**Разработка программ с использованием порождающих паттернов**

*Цель работы*: научиться разрабатывать программы с использованием таких структурных шаблонов проектирования из каталога GoF, как Строитель, Абстрактная фабрика, Фабричный метод.

***1.1 Необходимые теоретические сведения***

### Шаблон Абстрактная фабрика

Абстрактная фабрика – порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы.

Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение.

Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс. Этот шаблон предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Как и в реальной жизни фабрика имеет некую специализацию, создавая товары или устройства какого-либо определенного типа.

Фабрика, которая выпускает, например, мебель, не может производить, например, еще и компоненты для смартфонов.

В программировании фабрика объектов может создавать только объекты определенного типа, которые используют единый интерфейс.

Самыми главными преимуществами данного паттерна в С#, является упрощение создания объектов различных классов, использующих единый интерфейс.

Паттерн предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны.

От класса «абстрактная фабрика» наследуются классы конкретных фабрик, которые содержат методы создания конкретных объектов-продуктов, являющихся наследниками класса «абстрактный продукт», объявляющего интерфейс для их создания.

Клиент пользуется только интерфейсами, заданными в классах «абстрактная фабрика» и «абстрактный продукт» (рисунок 1.1).

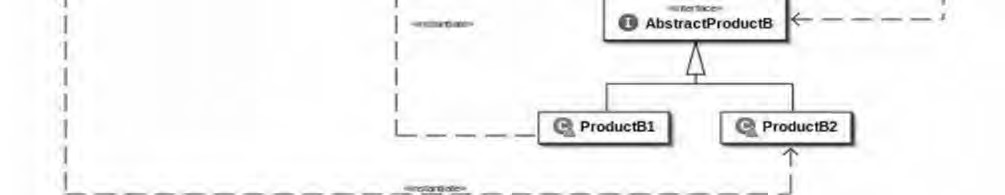
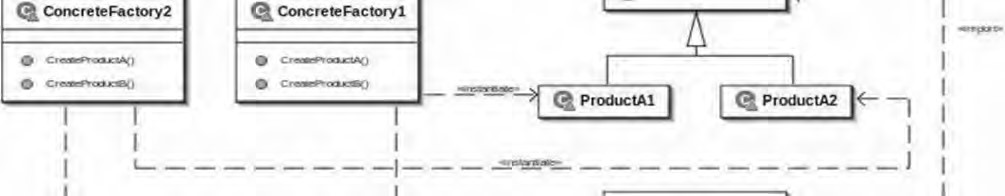


Рисунок 1.1 – UML-диаграмма шаблона Абстрактный продукт

Ниже приведена формальная реализация паттерна на языке C#.

abstract class AbstractFactory

{

public abstract AbstractProductA CreateProductA(); public abstract AbstractProductB CreateProductB();

}

class ConcreteFactory1: AbstractFactory

{

public override AbstractProductA CreateProductA()

{

return new ProductA1();

}

public override AbstractProductB CreateProductB()

{

return new ProductB1();

} }

class ConcreteFactory2: AbstractFactory

{

public override AbstractProductA CreateProductA()

{

return new ProductA2();

}

public override AbstractProductB CreateProductB()

{

return new ProductB2();

}

}

abstract class AbstractProductA

{}

abstract class AbstractProductB

{}

class ProductA1: AbstractProductA

{}

class ProductB1: AbstractProductB

{}

class ProductA2: AbstractProductA

{}

class ProductB2: AbstractProductB {}

class Client

{

private AbstractProductA abstractProductA; private AbstractProductB abstractProductB;

public Client(AbstractFactory factory)

{

abstractProductB = factory.CreateProductB(); abstractProductA = factory.CreateProductA();

}

public void Run()

{ }

}

### Шаблон Фабричный метод

Фабричный метод – порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса (рисунок 1.2).

В момент создания наследники могут определить, какой класс создавать.

Иными словами, Фабрика делегирует создание объектов наследникам родительского класса.

Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Шаблон определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

Фабричный метод позволяет классу делегировать создание подклассов.

Используется, когда:

* классу заранее неизвестно, объекты каких подклассов ему нужно создавать;
* класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами;
* класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

*Структура*:

Product – продукт; определяет интерфейс объектов, создаваемых абстрактным методом.

ConcreteProduct – конкретный продукт, реализует интерфейс Product.

Creator – создатель; объявляет фабричный метод, который возвращает объект типа Product.

Может также содержать реализацию этого метода «по умолчанию»; может вызывать фабричный метод для создания объекта типа Product.

ConcreteCreator – конкретный создатель; переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса ConcreteProduct.

