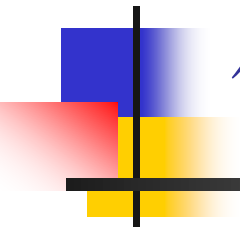
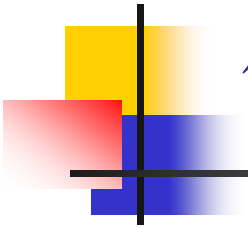


创建型模式



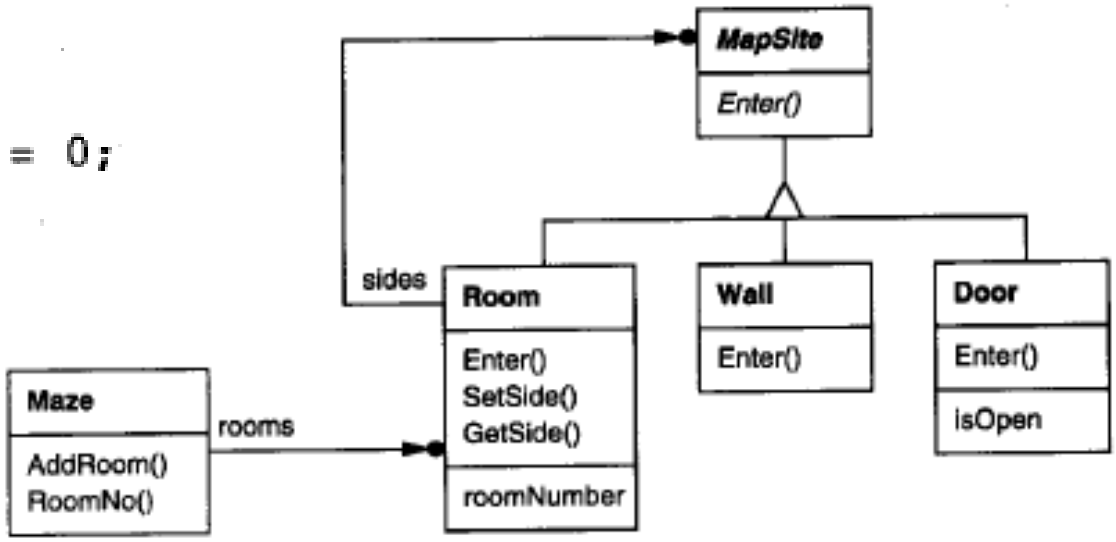
创建型模式

- 抽象了实例化过程
- 创建型模式的主旋律
  - 将关于该系统使用那些具体的类的信息封装起来
  - 隐藏了这些类的实例是如何被创建和放在一起的

Context

- 迷宫的创建
  - Room, Door, Wall
  - enum Direction {North, South, East, West }

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



Room

```
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;
};
```

其他MapSite对象的引用

房间号

Wall / Door

```
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;
};
```

Maze

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

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

查找特定的Room

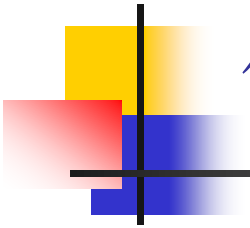
MazeGame：创建迷宫

```
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;
}
```

1. 存在的问题：不灵活！

2. 可能的变化：
   增加新的部件, e.g. DoorNeedingSpell

3. 改变的最大障碍：
   对被实例化的类进行硬编码

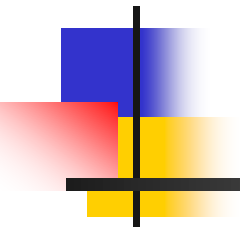


创建型模式提供的实例化方法

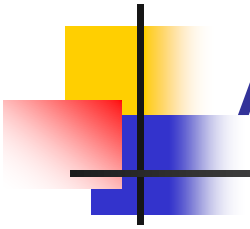
- Factory Method
  - CreateMaze调用虚函数来创建房间/墙壁/门
  - 则可以在MazeGame的子类中重定义这些虚函数，从而改变被实例化的类（房间/墙壁/门）
- Abstract Factory
  - 给CreateMaze传递参数来创建房间/墙壁/门
  - 则可以通过传递不同的参数来改变房间/墙壁/门的类

创建型模式提供的实例化方法

- Builder
  - 传递一个对象给CreateMaze
  - 该对象使用某些操作（增加房间/墙壁/门），来创建一个新的迷宫
  - 则可使用继承来改变迷宫的一部分或该迷宫被建造的方式
- Prototype
  - CreateMaze有多种原型对象（房间/墙壁/门对象）参数化，其拷贝并将这些对象增加到迷宫中。
  - 则可用不同的对象替换这些原型对象以改变迷宫的构成
- Singleton
  - 保证一个游戏中仅有一个迷宫
  - 游戏中所有对象均可访问该迷宫
  - 不使用全局变量或函数



3.1 Abstract Factory
抽象工程



Abstract Factory模式

1. 意图

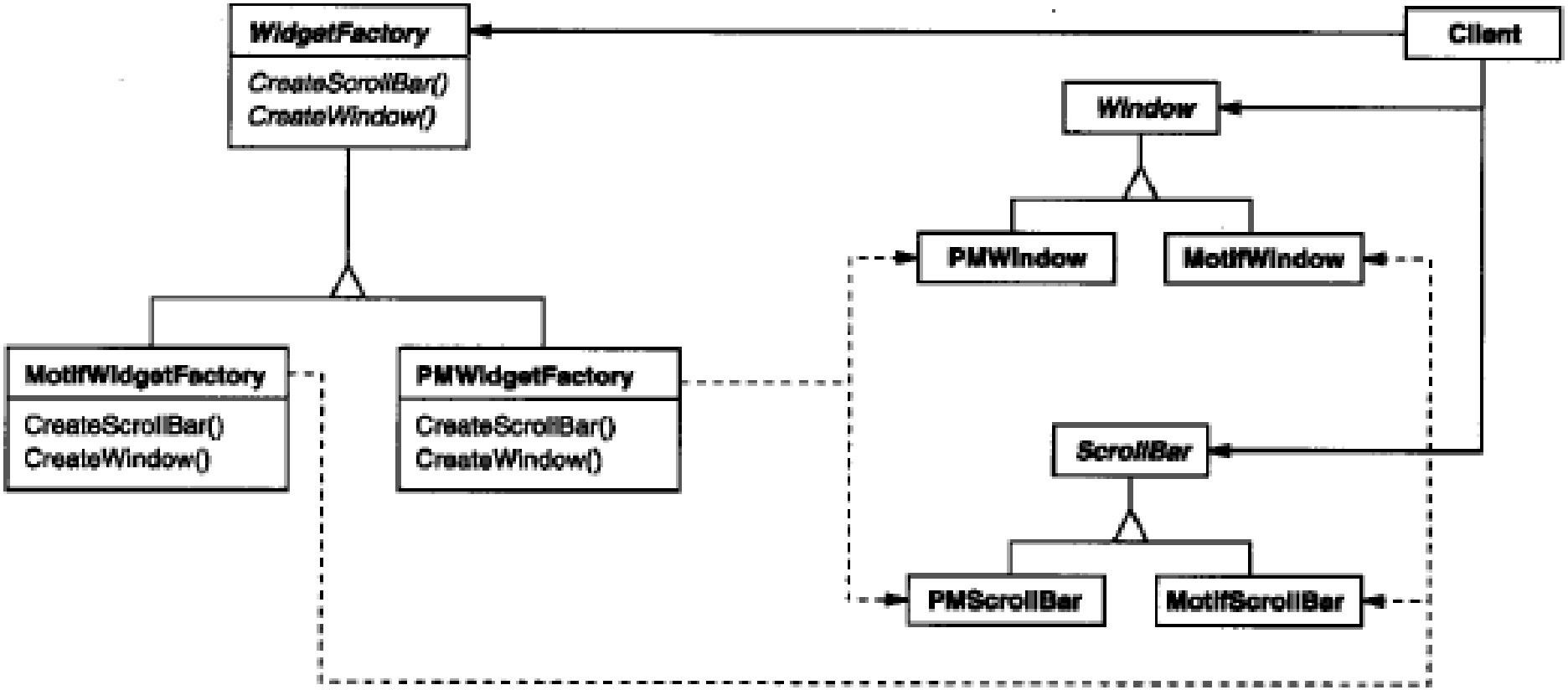
   提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

