

第3章 创建型模式

创建型模式抽象了实例化过程。它们帮助一个系统独立于如何创建、组合和表示它的那些对象。一个类创建型模式使用继承改变被实例化的类，而一个对象创建型模式将实例化委托给另一个对象。

随着系统演化得越来越依赖于对象复合而不是类继承，创建型模式变得更为重要。当这种情况发生时，重心从对一组固定行为的硬编码（hard-coding）转移为定义一个较小的基本行为集，这些行为可以被组合成任意数目的更复杂的行为。这样创建有特定行为的对象要求的不仅仅是实例化一个类。

在这些模式中，有两个不断出现的主旋律。第一，它们都将关于该系统使用哪些具体的类的信息封装起来。第二，它们隐藏了这些类的实例是如何被创建和放在一起的。整个系统关于这些对象所知道的是由抽象类所定义的接口。因此，创建型模式在什么被创建，谁创建它，它是怎样被创建的，以及何时创建这些方面给予你很大的灵活性。它们允许你用结构和功能差别很大的“产品”对象配置一个系统。配置可以是静态的（即在编译时指定），也可以是动态的（在运行时）。

有时创建型模式是相互竞争的。例如，在有些情况下 Prototype（3.4）或 Abstract Factory（3.1）用起来都很好。而在另外一些情况下它们是互补的：Builder（3.2）可以使用其他模式去实现某个构件的创建。Prototype（3.4）可以在它的实现中使用 Singleton（3.5）。

因为创建型模式紧密相关，我们将所有5个模式一起研究以突出它们的相似点和相异点。我们也将举一个通用的例子——为一个电脑游戏创建一个迷宫——来说明它们的实现。这个迷宫和游戏将随着各种模式不同而略有区别。有时这个游戏将仅仅是找到一个迷宫的出口；在这种情况下，游戏者可能仅能见到该迷宫的局部。有时迷宫包括一些要解决的问题和要战胜的危险，并且这些游戏可能会提供已经被探索过的那部分迷宫地图。

我们将忽略许多迷宫中的细节以及一个迷宫游戏中有一个还是多个游戏者。我们仅关注迷宫是怎样被创建的。我们将一个迷宫定义为一组房间，一个房间知道它的邻居；可能的邻居要么是另一个房间，要么是一堵墙、或者是到另一个房间的一扇门。

类 Room、Door 和 Wall 定义了我们所有的例子中使用到的构件。我们仅定义这些类中对创建一个迷宫起重要作用的一些部分。我们将忽略游戏者、显示操作和在迷宫中四处移动操作，以及其他一些重要的却与创建迷宫无关的功能。

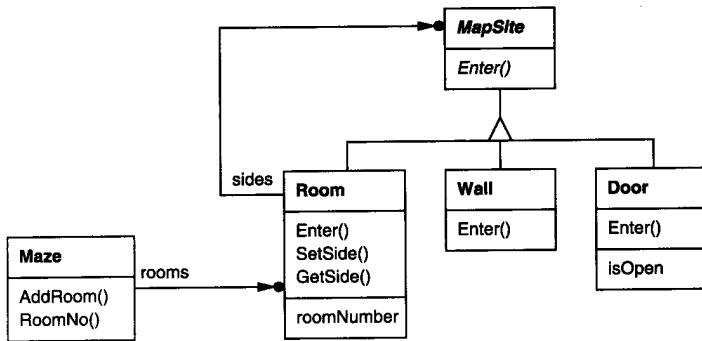
下页图表示了这些类之间的关系。

每一个房间有四面，我们使用 C++ 中的枚举类型 Direction 来指定房间的东南西北：

```
enum Direction {North, South, East, West};
```

Smalltalk 的实现使用相应的符号来表示这些方向。

类 MapSite 是所有迷宫组件的公共抽象类。为简化例子，MapSite 仅定义了一个操作 Enter，它的含义决定于你在进入什么。如果你进入一个房间，那么你的位置会发生改变。如果你试图进入一扇门，那么这两件事中就有一件会发生：如果门是开着的，你进入另一个房间。如



果门是关着的，那么你就会碰壁。

```

class MapSite {
public:
    virtual void Enter() = 0;
};
  
```

Enter为更加复杂的游戏操作提供了一个简单基础。例如，如果你在一个房间中说“向东走”，游戏只能确定直接在东边的是哪一个 MapSite并对它调用Enter。特定子类的Enter操作将计算出你的位置是发生改变，还是你会碰壁。在一个真正的游戏中，Enter可以将移动的游戏者对象作为一个参数。

Room是MapSite的一个具体的子类，而 MapSite定义了迷宫中构件之间的主要关系。Room有指向其他 MapSite对象的引用，并保存一个房间号，这个数字用来标识迷宫中的房间。

```

class Room : public MapSite {
public:
    Room(int roomNo);

    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);

    virtual void Enter();

private:
    MapSite* _sides[4];
    int _roomNumber;
};
  
```

下面的类描述了一个房间的每一面所出现的墙壁或门。

```

class Wall : public MapSite {
public:
    Wall();

    virtual void Enter();
};

class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);

    virtual void Enter();
    Room* OtherSideFrom(Room*);
};
  
```

```
private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

我们不仅需要知道迷宫的各部分，还要定义一个用来表示房间集合的 Maze 类。用 RoomNo 操作和给定的房间号，Maze 就可以找到一个特定的房间。

```
class Maze {
public:
    Maze();

    void AddRoom(Room*);
    Room* RoomNo(int) const;
private:
    // ...
};
```

RoomNo 可以使用线形搜索、hash 表、甚至一个简单数组进行一次查找。但我们在此处并不考虑这些细节，而是将注意力集中于如何指定一个迷宫对象的构件上。

我们定义的另一个类是 MazeGame，由它来创建迷宫。一个简单直接的创建迷宫的方法是使用一系列操作将构件增加到迷宫中，然后连接它们。例如，下面的成员函数将创建一个迷宫，这个迷宫由两个房间和它们之间的一扇门组成：

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

考虑到这个函数所做的仅是创建一个有两个房间的迷宫，它是相当复杂的。显然有办法使它变得更简单。例如，Room 的构造器可以提前用墙壁来初始化房间的每一面。但这仅仅是将代码移到了其他地方。这个成员函数真正的问题不在于它的大小而在于它不灵活。它对迷宫的布局进行硬编码。改变布局意味着改变这个成员函数，或是重定义它——这意味着重新实现整个过程——或是对它的部分进行改变——这容易产生错误并且不利于重用。

创建型模式显示如何使得这个设计更灵活，但未必会更小。特别是，它们将便于修改定

义一个迷宫构件的类。

假设你想在一个包含（所有的东西）施了魔法的迷宫的新游戏中重用已有的迷宫布局。施了魔法的迷宫游戏有新的构件，像 DoorNeedingSpell，它是一扇仅随着一个咒语才能被锁上和打开的门；以及 EnchantedRoom，一个可以有不寻常东西的房间，比如魔法钥匙或是咒语。你怎样才能较容易的改变 CreateMaze以让它用这些新类型的对象创建迷宫呢？

这种情况下，改变的最大障碍是对被实例化的类进行硬编码。创建型模式提供了多种不同方法从实例化它们的代码中除去对这些具体类的显式引用：

- 如果 CreateMaze 调用虚函数而不是构造器来创建它需要的房间、墙壁和门，那么你可以创建一个 MazeGame 的子类并重定义这些虚函数，从而改变被例化的类。这一方法是 Factory Method（3.3）模式的一个例子。
- 如果传递一个对象给 CreateMaze 作参数来创建房间、墙壁和门，那么你可以传递不同的参数来改变房间、墙壁和门的类。这是 Abstract Factory（3.1）模式的一个例子。
- 如果传递一个对象给 CreateMaze，这个对象可以在它所建造的迷宫中使用增加房间、墙壁和门的操作，来全面创建一个新的迷宫，那么你可以使用继承来改变迷宫的一些部分或该迷宫被建造的方式。这是 Builder（3.2）模式的一个例子。
- 如果 CreateMaze 由多种原型的房间、墙壁和门对象参数化，它拷贝并将这些对象增加到迷宫中，那么你可以用不同的对象替换这些原型对象以改变迷宫的构成。这是 Prototype（3.4）模式的一个例子。

剩下的创建型模式，Singleton（3.5），可以保证每个游戏中仅有一个迷宫而且所有的游戏对象都可以迅速访问它——不需要求助于全局变量或函数。Singleton 也使得迷宫易于扩展或替换，且不需变动已有的代码。

3.1 ABSTRACT FACTORY（抽象工厂）——对象创建型模式

1. 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。

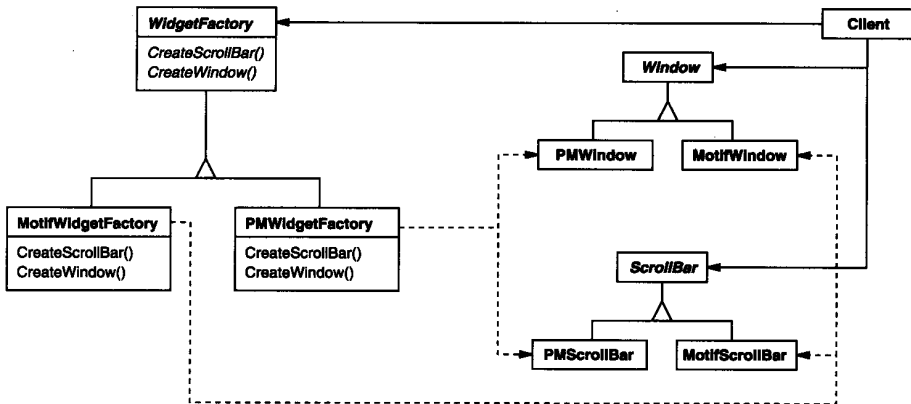
2. 别名

Kit

3. 动机

考虑一个支持多种视感（look-and-feel）标准的用户界面工具包，例如 Motif 和 Presentation Manager。不同的视感风格为诸如滚动条、窗口和按钮等用户界面“窗口组件”定义不同的外观和行为。为保证视感风格标准间的可移植性，一个应用不应该为一个特定的视感外观硬编码它的窗口组件。在整个应用中实例化特定视感风格的窗口组件类将使得以后很难改变视感风格。

为解决这一问题我们可以定义一个抽象的 WidgetFactory 类，这个类声明了一个用来创建每一类基本窗口组件的接口。每一类窗口组件都有一个抽象类，而具体子类则实现了窗口组件的特定视感风格。对于每一个抽象窗口组件类，WidgetFactory 接口都有一个返回新窗口组件对象的操作。客户调用这些操作以获得窗口组件实例，但客户并不知道他们正在使用的是哪些具体类。这样客户就不依赖于一般的视感风格，如下页图所示。



每一种视感标准都对应于一个具体的 **WidgetFactory** 子类。每一子类实现那些用于创建合适视感风格的窗口组件的操作。例如，**MotifWidgetFactory** 的 `CreateScrollBar` 操作实例化并返回一个 Motif 滚动条，而相应的 **PMWidgetFactory** 操作返回一个 Presentation Manager 的滚动条。客户仅通过 **WidgetFactory** 接口创建窗口组件，他们并不知道哪些类实现了特定视感风格的窗口组件。换言之，客户仅与抽象类定义的接口交互，而不使用特定的具体类的接口。

WidgetFactory 也增强了具体窗口组件类之间依赖关系。一个 Motif 的滚动条应该与 Motif 按钮、Motif 正文编辑器一起使用，这一约束条件作为使用 **MotifWidgetFactory** 的结果被自动加上。

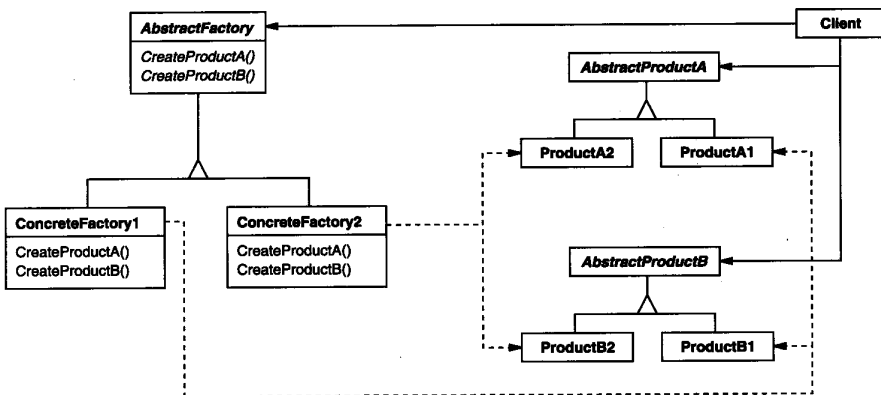
4. 适用性

在以下情况可以使用 Abstract Factory 模式

- 一个系统要独立于它的产品的创建、组合和表示时。
- 一个系统要由多个产品系列中的一个来配置时。
- 当你要强调一系列相关的产品对象的设计以便进行联合使用时。
- 当你提供一个产品类库，而只想显示它们的接口而不是实现时。

5. 结构

此模式的结构如下图所示。



6. 参与者

- **AbstractFactory** (**WidgetFactory**)

— 声明一个创建抽象产品对象的操作接口。

- ConcreteFactory (MotifWidgetFactory , PMWidgetFactory)

— 实现创建具体产品对象的操作。

- AbstractProduct (Windows , ScrollBar)

— 为一类产品对象声明一个接口。

- ConcreteProduct (MotifWindow , MotifScrollBar)

— 定义一个将被相应的具体工厂创建的产品对象。

— 实现AbstractProduct接口。

- Client

— 仅使用由AbstractFactory和AbstractProduct类声明的接口。

7. 协作

- 通常在运行时刻创建一个 ConcreteFactory类的实例。这一具体的工厂创建具有特定实现的产品对象。为创建不同的产品对象，客户应使用不同的具体工厂。
- AbstractFactory将产品对象的创建延迟到它的 ConcreteFactory子类。

8. 效果

AbstractFactory模式有下面的一些优点和缺点：

1) 它分离了具体的类 Abstract Factory模式帮助你控制一个应用创建的对象的类。因为一个工厂封装创建产品对象的责任和过程，它将客户与类的实现分离。客户通过它们的抽象接口操纵实例。产品的类名也在具体工厂的实现中被分离；它们不出现在客户代码中。

2) 它使得易于交换产品系列 一个具体工厂类在一个应用中仅出现一次——即在它初始化的时候。这使得改变一个应用的具体工厂变得很容易。它只需改变具体的工厂即可使用不同的产品配置，这是因为一个抽象工厂创建了一个完整的产品系列，所以整个产品系列会立刻改变。在我们的用户界面的例子中，我们仅需转换到相应的工厂对象并重新创建接口，就可实现从Motif窗口组件转换为Presentation Manager窗口组件。

3) 它有利于产品的一致性 当一个系列中的产品对象被设计成一起工作时，一个应用一次只能使用同一个系列中的对象，这一点很重要。而 AbstractFactory很容易实现这一点。

4) 难以支持新种类的产品 难以扩展抽象工厂以生产新种类的产品。这是因为 AbstractFactory接口确定了可以被创建的产品集合。支持新种类的产品就需要扩展该工厂接口，这将涉及 AbstractFactory类及其所有子类的改变。我们会在实现一节讨论这个问题的一个解决办法。

9. 实现

下面是实现 Abstract Factory模式的一些有用技术：

1) 将工厂作为单件 一个应用中一般每个产品系列只需一个 ConcreteFactory的实例。因此工厂通常最好实现为一个 Singleton (3.5)。

2) 创建产品 AbstractFactory仅声明一个创建产品的接口，真正创建产品是由 ConcreteProduct子类实现的。最通常的一个办法是为每一个产品定义一个工厂方法（参见 Factory Method (3.3)）。一个具体的工厂将为每个产品重定义该工厂方法以指定产品。虽然这样的实现很简单，但它却要求每个产品系列都要有一个新的具体工厂子类，即使这些产品系列的差别很小。

如果有多个可能的产品系列，具体工厂也可以使用 Prototype (3.4) 模式来实现。具体工厂使用产品系列中每一个产品的原型实例来初始化，且它通过复制它的原型来创建新的产品。在基于原型的方法中，使得不是每个新的产品系列都需要一个新的具体工厂类。

此处是 Smalltalk 中实现一个基于原型的工厂的方法。具体工厂在一个被称为 partCatalog 的字典中存储将被复制的原型。方法 make：检索该原型并复制它：