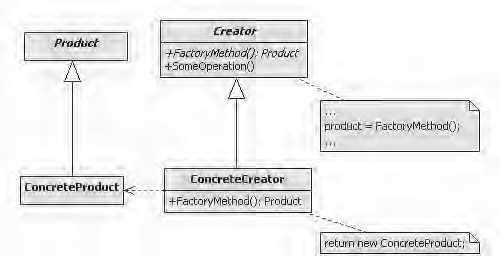


Рисунок 1.2 – UML-диаграмма шаблона Фабричный метод

Формальное определение паттерна на языке C# может выглядеть следующим образом.

abstract class Product

{}

class ConcreteProductA : Product

{}

class ConcreteProductB : Product

{}

abstract class Creator

{

public abstract Product FactoryMethod();

}

class ConcreteCreatorA : Creator

{

public override Product FactoryMethod() { return new

ConcreteProductA(); }

}

class ConcreteCreatorB : Creator

{

public override Product FactoryMethod() { return new

ConcreteProductB(); }

}

### Шаблон Строитель

Строитель (Builder) – шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

Паттерн Строитель рекомендуется использовать в следующих случаях:

* когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой;
* когда необходимо обеспечить получение различных вариаций объекта в процессе его создания.

UML-диаграмма паттерна показана на рисунке 1.3.

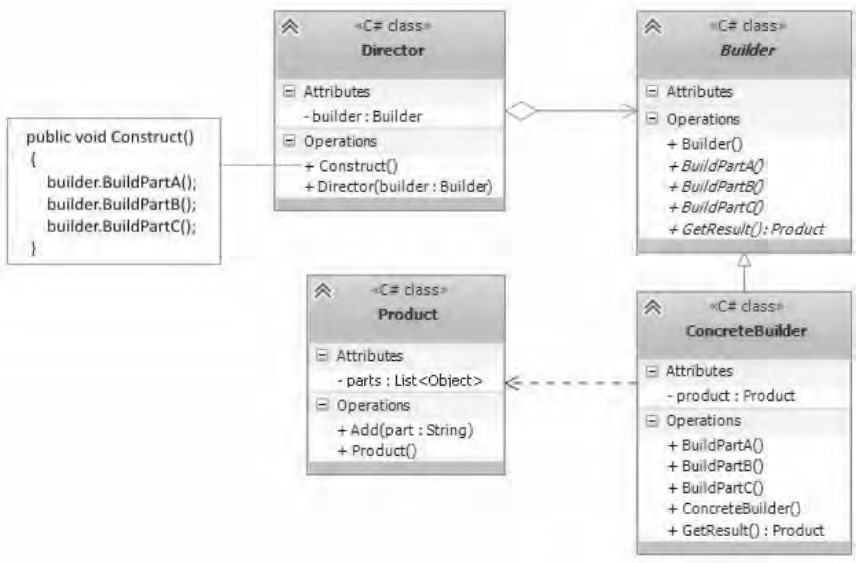


Рисунок 1.3 – UML-диаграмма шаблона Строитель

Формальное описание паттерна на языке C#.

class Client

{

void Main()

{

Builder builder = new ConcreteBuilder(); Director director = new Director(builder); director.Construct();

Product product = builder.GetResult();

} }

class Director

{

Builder builder;

public Director(Builder builder)

{

this.builder = builder;

}

public void Construct()

{

builder.BuildPartA();

builder.BuildPartB(); builder.BuildPartC();

}

}

abstract class Builder

{

public abstract void BuildPartA(); public abstract void BuildPartB(); public abstract void BuildPartC(); public abstract Product GetResult();

}

class Product

{

List<object> parts = new List<object>(); public void Add(string part)

{

parts.Add(part);

}

}

class ConcreteBuilder : Builder

{

Product product = new Product(); public override void BuildPartA()

{

product.Add("Part A");

}

public override void BuildPartB()

{  product.Add("Part B");

} public override void BuildPartC()

{

product.Add("Part C");

}

public override Product GetResult()

{

return product;

}

}

Участники паттерна:

* Product: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts;
* Builder: определяет интерфейс для создания различных частей объекта Product;
* ConcreteBuilder: конкретная реализация Buildera. Создает объект

Product и определяет интерфейс для доступа к нему;

* Director: распорядитель – создает объект, используя объекты Builder.

# Разработка программ с использованием структурных паттернов

*Цель работы: научиться разрабатывать программы с использованием структурных шаблонов проектирования Адаптер и Фасад из каталога GoF.*

***2.1 Необходимые теоретические сведения***

### Шаблон Адаптер

Адаптер – структурный шаблон проектирования, предназначенный для преобразования интерфейса класса к другому интерфейсу, на который рассчитан клиент.