2. 别名

   Kit

Abstract Factory模式

3. 动机



Abstract Factory模式

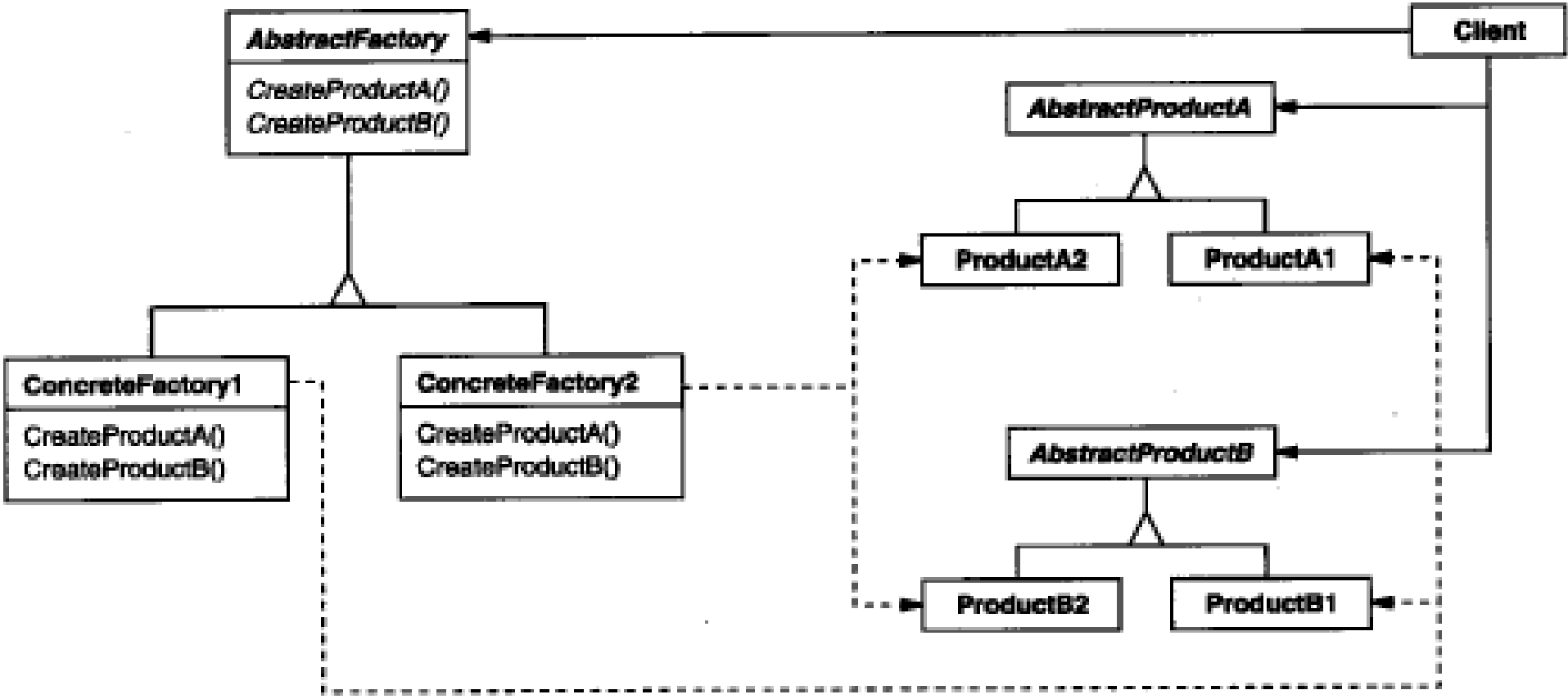
4. 适用性

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。

Abstract Factory模式

5. 结构



Abstract Factory模式

6. 参与者

- AbstractFactory (WidgetFactory)
  - 声明一个创建抽象产品对象的操作接口
- ConcreteFactory 具体工厂
  - 实现创建具体产品对象的操作
- AbstractProduct (Windows, ScrollBar)
  - 为一类产品对象什么一个接口
- ConcreteProduct 具体产品
  - 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“抽象工厂” / “抽象产品” 类声明的接口

Abstract Factory模式

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
  - 给Client一个抽象工厂指针，指向一个具体工厂对象
  - Client通过工厂获取抽象产品的指针，指向一个具体产品对象

- AbstractFactory将产品对象的创建延迟到它的子类（ConcreteFactory）

Abstract Factory模式

8. 效果

1）分离了具体的类

- 由工厂封装创建产品对象的职责和过程，从而将客户与类的实现分离。
- 客户通过抽象接口操作实例。
- 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。

Abstract Factory模式

8. 效果

2）使得易于交换产品系列

- 只需改变具体工厂对象，即可使用不同的产品系列。

3）有利于产品的一致性

- 一个应用一次只能使用同一个系列中的对象。

Abstract Factory模式

8. 效果

4）难于支持新种类的产品

- 原因：

  在AbstractFactory接口中确定了可以被创建的产品集合

Abstract Factory模式

9. 实现

1) 将工厂作为单件

- 一般情况下，一个应用中每个产品系列只需一个ConcreteFactory的实例
- 使用Singleton模式
  - 件具体工厂类的对象唯一化

Abstract Factory模式

9. 实现

2) 创建产品

具体工厂类的实现方法

- Factory模式
  - 在抽象类中为每个产品定义一个工厂方法
  - 在具体类中为每个产品重定义该工厂方法
  - 每个产品系列都要有一个新的具体工厂子类

- Prototype模式
  - 具体工厂使用产品系列中每个产品的原型实例来初始化
  - 通过复制它的原型来创建新的产品
  - 不是每个产品系列都要有一个新的具体工厂子类

Abstract Factory模式

9. 实现

3) 定义可扩展的工厂

- 问题：
  - 对每个新产品都要在抽象工厂类中添加新的接口
- 解决方法：用参数来指定被创建对象的种类

  AbstractProduct* Make(int id);
- 问题
  - 所有产品均是AbstractProduct的子类
  - 所有产品以AbstractProduct接口返回
  - 可能需要向下类型转换(downcast)