```
make : partName
    ^ (partCatalog at : partName) copy
```

具体工厂有一个方法用来向该目录中增加部件。

```
addPart : partTemplate named : partName
    partCatalog at : partName put : partTemplate
```

原型通过用一个符号标识它们，从而被增加到工厂中：

```
aFactory addPart : aPrototype named : #ACMEWidget
```

在将类作为第一类对象的语言中（例如 Smalltalk 和 ObjectiveC），这个基于原型的方法可能有所变化。你可以将这些语言中的一个类看成是一个退化的工厂，它仅创建一种产品。你可以将类存储在一个具体工厂中，这个具体工厂在变量中创建多个具体的产品，这很像原型。这些类代替具体工厂创建了新的实例。你可以通过使用产品的类而不是子类初始化一个具体工厂的实例，来定义一个新的工厂。这一方法利用了语言的特点，而纯基于原型的方法是与语言无关的。

像刚讨论过的 Smalltalk 中的基于原型的工厂一样，基于类的版本将有一个唯一的实例变量 partCatalog，它是一个字典，它的主键是各部分的名字。partCatalog 存储产品的类而不是存储被复制的原型。方法 make：现在是这样：

```
make : partName
    ^ (partCatalog at : partName) new
```

3) 定义可扩展的工厂 AbstractFactory 通常为每一种它可以生产的产品定义一个操作。产品的种类被编码在操作型构中。增加一种新的产品要求改变 AbstractFactory 的接口以及所有与它相关的类。一个更灵活但不太安全的设计是给创建对象的操作增加一个参数。该参数指定了将被创建的对象种类。它可以是一个类标识符、一个整数、一个字符串，或其他任何可以标识这种产品的东西。实际上使用这种方法，AbstractFactory 只需要一个“Make”操作和一个指示要创建对象种类参数。这是前面已经讨论过的基于原型的和基于类的抽象工厂的技术。

C++ 这样的静态类型语言与相比，这一变化更容易用在类似于 Smalltalk 这样的动态类型语言中。仅当所有对象都有相同的抽象基类，或者当产品对象可以被请求它们的客户安全的强制转换成正确类型时，你才能够在 C++ 中使用它。Factory Method (3.3) 的实现部分说明了怎样在 C++ 中实现这样的参数化操作。

该方法即使不需要类型强制转换，但仍有一个本质的问题：所有的产品将返回类型所给定的相同的抽象接口返回给客户。客户将不能区分或对一个产品的类别进行安全的假定。如果一个客户需要进行与特定子类相关的操作，而这些操作却不能通过抽象接口得到。虽然客户可以实施一个向下类型转换（downcast）（例如在 C++ 中用 dynamic_cast），但这并不总是可行或安全的，因为向下类型转换可能会失败。这是一个典型的高度灵活和可扩展接口的权衡

折衷。

10. 代码示例

我们将使用 Abstract Factory 模式创建我们在这章开始所讨论的迷宫。

类 MazeFactory 可以创建迷宫的组件。它建造房间、墙壁和房间之间的门。它可以用于一个从文件中读取迷宫说明图并建造相应迷宫的程序。或者它可以被用于一个随机建造迷宫的程序。建造迷宫的程序将 MazeFactory 作为一个参数，这样程序员就能指定要创建的房间、墙壁和门等类。

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
    { return new Maze; }
    virtual Wall* MakeWall() const
    { return new Wall; }
    virtual Room* MakeRoom(int n) const
    { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door(r1, r2); }
};
```

回想一下建立一个由两个房间和它们之间的门组成的小迷宫的成员函数 CreateMaze。CreateMaze 对类名进行硬编码，这使得很难用不同的组件创建迷宫。

这里是一个以 MazeFactory 为参数的新版本 CreateMaze，它修改了以上缺点：

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());
    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

我们创建 MazeFactory 的子类 EnchantedMazeFactory，这是一个创建施了魔法的迷宫的工厂。EnchantedMazeFactory 将重定义不同的成员函数并返回 Room，Wall 等不同的子类。

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
```



```
{ return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

现在假设我们想生成一个迷宫游戏，在这个游戏里，每个房间中可以有一个炸弹。如果这个炸弹爆炸，它将（至少）毁坏墙壁。我们可以生成一个 Room 的子类以明了是否有一个炸弹在房间中以及该炸弹是否爆炸了。我们也将需要一个 Wall 的子类以明了对墙壁的损坏。我们将称这些类为 RoomWithABomb 和 BombedWall。

我们将定义的最后一个是 BombedMazeFactory，它是 MazeFactory 的子类，保证了墙壁是 BombedWall 类的而房间是 RoomWithABomb 的。BombedMazeFactory 仅需重定义两个函数：

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

为建造一个包含炸弹的简单迷宫，我们仅用 BombedMazeFactory 调用 CreateMaze。

```
MazeGame game;
BombedMazeFactory factory;
```

```
game.CreateMaze(factory);
```

CreateMaze 也可以接收一个 EnchantedMazeFactory 实例来建造施了魔法的迷宫。

注意 MazeFactory 仅是工厂方法的一个集合。这是最通常的实现 Abstract Factory 模式的方式。同时注意 MazeFactory 不是一个抽象类；因此它既作为 AbstractFactory 也作为 ConcreteFactory。这是 Abstract Factory 模式的简单应用的另一个通常的实现。因为 MazeFactory 是一个完全由工厂方法组成的具体类，通过生成一个子类并重定义需要改变的操作，它很容易生成一个新的 MazeFactory。

