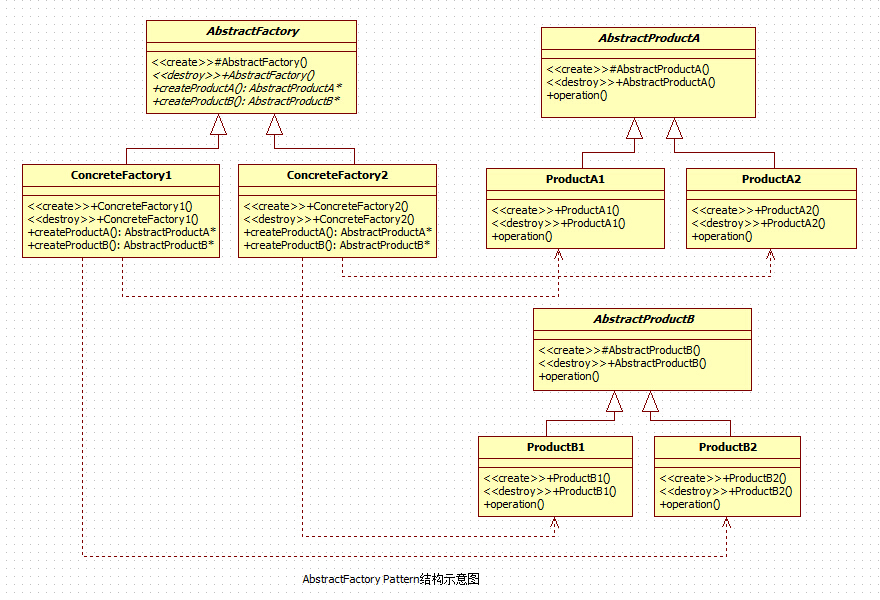
**AbstactFactory模式**

问题:

假设我们要开发一款游戏，当然为了吸引更多的人玩，游戏难度不能太大（让大家都没有信心了，估计游戏也就没有前途了），但是也不能太简单（没有挑战性也不符合玩家的心理）。于是我们就可以采用这样一种处理策略：为游戏设立等级，初级、中级、高级甚至有BT级。假设也是过关的游戏，每个关卡都有一些怪物（monster）守着，玩家要把这些怪物干掉才可以过关。作为开发者，我们就不得不创建怪物的类，然后初级怪物、中级怪物等都继承自怪物类（当然不同种类的则需要另创建类，但是模式相同）。在每个关卡，我们都要创建怪物的实例，例如初级就创建初级怪物（有很多种类）、中级创建中级怪物等。可以想象在这个系统中，将会有成千上万的怪物实例要创建，问题是还要保证创建的时候不会出错：初级不能创建BT级的怪物（玩家就郁闷了，玩家一郁闷，游戏也就挂挂了），反之也不可以。AbstractFactory模式就是用来解决这类问题的：要创建一组相关或者相互依赖的对象。

**模式选择:**

AbstractFactory模式典型的结构图为:



AbstractFactory 模式关键就是将这一组对象的创建封装到一个用于创建对象的类（ConcreteFactory）中，维护这样一个创建类总比维护n多相关对象的创建过程要简单的多。

**实现:**

完整代码示例（code）

// @ File Name : AbstactFactory.h

#if !defined(\_ABSTACTFACTORY\_H)

#define \_ABSTACTFACTORY\_H

class AbstractProductA;

class AbstractProductB;

class AbstractFactory

{

public:

virtual ~*AbstractFactory*();

virtual AbstractProductA\* *createProductA*() = 0;

virtual AbstractProductB\* *createProductB*() = 0;

protected:

AbstractFactory();

};

#endif //\_ABSTACTFACTORY\_H

// @ File Name : AbstactFactory.cpp

#include "AbstractFactory.h"

#include "AbstractProductA.h"

#include "AbstractProductB.h"

AbstractFactory::~*AbstractFactory*()

{

}

AbstractFactory::AbstractFactory()

{

}

// @ File Name : ConcreteFactory1.h

#if !defined(\_CONCRETEFACTORY1\_H)

#define \_CONCRETEFACTORY1\_H

#include "AbstractFactory.h"

class ConcreteFactory1 : public AbstractFactory

{

public:

ConcreteFactory1();

~*ConcreteFactory1*();

AbstractProductA\* *createProductA*();

AbstractProductB\* *createProductB*();

};

#endif //\_CONCRETEFACTORY1\_H

// @ File Name : ConcreteFactory1.cpp

#include "ConcreteFactory1.h"

#include "ProductA2.h"

#include "ProductB2.h"

ConcreteFactory1::ConcreteFactory1()

{

}

ConcreteFactory1::~*ConcreteFactory1*()

{

}

AbstractProductA\* ConcreteFactory1::*createProductA*()

{

return new ProductA2();

}

AbstractProductB\* ConcreteFactory1::*createProductB*()

{

return new ProductB2();

}

// @ File Name : ConcreteFactory2.h

#if !defined(\_CONCRETEFACTORY2\_H)

#define \_CONCRETEFACTORY2\_H

#include "AbstractFactory.h"

class ConcreteFactory2 : public AbstractFactory

{

public:

ConcreteFactory2();