    e.g. dynamic_cast

Abstract Factory模式

10. 代码示例

抽象工厂

```
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); }
};
```

Client

```
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;
}
```

Abstract Factory模式

10. 代码示例

```
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;
};
```

具体工厂

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

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

Client

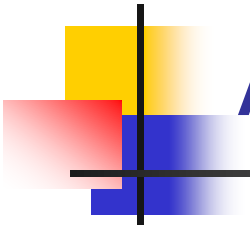
```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

Abstract Factory模式

10. 代码示例

- 上例是最通常的Abstract Factory模式的实现方式

- MazeFactory既作为抽象工厂，也作为具体工厂

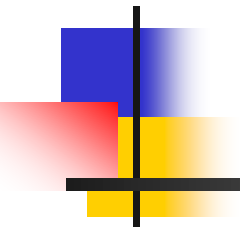


Abstract Factory模式

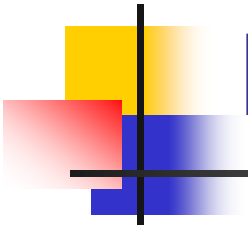
12. 相关应用

- AbstractFactory类通常使用“Factory Method”实现，也可以用Prototype实现

- 一个具体的工厂通常是一个单件（Singleton）



3.2 Builder
生成器

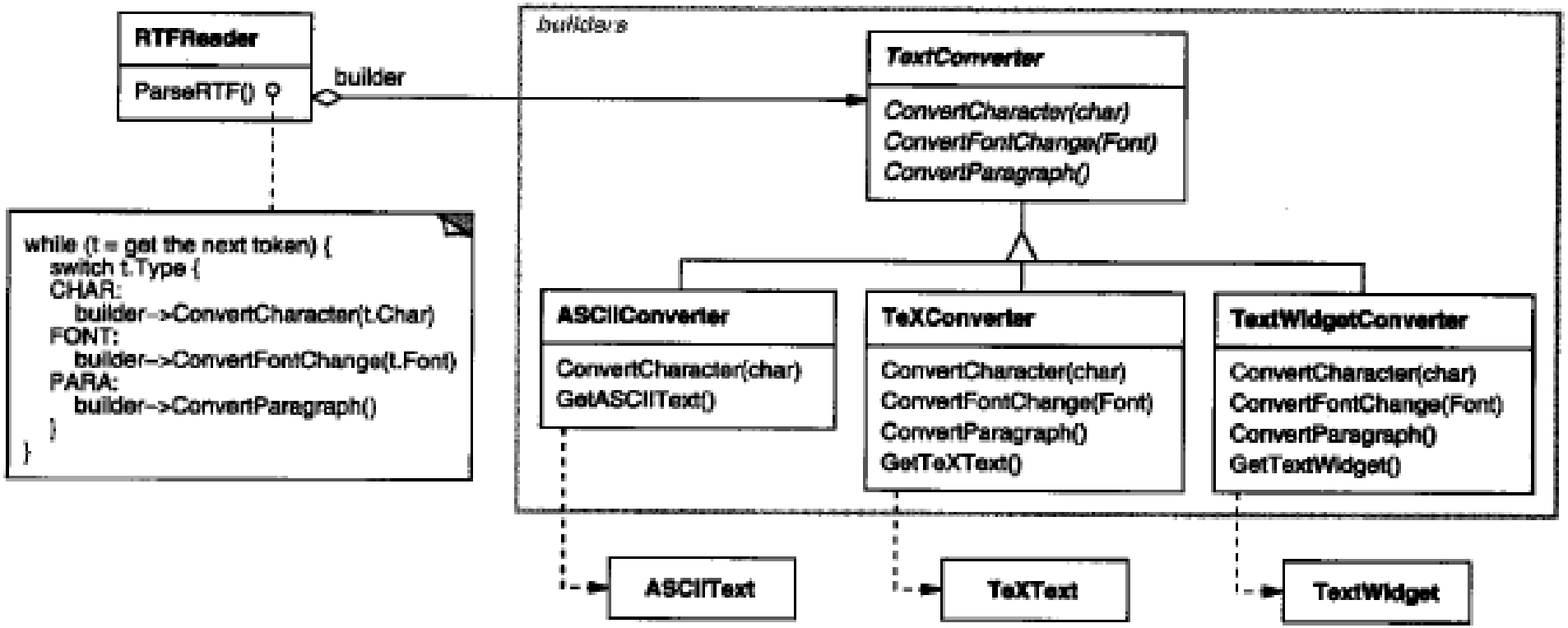


Builder模式

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



将RTF文档转换成多种正文格式

2. 动机

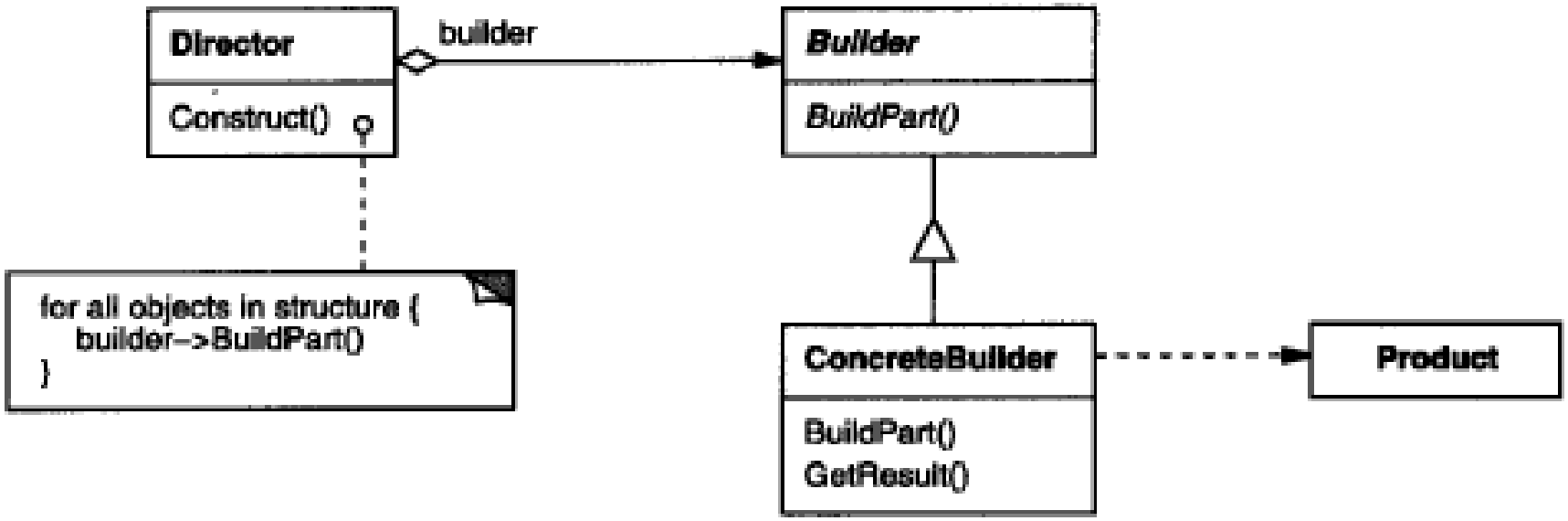
- 转换器　（Builder，生成器）
  - 封装创建和装配一个复杂对象的机制
- 阅读器　（Director，导向器）
  - 负责对一个RTF进行语法分析

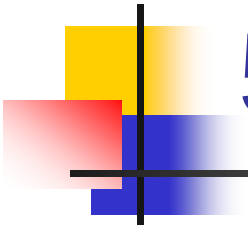
- Builder模式
  - 将分析文本格式的算法（Director）与描述怎样创建和表示一个转换后格式的算法（Builder）分离开来。

3. 适用性

- 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对象有不同的表示时。

4.结构


Director
builder
Construct()
Builder
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
BuildPart()
GetResult()
Product



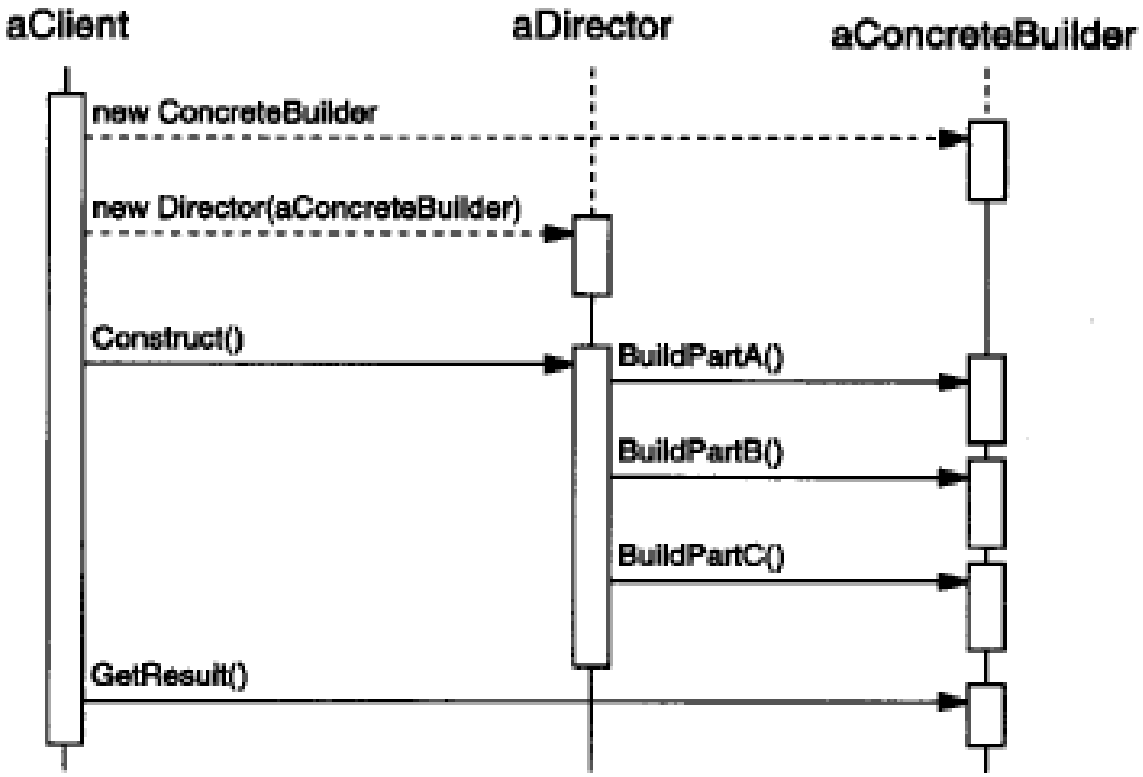
5. 参与者

- Builder (TextConverter)
  - 为创建一个Product对象的各个部件指定抽象接口
- ConcretBuilder
  - 实现Builder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 GetResult
    - e.g. GetASCIIText, GetTextWidget

5. 参与者

- 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

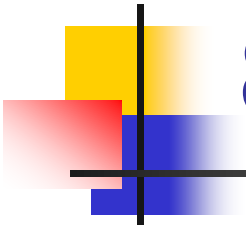
6. 协作



1. Client创建Director对象，并用Builder对象进行配置。
2. 当需要生成产品部件时，Director通知Builder
3. Builder处理Director请求，并将部件添加到产品中
4. Client从Builder中获取产品

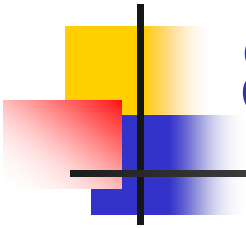
7. 效果

- 可以改变一个产品的内部表示
  - Builder封装了产品的内部表示，与Director分离
  - 通过更换Builder来改变产品的内部表示
- 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。
- 可对构造过程进行更精细的控制
  - 在Director控制下一步一步构造产品



8. 实现

- 装配和构造接口
  - Builder类接口必须足够普通，以便满足ConcreteBuilder的需要
  - 简单情况
    - 构造请求的结果是将新的部件添加到产品中
  - 复杂情况
    - 需要访问前面已经构造的产品部件



8. 实现

- 为什么产品没有抽象类
  - 通常，由ConcreteBuilder生成的产品差异很大
  - 与抽象工厂模式不同
- 在Builder中缺省的方法为空
  - 在Builder基类中定义空函数，而不是纯虚函数
  - 便于在ConcreteBuilder中只需要重定义需要的操作

9. 代码实例