CreateMaze 使用房间的 SetSide 操作以指定它们的各面。如果它用一个 BombedMazeFactory 创建房间，那么该迷宫将由有 BombedWall 面的 RoomWithABomb 对象组成。如果 RoomWithABomb 必须访问一个 BombedWall 的与特定子类相关的成员，那么它将不得不对它的墙壁引用以进行从 Wall* 到 BombedWall* 的转换。只要该参数确实是一个 BombedWall，这个向下类型转换就是安全的，而如果墙壁仅由一个 BombedMazeFactory 创建就可以保证这一点。

当然，像 Smalltalk 这样的动态类型语言不需要向下类型转换，但如果它们在应该是 Wall 的子类的地方遇到一个 Wall 类可能会产生运行时刻错误。使用 Abstract Factory 建造墙壁，通过确定仅有特定类型的墙壁可以被创建，从而有助于防止这些运行时刻错误。

让我们考虑一个 Smalltalk 版本的 MazeFactory，它仅有一个以要生成的对象种类为参数的 make 操作。此外，具体工厂存储它所创建的产品类的类。

首先，我们用 Smalltalk 写一个等价的 CreateMaze：

```
createMaze: aFactory
| room1 room2 aDoor |
room1 := (aFactory make: #room) number: 1.
room2 := (aFactory make: #room) number: 2.
aDoor := (aFactory make: #door) from: room1 to: room2.
```

```

room1 atSide: #north put: (aFactory make: #wall).
room1 atSide: #east put: aDoor.
room1 atSide: #south put: (aFactory make: #wall).
room1 atSide: #west put: (aFactory make: #wall).
room2 atSide: #north put: (aFactory make: #wall).
room2 atSide: #east put: (aFactory make: #wall).
room2 atSide: #south put: (aFactory make: #wall).
room2 atSide: #west put: aDoor.
^ Maze new addRoom: room1; addRoom: room2; yourself

```

正如我们在实现一节所讨论，MazeFactory 仅需一个实例变量 partCatalog 来提供一个字典，这个字典的主键为迷宫组件的类。也回想一下我们是如何实现 make: 方法的：

```

make: partName
^ (partCatalog at: partName) new

```

现在我们可以创建一个 MazeFactory 并用它来实现 CreateMaze。我们将用类 MazeGame 的一个方法 CreateMazeFactory 来创建该工厂。

```

createMazeFactory
^ (MazeFactory new
  addPart: Wall named: #wall;
  addPart: Room named: #room;
  addPart: Door named: #door;
  yourself)

```

通过将不同的类与它们的主键相关联，就可以创建一个 BombedMazeFactory 或 EnchantedMazeFactory。例如，一个 EnchantedMazeFactory 可以这样被创建：

```

createMazeFactory
^ (MazeFactory new
  addPart: Wall named: #wall;
  addPart: EnchantedRoom named: #room;
  addPart: DoorNeedingSpell named: #door;
  yourself)

```

11. 已知应用

InterView 使用 “Kit” 后缀 [Lin92] 来表示 AbstractFactory 类。它定义 WidgetKit 和 DialogKit 抽象工厂来生成与特定视感风格相关的用户界面对象。InterView 还包括一个 LayoutKit，它根据所需要的布局生成不同的组成（composition）对象。例如，一个概念上是水平的布局根据文档的定位（画像或是风景）可能需要不同的组成对象。

ET++ [WGM88] 使用 Abstract Factory 模式以达到在不同窗口系统（例如，X Windows 和 SunView）间的可移植性。WindowSystem 抽象基类定义一些接口，来创建表示窗口系统资源的对象（例如 MakeWindow、MakeFont、MakeColor）。具体的子类为某个特定的窗口系统实现这些接口。运行时刻，ET++ 创建一个具体 WindowSystem 子类的实例，以创建具体的系统资源对象。

12. 相关模式

AbstractFactory 类通常用工厂方法（Factory Method（3.3））实现，但它们也可以用 Prototype 实现。

一个具体的工厂通常是一个单件（Singleton（3.5））。

3.2 BUILDER（生成器）——对象创建型模式

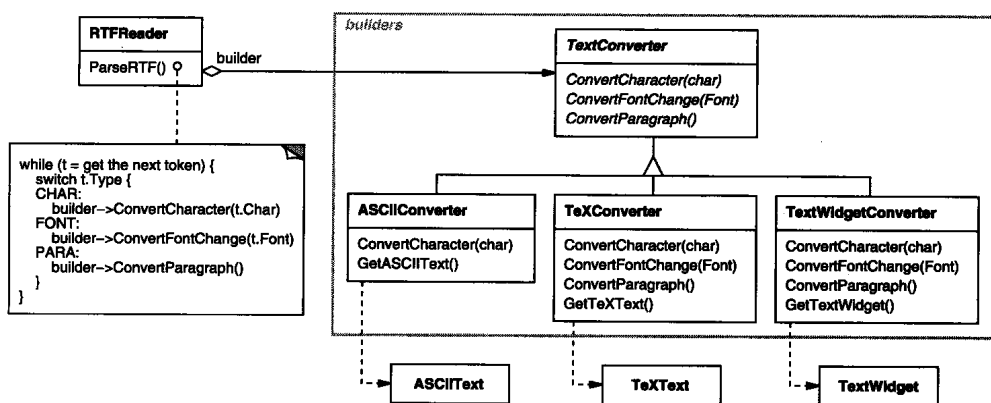
1. 意图

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

2. 动机

一个RTF (Rich Text Format) 文档交换格式的阅读器应能将 RTF转换为多种正文格式。该阅读器可以将RTF文档转换成普通ASCII文本或转换成一个能以交互方式编辑的正文窗口组件。但问题在于可能转换的数目是无限的。因此要能够很容易实现新的转换的增加，同时却不改变RTF阅读器。

一个解决办法是用一个可以将 RTF转换成另一种正文表示的 TextConverter对象配置这个 RTFReader类。当RTFReader对RTF文档进行语法分析时，它使用 TextConverter去做转换。无论何时RTFReader识别了一个RTF标记（或是普通正文或是一个 RTF控制字），它都发送一个请求给TextConverter去转换这个标记。TextConverter对象负责进行数据转换以及用特定格式表示该标记，如下图所示。



TextConvert的子类对不同转换和不同格式进行特殊处理。例如，一个 ASCIIConverter只负责转换普通文本，而忽略其他转换请求。另一方面，一个 TeXConverter将会为对所有请求的操作，以便生成一个获取正文中所有风格信息的 TEX表示。一个TextWidgetConverter将生成一个复杂的用户界面对象以便用户浏览和编辑正文。

每种转换器类将创建和装配一个复杂对象的机制隐含在抽象接口的后面。转换器独立于阅读器，阅读器负责对一个RTF文档进行语法分析。

Builder模式描述了所有这些关系。每一个转换器类在该模式中被称为生成器（ builder），而阅读器则称为导向器（ director）。在上面的例子中， Builder模式将分析文本格式的算法（即RTF文档的语法分析程序）与描述怎样创建和表示一个转换后格式的算法分离开来。这使我们可以重用RTFReader的语法分析算法，根据RTF文档创建不同的正文表示——仅需使用不同的TextConverter的子类配置该RTFReader即可。

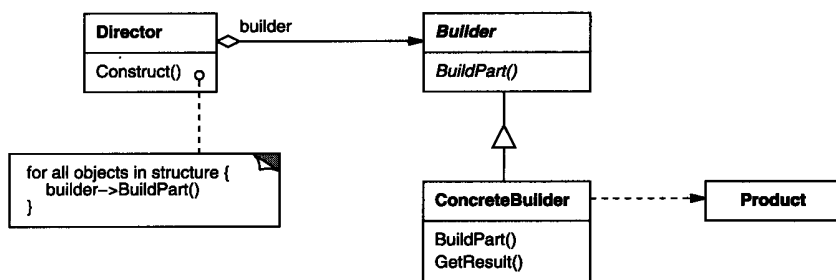
3. 适用性

在以下情况使用 Builder模式

- 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。
- 当构造过程必须允许被构造的对象有不同的表示时。

4. 结构

此模式结构如下页上图所示。



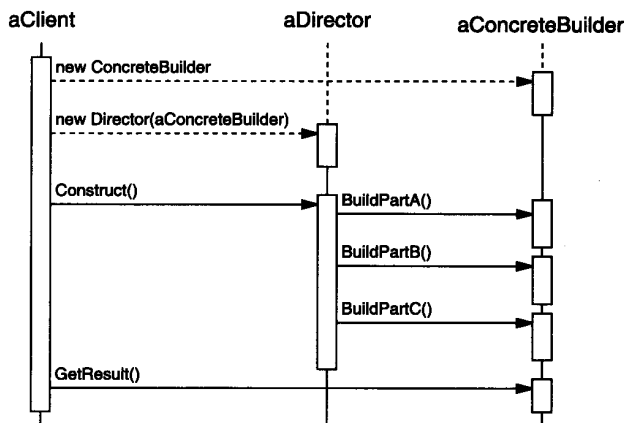
5. 参与者

- **Builder (TextConverter)**
 - 为创建一个 `Product` 对象的各个部件指定抽象接口。
- **ConcreteBuilder (ASCIIConverter、TeXConverter、TextWidgetConverter)**
 - 实现 `Builder` 的接口以构造和装配该产品的各个部件。
 - 定义并明确它所创建的表示。
 - 提供一个检索产品的接口（例如，`GetASCIIText` 和 `GetTextWidget`）。
- **Director (RTFReader)**
 - 构造一个使用 `Builder` 接口的对象。
- **Product (ASCIIText、TeXText、TextWidget)**
 - 表示被构造的复杂对象。`ConcreteBuilder` 创建该产品的内部表示并定义它的装配过程。
 - 包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

6. 协作

- 客户创建 `Director` 对象，并用它所想要的 `Builder` 对象进行配置。
- 一旦产品部件被生成，导向器就会通知生成器。
- 生成器处理导向器的请求，并将部件添加到该产品中。
- 客户从生成器中检索产品。

下面的交互图说明了 `Builder` 和 `Director` 是如何与一个客户协作的。



7. 效果

这里是 `Builder` 模式的主要效果：

1) 它使你可以改变一个产品的内部表示 Builder对象提供给导向器一个构造产品的抽象接口。该接口使得生成器可以隐藏这个产品的表示和内部结构。它同时也隐藏了该产品是如何装配的。因为产品是通过抽象接口构造的，你在改变该产品的内部表示时所要做的只是定义一个新的生成器。

2) 它将构造代码和表示代码分开 Builder模式通过封装一个复杂对象的创建和表示方式提高了对象的模块性。客户不需要知道定义产品内部结构的类的所有信息；这些类是不出现在Builder接口中的。每个 ConcreteBuilder包含了创建和装配一个特定产品的所有代码。这些代码只需要写一次；然后不同的 Director可以复用它在相同部件集合的基础上构造不同的 Product。在前面的 RTF例子中，我们可以为 RTF格式以外的格式定义一个阅读器，比如一个 SGMLReader，并使用相同的 TextConverter生成 SGML文档的 ASCIIText、TeXText和 TextWidget译本。

