壮程度
在遍历一个聚合的同时修改这个聚合，例如增加或删除其中的元素，可能会导致两次访问同一个元素或遗漏掉某个元素。
可以先拷贝这个聚合，然后对这个拷贝做遍历，但会增加操作的代价。
也可以由聚合对象在发生修改时，或者调整迭代器的状态，或者在聚合对象内部维护额外的信息以保证正确的遍历。
附加的迭代器操作
迭代器的最小接口包括First，Next，IsDone，CurrentItem。可以在这个基础上增加其它的操作。

例如Previous，SkipTo等
在C++中使用多态的迭代器
缺点
需要用一个额外的工厂方法
由于工厂方法需要动态分配迭代器对象，因此客户程序需要负责释放这个对象
解决方案
可以用一个Proxy作为实际迭代器的代理
用一个栈分配的Proxy，在其析构方法中删除这个迭代器
即使发生异常，也能够保证迭代器被释放
迭代器可以有访问特权
在C++中可以把迭代器定义为聚合的友元，这样就可以访问聚合中的protected和private成员。
这种方法不容易定义新的遍历，因为新的遍历需要改变聚合的定义，即增加一个新的迭代器友元。可以用以下方法绕开这个问题
在作为聚合的友元的迭代器类中定义一些protected操作（原来是private）来访问聚合类的重要的非公共成员
迭代器的子类可以继承这些protected操作得到特权访问的能力。
用于复合对象的迭代器
采用Composite模式的聚合对象不容易实现外部迭代器，因为外部迭代器不容易描述聚合对象中的每个元素的位置（在哪一层的哪个节点）。
这时应采用内部迭代器
可以为不同的遍历方法（前序、中序、后序、…）定义不同的迭代器。
空迭代器
一个空迭代器(NullIterator)是一个退化的迭代器，它的IsDone总是返回true。
利用空迭代器可以更容易以一种统一的方式遍历树形结构的聚合（如复合对象）
复合结构中的每个元素都能够接受请求，返回一个能够遍历当前元素各个子节点的具体的迭代器
叶子节点元素返回一个空迭代器
代码示例
定义列表和迭代器接口
实现迭代器子类
使用迭代器
支持多种列表实现
保证迭代器被删除
实现一个内部迭代器
定义列表和迭代器接口
在List接口中提供了足以支持前向和后向两种迭代的方法
Get(long index);
Iterator定义了迭代器的接口
实现Iterator的子类
ListIterator实现从前到后的遍历
类似的，可以定义ReverseListIterator实现从后向前的遍历。
例如在ReverseListIterator中的First操作把_current置为列表的末尾, next中执行_current--，IsDone中判断_current<=0。
使用迭代器
定义一个PrintEmployees操作进行遍历和打印。
生成一个Employee列表、一个正向迭代器和一个逆向迭代器。
传递给PrintEmployees，实现正向和逆向打印。
支持多种列表实现
可以为各种List引入一个超类AbstractList，在其中定义一个工厂方法CreateIterator返回对应于具体List的Iterator。
这样，客户代码中就不要指明具体的List类和Iterator类
保证迭代器被删除
采用Proxy模式，建立一个类IteratorPtr，其析构方法负责销毁所引用的迭代器，再定义两个操作符“->”和“*”以类似于指针的语法进行操作

```
Method() {
    …
    IteratorPtr<Employee*> iterator(employees->CreateIterator());
    …
  }
```

这里的iterator变量就是在栈中分配的，Method()方法返回的时候就会自动销毁，导致销毁迭代器
实现一个内部的ListIterator
内部迭代器包装了迭代过程的逻辑，不需要客户程序推动，由迭代器控制对聚合中的每个元素做同一操作。
可以有两种方法实现一个抽象的内部迭代器，支持不同的作用于列表各个元素的操作

给迭代器传递一个函数指针，迭代器在迭代过程中的每一步调用传递给它的操作。
需要额外的静态变量记住函数操作的累积状态。
为内部迭代器生成子类，遍历时通过调用在子类中重定义的方法对各个元素做操作
以下讨论这种方法
利用子类实现内部迭代器
可以用子类的实例变量存储状态。
例如一个内部迭代器ListTraverser的实现
内部依赖于一个私有成员--外部迭代器对象_iterator