```
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();
};
```

创建部件

获取产品

Builder基类

9. 代码实例

Director

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

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

    return builder.GetMaze();
}
```

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

    return builder.GetMaze();
}
```

Builder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

9. 代码实例

```
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;
};

StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

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

ConcreteBuilder

9. 代码实例

```
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);
    }
}
```

ConcreteBuilder

```
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);
}
```

9. 代码实例

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

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

Client

9. 代码实例

```
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;
};
```

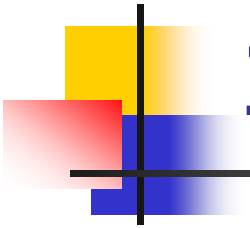
```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

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

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

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

另一个ConcreteBuilder
不创建迷宫；仅对各种部件进行计数

9. 代码实例

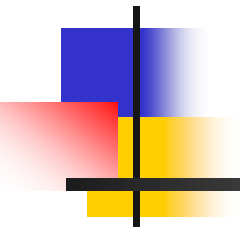
```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

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

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

Client



11. 相关模式

- 抽象工厂模式与Builder模式
  - Builder模式着重于一步步构造一个复杂对象。而抽象工厂着重于多个系列的产品对象。
  - Builder在最后一步返回产品；抽象工厂中，产品是立即返回的
- Composite通常用Builder生成



3.3 Factory Method
工厂方法

Factory Method模式

1. 意图

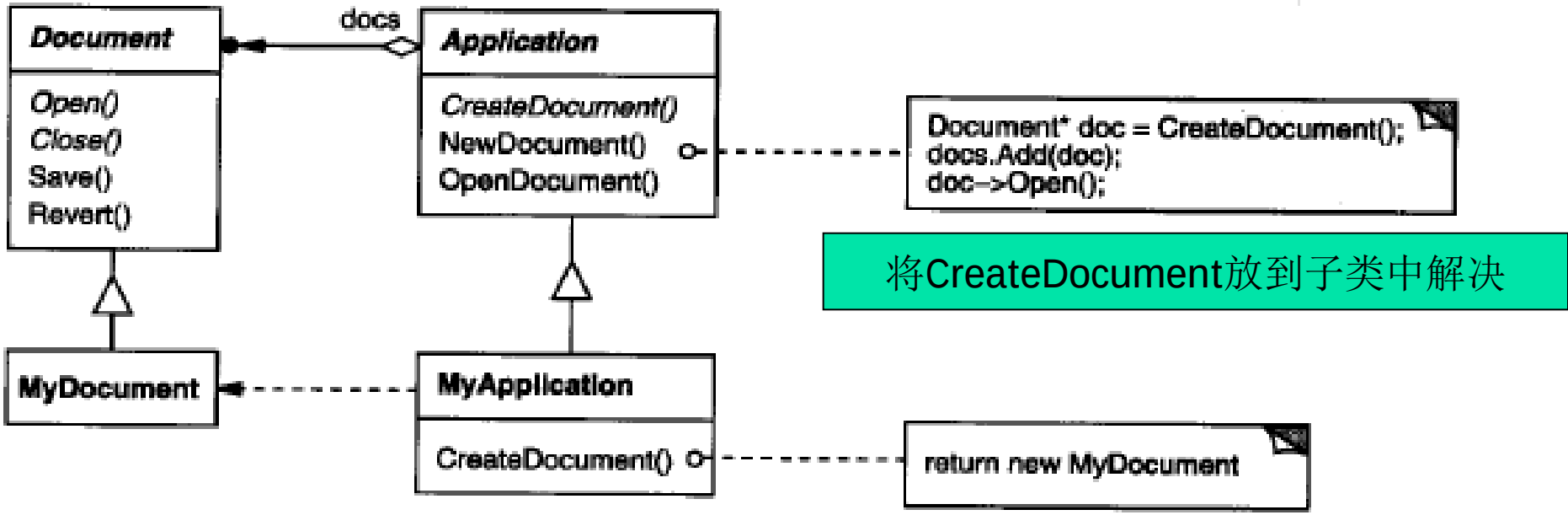
- 定义一个用于创建对象的接口，让子类决定实例化哪一个类
- Factory Method模式使一个类的实例化延迟到其子类。

