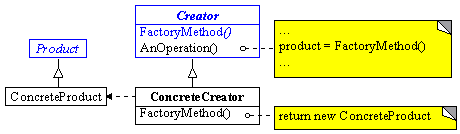
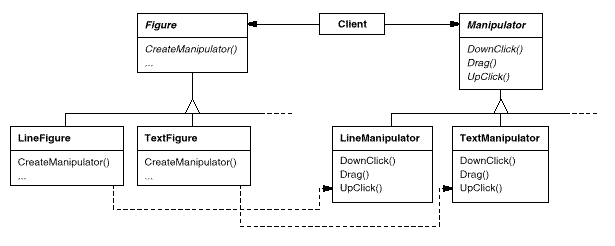
# 创建型设计模式

## Factory Method (virtual constructor)

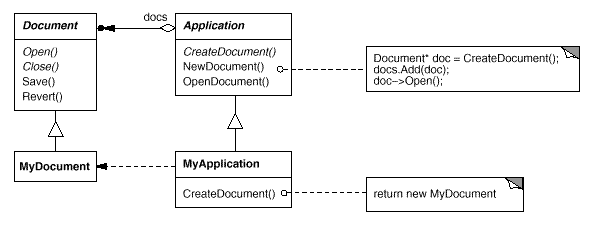
1. Aliases：virtual constructor
2. Intent
   1. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
3. Motivation
   1. Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.
4. Struct



1. Participants
   1. Product、ConcreteProduct、Creator、ConcreteCreator
2. 使用场景：
   1. 当一个类不知道它要创建何种方法时
   2. 当一个类想要它的子类明确指定它要创建的类时
   3. Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.
3. Evaluation
   1. 多态性：客户代码可以做到与特定应用无关，适用于任何实体类
   2. 缺点：需要Creator和相应的子类作为factory method的载体，如果应用模型确实需要creator和子类存在，则很好；否则的话，需要增加一个类层次
   3. 优点：
      1. Provides hooks for subclasses。基类为factory method提供缺省实现，子类可以重写新的实现，也可以继承父类的实现。  
         体现了：加一层间接性，增加了灵活性
      2. Connects parallel class hierarchies
4. Connects parallel class hierarchies



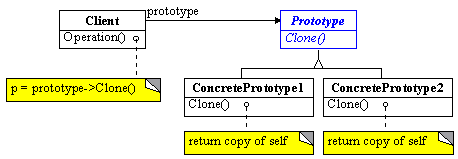
1. Implementation
   1. 父类是否提供缺省的实现
   2. factory method的参数
   3. Language-specific variants and issues
      1. SmallTalk，使用类型
      2. C++，使用lazy initialization技术
   4. Using templates to avoid subclassing
2. Related Patterns
   1. Abstract factory
   2. Prototype
3. Examples



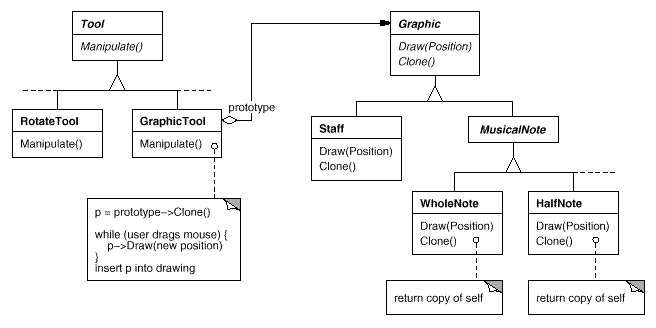
## Abstract Factory

## Prototype

1. Intent
   1. Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
2. Motivation
   1. 以一个已有的对象作为原型，通过它来创建新的对象。在增加新的对象的时候，新对象的细节创建工作由自己来负责，从而使新对象的创建过程与框架隔离开来
3. Applicability
   1. 当产品的创建过程要独立于系统时
   2. 当产品的类型是在runtime时被指定的情况下
   3. 避免创建一个与product层次平行的factory层次时
   4. 当产品类的实例只能是集中确定的不同势力状态中的一种时
4. Structure