从ListTraverser外部看，ListTraverser是一个内部迭代器；从ListTraverser内部看，_iterator是一个外部迭代器。
ListTraverser的迭代方法Traverse通过_iterator完成迭代过程。
ListTraverser的ProcessItem是一个纯抽象方法。
定义子类实现不同的迭代逻辑。
例如子类PrintNEmployees：重定义了ProcessItem，用实例变量_count计数已打印的雇员数。
内部迭代器的优点
关键的区别在于推动遍历的循环是在迭代器（ 这里是ListTraverser ）内部还是外部定义
不需要在使用内部迭代器的每个程序中都重复定义这个循环，这是内部迭代器的主要优点。
作为比较的一个外部迭代器
ListIterator<Employee*> i(employees)…（P179）
需要在外部迭代器中指定对迭代的（循环）控制过程
可以用不同的内部迭代器封装不同类型的迭代
例如FilteringListTraverser仅处理能通过测试的那些列表元素。
例：java中的迭代器
Iterator\Test.java下的例子
通过迭代器，以相同的方式访问不同的数据结构
相关模式
Composite：聚合的常用模式
Factory Method：用于实现多态迭代
Memento：用于保存迭代的状态
5.5 中介器模式
1. 意图
用一个中介对象封装一系列对象的交互，使得这些对象不需要显式地相互引用，达到松散耦合的效果。

改变中介对象就可以改变这些对象的交互方式。

与外观模式不同，外观模式包装一组对象，对外提供服务
中介模式在同一组对象之间协作
问题
应用程序由许多对象组成，大量的对象之间需要大量的相互连接。
例如在一个对话框中包含多个GUI组件，相互关联
例如Save对话框中如果不指定文件名，“Save”按钮就不会有效。
可以让这些GUI组件互相联系，例如让文件名输入域与“Save”按钮关联
如果有很多GUI组件，每个GUI组件都要引用很多其它GUI组件，修改每个组件（例如定义其子类）都要在所定义的子类中修改大量的引用关系。
导致应用程序的修改困难
解决方案
把部件之间的关联关系提取出来，定义在另一个类中，这个类就是一个中介器。
新的应用中只要重新定义中介器，其它的GUI部件的代码不需要改变。
本节的图
代码示例（p185-187）
例子：MediatorTest.java
其中的button和textArea并没有直接的联系
5.6 备忘录模式
1. 意图
在不破坏封装性的前提下，取得一个对象的内部状态，并在这个对象之外保存这个状态。
将来可以把这个对象恢复到原先的状态
问题
为了能够实现Undo操作，需要在改变一个对象的状态之前保存其状态。
但由于封装性，其它对象不应该具有访问这些状态信息的权限。
例如
一个图形编辑器，对两个通过最短连线连接的几何形状，在移动其中某个几何形状之后，能自动维持这种最短连线。
为了实现Undo，可以有两种方法：
把被移动的几何形状反方向移动相同距离，然后重新计算。
可能无法精确恢复到先前的状态
例如本节第1、2张图（p188-189）
采用备忘录模式
备忘录
备忘录（memento）是一个对象，存储另一个对象在某个时间点的状态，后者被称为备忘录的原发器（originator）。
在状态改变之前，原发器根据当前状态初始化一个备忘录对象。
只有原发器可以访问备忘录中的信息。
原发器可以利用备忘录中的信息恢复状态。
结构图
效果
简化了原发器

如果把备忘录交给原发器自己管理能最好地保证封装性，但这会增加原发器的复杂性，而且要求客户在工作结束时通知原发器以便释放保存的备忘录。更多的时候交给客户来管理，但客户不能访问到备忘录的内部状态。
备忘录为原发器提供一个宽接口，为客户提供的是一个窄接口—不能访问内部数据。
代码示例
例1：
P193
_target->Move(-_delta)之后，不是由ConstraintSolver计算，而是直接通过solver->SetMemento(_state)恢复。
例2：
在集合中使用备忘录实现迭代：不需要为了支持迭代而破坏一个集合的封装性，备忘录仅由集合自己处理，对外不可见。

```
While (…) {
  aCollection.CurrentItem(state)->Process();
  …
}
```