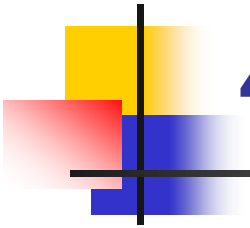
2. 别名

虚构造器　Virtual Constructor

3. 动机

- 基于Application – Document的应用框架

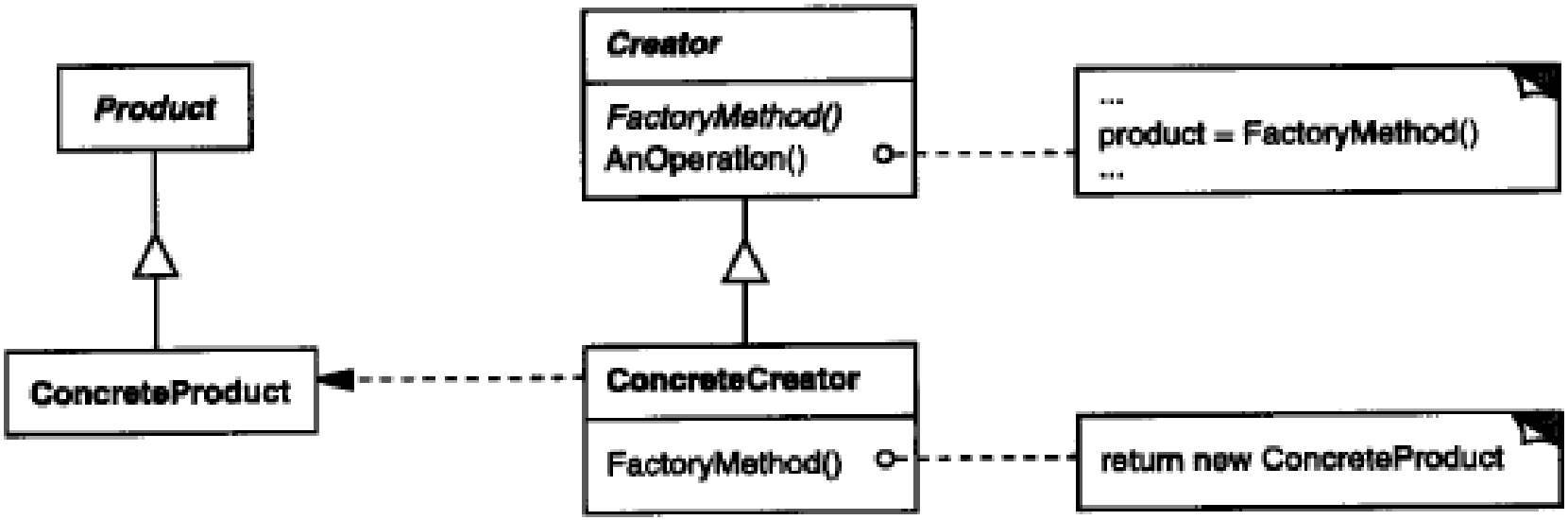




4. 使用性

- 当一个类不知道它所必须创建的对象的类的时候
  - Application不知道要创建的Doc的类
- 当一个类希望由它的子类来指定它所创建的对象的时候
  - Application由MyApplication类来处理
- 当类将职责委托给多个辅助子类中的一个，而且将“具体是哪个辅助子类”的信息局部化。

5. 结构


Product
Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

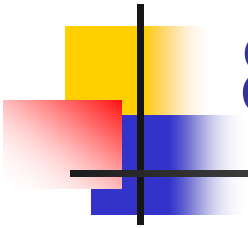
6. 参与者

- Product (Document)
  - 定义所创建对象的接口
- ConcreteProduct (MyDocument)
  - 实现Product接口
- Creator (Application)
  - 声明工厂方法，
  - 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

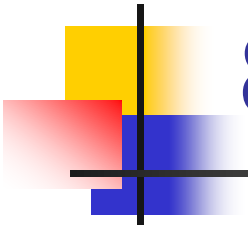
- Creator依赖于其子类来定义工厂方法，从而返回一个适当的ConcreteProduct实例。

- 即： 由子类返回具体的产品对象



8. 效果

- 从Creator的角度
  - 通过工厂方法，不再将与特定应用相关的类绑定到Creator代码中
  - Creator代码仅处理Product接口
- 问题：当仅处理一个具体产品时
  - 可能仅为创建一个特定的ConcreteProduct对象，而必须创建Creator子类



8. 效果

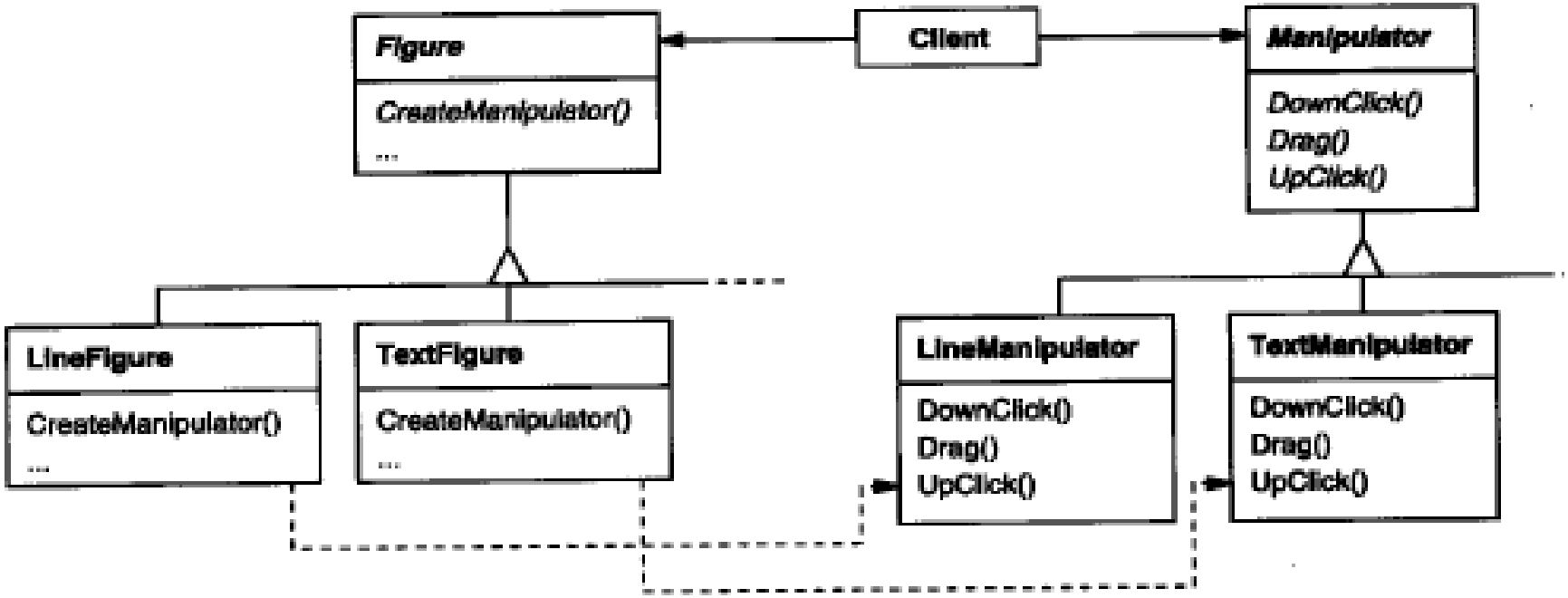
1）为子类提供挂钩（hook）

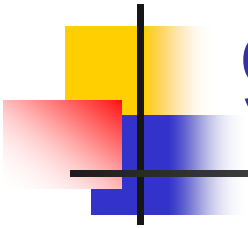
- 用工厂方法在一个类内部创建对象通常比直接创建对象更灵活
- Factory Method给子类一个挂钩以提供对象的扩展版本

8. 效果

2）连接平行的类层次

- 平行类层次
  - 一个类需要将一些职责委托给一个独立的类





9. 实现

1）两种情况

A. Creator是一个抽象类

B. Creator是一个具体类，并为工厂方法提供了一个缺省实现

9. 实现

2）参数化工厂方法

- 工厂方法采用一个参数，标识要创建的对象种类

```
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;
}
```

基类

```
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;

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

子类

9. 实现

3）特定语言的变化和问题

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

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

lazy initialization

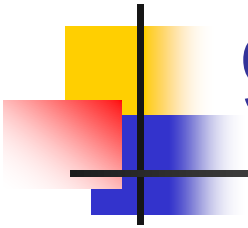
9. 实现

4）使用模板以避免创建子类

```
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;
}
```

避免仅为一个Product对象而制作新的Creator子类

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

StandardCreator<MyProduct> myCreator;
```



9. 实现

5）命名约定

Class* DoMakeClass();

e.g. Document* DoMakeDocument()

10. 代码示例

```
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); }
};
```

Creator类

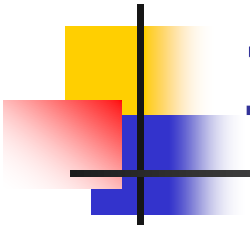
```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    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;
}
```

