练习

Review:面向对象设计原则

设计原则名称	设计原则简介	重要性
单一职责原则(SRP)	类的职责要单一，不能将太多的职责放在一个类中	★★★★☆
开闭原则 (OCP)	软件实体对扩展是开放的，但对修改是关闭的，即在不修改一个软件实体的基础上去扩展其功能	★★★★★
里氏代换原则(LSP)	在软件系统中，一个可以接受基类对象的地方必然可以接受一个子类对象	★★★★☆
依赖倒置原则(DIP)	要针对抽象层编程，而不要针对具体类编程	★★★★★
接口隔离原则(ISP)	使用多个专门的接口来取代一个统一的接口	★★☆☆☆
组合复用原则(CRP)	在系统中应该尽量多使用组合和聚合关联关系，尽量少使用甚至不使用继承关系	★★★★☆

单选题

- Open-Close原则的含义是一个软件实体

  A.应当对扩展开放，对修改关闭.

  B.应当对修改开放，对扩展关闭

  C.应当对继承开放，对修改关闭

  D.以上都不对

- 答： A

单选题

- 要依赖于抽象，不要依赖于具体。即针对接口编程，不要针对实现编程,是(　)的表述

  A.开-闭原则

  B.接口隔离原则

  C.里氏代换原则

  D.依赖倒转原则

- 答： D

单选题

- 依据设计模式思想, 软件开发中应优先使用的是(　) 关系实现复用。

  A. 组合聚合

  B. 继承

  C. 创建

  D. 以上都不对

- 答： A

单选题

- 关于继承表述错误的是：

  A.继承是一种通过扩展一个已有对象的实现，从而获得新功能的复用方法。

  B.泛化类（超类）可以显式地捕获那些公共的属性和方法。特殊类（子类）则通过附加属性和方法来进行实现的扩展。

  C.破坏了封装性，因为这会将父类的实现细节暴露给子类。

  D.继承本质上是“白盒复用”，对父类的修改，不会影响到子类。

- 答： D

CREATIONAL PATTERNS

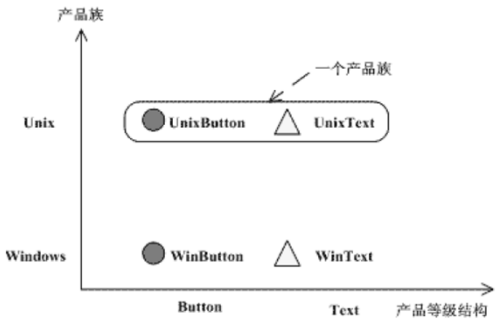
Review：Creational Patterns

- Factory Method
  - 本质：用一个virtual method完成创建过程
- Abstract Factory
  - 一个product族的factory method构成了一个factory接口
- Builder
  - 通过一个构造算法和builder接口把构造过程与客户隔离开
- Prototype
  - 通过product原型来构造product，Clone + prototype manager
- Singleton
  - 单实例类型。由构造函数，一个静态变量，以及一个静态方法对实例化进行控制和限制

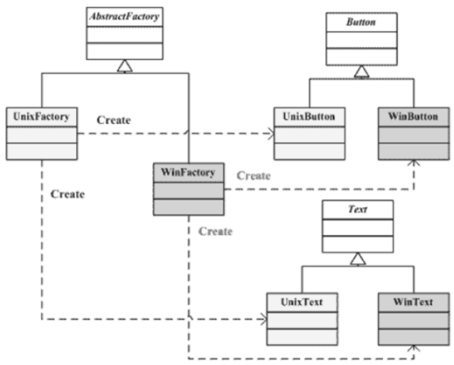
练习

- 创建在不同操作系统的视窗环境下都能够运行的系统
  - 两种操作系统：Windows和Unix
    - Windows操作系统下，使用具有Windows风格的视窗构件（这里设为WindowsButton对象和WindowsText对象）
    - Unix操作系统下，使用具有Unix风格的视窗构件UnixButton对象和UnixText对象
  - 如何进行设计，使得…
    - 当需要增加对新操作系统的支持时（如系统还需要支持Solaris），现有代码不必修改
    - 在系统的设计中约束用户使用的各种构件一定属于同一操作系统（不会出现将WindowsButton和UnixText一起使用这种情况）
- 问：采用哪种设计模式进行设计？

从产品族的角度来看...