5.7 观察者(Observer)模式
别名：
发布/订阅（Publish/Subscribe）模式
模型/视图（Model/View）模式
源/监听器（Source/Listener）模式。
依赖（Dependents）模式。
意图：
定义对象间的一种1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。
动机
例如在应用程序中，保存数据和显示数据可以由不同的对象实现。
同一个数据对象中的数据可以用一个表格对象显示，也可以用一个曲线图对象显示，还可以用一个直方图对象显示。
可以把数据对象看成Subject，把各种显示对象看成Observer。要在它们之间保持状态一致，就把这些显示对象注册在数据对象上，数据对象随时把数据的变化通知它们。
本节第一个图（p194）
适用性
当一个抽象模型有两个方面，其中一个方面依赖于另一方面。
将二者封装在独立的对象中以使它们可以各自独立地改变和复用。
当对一个对象的改变需要同时改变其它对象，而又不能假定其它对象是谁，会有多少这样的其它对象。

从而保持了这些对象之间的松散耦合

观察者(Observer)模式结构
参与观察者模式的角色
主体Subject
定义被观察者，提供注册和注销观察者对象的方法。

具体主体ConcreteSubject
保存状态，当它的状态发生改变时，向它的各个观察者发出通知。
观察者Observer
定义观察者接口。
具体观察者ConcreteObserver
具体实现的观察者。
观察者模式中的协作关系
（本节第3个图）P196
被观察者的状态改变
例如由某个具体的观察者调用被观察者的SetState方法设置或改变其状态；
被观察者对象执行Notify方法
在Notify方法中依次调用每个具体观察者的Update方法
其中通过调用主体对象的GetState方法取得当前状态
效果
观察者模式的特点
Subject和Observer之间通过Observer的简单接口联系，是抽象而且松散的耦合。
Subject和Observer之间是以广播方式通信，支持任意数量的Observer。
由于Observer之间互相独立，一个Observer对一个Subject的改变可能导致其它Observer无法预期的动作。
有关观察者模式的一些讨论
建立Subject和Observer之间的联系
一个Observer依赖于多个Subject时
由谁调用Notify
对象删除
先完成状态更新再调用Notify
Subject和Observer的推/拉关系模型
显式指定感兴趣的改变
封装复杂的语义
建立Subject和Observer的联系
通常在Subject中保存对Observer的引用。
当有很多Subject和很少Observer时，这种方式导致较高的存储代价。
？：这应该指用定长数组存放指向Observer的指针列表的情况
解决方案
用独立的关联表维护Subject和Observer的关联。
优点：没有Observer的Subject没有存储开销
缺点： 每次要查表后才知道要通知哪些Observer
一个Observer依赖于多个Subject

一个Observer依赖于多个Subject时，可能需要知道一个通知究竟来自哪个Subject。
解决方案
扩展update，把Subject作为其中的一个参数。
由谁调用Notify
通常Subject在状态变化后自动调用notify通知各个Observer。
如果客户对象连续触发Subject状态变化， Subject将发出一系列通知，引起Observer频繁更新。
解决方案
可以由客户对象自己在必要时调用notify。
缺点：增加了客户对象的责任，如果遗漏对notify的调用可能会导致错误。
对象删除
删除一个Subject可能会在其观察者中留下对这个Subject的悬挂引用
解决方案：当一个Subject被删除时，让它通知它的观察者清除对这个Subject的引用。
删除一个Observer可能会在一个或多个Subject中留下悬挂引用
解决方案：需要记住一个Observer被哪些Subject引用，并通过detach接口清除这些引用。
先完成状态更新再调用Notify
Subject中每次应该在完成了状态更新后再调用Notify通知各个Observer。
解决方案：
可以采用模板方法模式，在Subject中定义一个模板方法，其中调用各种可以在子类中被重载的更新状态的操作，在最后调用Notify方法。
Subject和Observer的推/拉关系
通常Subject发出的通知中提供了状态变化的信息，这称为推模型。
这可能不利于Observer的复用。
由于依赖于Subject的消息结构和内容
也可以采用拉模型， Subject只发出最简的通知，Observer随后自行查询Subject中的状态变化。
拉模型可能效率不高。
Observer需要自行检查Subject看哪些内容被改变了?其实可能更不利于Observer复用，因为Observer会依赖于Subject的查询接口
显式指定感兴趣的改变
可以扩展Subject的注册接口，让每个Observer注册为仅对特定事件感兴趣。
Subject的注册方法为：
Subject::Attach(Observer*, Aspect&)
Observer的更新方法为：
Observer::Update(Subject*, Aspect&)
封装复杂的语义
当Subject和Observer之间存在复杂的依赖关系时，可以用一个称为更改管理器（ChangeManager）的对象维护这些关系，其功能包括：
维持Subject和Observer之间的映射关系和调用工作
实现特定的更新策略
例如在完成相关的多个Subject的更新后才通知相应的Observer，以避免因为一次改变反复通知一个Observer
这个例子中ChangeManager相当于一个中介器