10. 代码示例

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

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

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

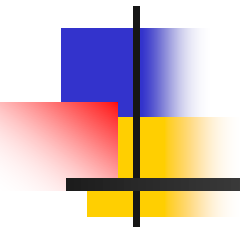
两个ConcreteCreator

```
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;
};
```

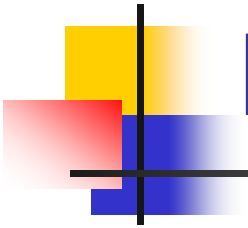


12. 相关模式

- Abstract Factory经常用工厂方法来实现
- 工厂方法通常在Template Methods中被调用
- Prototypes不需要创建子类
  - Prototypes模式要求一个针对Product类的Initialize操作
  - Factory Method模式不需要这样的操作



3.4 Prototype
原型



Prototype模式

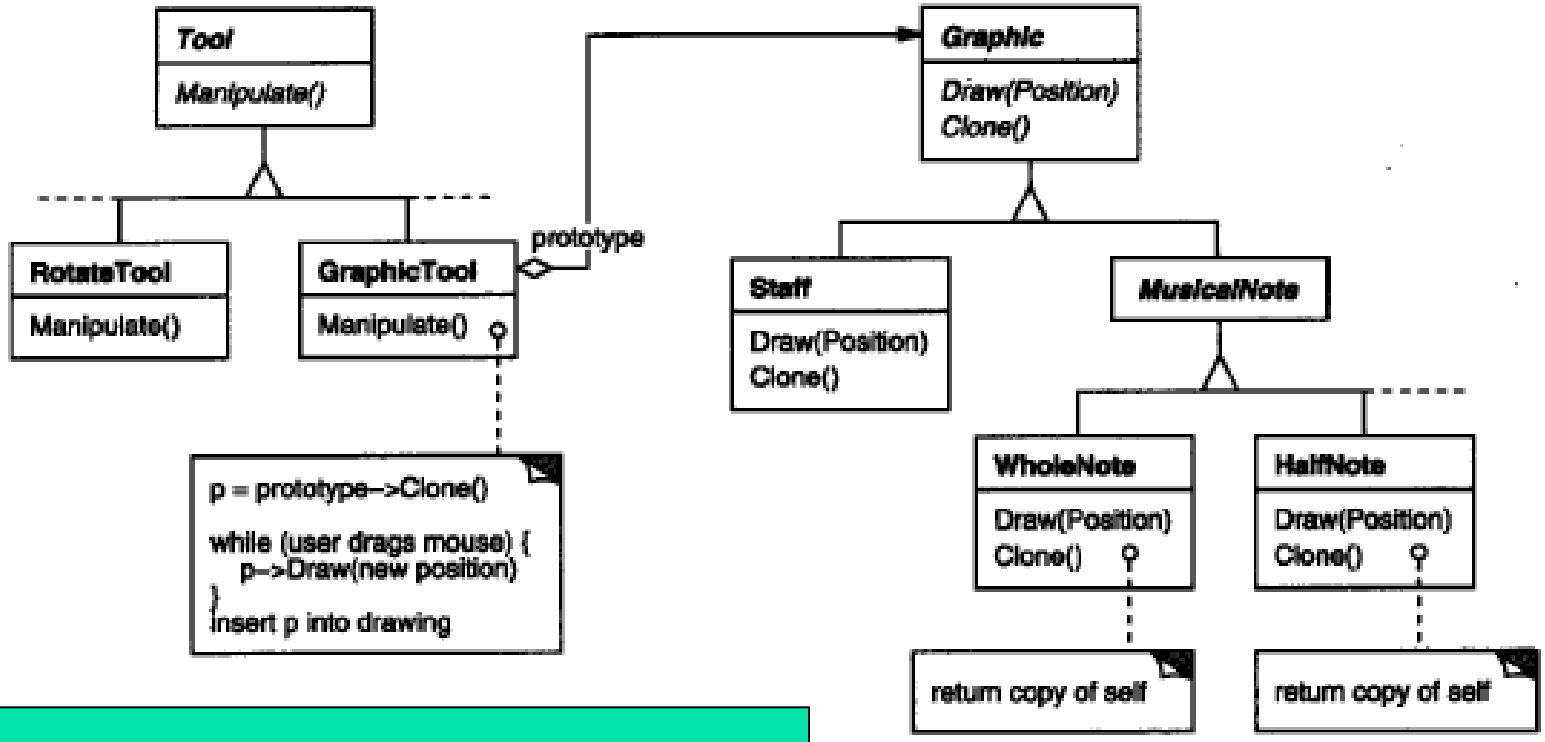
1. 意图

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

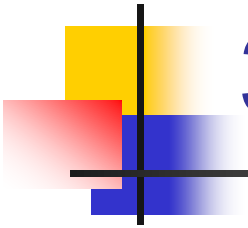
在一个通用的图形编辑器框架上构造一个乐谱编辑器

- 乐谱编辑器



GraphicTool属于通用的图形编辑器框架并不知道特定于应用的类

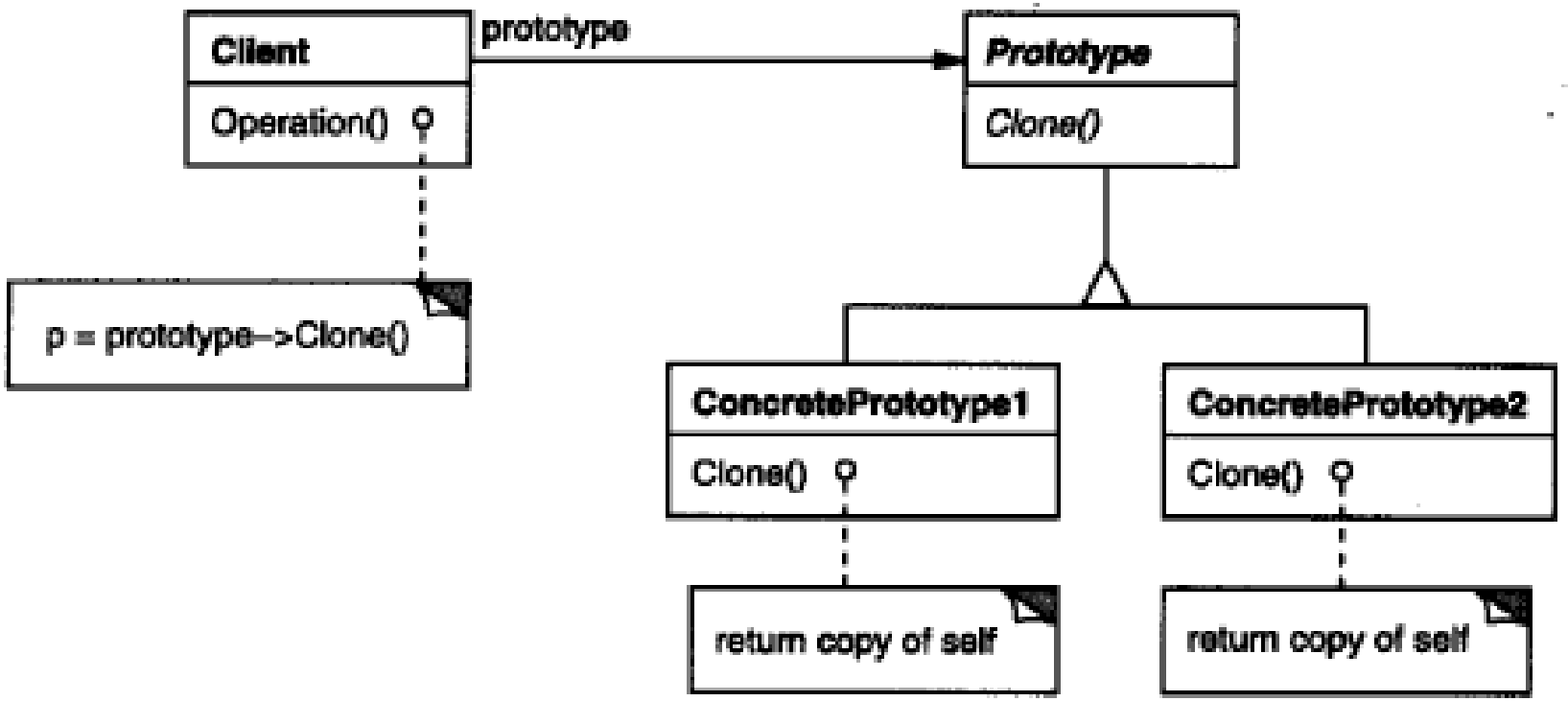
解决：GraphicTool通过Clone一个Graphic子类的实例来创建新的Graphic