*Проблема.*

Как обеспечить взаимодействие несовместимых интерфейсов или как создать единый устойчивый интерфейс для нескольких классов с разными интерфейсами?

*Решение.*

Преобразовать исходный интерфейс класса к другому виду с помощью промежуточного объекта-адаптера.

Основными участниками решения являются:

* Client – клиент, использующий целевой интерфейс;
* ITarget – целевой интерфейс;
* Adapter – адаптер, реализующий интерфейс ITarget;
* Adapted – адаптируемый класс, имеющий интерфейс, несовместимый с интерфейсом клиента.
* Работа клиента с адаптируемым объектом происходит следующим образом:
* клиент обращается с запросом к объекту-адаптеру Adapter, вызывая его метод Request() через целевой интерфейс ITarget;
* адаптер Adapter преобразует запрос в один или несколько вызовов к адаптируемому объекту Adapted;
* клиент получает результаты вызова, не зная ничего о преобразованиях, выполненных адаптером.

Шаблон Адаптер имеет следующие разновидности:

* адаптер объекта применяет для адаптации одного интерфейса к другому композицию объектов адаптируемого класса (рисунок 2.1).
* адаптер класса использует для адаптации наследование адаптируемому классу (рисунок 2.2).

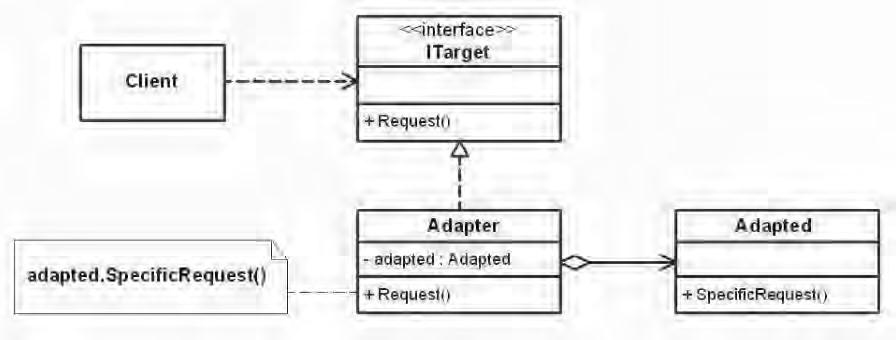


Рисунок 2.1 – Диаграмма классов шаблона Адаптер объекта

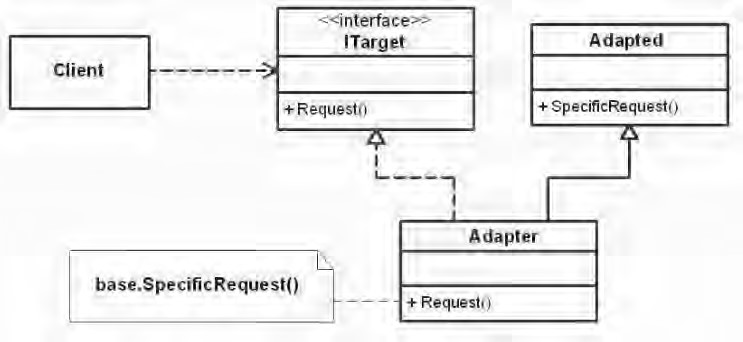


Рисунок 2.2 – Диаграмма классов шаблона Адаптер класса

*Результаты.*

Система становится независимой от интерфейса внешних классов (компонентов, библиотек). При переходе на использование внешних классов не требуется переделывать всю систему, достаточно переделать один класс Adapter.

Результаты применения адаптеров классов и объектов различаются. **Адаптер объекта:**

* позволяет работать объекту Adapter со многими адаптируемыми объектами (например, с объектами класса Adapted и его производных классов);
* отличается сложностью при замещении операций класса Adapted (для этого может потребоваться создать класс производный от Adapted, и добавить в класс Adapter ссылку на этот производный класс).

**Адаптер классов:**

* обеспечивает простой доступ к элементам адаптируемого класса, поскольку Adapter является производным классом от Adapted;
* характеризуется легкостью изменения адаптером операций адаптируемого класса Adapted;
* обладает возможностью работы только с одним адаптируемым классом (возможность адаптировать классы, производные от Adaptee, отсутствует).

*Реализация.*

Простейший пример реализации шаблона Адаптер объекта на языке C# представлен ниже.