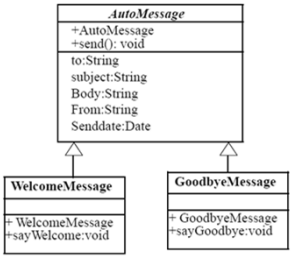


UML示意 （Abstract Factory模式）

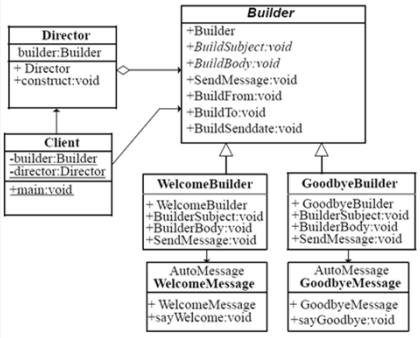


*练习

- 一个电子杂志系统，定期向用户的电子邮箱发送电子杂志。用户可以通过网页订阅电子杂志，也可结束订阅。当用户开始订阅时，系统发送一个电子邮件表示欢迎，当客户结束订阅时，系统发送一个电子邮件表示欢送。本例就是该系统负责发送“欢迎”和“欢送”邮件的模块。



基于Builder模式的设计



由于电子邮件可以划分成发件人地址、收件人地址、主题、内容等部分(或称零件)，因此，电子邮件建造过程可以划分成几个零件的建造过程，包括**buildTo()**、**buildFrom()**、**buildSendDate()**、**buildSubject()**以及**buildBody()**.

邮件是最终产品，而**To**、**From**、**SendDate**、**Subject**以及**Body**均为产品零件。

*练习

- 一个基于web的系统，它允许用户登录进入一个web服务器的受保护区域，该系统有一个包含用户名、口令及其他用户属性的数据库。该数据库通过第三方API进行访问。可以在每个需要读写用户信息的模块中直接访问数据库，但会带来问题：
  - 对第三方API的访问分散在整个代码中；
  - 无法强制实施一些访问或者结构方面的约定。
- 怎么解决？

UserDatabase接口：

```
publicinterfaceUserDatabase
{
User readUser(StringUsername);
void writeUser(User user);
}
```


UserDatabaseSource Singleton类：

```
Public class UserDatabaseSource implements UserDatabase
{
  private static UserDatabase theInstance=new UserDatabaseSource();
  public static UserDatabaseInstance(){
      return theInstance;
  }
public User readUser(StringUsername)
  {
     return null;
  }
public void writeUser(User user)
  {
  }
}
```

确保了所有对数据库的访问都通过UserDatabaseSource类的单一实例进行，这样就可以任意在UserDatabaseSource类中放入检查、计数、锁等机制来强制实施访问及结构方面的约定

STRUCTURAL PATTERNS

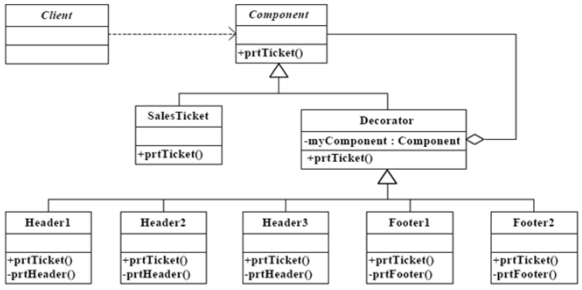
Review：Structural Patterns

- 结构型模式，关注的是对象之间组合的方式
  - Adapter模式，在不改变原有实现的基础上，将原先不兼容的接口转换成兼容的接口
  - Decorator模式，描述的就是对象间可能存在的多种组合方式，这种组合方式是一种装饰者与被装饰者之间的关系
  - Bridge模式，封装的则是对象实现的依赖关系
  - Composite模式
  - Facade模式
  - Proxy模式
  - Flyweight模式

练习：发票打印

- 有一个程序打印发票，所有发票都需要有正文部分，同时根据用户需要有的订单需要有表头，有的需要有页脚。表头有3种，分别叫做“表头1”、“表头2”、“表头3”；页脚有2种，分别叫做“页脚1”、“页脚2”。不同的用户可能要求有表头，没有表头；有1种表头，或者同时有2个表头，或者3个表头；有页脚，没有页脚；有1个页脚，或者2个不同的页脚。
- 问：采用哪种设计模式进行设计？

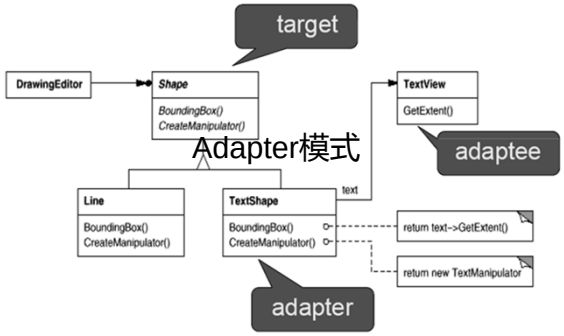
采用Decorator模式



练习

- 有一个绘图编辑器，这个编辑器允许用户绘制和排列基本图元(线、多边型和正文等)生成图片和图表。这个绘图编辑器的关键抽象是图形对象。图形对象有一个可编辑的形状，并可以绘制自身。图形对象的接口由一个称为Shape的抽象类定义。基本几何图形的类比较容易实现，但是对于可以显示和编辑正文的TextShape子类来说，实现相当困难。
- 成品的用户界面工具箱可能已经提供了一个复杂的TextView类用于显示和编辑正文。但TextView和Shape对象不能互换。
- 问：采用哪种设计模式进行设计？

采用Adapter模式



练习

- 设计一组容器：
  - 容器包括堆栈、队列
  - 这些容器可采用数组，也可采用链表来实现
- 问：采用哪种设计模式进行设计？

采用Bridge模式