~*ConcreteFactory2*();

AbstractProductA\* *createProductA*();

AbstractProductB\* *createProductB*();

};

#endif //\_CONCRETEFACTORY2\_H

// @ File Name : ConcreteFactory2.cpp

#include "ConcreteFactory2.h"

#include "ProductA1.h"

#include "ProductB1.h"

ConcreteFactory2::ConcreteFactory2()

{

}

ConcreteFactory2::~*ConcreteFactory2*()

{

}

AbstractProductA\* ConcreteFactory2::*createProductA*()

{

return new ProductA1();

}

AbstractProductB\* ConcreteFactory2::*createProductB*()

{

return new ProductB1();

}

// @ File Name : AbstractProductA.h

#if !defined(\_ABSTRACTPRODUCTA\_H)

#define \_ABSTRACTPRODUCTA\_H

class AbstractProductA

{

public:

virtual ~*AbstractProductA*();

void operation();

protected:

AbstractProductA();

};

#endif //\_ABSTRACTPRODUCTA\_H

// @ File Name : AbstractProductA.cpp

#include "AbstractProductA.h"

AbstractProductA::~*AbstractProductA*()

{

}

void AbstractProductA::operation()

{

}

AbstractProductA::AbstractProductA()

{

}

// @ File Name : AbstractProductB.h

#if !defined(\_ABSTRACTPRODUCTB\_H)

#define \_ABSTRACTPRODUCTB\_H

class AbstractProductB

{

public:

virtual ~*AbstractProductB*();

void operation();

protected:

AbstractProductB();

};

#endif //\_ABSTRACTPRODUCTB\_H

// @ File Name : AbstractProductB.cpp

#include "AbstractProductB.h"

AbstractProductB::~*AbstractProductB*()

{

}

void AbstractProductB::operation()

{

}

AbstractProductB::AbstractProductB()

{

}

// @ File Name : ProductA1.cpp

#include "ProductA1.h"

#include <iostream>

ProductA1::ProductA1()

{

std::cout << "ProductA1..." << std::endl;

}

ProductA1::~*ProductA1*()

{

}

void ProductA1::operation()

{

}

// @ File Name : ProductA1.h

#if !defined(\_PRODUCTA1\_H)

#define \_PRODUCTA1\_H

#include "AbstractProductA.h"

class ProductA1 : public AbstractProductA

{

public:

ProductA1();

~*ProductA1*();

void operation();

};

#endif //\_PRODUCTA1\_H

// @ File Name : ProductA2.h

#if !defined(\_PRODUCTA2\_H)

#define \_PRODUCTA2\_H

#include "AbstractProductA.h"

class ProductA2 : public AbstractProductA

{

public:

ProductA2();

~*ProductA2*();

void operation();

};

#endif //\_PRODUCTA2\_H

// @ File Name : ProductA2.cpp

#include "ProductA2.h"

#include <iostream>

ProductA2::ProductA2()

{

std::cout << "ProductA2..." << std::endl;

}

ProductA2::~*ProductA2*()

{

}

void ProductA2::operation()

{

}

// @ File Name : ProductB1.cpp

#include "ProductB1.h"

#include <iostream>

ProductB1::ProductB1()

{

std::cout << "ProductB1..." << std::endl;

}

ProductB1::~*ProductB1*()

{

}

void ProductB1::operation()

{

}

// @ File Name : ProductB2.h

#if !defined(\_PRODUCTB2\_H)

#define \_PRODUCTB2\_H

#include "AbstractProductB.h"

class ProductB2 : public AbstractProductB

{

public:

ProductB2();

~*ProductB2*();

void operation();

};

#endif //\_PRODUCTB2\_H

// @ File Name : ProductB2.cpp

#include "ProductB2.h"

#include <iostream>

ProductB2::ProductB2()

{

std::cout << "ProductB2..." << std::endl;

}

ProductB2::~*ProductB2*()

{

}

void ProductB2::operation()

{

}

#include "AbstractFactory.h"

#include "ConcreteFactory1.h"

#include "ConcreteFactory2.h"

#include <iostream>

using namespace std;

int main()

{

AbstractFactory\* factory1 = new ConcreteFactory1();

factory1->*createProductA*();

factory1->*createProductB*();

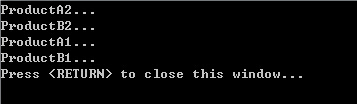
AbstractFactory\* factory2 = new ConcreteFactory2();

factory2->*createProductA*();

factory2->*createProductB*();

return 0;

}



**代码说明:**

AbstractFactory模式的实现代码很简单，在测试程序中可以看到，当我们要创建一组对象（ProductA1，ProductA2）的时候我们只用维护一个创建对象（ConcreteFactory1），大大简化了维护的成本和工作。

AbstractFactory模式和Factory模式的区别是初学（使用）设计模式时候的一个容易引起困惑的地方。实际上，AbstractFactory模式是为创建一组（有多类）相关或依赖的对象提供创建接口，而Factory模式正如我在相应的文档中分析的是为一类对象提供创建接口或延迟对象的创建到子类中实现。并且可以看到，AbstractFactory模式通常都是使用Factory模式实现（ConcreteFactory1）。