// Представляет целевой интерфейс public interface ITarget

{ void Request();

}

// Представляет адаптируемые объекты class Adapted

{

public void SpecificRequest()

{

Console.WriteLine("Вызван SpecificRequest()"); }

}

// Представляет объекты-адаптеры public class Adapter : ITarget

{

Adapted adapted = new Adapted(); public void Request()

{ adapted.SpecificRequest();

}

}

// Клиент class Client

{ static void Main(string[] args)

{

ITarget target = new Adapter(); target.Request();

}

}

Отличие реализации шаблона Адаптер класса будет заключаться только в коде класса Adapter.

// Представляет объекты-адаптеры public class Adapter : Adapted, ITarget

{ public void Request()

{

base.SpecificRequest();

}

}

### Шаблон Фасад

Фасад – структурный шаблон проектирования, позволяющий скрыть сложность подсистемы путем сведения всех возможных вызовов к одному объекту (фасадному объекту), делегирующему их объектам подсистемы (рисунок 2.3).

*Проблема*.

Как обеспечить унифицированный интерфейс с подсистемой, если нежелательна высокая степень связанности с этой подсистемой или реализация подсистемы может измениться?

*Решение.*

Определить одну точку взаимодействия с подсистемой – фасадный объект, обеспечивающий единый упрощенный интерфейс с подсистемой и возложить на него обязанность по взаимодействию с классами подсистемы.

Участники решения:

* Client – взаимодействует с фасадом и не имеет доступа к классам подсистемы;
* Facade – перенаправляет запросы клиентов к классам подсистемы;
* Классы подсистемы – выполняют работу, порученную объектом Facade, ничего не зная о существовании фасада, то есть не хранят ссылок на него.

*Результаты:*

* клиенты изолируются от классов (компонентов) подсистемы, что уменьшает число объектов, с которыми клиенты взаимодействуют, и упрощает работу с подсистемой;
* снижается степень связанности между клиентами и подсистемой, что позволяет изменять классы подсистемы, не затрагивая при этом клиентов.

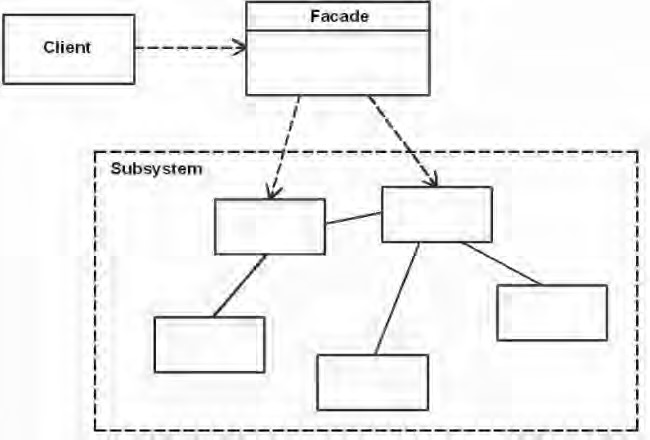


Рисунок 2.3 – Диаграмма классов шаблона Фасад

*Реализация.*

Простой пример реализации шаблона Фасад на языке C# представлен ниже.

namespace Subsystem // Содержит классы подсистемы

{ internal class ClassA

{ public string A1()

{

return "Метод A1() класса ClassA";

}

public string A2()

{ return "Метод A2() класса ClassA"; }

}

internal class ClassB

{ public string B1()

{ return "Метод B1() класса ClassB";

}

public string B2()

{ return "Метод B2() класса ClassB"; }

}

}

// Представляет фасадные объекты public class Facade

{

Subsystem.ClassA a; Subsystem.ClassB b; public Facade()

{ a = new Subsystem.ClassA(); b = new Subsystem.ClassB();

}

public void F1()

{

Console.WriteLine("Метод F1() класса Facade:\n"

+"\* {0}\n" +"\* {1}\n", a.A1(), b.B2());

} public void F2()

{

Console.WriteLine("Метод F2() класса Facade:\n" +"\* {0}\n" +"\* {1}\n" + "\* {2}\n", a.A2(), b.B1(), b.B2());

}

}

// Клиент class Client

{ static void Main(string[] args)

{

Facade facade = new Facade(); facade.F1();

facade.F2();

}

}

# Разработка программ с использованием паттернов поведения

*Цель работы: научиться разрабатывать программы с использованием поведенческих шаблонов проектирования Стратегия, Состояние и Шаб- лонный метод.*

***3.1 Необходимые теоретические сведения***

### Поведенческие шаблоны проектирования. Диаграммы конечных автоматов UML

Поведенческие шаблоны (англ. «Behavioral Patterns») предназначены для определения алгоритмов и способов реализации взаимодействия различных объектов и классов.

Одним из средств описания поведения программных объектов в языке UML являются диаграммы конечных автоматов (диаграммы состояний).

Диаграммы конечных автоматов (англ. «state machine diagrams») UML отображают жизненный цикл объекта с помощью состояний, собы- тий и переходов.

