Бакулин Никита

ПИ-15-1

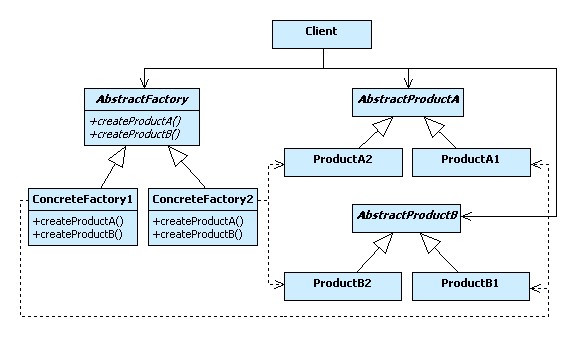
**Абстрактная фабрика**

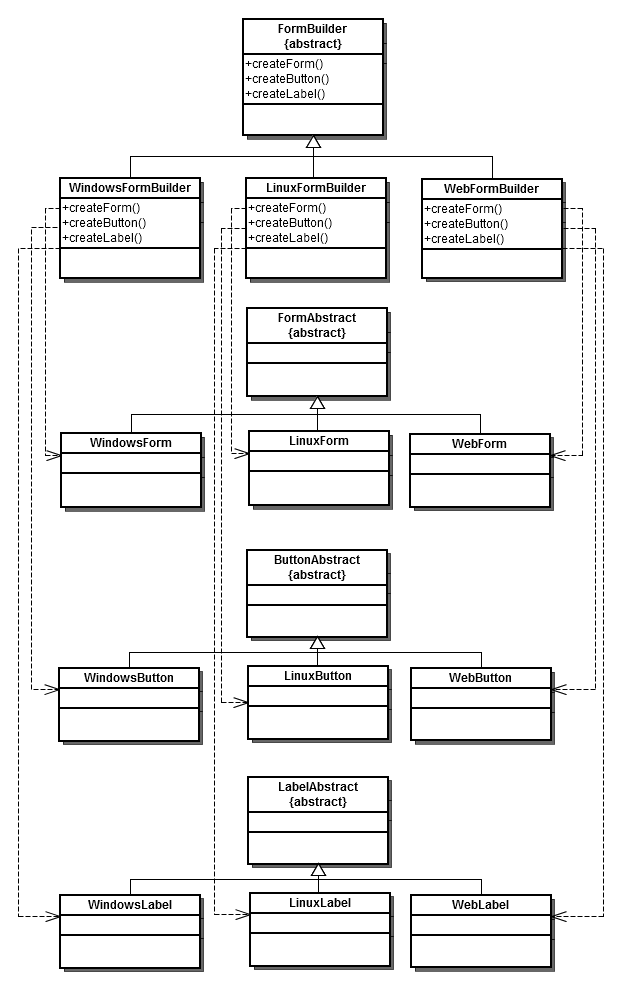
1. Назначение и цель применения

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

* Если система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов.
* Если необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

1. UML-диаграмма





1. Области применения

* Система не должна зависеть от того, как создаются, компонуются и представляются входящие в неё объекты.
* Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения.
* Система должна конфигурироваться одним из семейств составляющих её объектов.
* Требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

1. Особенности

Плюсы:

* Изолирует конкретные классы.
* Упрощает замену семейств продуктов.
* Гарантирует сочетаемость продуктов.
* Упрощает добавление новых продуктов в программу.

Минусы:

* Требует наличия всех типов продуктов каждого класса.
* Усложняется код программы за счёт множества дополнительных классов.

1. Примеры реализации
   1. C#

using System;

namespace DoFactory.GangOfFour.Abstract.Structural

{

class MainApp

{

/// <summary>

/// Точка входа в приложение

/// </summary>

public static void Main()

{

// Вызов абстрактной фабрики №1

AbstractFactory factory1 = new ConcreteFactory1();

Client client1 = new Client(factory1);

client1.Run();

// Вызов абстрактной фабрики №2

AbstractFactory factory2 = new ConcreteFactory2();

Client client2 = new Client(factory2);

client2.Run();

// Ожидание ввода

Console.ReadKey();

}

}

/// <summary>

/// Класс абстрактной фабрики

/// </summary>

abstract class AbstractFactory

{

public abstract AbstractProductA CreateProductA();

public abstract AbstractProductB CreateProductB();

}

/// <summary>

/// Класс фабрики №1

/// </summary>

class ConcreteFactory1 : AbstractFactory

{

public override AbstractProductA CreateProductA()

{

return new ProductA1();

}

public override AbstractProductB CreateProductB()

{

return new ProductB1();

}

}

/// <summary>

/// Класс фабрики №2

/// </summary>

class ConcreteFactory2 : AbstractFactory

{

public override AbstractProductA CreateProductA()

{

return new ProductA2();

}

public override AbstractProductB CreateProductB()

{

return new ProductB2();

}

}

/// <summary>

/// Абстрактный класс продукта А

/// </summary>

abstract class AbstractProductA

{

}

/// <summary>

/// Абстрактный класс продукта В

/// </summary>

abstract class AbstractProductB

{

public abstract void Interact(AbstractProductA a);

}

/// <summary>

/// Первый класс продукта типа А

/// </summary>

class ProductA1 : AbstractProductA

{

}

/// <summary>

/// Первый класс продукта типа В

/// </summary>

class ProductB1 : AbstractProductB

{

public override void Interact(AbstractProductA a)

{

Console.WriteLine(this.GetType().Name +

" interacts with " + a.GetType().Name);

}

}

/// <summary>

/// Второй класс продукта типа А

/// </summary>

class ProductA2 : AbstractProductA

{

}

/// <summary>

/// Второй класс продукта типа В

/// </summary>

class ProductB2 : AbstractProductB

{

public override void Interact(AbstractProductA a)

{

Console.WriteLine(this.GetType().Name +

" interacts with " + a.GetType().Name);

}

}

/// <summary>

/// Класс клиента, в котором происходит взаимодействие между объектами

/// </summary>

class Client

{

private AbstractProductA \_abstractProductA;

private AbstractProductB \_abstractProductB;

// Конструктор

public Client(AbstractFactory factory)

{

\_abstractProductB = factory.CreateProductB();

\_abstractProductA = factory.CreateProductA();

}

public void Run()

{

\_abstractProductB.Interact(\_abstractProductA);

}

}

}