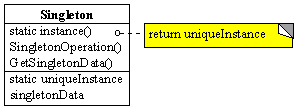


1. Participants
   1. Prototype、ConcretePrototype、Client
2. Evaluation
   1. Adding and removing products at run-time
   2. Specifying new objects by varying values，降低系统中类的数目
   3. Configuring an application with classes dynamically
   4. 要求：每一个product类都必须实现Clone操作
   5. 对于C++语言特别有意义：C++的class不是first-class objects
3. Implementation
   1. Using a prototype manager
   2. Implementing the Clone operation
      1. shallow copy versus deep copy
      2. Save & Load
   3. Initializing clones
      1. 两阶段构造
4. Related patterns
   1. Prototype与Abstract Factory往往是相互竞争的
   2. factory method
5. Examples
   1. DrawCli，music editor



## Singleton

1. Intent
   1. Ensure a class only has one instance, and provide a global point of access to it.
2. Motivation
   1. It's important for some classes to have exactly one instance.
   2. Instance-controlled class
3. Applicability, Use the Singleton pattern when
   1. there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
   2. when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
4. Structure

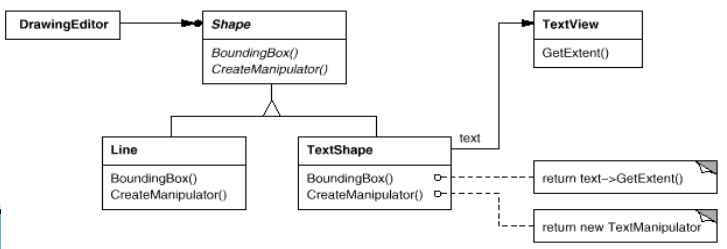


1. Participants
   1. Singleton
2. Collaborations
   1. Clients access a Singleton instance solely through Singleton's Instance operation.
3. Evaluation
   1. Controlled access to sole instance
   2. Reduced name space 避免使用全局变量 (why not 全局静态对象)
   3. Permits refinement of operations and representation，允许子类化
   4. Permits a variable number of instances （How?）
   5. More flexible than class operations (static member functions in C++ )
   6. 这种思想比较适用于Object-Based中的许多情形
4. Implementation
   1. Ensuring a unique instance
      1. 考虑使用lazy initialize
      2. 使用global/static object的缺点
   2. Subclassing the Singleton class
5. Related patterns
   1. Singleton与其他创建型模式并不矛盾，可以用singleton来实现其他模式中的对象。包括Abstract Factory、Builder、Prototype等。  
       多个实例对于构造过程往往并无意义，所以在许多情况下singleton模式比较符合应用背景
6. Examples
   1. MFC中的CWinApp派生类实例theApp
   2. ……