Основными элементами диаграмм конечных автоматов являются:

* состояние – это ситуация во время жизни объекта, в которой он удовлетворяет определённым условиям, выполняет определённую деятельность или находится в ожидании событий. Состояния изображаются на диаграмме в виде прямоугольников со скруглёнными углами. Специальными видами состояний являются начальное и конечное состояния, изображаемые соответственно символами:



* переход – это отношение между двумя состояниями, указывающее на то, что объект из первого состояния перейдёт во второе, при выполнении определённого условия. Переходы на диаграммах конечных автоматов изображают стрелками, ведущими от одного состояния к другому;
* событие – это значимое или заслуживающее внимания происшествие, которое может инициализировать переход объекта от одного состояния к другому. Событием может являться поступление сигнала, выполнение какого-либо условия, истечение определённого периода времени. События разделяют на внешние, внутренние и временные.

Пример простой диаграммы конечных автоматов для объекта «входная дверь» показан на рисунке 3.1.

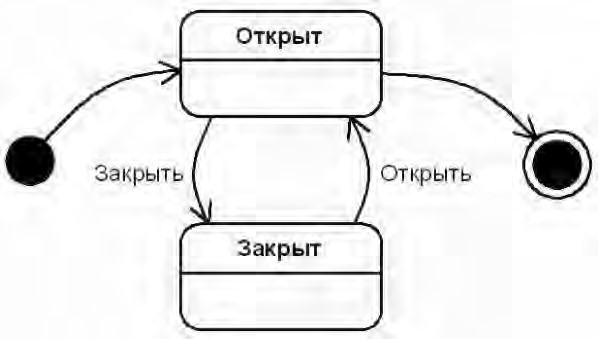


Рисунок 3.1 – Простая диаграмма конечных автоматов UML

### Шаблон Состояние

Шаблон Состояние (англ. «State») управляет изменением поведения объекта в зависимости от его внутреннего состояния.

*Проблема*.

Как изменять поведение объекта в зависимости от его внутреннего состояния?

*Решение.*

Определить для каждого состояния отдельный класс со стандартным интерфейсом.

*Структура.*

Диаграмма классов шаблона Состояние представлена на рисунке 3.2.

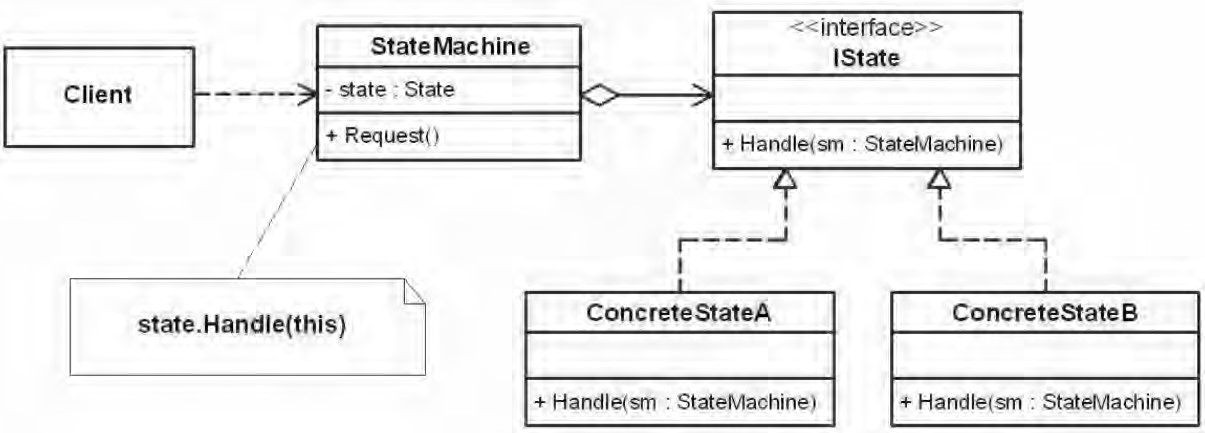


Рисунок 3.2 – Диаграмма классов шаблона Состояние

Участники шаблона:

* State – абстрактный класс, определяющий общий интерфейс для всех конкретных состояний;
* StateMachine – класс с несколькими внутренними состояниями, хранит экземпляр класса State;
* ConcreteStateA, ConcreteStateB – классы конкретных состояний, обрабатывающие запросы от класса StateMachine; каждый класс предоставляет собственную реализацию обработки запроса.