3) 它使你可对构造过程进行更精细的控制 Builder模式与一下子就生成产品的创建型模式不同，它是在导向者的控制下一步一步构造产品的。仅当该产品完成时导向者才从生成器中取回它。因此 Builder接口相比其他创建型模式能更好的反映产品的构造过程。这使你可以更精细的控制构建过程，从而能更精细的控制所得产品的内部结构。

8. 实现

通常有一个抽象的 Builder类为导向者可能要求创建的每一个构件定义一个操作。这些操作缺省情况下什么都不做。一个 ConcreteBuilder类对它有兴趣创建的构件重定义这些操作。

这里是其他一些要考虑的实现问题：

1) 装配和构造接口 生成器逐步的构造它们的产品。因此 Builder类接口必须足够普遍，以便为各种类型的具体生成器构造产品。

一个关键的设计问题在于构造和装配过程的模型。构造请求的结果只是被添加到产品中，通常这样的模型就已足够了。在 RTF的例子中，生成器转换下一个标记并将它添加到它已经转换了的正文中。

但有时你可能需要访问前面已经构造了的产品部件。我们在代码示例一节所给出的 Maze例子中，MazeBuilder接口允许你在已经存在的房间之间增加一扇门。像语法分析树这样自底向上构建的树型结构就是另一个例子。在这种情况下，生成器会将子结点返回给导向者，然后导向者将它们回传给生成者去创建父结点。

2) 为什么产品没有抽象类 通常情况下，由具体生成器生成的产品，它们的表示相差是如此之大以至于给不同的产品以公共父类没有太大意思。在 RTF例子中，ASCIIText和 TextWidget对象不太可能有公共接口，它们也不需要这样的接口。因为客户通常用合适的具体生成器来配置导向者，客户处于的位置使它知道 Builder的哪一个具体子类被使用和能相应的处理它的产品。

3) 在Builder中却省的方法为空 C++中，生成方法故意不声明为纯虚成员函数，而是把它们定义为空方法，这使客户只重定义他们所感兴趣的操作。

9. 代码示例

我们将定义一个 CreateMaze成员函数的变体，它以类 MazeBuilder的一个生成器对象作为参数。

MazeBuilder类定义下面的接口来创建迷宫：

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

该接口可以创建：1) 迷宫。2) 有一个特定房间号的房间。3) 在有号码的房间之间的门。GetMaze操作返回这个迷宫给客户。MazeBuilder的子类将重定义这些操作，返回它们所创建的迷宫。

MazeBuilder的所有建造迷宫的操作缺省时什么也不做。不将它们定义为纯虚函数是为了便于派生类只重定义它们所感兴趣的那些方法。

用MazeBuilder接口，我们可以改变CreateMaze成员函数，以生成器作为它的参数。

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

将这个CreateMaze版本与原来的相比，注意生成器是如何隐藏迷宫的内部表示的——即定义房间、门和墙壁的那些类——以及这些部件是如何组装成最终的迷宫的。有人可能猜测到有一些类是用来表示房间和门的，但没有迹象显示哪个类是用来表示墙壁的。这就使得改变一个迷宫的表示方式要容易一些，因为所有MazeBuilder的客户都不需要被改变。

像其他创建型模式一样，Builder模式封装了对象是如何被创建的，在这个例子中是通过MazeBuilder所定义的接口来封装的。这就意味着我们可以重用MazeBuilder来创建不同种类的迷宫。CreateComplexMaze操作给出了一个例子：

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

注意MazeBuilder自己并不创建迷宫；它的主要目的仅仅是为创建迷宫定义一个接口。它主要为方便起见定义一些空的实现。MazeBuilder的子类做实际工作。

子类StandardMazeBuilder是一个创建简单迷宫的实现。将它正在创建的迷宫放在变量_currentMaze中。

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
```



```

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};

```

CommonWall是一个功能性操作，它决定两个房间之间的公共墙壁的方位。

StandardMazeBuilder的构造器只初始化了_currentMaze。

```

StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}

```

BuildMaze实例化一个Maze，它将被其他操作装配并最终返回给客户（通过 GetMaze）。

```

void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

```

```

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}

```

BuildRoom操作创建一个房间并建造它周围的墙壁：

```

void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}

```

为建造一扇两个房间之间的门，StandardMazeBuilder查找迷宫中的这两个房间并找到它们相邻的墙：

```

void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}

```

客户现在可以用CreateMaze和StandardMazeBuilder来创建一个迷宫：

```

Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();

```

我们本可以将所有的StandardMazeBuilder操作放在Maze中并让每一个Maze创建它自身。但将Maze变得小一些使得它能更容易被理解和修改，而且StandardMazeBuilder易于从Maze中分离。更重要的是，将两者分离使得你可以有多种MazeBuilder，每一种使用不同的房间、墙壁和门的类。

一个更特殊的MazeBuilder是CountingMazeBuilder。这个生成器根本不创建迷宫；它仅仅对已被创建的不同种类的构件进行计数。

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

构造器初始化该计数器，而重定义了MazeBuilder操作只是相应的增加计数。