5.8 状态模式

一个对象在不同的状态下有不同的行为。
一种直观的实现方法是用条件判断结构根据对象的属性值判断对象的当前状态，然后确定行为。
条件判断结构可能很复杂，而且不容易灵活增加新的判断分枝
状态模式解决方案是让这个对象包含（引用）一个状态对象。
改变状态对象就改变了当前对象的行为。这样可以灵活地改变对象的行为，不需要复杂的分支结构。
例：状态变迁的几种实现
问题：销售订单具有不同的状态，例如“NewOrder”、“Registered”、“Granted”、“Shipped”、“Invoiced”、“Cancelled”。
订单可以从什么状态进入什么状态依赖于严格的规定。例如不能从Registered直接进入Shipped。
不同状态下也有不同的行为。例如当Cancelled时，就不能调用AddOrderLine()为订单增加更多的项。
另外，特定行为还会导致状态的转换，例如调用AddOrderLine()时将从Granted转换回New Order。
解决方案1：用属性记录所处状态
在与状态有关的方法中检查所处的状态，并且根据当前状态确定可以做什么操作，也就是用条件判断分支结构
当状态较多的时候，这个分支结构会很复杂
而且将来这些状态之间关系变化的时候要修改这个分支结构，这需要修改代码。
但对于状态较少而且稳定的情况则也还算一个简便的好方案。
解决方案2：用一个状态转换表
优点：只要修改数据就可以改变转换关系
缺点：大的表的查找会影响性能。
解决方案3：状态模式
优点：每个状态类中只有小段的简单代码，修改起来较为方便。

5.9 策略模式（略）

对象的某个功能有多种实现方式（算法）
可以用分支结构实现，但太不灵活，不清晰，而且很多代码可能用不到。
把算法包装成策略对象。需要不同的算法时只要引用不同的策略对象就可以了。
与状态模式其实差不多，只是没有状态变迁的概念。

与Decorate模式比较

Decorate通过在对象的外部增加外壳改变对象的行为。
对于界面复杂的对象，工作量会很高
策略模式通过改变对象的内核改变对象的行为。
5.10 模板方法（Template Method）
意图
定义算法的骨架，算法中的某些步骤的具体实现可以在子类中重新定义，这样子类只要对某些步骤做所需要的修改就可以实现特定的任务。
动机
每次要打开一个文档都要执行以下步骤
检查指定的文档能否打开
创建与应用相关的Document对象
将这个对象加入文档集合中
从指定文件读入数据初始化这个对象
但不同的应用程序可能打开不同的文档
例如Word打开的是DOC类型的文档，而PPT打开的是PPT类型的文档
怎样保证每次打开文档时都按上述步骤进行，同时允许在不同的应用程序中打开不同类型的文档?
解决方案
本节的第一个图（p215）
在抽象类Application中定义OpenDocument方法，其中定义了打开文档的步骤
同时定义了对应每个步骤的方法的型构
例如DoCreateDocument，…
在Application的子类中继承OpenDocument方法，同时在必要时覆盖对应特定步骤的方法
例如DoCreateDocument，…
适用性
一次性实现一个算法的不变部分，并将可变的行为留给子类来实现
只允许子类在特定的方面进行扩展
模板方法模式结构
模板方法的参与者
AbstractClass
定义抽象的基本操作，具体的子类将重新定义这些基本操作以实现一个算法的各步骤。
子类中必须重写的基本操作应该做成抽象方法，其他的基本方法可以提供缺省的实现。
实现一个模板方法，其中定义一个算法的骨架，包含了对基本操作的调用
ConcreteClass
覆盖并实现抽象类中的基本操作。
（略）
协作
效果
实现
代码示例
5.11 访问者模式
1. 意图
表示一个作用于某对象结构中的各元素的操作。可以在不改变各元素的类定义的前提下定义作用于这些元素的新操作。
问题
例如编译器将源程序表示为一个抽象语法树，需要在抽象语法树上实施某些操作：
检查变量的定义、类型检查、代码优化、…
每个操作对不同的语法树结点的处理过程不同
不同的语法树结点属于不同的类
例如用于赋值语句的类、用于变量的类、用于算术表达式的类、…。
如果在语法树结点中包含这些操作将导致难以理解和修改维护。
增加新的操作还要重新编译所有这些类
本节第1个图（p218）
解决方案
可以把这些操作包装成独立的对象（Visitor），在遍历抽象语法树时将Visitor传递给每个Node。
需要定义两个类层次
接收操作的元素（Node）层次
访问者层次
每个Node接受Visitor后，调用Visitor相应的操作，并把自身作为参数传入
AssignmentNode调用VisitAssignment操作
VariableRefNode调用VisitVariableRef操作
本节第2个图（p219），与第一个图（p218）对比。