*Реализация.*

|  |  |
| --- | --- |
| Простой пример реализации шаблона Состояние приведён ниже.    // Представляет состояния interface IState  { void Handle(StateMachine sm); }  // Представляет конкретные состояния A class ConcreteStateA : IState  { public void Handle(StateMachine sm) { | на языке C# |
| Console.Write("Объект {0} в sm.ToString());  }  }  // Представляет конкретные состояния B class ConcreteStateB : IState  { public void Handle(StateMachine sm) { | сост. A", |
| Console.Write("Объект {0} в  sm.ToString());  } | сост. B", |

}

// Представляет конечные автоматы class StateMachine

{ public IState State { get; set; } public void Request()

{

State.Handle(this);

}

}

// Представляет клиентов class Client

{ static void Main(string[] args)

{

StateMachine sm = new StateMachine(); sm.State = new ConcreteStateA(); sm.Request();

sm.State = new ConcreteStateB(); sm.Request();

}

}

### Шаблоны Стратегия и Шаблонный метод

Шаблон Стратегия (англ., «Strategy» или «Policy») позволяет менять алгоритм независимо от клиентов, которые его используют.

*Проблема.*

Как спроектировать изменяемые, но надёжные алгоритмы (стратегии)? *Решение.*

Определить для каждого алгоритма (стратегии) отдельный класс со стандартным интерфейсом.

*Структура.*

Диаграмма классов шаблона Стратегия показана на рисунке 3.3.

Участники шаблона:

* Strategy – класс, определяющий общий для всех поддерживаемых алгоритмов интерфейс;
* Context – класс, хранящий ссылку на экземпляр класса Стратегия;
* ConcreteStrategyA, ConcreteStrategyB – классы, представляющие конкретные стратегии, использующие интерфейс класса Strategy для реализации алгоритма.

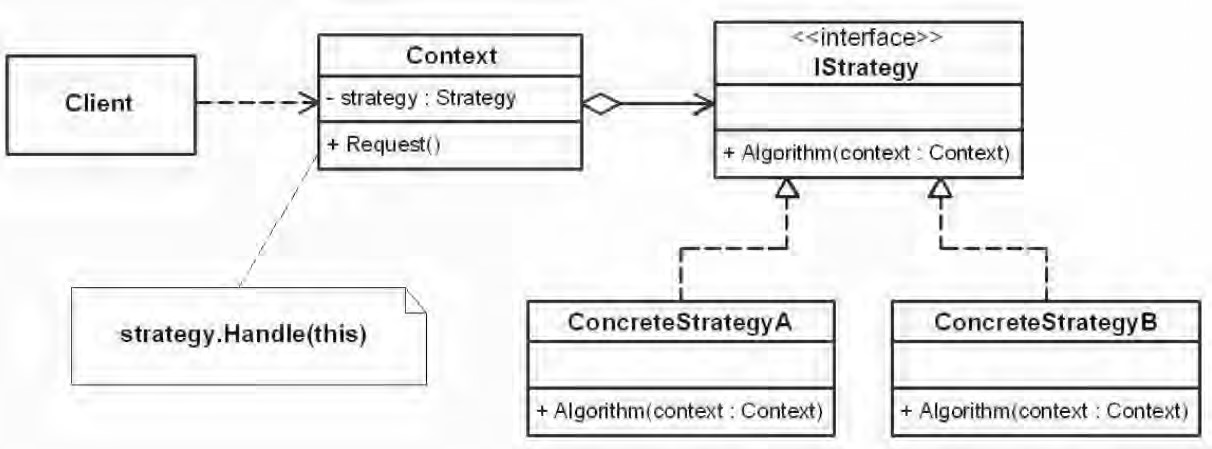


Рисунок 3.3 – Диаграмма классов шаблона Стратегия

*Реализация.*

Простой пример реализации шаблона Стратегия на языке C# приведён ниже.

// Представляет стратегии, реализуемые классами public interface IStrategy

{ void Algorithm();

}

// Представляет конкретные стратегии A public class ConcreteStrategyA : IStrategy

{ public void Algorithm()

{

Console.WriteLine("Алгоритм стратегии A"); }

}

// Представляет конкретные стратегии B public class ConcreteStrategyB : IStrategy

{

public void Algorithm()

{

Console.WriteLine("Алгоритм стратегии B"); }

}

// Представляет контексты, использующие стратегии public class Context

{ public IStrategy Strategy { get; set; } public void Request()

{

Strategy.Algorithm();

}

}

// Представляет клиентов class Client

{

static void Main(string[] args)

{

Context context = new Context(); context.Strategy = new ConcreteStrategyA(); context.Request();

context.Strategy = new ConcreteStrategyB(); context.Request();

}

}

Шаблон Шаблонный метод (англ., «Template Method») определяет основу алгоритма (шаблонный метод) в базовом классе и позволяет производным классам переопределять отдельные шаги алгоритма, не изменяя его структуру в целом.