```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

下面是一个客户可能怎样使用 CountingMazeBuilder：

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "The maze has "
    << rooms << " rooms and "
    << doors << " doors" << endl;
```

10. 已知应用

RTF转换器应用来自ET++[WGM88]。它的正文生成模块使用一个生成器处理以RTF格式存储的正文。

生成器在Smalltalk-80[Par90]中是一个通用的模式：

- 编译子系统中的Parser类是一个Director，它以一个ProgramNodeBuilder对象作为参数。每当Parser对象识别出一个语法结构时，它就通知它的ProgramNodeBuilder对象。当这个语法分析器做完时，它向该生成器请求它生成的语法分析树并将语法分析树返回给客户。

- ClassBuilder是一个生成器，Class使用它为自己创建子类。在这个例子中，一个Class既是Director也是Product。
- ByteCodeStream是一个生成器，它将一个被编译了的方法创建为字节数组。ByteCodeStream不是Builder模式的标准使用，因为它生成的复杂对象被编码为一个字节数组，而不是正常的Smalltalk对象。但ByteCodeStream的接口是一个典型的生成器，而且将很容易用一个将程序表示为复合对象的不同的类来替换ByteCodeStream。

自适应通讯环境（Adaptive Communications Environment）中的服务配置者（Service Configurator）框架使用生成器来构造运行时刻动态连接到服务器的网络服务构件 [SS94]。这些构件使用一个被LALR（1）语法分析器进行语法分析的配置语言来描述。这个语法分析器的语义动作对将信息加载给服务构件的生成器进行操作。在这个例子中，语法分析器就是Director。

11. 相关模式

Abstract Factory（3.1）与Builder相似，因为它也可以创建复杂对象。主要的区别是Builder模式着重于一步步构造一个复杂对象。而Abstract Factory着重于多个系列的产品对象（简单的或是复杂的）。Builder在最后的一步返回产品，而对于Abstract Factory来说，产品是立即返回的。

Composite（4.3）通常是用Builder生成的。

3.3 FACTORY METHOD（工厂方法）——对象创建型模式

1. 意图

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使一个类的实例化延迟到其子类。

2. 别名

虚构造器（Virtual Constructor）

3. 动机

框架使用抽象类定义和维护对象之间的关系。这些对象的创建通常也由框架负责。

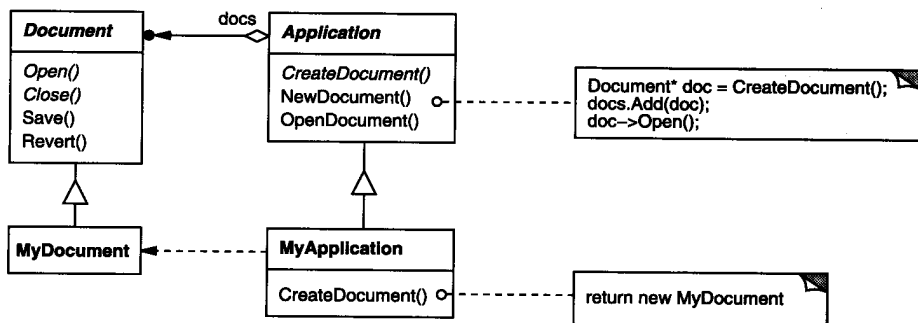
考虑这样一个应用框架，它可以向用户显示多个文档。在这个框架中，两个主要的抽象是类Application和Document。这两个类都是抽象的，客户必须通过它们的子类来做与具体应用相关的实现。例如，为创建一个绘图应用，我们定义类DrawingApplication和DrawingDocument。Application类负责管理Document并根据需要创建它们——例如，当用户从菜单中选择Open或New的时候。

因为被实例化的特定Document子类是与特定应用相关的，所以Application类不可能预测到哪个Document子类将被实例化——Application类仅知道一个新的文档何时应被创建，而不知道哪一种Document将被创建。这就产生了一个尴尬的局面：框架必须实例化类，但是它只知道不能被实例化的抽象类。

Factory Method模式提供了一个解决办法。它封装了哪一个Document子类将被创建的信息并将这些信息从该框架中分离出来，如下页上图所示。

Application的子类重定义Application的抽象操作CreateDocument以返回适当的Document子类对象。一旦一个Application子类实例化以后，它就可以实例化与应用相关的文档，而无

需知道这些文档的类。我们称 CreateDocument 是一个工厂方法 (factory method), 因为它负责 “生产” 一个对象。

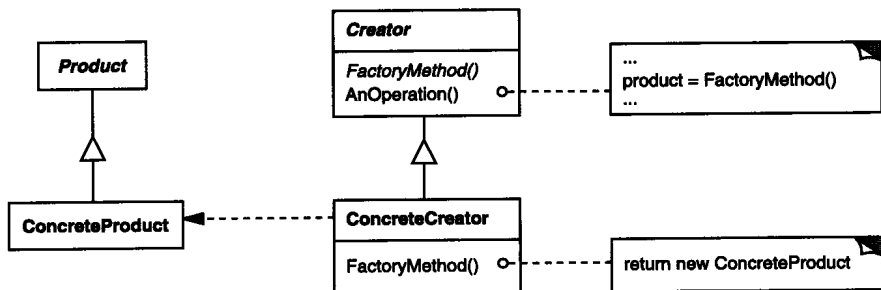


4. 适用性

在下列情况下可以使用 Factory Method 模式：

- 当一个类不知道它所必须创建的对象类的時候。
- 当一个类希望由它的子类来指定它所创建的对象的时候。
- 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

5. 结构



6. 参与者

- **Product (Document)**
— 定义工厂方法所创建的对象接口。
- **ConcreteProduct (MyDocument)**
— 实现 Product 接口。
- **Creator (Application)**
— 声明工厂方法，该方法返回一个 Product 类型的对象。Creator 也可以定义一个工厂方法的缺省实现，它返回一个缺省的 ConcreteProduct 对象。
— 可以调用工厂方法以创建一个 Product 对象。
- **ConcreteCreator (MyApplication)**
— 重定义工厂方法以返回一个 ConcreteProduct 实例。

7. 协作

- **Creator** 依赖于它的子类来定义工厂方法，所以它返回一个适当的 ConcreteProduct 实例。

8. 效果

工厂方法不再将与特定应用有关的类绑定到你的代码中。代码仅处理 `Product` 接口；因此它可以与用户定义的任何 `ConcreteProduct` 类一起使用。

工厂方法的一个潜在缺点在于客户可能仅仅为了创建一个特定的 `ConcreteProduct` 对象，就不得不创建 `Creator` 的子类。当 `Creator` 子类不必用时，客户现在必然要处理类演化的其他方面；但是当客户无论如何必须创建 `Creator` 的子类时，创建子类也是可行的。

下面是 `Factory Method` 模式的另外两种效果：

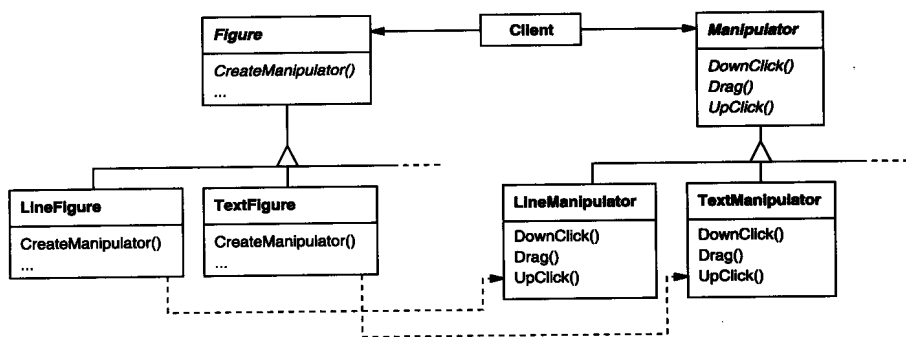
1) 为子类提供挂钩 (hook) 用工厂方法在一个类的内部创建对象通常比直接创建对象更灵活。`Factory Method` 给子类一个挂钩以提供对象的扩展版本。

在 `Document` 的例子中，`Document` 类可以定义一个称为 `CreateFileDialog` 的工厂方法，该方法为打开一个已有的文档创建默认的文件对话框对象。`Document` 的子类可以重定义这个工厂方法以定义一个与特定应用相关的文件对话框。在这种情况下，工厂方法就不再抽象了而是提供了一个合理的缺省实现。

2) 连接平行的类层次 迄今为止，在我们所考虑的例子中，工厂方法并不往往只是被 `Creator` 调用，客户可以找到一些有用的工厂方法，尤其在平行类层次的情况下。

当一个类将它的一些职责委托给一个独立的类的时候，就产生了平行类层次。考虑可以被交互操纵的图形；也就是说，它们可以用鼠标进行伸展、移动，或者旋转。实现这样一些交互并不总是那么容易，它通常需要存储和更新在给定时刻记录操纵状态的信息，这个状态仅仅在操纵时需要。因此它不需要被保存在图形对象中。此外，当用户操纵图形时，不同的图形有不同的行为。例如，将直线图形拉长可能会产生一个端点被移动的效果，而伸展正文图形则可能会改变行距。

有了这些限制，最好使用一个独立的 `Manipulator` 对象实现交互并保存所需要的任何与特定操纵相关的状态。不同的图形将使用不同的 `Manipulator` 子类来处理特定的交互。得到的 `Manipulator` 类层次与 `Figure` 类层次是平行（至少部分平行），如下图所示。



`Figure` 类提供了一个 `CreateManipulator` 工厂方法，它使得客户可以创建一个与 `Figure` 相对应的 `Manipulator`。`Figure` 子类重定义该方法以返回一个合适的 `Manipulator` 子类实例。做为一种选择，`Figure` 类可以实现 `CreateManipulator` 以返回一个默认的 `Manipulator` 实例，而 `Figure` 子类可以只是继承这个缺省实现。这样的 `Figure` 类不需要相应的 `Manipulator` 子类——因此该层次只是部分平行的。

注意工厂方法是怎样定义两个类层次之间的连接的。它将哪些类应一同工作工作的信息局部化了。

9. 实现

当应用Factory Method模式时要考虑下面一些问题：

1) 主要有两种不同的情况 Factory Method模式主要有两种不同的情况：1) 第一种情况是，Creator类是一个抽象类并且不提供它所声明的工厂方法的实现。2) 第二种情况是，Creator是一个具体的类而且为工厂方法提供一个缺省的实现。也有可能有一个定义了缺省实现的抽象类，但这不太常见。

第一种情况需要子类来定义实现，因为没有合理的缺省实现。它避免了不得不实例化不可预见类的问题。在第二种情况中，具体的Creator主要因为灵活性才使用工厂方法。它所遵循的准则是，“用一个独立的操作创建对象，这样子类才能重定义它们的创建方式。”这条准则保证了子类的设计者能够在必要的时候改变父类所实例化的对象的类。

2) 参数化工厂方法 该模式的另一种情况使得工厂方法可以创建多种产品。工厂方法采用一个标识要被创建的对象种类的参数。工厂方法创建的所有对象将共享Product接口。在Document的例子中，Application可能支持不同种类的Document。你给CreateDocument传递一个外部参数来指定将要创建的文档的种类。

图形编辑框架Unidraw [VL90]使用这种方法来重构存储在磁盘上的对象。Unidraw定义了一个Creator类，该类拥有一个以类标识符为参数的工厂方法Create。类标识符指定要被实例化的类。当Unidraw将一个对象存盘时，它首先写类标识符，然后是它的实例变量。当它从磁盘中重构该对象时，它首先读取的是类标识符。

一旦类标识符被读取后，这个框架就将该标识符作为参数，调用Create。Create到构造器中查询相应的类并用它实例化对象。最后，Create调用对象的Read操作，读取磁盘上剩余的信息并初始化该对象的实例变量。

一个参数化的工厂方法具有如下的一般形式，此处MyProduct和YourProduct是Product的子类：

```
class Creator {
public:
    virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
    if (id == MINE) return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // repeat for remaining products...

    return 0;
}
```

重定义一个参数化的工厂方法使你可以简单而有选择性的扩展或改变一个Creator生产的产品。你可以为新产品引入新的标识符，或可以将已有的标识符与不同的产品相关联。

例如，子类MyCreator可以交换MyProduct和YourProduct并且支持一个新的子类TheirProduct：

```
Product* MyCreator::Create (ProductId id) {
    if (id == YOURS) return new MyProduct;
    if (id == MINE) return new YourProduct;
    // N.B.: switched YOURS and MINE

    if (id == THEIRS) return new TheirProduct;
```



```
    return Creator::Create(id); // called if all others fail
}
```

注意这个操作所做的最后一件事是调用父类的 Create。这是因为 MyCreator::Create 仅在对 YOURS、MINE 和 THEIRS 的处理上和父类不同。它对其他类不感兴趣。因此 MyCreator 扩展了所创建产品的种类，并且将除少数产品以外所有产品的创建职责延迟给了父类。

3) 特定语言的变化和问题 不同的语言有助于产生其他一些有趣的变化和警告 (caveat)。

Smalltalk 程序通常使用一个方法返回被实例化的对象的类。Creator 工厂方法可以使用这个值去创建一个产品，并且 ConcreteCreator 可以存储甚至计算这个值。这个结果是对实例化的 ConcreteProduct 类型的一个更迟的绑定。

Smalltalk 版本的 Document 的例子可以在 Application 中定义一个 documentClass 方法。该方法为实例化文档返回合适的 Document 类，其在 MyApplication 中的实现返回 MyDocument 类。这样在类 Application 中我们有

```
clientMethod
    document := self documentClass new.
```

```
documentClass
    self subclassResponsibility
```

在类 MyApplication 中我们有

```
documentClass
    ^ MyDocument
```

它把将被实例化的类 MyDocument 返回给 Application。一个更灵活的类似于参数化工厂方法的办法是将被创建的类存储为 Application 的一个类变量。你用这种方法在改变产品时就无需用到 Application 的子类。

C++ 中的工厂方法都是虚函数并且常常是纯虚函数。一定要注意在 Creator 的构造器中不要调用工厂方法——在 ConcreteCreator 中该工厂方法还不可用。

只要你使用按需创建产品的访问者操作，很小心地访问产品，你就可以避免这一点。构造器只是将产品初始化为 0，而不是创建一个具体产品。访问者返回该产品。但首先它要检查确定该产品的存在，如果产品不存在，访问者就创建它。这种技术有时被称为 lazy initialization。下面的代码给出了一个典型的实现：

```
class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};

Product* Creator::GetProduct () {
    if (_product == 0) {
        _product = CreateProduct();
    }
    return _product;
}
```

4) 使用模板以避免创建子类 正如我们已经提及的，工厂方法另一个潜在的问题是它们可能仅为了创建适当的 Product 对象而迫使你创建 Creator 子类。在 C++ 中另一个解决方法是提供 Creator 的一个模板子类，它使用 Product 类作为模板参数：

```

class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}

```

使用这个模板，客户仅提供产品类——而不需要创建Creator的子类。

```

class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;

```

5) 命名约定 使用命名约定是一个好习惯，它可以清楚地说明你正在使用工厂方法。例如，Macintosh的应用框架MacApp [App89]总是声明那些定义为工厂方法的抽象操作为 Class* DoMakeClass()，此处Class是Product类。

10. 代码示例

函数CreateMaze（第3章）建造并返回一个迷宫。这个函数存在的一个问题是它对迷宫、房间、门和墙壁的类进行了硬编码。我们将引入工厂方法以使子类可以选择这些构件。首先我们将在MazeGame中定义工厂方法以创建迷宫、房间、墙壁和门对象：

```

class MazeGame {
public:
    Maze* CreateMaze();

    // factory methods:

    virtual Maze* MakeMaze() const
    { return new Maze; }
    virtual Room* MakeRoom(int n) const
    { return new Room(n); }
    virtual Wall* MakeWall() const
    { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door(r1, r2); }
};

```

每一个工厂方法返回一个给定类型的迷宫构件。MazeGame提供一些缺省的实现，它们返回最简单的迷宫、房间、墙壁和门。

现在我们可以用这些工厂方法重写 CreateMaze：

```

Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
}

```

```

Door* theDoor = MakeDoor(r1, r2);

aMaze->AddRoom(r1);
aMaze->AddRoom(r2);

r1->SetSide(North, MakeWall());
r1->SetSide(East, theDoor);
r1->SetSide(South, MakeWall());
r1->SetSide(West, MakeWall());

r2->SetSide(North, MakeWall());
r2->SetSide(East, MakeWall());
r2->SetSide(South, MakeWall());
r2->SetSide(West, theDoor);
return aMaze;
}

```

不同的游戏可以创建 MazeGame 的子类以特别指明一些迷宫的部件。MazeGame 子类可以重定义一些或所有的工厂方法以指定产品中的变化。例如，一个 BombedMazeGame 可以重定义产品 Room 和 Wall 以返回爆炸后的变体：