# 结构型设计模式

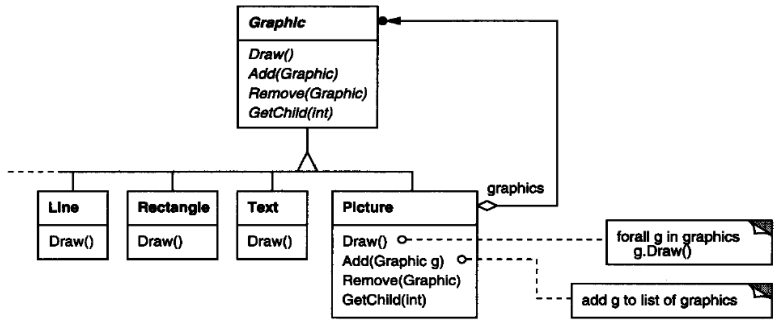
## Adapter

1. 别名: Wrapper
2. 目的（Intent）：Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. 将一个类的接口转换为客户所需的样子。Adapter能让接口不兼容的类协同工作。
3. 动机（Motivation）：Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface that an application requires. 有时，一个本来设计的时候是为了重复使用的工具类，会因为接口不符合某些应用程序所要求的具体领域接口，而没办法重用。
4. 适用范围（Applicability）：Use the Adapter patt在以下情况用：
   1. 要使用已存在的类，但是它的接口不符合需要
   2. 想创建一个可重用的类，并且它能与无关的、创建的时候没有遇见的类
   3. 要使用几个已存在的子类，但为了统一接口去继承把这些子类全部都继承一遍又不现实。这时，我们可以为这些子类的共同父类创建一个Adapter。
5. 结构（Struct）：
   1. 类Adapter：
   2. 对象Adapter：
6. 参与者（Participants）：Client, Target, Adaptee, Adapter
7. 贡献者（Collaborations）：
   1. 类Adapter——代理
   2. 对象Adapter——容器
8. 评价（Evaluation）：
   1. 本质上是两种重用模型：
      1. 类Adapter——无法适配adaptee的子类，但是可以重载adaptee的行为
      2. 对象Adapter——可以适配adaptee的所有子类
   2. 用双向Adapter来提供透明性
   3. 针对类Adapter，用多重继承来实现
9. 实现（Implementation）：
   1. 使用C++继承机制实现类Adapter
   2. 使用内嵌对象技术实现对象Adapter
   3. 对于Pluggable adapters有三种实现方案：
      1. 使用抽象方法定义
      2. 使用代理对象
      3. 参数化技术
   4. 以上三种方法的实质：如何在一个类中定义抽象操作，供客户插入？Hook技术
10. 相关模式（Related Patterns）：Bridge, Decorator, Proxy
11. 例子：

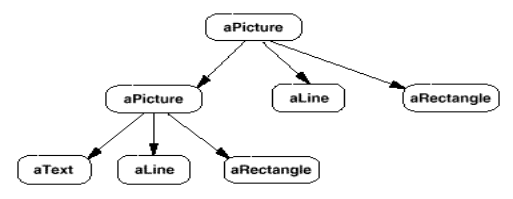


## Composite

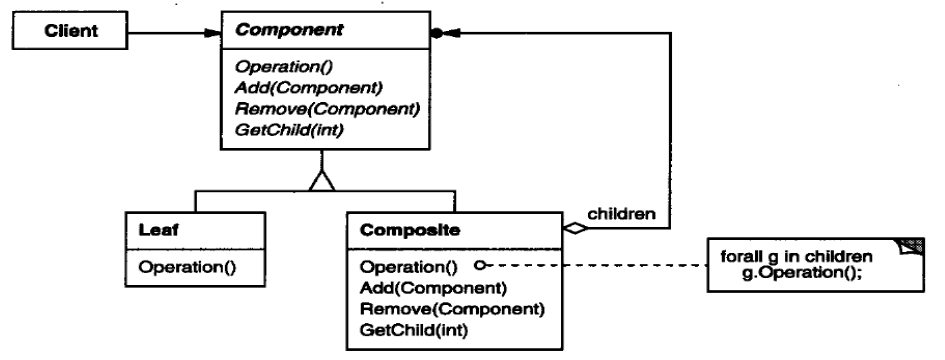
在绘图编辑器和图形捕捉系统这样的图形应用程序中，用户可以使用简单的图元来创建复杂的图标。用户可以组合多个简单图元以形成一些较大的图表。一个简单的实现方法是为Text和Line这样的图元定义一些类，另外定义一些类作为这些图元的容器类。但是这样的问题是：使用这些类的代码必须区别对待图元对象和容器对象，而实际上大多数情况下用户认为他们一样。



1. 目的：将对象组合成树结构，来表示他们的部分-整体关系。Composite可以使客户类统一对待个体对象和对象的组合。
2. 动机：一些部件对象经过组合构成的符合部件对象仍然具有单个部件对象的接口，这样的符合部件对象被称为容器。符合部件与单个部件具有同样的接口，所有接口包含两个部分：单个部件的功能、管理子部件的功能。递归组合如下：