*Проблема.*

Как определить алгоритм и реализовать возможность переопределения некоторых шагов алгоритма в производных классах без изменения общей структуры алгоритма?

*Решение.*

Создать абстрактный класс, определяющий следующие операции:

* шаблонный метод, который задаёт основу алгоритма;
* абстрактные операции, которые замещаются в производных классах для реализации шагов алгоритма.

*Структура.*

Диаграмма классов шаблона Шаблонный метод показана на рисунке 3.4.

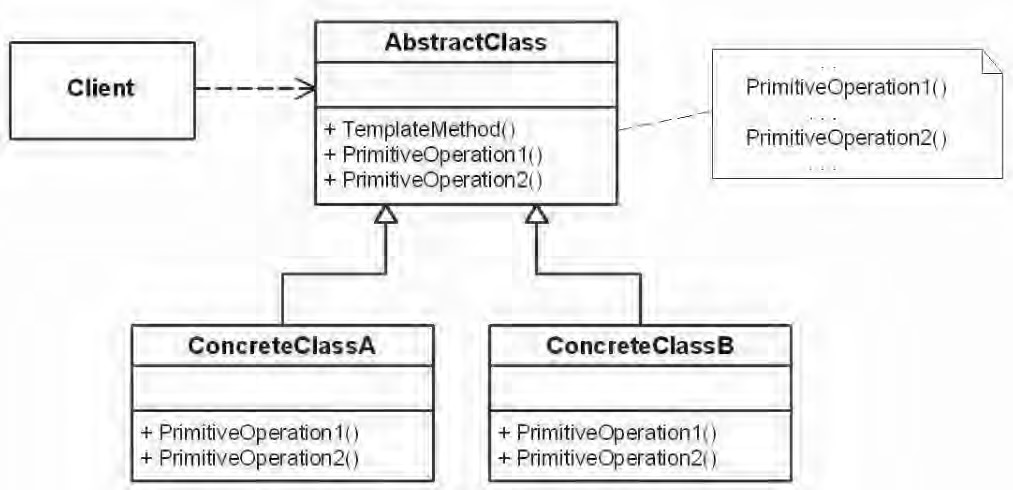


Рисунок 3.4 – Диаграмма классов шаблона Шаблонный метод

Участники шаблона:

* AbstractClass определяет абстрактные частные операции и содержит шаблонный метод, который вызывает эти операции;
* ConcreteClassA, ConcreteClassB реализуют частные операции, вызываемые шаблонным методом.

*Реализация.*

Простой пример реализации шаблона Шаблонный метод на языке C# приведён ниже.

// Представляет Абстрактный класс public abstract class AbstractClass

{ int a = 10;

// Шаблонный метод

public int TemplateMethod(int x)

{ int y = 3;

if (x > a) y += PartMethod1(x); else y += PartMethod2(x); y -= a; return y;

}

public abstract int PartMethod1(int x); public abstract int PartMethod2(int x);

}

// Представляет Конкретный класс A public class ConcreteClassA : AbstractClass

{ int k = 5;

public override int PartMethod1(int x)

{ return k + x; }

public override int PartMethod2(int x)

{ return k \* x; }

}

// Представляет Конкретный класс B public class ConcreteClassB : AbstractClass

{

int k = 3;

public override int PartMethod1(int x)

{ return x \* x - k; }

public override int PartMethod2(int x)

{ return x + x \* k; }

}

// Представляет клиентов class Client

{ static void Main(string[] args)

{

AbstractClass a = new ConcreteClassA();

AbstractClass b = new ConcreteClassB(); int x = 20;

Console.WriteLine(a.TemplateMethod(x));

Console.WriteLine(b.TemplateMethod(x)); }

}

# Разработка программ с использованием шаблонов GRASP

*Цель работы: получить навыки разработки программ с использованием шаблонов GRASP.*

***4.1 Необходимые теоретические сведения***

В разрабатываемой программной системе могут быть определены десятки и сотни различных классов, на которые могут быть возложены сотни и тысячи обязанностей.

Во время объектно-ориентированного проектирования при определении принципов взаимодействия объектов необходимо распределить обязанности между классами. При правильном выполнении этой задачи система становится гораздо проще для понимания, поддержки и расширения. Кроме того, появляется возможность повторного использования уже разработанных компонентов в последующих приложениях.

Проектирование на основе обязанностей (англ. Responsibility-Driven Design, RDD) – это общий подход к проектированию классов, в котором считается, что классы программной системы имеют определённые обязанности и должны взаимодействовать с другими классам для их выполнения.