```

class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
    { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
    { return new RoomWithABomb(n); }
};

```

一个 EnchantedMazeGame 变体可以像这样定义：

```

class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};

```

11. 已知应用

工厂方法主要用于工具包和框架中。前面的文档例子是 MacApp 和 ET++ [WGM88] 中的一个典型应用。操纵器的例子来自 Unidraw。

Smalltalk-80 Model/View/Controller 框架中的类视图 (Class View) 有一个创建控制器的方法 defaultController，它有点类似于一个工厂方法 [Par90]。但是 View 的子类通过定义 defaultControllerClass 来指定它们默认的控制器的类。defaultControllerClass 返回 defaultController 所创建实例的类，因此它才是真正的工厂方法，即子类应该重定义它。

Smalltalk-80 中一个更为深奥的例子是由 Behavior (用来表示类的所有对象的超类) 定义的工厂方法 parserClass。这使得一个类可以对它的源代码使用一个定制的语法分析器。例如，

一个客户可以定义一个类 `SQLParser` 来分析嵌入了 SQL 语句的类的源代码。 `Behavior` 类实现了 `parserClass`，返回一个标准的 `Smalltalk Parser` 类。一个包含嵌入 SQL 语句的类重定义了该方法（以类方法的形式）并返回 `SQLParser` 类。

IONA Technologies 的 Orbix ORB 系统 [ION94] 在对象给一个远程对象引用发送请求时，使用 `Factory Method` 生成一个适当类型的代理（参见 `Proxy (4.7)`）。 `Factory Method` 使得易于替换缺省代理。比如说，可以用一个使用客户端高速缓存的代理来替换。

12. 相关模式

`Abstract Factory (3.1)` 经常用工厂方法来实现。 `Abstract Factory` 模式中动机一节例子也对 `Factory Method` 进行了说明。

工厂方法通常在 `Template Methods (5.10)` 中被调用。在上面的文档例子中， `NewDocument` 就是一个模板方法。

`Prototypes (3.4)` 不需要创建 `Creator` 的子类。但是，它们通常要求一个针对 `Product` 类的 `Initialize` 操作。 `Creator` 使用 `Initialize` 来初始化对象。而 `Factory Method` 不需要这样的操作。

3.4 PROTOTYPE（原型）——对象创建型模式

1. 意图

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

2. 动机

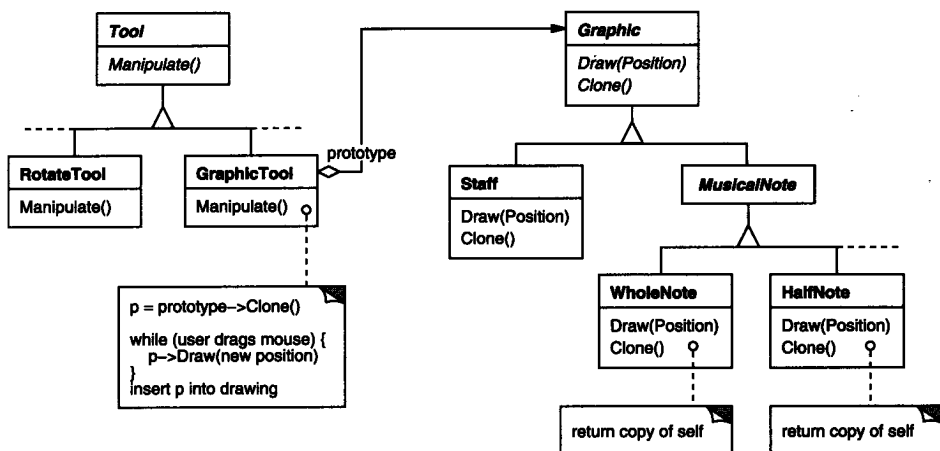
你可以通过定制一个通用的图形编辑器框架和增加一些表示音符、休止符和五线谱的新对象来构造一个乐谱编辑器。这个编辑器框架可能有一个工具选择板用于将这些音乐对象加到乐谱中。这个选择板可能还包括选择、移动和其他操纵音乐对象的工具。用户可以点击四分音符工具并使用它将四分音符加到乐谱中。或者他们可以使用移动工具在五线谱上上下下移动一个音符，从而改变它的音调。

我们假定该框架为音符和五线谱这样的图形构件提供了一个抽象的 `Graphics` 类。此外，为定义选择板中的那些工具，还提供一个抽象类 `Tool`。该框架还为一些创建图形对象实例并将它们加入到文档中的工具预定义了一个 `GraphicTool` 子类。

但 `GraphicTool` 给框架设计者带来一个问题。音符和五线谱的类特定于我们的应用，而 `GraphicTool` 类却属于框架。 `GraphicTool` 不知道如何创建我们的音乐类的实例，并将它们添加到乐谱中。我们可以为每一种音乐对象创建一个 `GraphicTool` 的子类，但这样会产生大量的子类，这些子类仅仅在它们所初始化的音乐对象的类别上有所不同。我们知道对象复合是比创建子类更灵活的一种选择。问题是，该框架怎么样用它来参数化 `GraphicTool` 的实例，而这些实例是由 `Graphic` 类所支持创建的。

解决办法是让 `GraphicTool` 通过拷贝或者“克隆”一个 `Graphic` 子类的实例来创建新的 `Graphic`，我们称这个实例为一个原型。 `GraphicTool` 将它应该克隆和添加到文档中的原型作为参数。如果所有 `Graphic` 子类都支持一个 `Clone` 操作，那么 `GraphicTool` 可以克隆所有种类的 `Graphic`，如下页上图所示。

因此在我们的音乐编辑器中，用于创建个音乐对象的每一种工具都是一个用不同原型进行初始化的 `GraphicTool` 实例。通过克隆一个音乐对象的原型并将这个克隆添加到乐谱中，每个 `GraphicTool` 实例都会产生一个音乐对象。



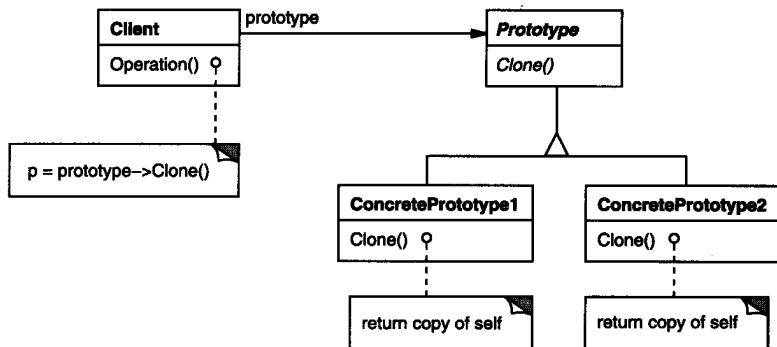
我们甚至可以进一步使用 Prototype 模式来减少类的数目。我们使用不同的类来表示全音符和半音符，但可能不需要这么做。它们可以是使用不同位图和时延初始化的相同的类的实例。一个创建全音符的工具就是这样的 GraphicTool，它的原型是一个被初始化成全音符的 MusicalNote。这可以极大的减少系统中类的数目，同时也更易于在音乐编辑器中增加新的音符。

3. 适用性

当一个系统应该独立于它的产品创建、构成和表示时，要使用 Prototype 模式；以及

- 当要实例化的类是在运行时时刻指定时，例如，通过动态装载；或者
- 为了避免创建一个与产品类层次平行的工厂类层次时；或者
- 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

4. 结构



5. 参与者

- Prototype (Graphic)
— 声明一个克隆自身的接口。
- ConcretePrototype (Staff、WholeNote、HalfNote)
— 实现一个克隆自身的操作。
- Client (GraphicTool)
— 让一个原型克隆自身从而创建一个新的对象。

6. 协作

- 客户请求一个原型克隆自身。

7. 效果

Prototype有许多和Abstract Factory (3.1) 和Builder (3.2) 一样的效果：它对客户隐藏了具体的产品类，因此减少了客户知道的名字的数目。此外，这些模式使客户无需改变即可使用与特定应用相关的类。

下面列出Prototype模式的另外一些优点。

1) 运行时时刻增加和删除产品 Prototype允许只通过客户注册原型实例就可以将一个具体的产品类并入系统。它比其他创建型模式更为灵活，因为客户可以在运行时时刻建立和删除原型。

2) 改变值以指定新对象 高度动态的系统允许你通过对象复合定义新的行为——例如，通过为一个对象变量指定值——并且不定义新的类。你通过实例化已有类并且将这些实例注册为客户对象的原型，就可以有效定义新类别的对象。客户可以将职责代理给原型，从而表现出新的行为。

这种设计使得用户无需编程即可定义新“类”。实际上，克隆一个原型类似于实例化一个类。Prototype模式可以极大的减少系统所需要的类的数目。在我们的音乐编辑器中，一个GraphicTool类可以创建无数种音乐对象。

3) 改变结构以指定新对象 许多应用由部件和子部件来创建对象。例如电路设计编辑器就是由子电路来构造电路的[⊖]。为方便起见，这样的应用通常允许你实例化复杂的、用户定义的结构，比方说，一次又一次的重复使用一个特定的子电路。

Prototype模式也支持这一点。我们仅需将这个子电路作为一个原型增加到可用的电路元素选择板中。只要复合电路对象将Clone实现为一个深拷贝 (deep copy)，具有不同结构的电路就可以是原型了。

4) 减少子类的构造 Factory Method (3.3) 经常产生一个与产品类层次平行的Creator类层次。Prototype模式使得你克隆一个原型而不是请求一个工厂方法去产生一个新的对象。因此你根本不需要Creator类层次。这一优点主要适用于像C++这样不将类作为一级类对象的语言。像Smalltalk和Objective C这样的语言从中获益较少，因为你总是可以用一个类对象作为生成者。在这些语言中，类对象已经起到原型一样的作用了。

5) 用类动态配置应用 一些运行时环境允许你动态将类装载到应用中。在像C++这样的语言中，Prototype模式是利用这种功能的关键。

一个希望创建动态载入类的实例的应用不能静态引用类的构造器。而应该由运行环境在载入时自动创建每个类的实例，并用原型管理器来注册这个实例（参见实现一节）。这样应用就可以向原型管理器请求新装载的类的实例，这些类原本并没有和程序相连接。ET++应用框架[WGM88]有一个运行系统就是使用这一方案的。

Prototype的主要缺陷是每一个Prototype的子类都必须实现Clone操作，这可能很困难。例如，当所考虑的类已经存在时就难以新增Clone操作。当内部包括一些不支持拷贝或有循环引用的对象时，实现克隆可能也会很困难的。

8. 实现

⊖ 这样的应用反映了Composite (4.3) 和Decorator (4.4) 模式。

因为在像C++这样的静态语言中，类不是对象，并且运行时刻只能得到很少或者得不到任何类型信息，所以Prototype特别有用。而在Smalltalk或Objective C这样的语言中Prototype就不是那么重要了，因为这些语言提供了一个等价于原型的东西（即类对象）来创建每个类的实例。Prototype模式在像Self[US87]这样基于原型的语言中是固有的，所有对象的创建都是通过克隆一个原型实现的。

当实现原型时，要考虑下面一些问题：

1) 使用一个原型管理器 当一个系统中原型数目不固定时（也就是说，它们可以动态创建和销毁），要保持一个可用原型的注册表。客户不会自己来管理原型，但会在注册表中存储和检索原型。客户在克隆一个原型前会向注册表请求该原型。我们称这个注册表为原型管理器（prototype manager）。

原型管理器是一个关联存储器（associative store），它返回一个与给定关键字相匹配的原型。它有一些操作可以用来通过关键字注册原型和解除注册。客户可以在运行时更改甚或浏览这个注册表。这使得客户无需编写代码就可以扩展并得到系统清单。

2) 实现克隆操作 Prototype模式最困难的部分在于正确实现Clone操作。当对象结构包含循环引用时，这尤为棘手。

大多数语言都对克隆对象提供了一些支持。例如，Smalltalk提供了一个copy的实现，它被所有Object的子类所继承。C++提供了一个拷贝构造器。但这些设施并不能解决“浅拷贝和深拷贝”问题[GR83]。也就是说，克隆一个对象是依次克隆它的实例变量呢，或者还是由克隆对象和原对象共享这些变量？

浅拷贝简单并且通常也足够了，它是Smalltalk所缺省提供的。C++中的缺省拷贝构造器实现按成员拷贝，这意味着在拷贝的和原来的对象之间是共享指针的。但克隆一个结构复杂的原型通常需要深拷贝，因为复制对象和原对象必须相互独立。因此你必须保证克隆对象的构件也是对原型的构件的克隆。克隆迫使你决定如果所有东西都被共享了该怎么办。

如果系统中的对象提供了Save和Load操作，那么你只需通过保存对象和立刻载入对象，就可以为Clone操作提供一个缺省实现。Save操作将该对象保存在内存缓冲区中，而Load则通过从该缓冲区中重构这个对象来创建一个复本。

3) 初始化克隆对象 当一些客户对克隆对象已经相当满意时，另一些客户将会希望使用他们所选择的一些值来初始化该对象的一些或是所有的内部状态。一般来说不可能在Clone操作中传递这些值，因为这些值的数目由于原型的类的不同而会有所不同。一些原型可能需要多个初始化参数，另一些可能什么也不要。在Clone操作中传递参数会破坏克隆接口的统一性。

可能会这样，原型的类已经为（重）设定一些关键的状态值定义好了操作。如果这样的话，客户在克隆后马上就可以使用这些操作。否则，你就可能不得不引入一个Initialize操作（参见代码示例一节），该操作使用初始化参数并据此设定克隆对象的内部状态。注意深拷贝Clone操作——一些复制在你重新初始化它们之前可能必须要被删除掉（删除可以显式地做也可以在Initialize内部做）。

9. 代码示例

我们将定义MazeFactory（3.1）的子类MazePrototypeFactory。该子类将使用它要创建的原型的原型来初始化，这样我们就不需要仅仅为了改变它所创建的墙壁或房间的类而生成子

类了。

MazePrototypeFactory用一个以原型为参数的构造器来扩充 MazeFactory接口：

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

新的构造器只初始化它的原型：

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
```