1. 结构：

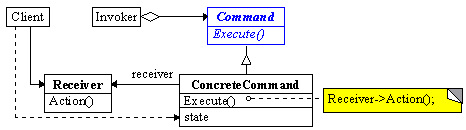


1. 参与者：Client, Component, Leaf, Composite
2. 适用范围：
   1. 想要表示对象的部分-整体层次结构
   2. 希望用户类忽略个体对象和组合对象的区别，希望用户类统一对待Composite中的各个组成部分时。
3. 评价：定义了包含leaf对象和composite对象的类层次接口（递归结构），简化了客户类，抑郁增加新组件，让你的设计能够尽可能泛化。然而，容易增加新组建这个特性也带来了难题，那就是很难限制组合中的组件，因为有时希望一个组合只能有某些特定的组件。使用composite时，不能依赖类型系统来施加这些约束，而必须在运行时检查。
4. 实现：
   1. 显式的父引用：保持从子部件到父部件的引用，能简化组合结构的遍历和管理。父部件引用可以简化结构的上移和组件的删除。通常在Component类中定义父部件引用。Leaf和composite类可以继承这个引用以及管理这个引用的操作。
   2. 最大化组件接口：Composite模式的目的之一是是的用户不知道他们正在使用的具体的Leaf和Composite类。为了达到这一目的，Composite类应该为Leaf和Composite类尽可能多提供一些公共操作。Composite类通常为这些操作提供缺省的实现，而Leaf和Composite子类可以对他们进行重定义。
   3. 声明子部件的管理操作：

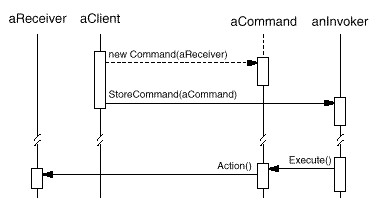
# 行为型模式

## Command

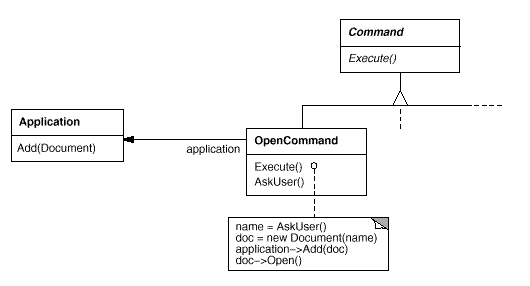
1. Aliases
   1. Action, Transaction
   2. functor (function object)
2. Intent
   1. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
3. Motivation
   1. 把请求信息和请求执行过程封装起来
   2. framework往往需要把命令请求与处理请求的对象分开，command模式可以把调用操作的对象与操作的目标对象分开
   3. 允许通过多种途径调用同一个请求。——请求的重用
4. Applicability：Use the Command pattern when :
   1. parameterize objects by an action to perform，代替回调
   2. specify, queue, and execute requests at different times
   3. support undo
   4. support logging changes so that they can be reapplied in case of a system crash
   5. structure a system around high-level operations built on primitives operations —— transactions
5. 结构：



1. 参与者：Client, Command、ConcreteCommand、Invoker、Receiver
2. Collaborations：

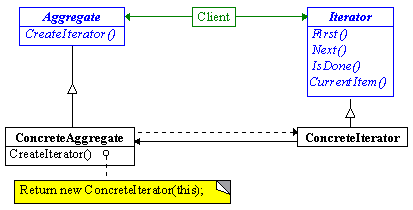


1. 评价：
   1. Command decouples the object that invokes the operation from the one that knows how to perform it.
   2. Commands are first-class objects. They can be manipulated and extended like any other object.
   3. You can assemble commands into a composite command. An example is MacroCommand.
   4. It's easy to add new Commands, because you don't have to change existing classes.
2. 实现：
   1. How intelligent should a command be?
   2. Supporting undo and redo
   3. Avoiding error accumulation in the undo process
   4. Using C++ templates
3. Related Patterns
   1. Composite模式可用来实现command组合
   2. 为实现undo/redo，可以用其他行为模式来管理状态，如memento模式。Command被放到history list之前，可以用prototype模式复制自身
4. 例子：