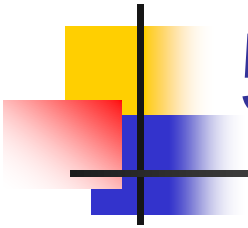


3. 适用性

- 一个系统应该独立于它的产品创建、构成和表示时
- 当要实例化的类是运行时刻指定的。e.g. 通过动态加载
- 避免创建一个与产品类层次平行的工厂类层次
  - 这种是Abstract Factory
- 当一个类的实例只能有几个不同状态组合中的一种时

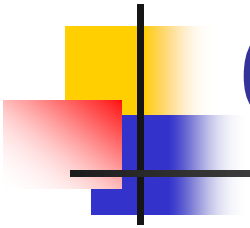
4. 结构


Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self



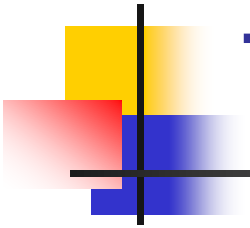
5. 参与者

- Prototype (Graphic)
  - 声明一个克隆自己的接口
- ConcretePrototype (Staff, WholeNote, HalfNote)
  - 实现一个克隆自己的操作
- 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
  - 对客户隐藏了具体的产品类
  - 使得客户无需改变即可使用与特定应用相关的类
    - GraphicTool无需修改即可使用与特定应用相关的类(Staff, WholeNote, HalfNote)

7. 效果

1）运行时刻增加和删除产品

- 通过注册原型实例即可将新的产品类引入系统。
- 灵活：在运行时刻建立和删除原型

2）改变值以指定新对象

- 通过对象复合定义新的行为
- 使得用户无需编程即可定义新“类”

7. 效果

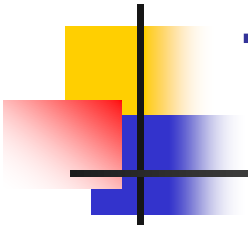
3）改变结构以指定新对象

- 将一个组合对象作为原型

4）减少子类的构造

- 不需要Creator类层次

  e.g. Factory Method经常需要一个与产品类层次平行的Creator类层次



7. 效果

5）用类动态配置应用

- 在运行时刻动态将类装载到应用中

- 缺点
  - 每一个Prototype的子类都必须实现Clone操作

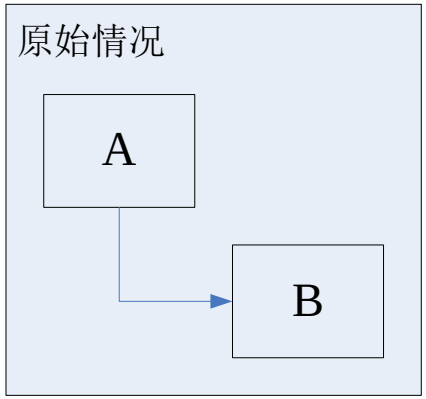
8. 实现

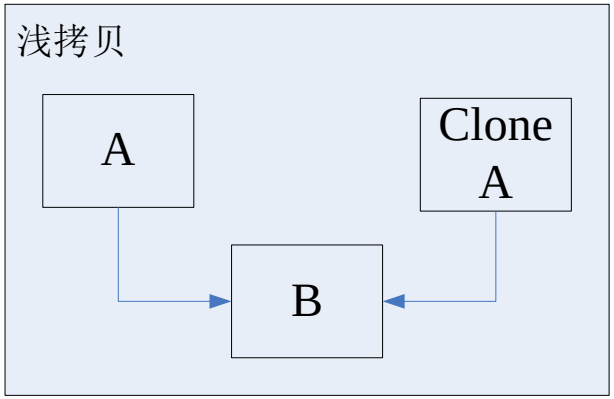
- 使用一个原型管理器
  - 当系统中原型数目不固定时（可以动态创建和销毁），需保持一个可用原型的注册表
  - 客户在注册表中存储和检索原型
  - 注册表：

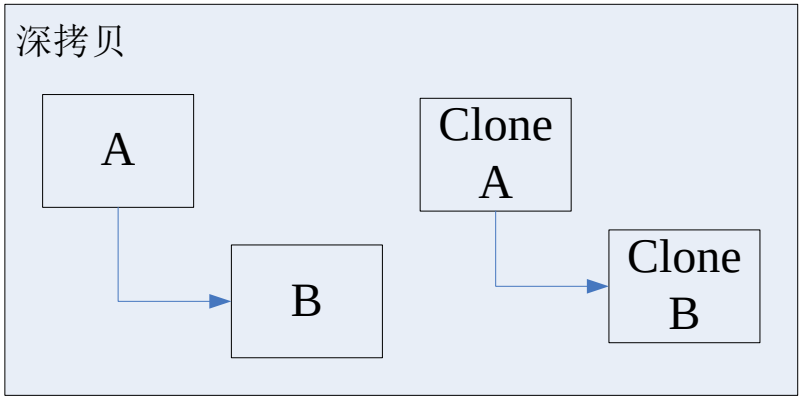
    原型管理器（Prototype manager）
  - 功能
    - 管理存储器（associative store），返回与给定关键字相匹配的原型
    - 通过关键字注册原型和解除注册
    - 客户在运行时可以更改或浏览注册表

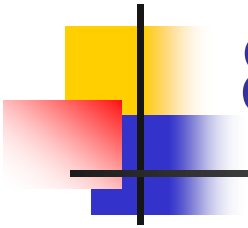
8. 实现

- 实现克隆操作
  - 对象结构包含循环引用时的Clone操作
  - 浅拷贝与深拷贝









8. 实现

- 初始化克隆对象
  - 问题
    - 克隆对象后，Client需要初始化对象的某些内部状态
    - 在Clone操作中传递参数会破坏克隆结构的统一性
  - 解决
    - 原型类为（重）设定的关键状态值定义了操作

    或
    - 引入Initialize操作

9. 代码示例

```
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;
};
```

初始化原型

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

通过原型创建对象

```
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;
}
```

Client使用

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

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

更换原型

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

9. 代码示例

原型对象

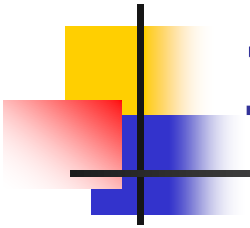
```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    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);
}
```

Initialize

Clone

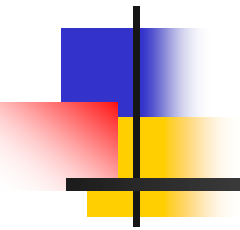
子类实现

```
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);
}
```