用于创建墙壁、房间和门的成员函数是相似的：每个都要克隆一个原型，然后初始化。

下面是MakeWall和MakeDoor的定义：

```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

我们只需使用基本迷宫构件的原型进行初始化，就可以由 MazePrototypeFactory来创建一个原型的或缺省的迷宫：

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

为了改变迷宫的类型，我们用一个不同的原型集合来初始化 MazePrototypeFactory。下面的调用用一个BombedDoor和一个RoomWithABomb创建了一个迷宫：

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

一个可以被用作原型的对象，例如 Wall的实例，必须支持Clone操作。它还必须有一个拷

贝构造器用于克隆。它可能还需要一个独立的操作来重新初始化内部状态。我们将给 Door 增加 Initialize 操作以允许客户初始化克隆对象的房间。

将下面 Door 的定义与第 3 章的进行比较：

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;
    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

BombedWall 子类必须重定义 Clone 并实现相应的拷贝构造器。

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

虽然 BombedWall::Clone 返回一个 Wall*，但它的实现返回了一个指向子类的新实例的指针，即 BombedWall*。我们在基类中这样定义 Clone 是为了保证克隆原型的客户不需要知道具体的子类。客户决不需要将 Clone 的返回值向下类型转换为所需类型。

在 Smalltalk 中，你可以重用从 Object 中继承的标准 copy 方法来克隆任一 MapSite。你可以

用MazeFactory来生成你需要的原型；例如，你可以提供名字 #room来创建一个房间。MazeFactory有一个将名字映射为原型的字典。它的 make:方法如下：

```
make: partName
    ^ (partCatalog at: partName) copy
```

假定有用原型初始化 MazeFactory的适当方法，你可以用下面代码创建一个简单迷宫：

```
CreateMaze
on: (MazeFactory new
    with: Door new named: #door;
    with: Wall new named: #wall;
    with: Room new named: #room;
    yourself)

其中CreateMaze的类方法 on:的定义将是

on: aFactory
| room1 room2 |
room1 := (aFactory make: #room) location: 1@1.
room2 := (aFactory make: #room) location: 2@1.
door := (aFactory make: #door) from: room1 to: room2.

room1
    atSide: #north put: (aFactory make: #wall);
    atSide: #east put: door;
    atSide: #south put: (aFactory make: #wall);
    atSide: #west put: (aFactory make: #wall).
room2
    atSide: #north put: (aFactory make: #wall);
    atSide: #east put: (aFactory make: #wall);
    atSide: #south put: (aFactory make: #wall);
    atSide: #west put: door.
^ Maze new
    addRoom: room1;
    addRoom: room2;
    yourself
```

10. 已知应用

可能Prototype模式的第一个例子出现于 Ivan Sutherland的Sketchpad系统中[Sut63]。该模式在面向对象语言中第一个广为人知的应用是在 ThingLab中，其中用户能够生成复合对象，然后把它安装到一个可重用的对象库中从而促使它成为一个原型 [Bor81]。Goldberg和Robson都提出原型是一种模式[GR83]，但Coplien[Cop92]给出了一个更为完整的描述。他为 C++描述了与Prototype模式相关的术语并给出了很多例子和变种。

etgdb是一个基于ET++的调试器前端，它为不同的行导向（line-oriented）调试器提供了一个点触式（point-and-click）接口。每个调试器有相应的 DebuggerAdaptor子类。例如，GdbAdaptor使etgdb适应GNU的gdb命令语法，而SunDbxAdaptor则使etgdb适应Sun的dbx调试器。etgdb没有一组硬编码于其中的DebuggerAdaptor类。它从环境变量中读取要用到的适配器的名字，在一个全局表中根据特定名字查询原型，然后克隆这个原型。新的调试器通过与该调试器相对应的DebuggerAdaptor链接，可以被添加到etgdb中。

Mode Composer中的“交互技术库”（interaction technique library）存储了支持多种交互技术的对象的原型[Sha90]。将Mode Composer创建的任一交互技术放入这个库中，它就可以被作为一个原型使用。Prototype模式使得Mode Composer可支持数目无限的交互技术。

前面讨论过的音乐编辑器的例子是基于 Unidraw绘图框架的[VL90]。

11. 相关模式

正如我们在这一章结尾所讨论的那样，Prototype和Abstract Factory（3.1）模式在某种方面是相互竞争的。但是它们也可以一起使用。Abstract Factory可以存储一个被克隆的原型的集合，并且返回产品对象。

大量使用Composite（4.3）和Decorator（4.4）模式的设计通常也可从Prototype模式处获益。

3.5 SINGLETON（单件）——对象创建型模式

1. 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。

2. 动机

对一些类来说，只有一个实例是很重要的。虽然系统中可以有許多打印机，但却只应该有一个打印假脱机（printer spooler），只应该有一个文件系统和一个窗口管理器。一个数字滤波器只能有一个A/D转换器。一个会计系统只能专用于一个公司。

我们怎样才能保证一个类只有一个实例并且这个实例易于被访问呢？一个全局变量使得一个对象可以被访问，但它不能防止你实例化多个对象。

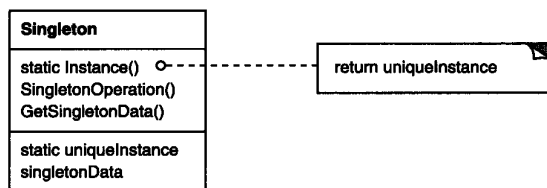
一个更好的办法是，让类自身负责保存它的唯一实例。这个类可以保证没有其他实例可以被创建（通过截取创建新对象的请求），并且它可以提供一个访问该实例的方法。这就是Singleton模式。

3. 适用性

在下面的情况下可以使用Singleton模式

- 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
- 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

4. 结构



5. 参与者

• Singleton

— 定义一个Instance操作，允许客户访问它的唯一实例。Instance是一个类操作（即Smalltalk中的一个类方法和C++中的一个静态成员函数）。

— 可能负责创建它自己的唯一实例。

6. 协作

- 客户只能通过Singleton的Instance操作访问一个Singleton的实例。

7. 效果

Singleton模式有许多优点：

- 1) 对唯一实例的受控访问 因为Singleton类封装它的唯一实例，所以它可以严格的控制

客户怎样以及何时访问它。

2) 缩小名空间 Singleton模式是对全局变量的一种改进。它避免了那些存储唯一实例的全局变量污染名空间。

3) 允许对操作和表示的精细化 Singleton类可以有子类,而且用这个扩展类的实例来配置一个应用是很容易的。你可以用你所需要的类的实例在运行时刻配置应用。

4) 允许可变数目的实例 这个模式使得你易于改变你的想法,并允许 Singleton类的多个实例。此外,你可以用相同的方法来控制应用所使用的实例的数目。只有允许访问 Singleton实例的操作需要改变。

5) 比类操作更灵活 另一种封装单件功能的方式是使用类操作(即C++中的静态成员函数或者是Smalltalk中的类方法)。但这两种语言技术都难以改变设计以允许一个类有多个实例。此外,C++中的静态成员函数不是虚函数,因此子类不能多态的重定义它们。

8. 实现

下面是使用Singleton模式时所要考虑的实现问题:

1) 保证一个唯一的实例 Singleton模式使得这个唯一实例是类的一般实例,但该类被写成只有一个实例能被创建。做到这一点的一个常用方法是将创建这个实例的操作隐藏在一个类操作(即一个静态成员函数或者是一个类方法)后面,由它保证只有一个实例被创建。这个操作可以访问保存唯一实例的变量,而且它可以保证这个变量在返回值之前用这个唯一实例初始化。这种方法保证了单件在它的首次使用前被创建和使用。

在C++中你可以用Singleton类的静态成员函数 Instance来定义这个类操作。Singleton还定义了一个静态成员变量 _instance,它包含了一个指向它的唯一实例的指针。

Singleton 类定义如下

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

相应的实现是

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

客户仅通过 Instance成员函数访问这个单件。变量 _instance初始化为0,而静态成员函数 Instance返回该变量值,如果其值为0则用唯一实例初始化它。Instance使用惰性(lazy)初始化;它的返回值直到被第一次访问时才创建和保存。

注意构造器是保护型的。试图直接实例化 Singleton的客户将得到一个编译时的错误信息。这就保证了仅有一个实例可以被创建。

此外,因为_instance是一个指向Singleton对象的指针, Instance成员函数可以将一个指向Singleton的子类的指针赋给这个变量。我们将在代码示例一节给出一个这样的例子。

关于C++的实现还有一点需要注意。将单件定义为一个全局或静态的对象，然后依赖于自动的初始化，这是不够的。有如下三个原因：

a) 我们不能保证静态对象只有一个实例会被声明。

b) 我们可能没有足够的信息在静态初始化时实例化每一个单件。单件可能需要在程序执行中稍后被计算出来的值。

c) C++没有定义转换单元（translation unit）上全局对象的构造器的调用顺序 [ES90]。这意味着单件之间不存在依赖关系；如果有，那么错误将是不可避免的。

使用全局/静态对象的实现方法还有另一个（尽管很小）缺点，它使得所有单件无论用到与否都要被创建。使用静态成员函数避免了所有这些问题。

Smalltalk中，返回唯一实例的函数被实现为 Singleton 类的一个类方法。为保证只有一个实例被创建，重定义了 new 操作。得到的 Singleton 类可能有下列两个类方法，其中 SoleInstance 是一个其他地方并不使用的类变量：