适用性
一个结构中包含多种类型的对象，这些对象有不同的接口，需要对它们实施一些依赖于其所属类的操作。
需要对一个结构中的对象进行很多不同的并且不相关的操作，同时要避免这些操作影响这些对象的类定义。
Visitor把相关的操作集中定义在一个类中
当所要访问对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作
结构中对象所属的类很少改变，但经常需要在此结构上定义新的操作。
模式图示
本节第3个图（P220）
参与者（1）
Visitor（访问者，如NodeVisitor）
为结构中的对象的每个类声明一个Visit操作。
ConcreteVisitor（具体访问者，如TypeCheckingVisitor）
实现Visitor中声明的操作
Element（元素，如Node）
定义一个Accept操作，它以一个Visitor为参数
在Accept中调用Visitor对应于当前Element类型的操作方法

参与者（2）
ConcreteElement（具体元素，如AssignmentNode，VariableRefNode）
实现具体元素的Accept操作
ObjectStructure（结构，如Program）
能枚举它的元素
可以提供一个高层的接口以允许Visitor访问它的元素
可以是一个复合对象或一个集合
协作
一个使用Visitor模式的客户必须创建一个ConcreteVisitor对象，然后用这个访问者访问每个元素。
当一个元素被访问时，这个元素调用对应于它的类的Visitor操作。
效果（1）
容易增加新的操作
只要增加新的Visitor子类就可以增加新的操作。
Visitor集中相关的操作而分离无关的操作
相关的行为集中在一个Visitor子类中
无关行为则分别放在各自的Visitor子类中
增加新的ConcreteElement类很困难
因为需要在Visitor中定义新的操作并且在ConcreteVisitor中实现这些操作。
导致整个Visitor继承层次的修改
效果（2）
与迭代器模式比较
迭代器访问的聚合中的元素需要具有相同的超类
Visitor则可以访问完全无关的对象。
当然，都要有accept方法
累积状态
访问过程中的累积状态可以保存在Visitor中而不需要作为全局数据
破坏封装
Visitor可能需要ConcreteElement提供接口访问其中的内部状态，可能会破坏ConcreteElement的封装性。
实现
双分派：根据请求的名称和接收者类型决定执行的操作
Accept的动作根据Visitor和Element的类型确定
在Element的accept方法中调用Visitor的处理方法时，具体的Visitor的类型决定了处理方法是什么。
谁负责遍历对象结构
对象结构、Visitor、独立的迭代器对象都可以。
演示例子
Visitor \Test.java
不改变结构以及组成结构的元素的类定义，只要接受不同的visitor，就会执行不同的动作。
代码示例
为复合类Equipment中添加Accept操作
Equipment子类以基本相同的方式定义Accept
调用EquipmentVisitor中的对应于当前类的操作
聚合型的Equipment的Accept需要遍历其各个子构件并调用它们各自的Accept，然后对自身调用Visit操作
例如Chassis。
定义EquipmentVisitor类，其中为每个设备子类定义一个Visit虚函数（例如VisitCard）
具体的EquipmentVisitor子类实现特定的操作
PricingVisitor
InventoryVisitor
…
相关模式
Composite
Interpreter
5.12 行为模式的讨论
封装变化
Strategy, State, Mediator, Iterator
对象作为参数
Visitor, Command, Memento
通信应该被封装还是被分布
Mediator, Observer
对发送者和接收者解耦
Command, Observer, Mediator, Chain of responsibility
作业
见elearning上的作业说明