11.相关模式

- Prototype 和 Abstract Factory模式
- Composite模式和Decorator模式中可以使用Prototype模式



3.5 Singleton
單件

Singleton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

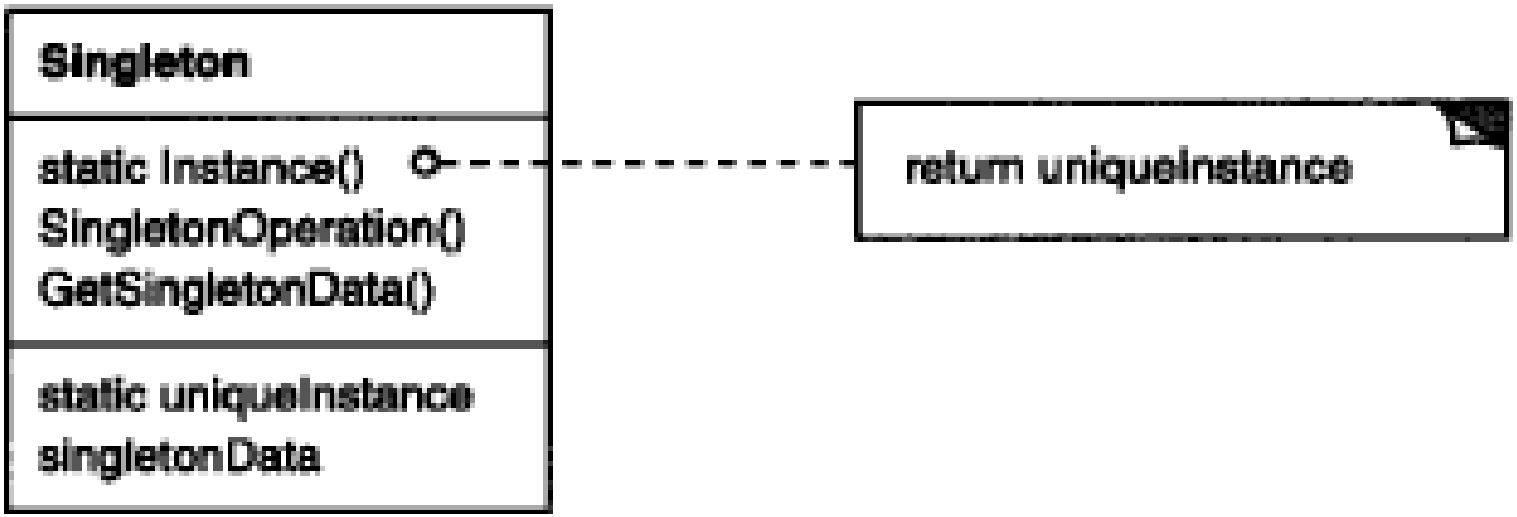
   - 对于一些类，只能有一个实例
   - 使用全局变量可以去访问一个对象，当时不能防止用户实例化多个对象
   - Singleton模式

     让类自身负责保存它的唯一实例

3. 适用性

- 当类只有一个实例，且Client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可扩展的，且Client应该无需更改代码就能使用一个扩展的实例时。

4. 结构


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

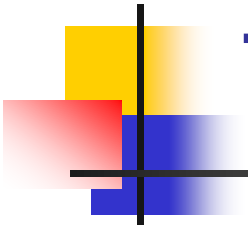
Singleton模式

5. 参与者

- Singleton
  - 定义一个Instance操作，运行Client访问它的唯一实例
    - static Instance()
  - 可能负责创建它自己的唯一实例。

6. 协作

Client只能通过Instance操作来访问一个Singleton的实例。



7. 效果

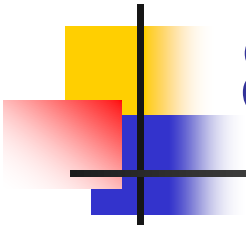
- 对唯一实例的受控访问
  - 通过Singleton类来进行控制
- 缩小名空间
  - 避免全局变量
- 允许对操作和表示的精化
  - Singleton类可以有子类
- 允许可变数目的实例
  - 可以是允许Singleton的多个实例
- 比类操作更灵活
  - 使用类操作(C++的静态成员函数)可以实现单件功能
  - 但其难以改变设计以允许一个类有多个实例
  - 且子类不能对静态成员函数进行多态的重定义

8. 实现

1）保证一个唯一的实例

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

Singleton* Singleton::_instance = 0;

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

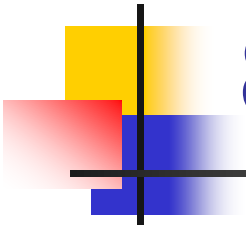
惰性(Lazy)初始化
直到第一次访问时才创建和保存对象



8. 实现

1）保证一个唯一的实例

- 另一种实现方式
  - 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
- 存在的问题
  - 不能保证静态对象只有一个实例被声明
  - 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过程中计算出来的信息
  - 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
  - 使得所有单件无论用到与否，都要被创建



8. 实现

2）创建Singleton类的子类

- 问题：如何选择某个Singleton子类，来创建实例
  - 在Singleton的Instance操作中决定使用哪个单件
  - 将Instance的实现从父类分离出来放入子类
  - 使用单件注册表（registry of singleton）
    - 更灵活的方式

8. 实现

registry of singleton

2）创建Singleton类的子类：

```
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;
};

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;
}
```

用环境变量指定需要的单件的名称
根据注册表信息进行查找

可以在子类的构造器中进行注册

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

此时需要定义一个Mysingleton的静态实例
从而保证构造器被调用，完成注册

```
static MySingleton theSingleton;
```

该情况下，Singleton类不再负责创建单件

9. 代码实例

- 不生成MazeFactory子类的情况

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

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

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

```
MazeFactory* MazeFactory::_instance = 0;

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

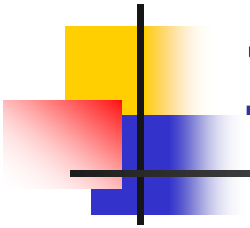
9. 代码实例

- 存在多个MazeFactory子类的情况

```
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;
}
```

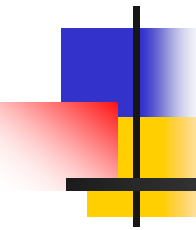
定义一个新的MazeFactory子类时，需要修改这里的Instance操作。

可用registry of singleton解决此问题



11.相关模式

- 可用Singleton模式实现的：
  - Abstract Factory
  - Builder
  - Prototype



3.6 创建型模式的讨论

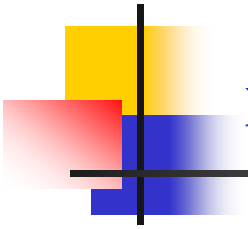
问题：parameterize a system by the classes of objects it creates

1. 生成创建对象的类的子类

- Factory Method模式
- 缺点：
  - 仅为改变产品类，就可能需要创建一个新的子类
  - 这样的改变是级联的（cascade）

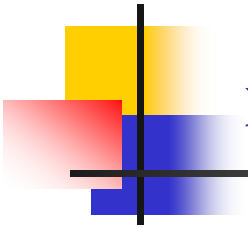
问题：parameterize a system by the classes of objects it creates

2. 依赖于对象复合
   - 定义一个对象（工厂）负责明确产品对应的类，并将它作为该系统的参数
     - Abstract Factory, Builder, Prototype模式的关键特征
     - Abstract Factory由工厂对象产生多个类的对象
     - Builder由工厂对象使用相对复杂的协议，逐步创建一个产品对象
     - Prototype由工厂对象通过拷贝原型对象来创建产品对象



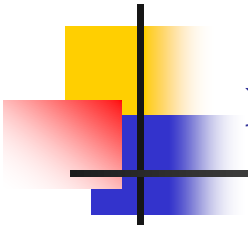
实例：绘图编辑器框架

- 通过产品类来参数化GraphicTool
  - Factory Method模式
    - 针对每个Graphic子类创建一个GraphicTool子类
    - GraphicTool有NewGraphic操作，其子类进行重定义
  - 缺点：
    - GraphicTool子类数目激增，且它们并没有做很多事情



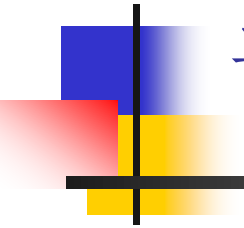
实例：绘图编辑器框架

- 通过产品类来参数化GraphicTool
  - Abstract Factory模式
    - 有一个GraphicsFactory类层次对应每个Graphic的子类，每个工厂仅创建一个产品
    - GraphicTool将合适的工厂对象作为参数
  - 缺点：
    - 需要一个庞大的GraphicsFactory类层次



实例：绘图编辑器框架

- 通过产品类来参数化GraphicTool
  - Prototype模式
    - 每个Graphic的子类实现Clone操作
    - GraphicTool将合适的原型作为参数
  - 评价
    - 仅需要为每个Graphics类实现一个clone操作
    - 减少了类的数目
    - 且Clone操作还可以应用于其他目的



谢谢