```
new
    self error: 'cannot create new object'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

2) 创建 Singleton 类的子类 主要问题与其说是定义子类不如说是建立它的唯一实例，这样客户就可以使用它。事实上，指向单件实例的变量必须用子类的实例进行初始化。最简单的技术是在 Singleton 的 Instance 操作中决定你想使用的是哪一个单件。代码示例一节中的一个例子说明了如何用环境变量实现这一技术。

另一个选择 Singleton 的子类的方法是将 Instance 的实现从父类（即 MazeFactory）中分离出来并将它放入子类。这就允许 C++ 程序员在链接时刻决定单件的类（即通过链入一个包含不同实现的对象文件），但对单件的客户则隐蔽这一点。

链接的方法在链接时刻确定了单件类的选择，这使得难以在运行时刻选择单件类。使用条件语句来决定子类更加灵活一些，但这硬性限定（hard-wire）了可能的 Singleton 类的集合。这两种方法不是在所有的情况都足够灵活的。

一个更灵活的方法是使用一个单件注册表（registry of singleton）。可能的 Singleton 类的集合不是由 Instance 定义的，Singleton 类可以根据名字在一个众所周知的注册表中注册它们的单件实例。

这个注册表在字符串名字和单件之间建立映射。当 Instance 需要一个单件时，它参考注册表，根据名字请求单件。

注册表查询相应的单件（如果存在的话）并返回它。这个方法使得 Instance 不再需要知道所有可能的 Singleton 类或实例。它所需要的只是所有 Singleton 类的一个公共的接口，该接口包括了对注册表的操作：

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
```

```
static List<NameSingletonPair>* _registry;
};
```

Register以给定的名字注册 Singleton实例。为保证注册表简单，我们将让它存储一系列 NameSingletonPair对象。每个 NameSingletonPair将一个名字映射到一个单件。Lookup操作根据给定单件的名字进行查找。我们假定一个环境变量指定了所需要的单件的名字。

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
```

Singleton类在何处注册它们自己？一种可能是在它们的构造器中。例如， MySingleton子类可以像下面这样做：

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

当然，除非实例化类否则这个构造器不会被调用，这正反映了 Singleton模式试图解决的问题！在C++中我们可以定义 MySingleton的一个静态实例来避免这个问题。例如，我们可以在包含 MySingleton实现的文件中定义：

```
static MySingleton theSingleton;
```

Singleton类不再负责创建单件。它的主要职责是使得供选择的单件对象在系统中可以被访问。静态对象方法还是有一个潜在的缺点——也就是所有可能的 Singleton子类的实例都必须被创建，否则它们不会被注册。

9. 代码示例

假定我们定义一个 MazeFactory类用于建造在第3章所描述的迷宫。MazeFactory定义了一个建造迷宫的不同部件的接口。子类可以重定义这些操作以返回特定产品类的实例，如用 BombedWall对象代替普通的 Wall对象。

此处相关的问题是 Maze应用仅需迷宫工厂的一个实例，且这个实例对建造迷宫任何部件的代码都是可用的。这样就引入了 Singleton模式。将 MazeFactory作为单件，我们无需借助全局变量就可使迷宫对象具有全局可访问性。

为简单起见，我们假定不会生成 MazeFactory的子类。（我们随后将考虑另一个选择。）我们通过增加静态的 Instance操作和静态的用以保存唯一实例的成员 _instance，从而在C++中生成一个 Singleton类。我们还必须保护构造器以防止意外的实例化，因为意外的实例化可能会导致多个实例。

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
```

```
private:
    static MazeFactory* _instance;
};

相应的实现是：

MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

现在让我们考虑当存在 MazeFactory 的多个子类，而且应用必须决定使用哪一个子类时的情况。我们将通过环境变量选择迷宫的种类并根据该环境变量的值增加代码用于实例化适当的 MazeFactory 子类。Instance 操作是增加这些代码的好地方，因为它已经实例化了

MazeFactory：

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... other possible subclasses

        } else { // default
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

注意，无论何时定义一个新的 MazeFactory 的子类，Instance 都必须被修改。在这个应用中这可能没什么关系，但对于定义在一个框架中的抽象工厂来说，这可能是一个问题。

一个可能的解决办法将是使用在实现一节中所描述过的注册表的方法。此处动态链接可能也很有用——它使得应用不需要装载那些用不着的子类。

10. 已知应用

在 Smalltalk-80[Par90] 中 Singleton 模式的例子是改变代码的集合，即 ChangeSet current。一个更巧妙的例子是类和它们的元类（metaclass）之间的关系。一个元类是一个类的类，而且每一个元类有一个实例。元类没有名字（除非间接地通过它们的唯一实例），但它们记录了它们的唯一实例并且通常不会再创建其他实例。

InterViews 用户界面工具箱 [LCI-92] 使用 Singleton 模式在其他类中访问 Session 和 WidgetKit 类的唯一实例。Session 定义了应用的主事件调度循环、存储用户的风格偏好数据库，并管理与一个或多个物理显示的连接。WidgetKit 是一个 Abstract Factory (3.1)，用于定义用户的窗口组件的视感风格。WidgetKit::instance() 操作决定了特定的 WidgetKit 子类，该子类根据 Session 定义的环境变量进行实例化。Session 的一个类似操作决定了支持单色还是彩色显示并

据此配置单件Session的实例。

11. 相关模式

很多模式可以使用 Singleton模式实现。参见 Abstract Factory (3.1)、Builder (3.2), 和 Prototype (3.4)。

3.6 创建型模式的讨论

用一个系统创建的那些对象的类对系统进行参数化有两种常用方法。一种是生成创建对象的类的子类；这对应于使用 Factory Method (3.3) 模式。这种方法的主要缺点是，仅为了改变产品类，就可能需要创建一个新的子类。这样的改变可能是级联的 (cascade)。例如，如果产品的创建者本身是由一个工厂方法创建的，那么你也必须重定义它的创建者。

另一种对系统进行参数化的方法更多的依赖于对象复合：定义一个对象负责明确产品对象的类，并将它作为该系统的参数。这是 Abstract Factory (3.1)、Builder (3.2) 和 Prototype (3.4) 模式的关键特征。所有这三个模式都涉及到创建一个新的负责创建产品对象的“工厂对象”。Abstract Factory由这个工厂对象产生多个类的对象。Builder由这个工厂对象使用一个相对复杂的协议，逐步创建一个复杂产品。Prototype由该工厂对象通过拷贝原型对象来创建产品对象。在这种情况下，因为原型负责返回产品对象，所以工厂对象和原型是同一个对象。

考虑在 Prototype 模式中描述的绘图编辑器框架。可以有多种方法通过产品类来参数化 GraphicTool：

- 使用 Factory Method 模式，将为选择板中的每个 Graphic 的子类创建一个 GraphicTool 的子类。GraphicTool 将有一个 NewGraphic 操作，每个 GraphicTool 的子类都会重定义它。
- 使用 Abstract Factory 模式，将有一个 GraphicsFactory 类层次对应于每个 Graphic 的子类。在这种情况下每个工厂仅创建一个产品：CircleFactory 将创建 Circle，LineFactory 将创建 Line，等等。GraphicTool 将以创建合适种类 Graphic 的工厂作为参数。
- 使用 Prototype 模式，每个 Graphic 的子类将实现 Clone 操作，并且 GraphicTool 将以它所创建的 Graphic 的原型作为参数。

究竟哪一种模式最好取决于诸多因素。在我们的绘图编辑器框架中，第一眼看来，Factory Method 模式使用是最简单的。它易于定义一个新的 GraphicTool 的子类，并且仅当选择板被定义了的时候，GraphicTool 的实例才被创建。它的主要缺点在于 GraphicTool 子类数目的激增，并且它们都没有做很多事情。

Abstract Factory 并没有很大的改进，因为它需要一个同样庞大的 GraphicsFactory 类层次。只有当早已存在一个 GraphicsFactory 类层次时，Abstract Factory 才比 Factory Method 更好一点——或是因为编译器自动提供（像在 Smalltalk 或是 Objective C 中）或是因为系统的其他部分需要这个 GraphicsFactory 类层次。

总的来说，Prototype 模式对绘图编辑器框架可能是最好的，因为它仅需要为每个 Graphics 类实现一个 Clone 操作。这就减少了类的数目，并且 Clone 可以用于其他目的而不仅仅是纯粹的实例化（例如，一个 Duplicate 菜单操作）。

Factory Method 使一个设计可以定制且只略微有一些复杂。其他设计模式需要新的类，而 Factory Method 只需要一个新的操作。人们通常将 Factory Method 作为一种标准的创建对象的

方法。但是当被实例化的类根本不发生变化或当实例化出现在子类可以很容易重定义的操作中（比如在初始化操作中）时，这就并不必要了。

使用Abstract Factory、Prototype或Builder的设计甚至比使用Factory Method的那些设计更灵活，但它们也更加复杂。通常，设计以使用Factory Method开始，并且当设计者发现需要更大的灵活性时，设计便会向其他创建型模式演化。当你在设计标准之间进行权衡的时候，了解多个模式可以给你提供更多的选择余地。