## Iterator

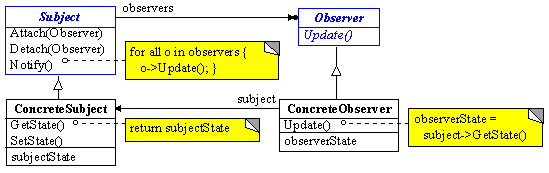
1. Aliases ：Cursor
2. Intent
   1. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
3. Motivation
   1. An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.
   2. Separating the traversal mechanism from the List object lets us define iterators for different traversal policies without enumerating them in the List interface.
4. Applicability：Use the Iterator pattern when :
   1. to access an aggregate object's contents without exposing its internal representation.
   2. to support multiple traversals of aggregate objects.
   3. to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).
5. Struct



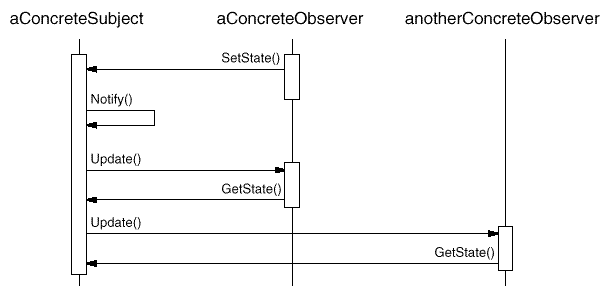
1. Participants
   1. Iterator、ConcreteIterator、Aggregate、ConcreteAggregate
2. Collaborations
3. Evaluation
   1. It supports variations in the traversal of an aggregate
   2. Iterators simplify the Aggregate interface
   3. More than one traversal can be pending on an aggregate
4. Implementation
   1. 实现可以非常灵活
   2. Who controls the iteration?
      1. external iterator *versus* internal iterator
   3. Who defines the traversal algorithm?
      1. Aggregate本身定义算法 —— Cursor mode
      2. iterator定义算法 —— iterator如何访问数据
   4. How robust is the iterator?
   5. Additional Iterator operations.
   6. 基本操作：First, Next, IsDone, and CurrentItem
   7. Using polymorphic iterators —— iterator资源释放
   8. Iterators may have privileged access
   9. Iterators for composites —— 适合于internal iterator或者cursor方式的iterator
   10. Null iterators
5. Related Patterns
   1. Composite：iterator常被用于composite模式的复合结构
   2. Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.
6. Examples
   1. COM enumerator：connectable object、…
   2. ADO/OLE DB
   3. C++ STL
   4. 在STL中，iterator是连接algorithm和container的桥梁

## Observer

1. Aliases ：Dependents, Publish-Subscribe
2. Intent
   1. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
3. Motivation
   1. 把系统分成一些相互关联的类或者对象，如何维护这些类的实例一致性？
   2. The key objects in this pattern are subject and observer
      1. One-to-many relationship
      2. A subject may have any number of dependent observers.
      3. All observers are notified whenever the subject undergoes a change in state.
4. Applicability：Use the Observer pattern when :
   1. When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
   2. When a change to one object requires changing others, and you don't know how many objects need to be changed.
   3. When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.
5. Struct



1. Participants
   1. Subject、ConcreteSubject、Observer、ConcreteObserver
2. Collaborations



1. Evaluation
   1. Abstract coupling between Subject and Observer
   2. Support for broadcast communication
   3. Unexpected updates
2. Implementation
   1. Mapping subjects to their observers.
   2. Observing more than one subject
   3. Who triggers the update? Client or subject？
   4. Making sure Subject state is self-consistent before notification
   5. subject向observer传递变化信息
   6. 中间插入ChangeManager
3. Related Patterns
   1. Mediator：用Mediator模式封装复杂的更新语义
4. Examples
   1. COM property page
   2. COM+ Event Model，
   3. MVC