В общем случае выделяют два типа обязанностей: знание (knowing) и действие (doing).

Обязанности, относящиеся к знаниям объекта (вытекают из модели предметной области):

* наличие информации о закрытых инкапсулированных данных;
* наличие информации о связанных объектах;
* наличие информации о следствиях или вычисляемых величинах.

Обязанности, относящиеся к действиям объекта:

* выполнение некоторых действий самим объектом (например, создание экземпляра или выполнение вычислений);
* инициирование действий других объектов;
* управление действиями других объектов и их координирование.

Выявить и описать основные принципы распределения обязанностей, положенные в основу подхода RDD, позволяют общие шаблоны распределения обязанностей (англ. General Responsibility Assignment Software Patterns, GRASP).

К основным шаблонам GRASP относятся:

* Information Expert – информационный эксперт;
* Creator – создатель экземпляров класса;
* Low Coupling – слабая связанность;
* High Cohesion – сильное сцепление.

**Шаблоны Information Expert и Creator**

Шаблон Information Expert является наиболее общим шаблоном GRASP и при распределении обязанностей используется гораздо чаще других шаблонов.

*Проблема*.

Каков наиболее общий принцип распределения обязанностей между классами при объектно-ориентированном проектировании?

*Решение.*

Назначить обязанность информационному эксперту – классу, который обладает достаточной информацией для выполнения этой обязанности.

*Результаты:*

* поддержка инкапсуляции; для выполнения требуемых задач объекты используют собственные данные;
* более простое понимание и поддержка системы классов, моделирующих заданную систему.
* Создание объектов в объектно-ориентированной системе является одним из наиболее стандартных видов деятельности. Поэтому при распределении обязанностей, связанных с созданием объектов, следует руководствоваться шаблоном Creator.

*Проблема.*

Какой класс должен отвечать за создание нового экземпляра выбранного класса A?

*Решение.*

Назначить классу B обязанность создавать объекты другого класса A, если выполняется одно из следующих условий:

* класс B агрегирует или содержит объекты A;
* класс B активно использует объекты A;
* класс B обладает данными инициализации для объектов A.

*Результаты:*

* использование шаблона не увеличивает число связей между классами, поскольку созданный класс, как правило, виден только для класса-создателя.
* если процедура создания объекта достаточно сложная, то предпочтительно использовать такой шаблон, как Фабрика.

**Шаблоны Low Coupling и High Cohesion**

Шаблон Слабая связанность позволяет снизить влияние изменений в одном классе системы на другие связанные с ним классы.

Степень связанности (англ. «coupling») – это мера, определяющая, насколько жестко один элемент программной системы связан с другими элементами, либо каким количеством данных о других элементах он обладает.

Класс с низкой степенью связанности (слабо связанный) зависит от не большого числа других классов.

Класс с высокой степенью связанности (жестко связанный) зависит от множества других классов. Наличие таких классов нежелательно, поскольку оно приводит к возникновению следующих проблем:

* изменения в связанных классах приводят к изменениям в данном классе;
* затрудняется понимание каждого класса в отдельности;
* усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

*Проблема.*

Как уменьшить влияние изменений, вносимых в данный класс, на другие классы?

*Решение.*

Распределить обязанности между классами таким образом, чтобы степень связанности системы оставалась низкой.

*Результаты.*

Улучшаются возможности для повторного использования классов системы.

Появляется возможность поручить разработку отдельных частей системы разным разработчикам.

При злоупотреблении шаблоном система будет состоять из набора изолированных сложных классов, самостоятельно выполняющих все операции, и набора классов, хранящих данные.

Ещё одним важным шаблоном GRASP является шаблон Сильное сцепление, который позволяет создавать простые для понимания классы, обладающие возможностью повторного использования.

В терминах объектно-ориентированного проектирования функциональное сцепление (англ., cohesion) – это мера связанности или сфокусированности обязанностей класса (подсистемы).

Класс обладает высокой степенью сцепления (сильным сцеплением), если его обязанности тесно связанны между собой, и он не выполняет непомерных объёмов работы.

Класс с низкой степенью сцепления (слабым сцеплением) выполняет много несвязанных между собой обязанностей. Такие классы приводят к возникновению следующих проблем:

* сложность понимания системы;
* сложность при повторном использовании;
* сложность поддержки;
* ненадёжность, постоянная подверженность изменениям.

*Проблема.*

Как обеспечить сфокусированность обязанностей классов, их управляемость и ясность для понимания?

*Решение.*

Обеспечить высокую степень сцепления при распределении обязанностей.

*Результаты.*

Классы с высокой степенью сцепления просты в поддержке и повторном использовании.

В некоторых случаях неоправданно использовать высокое сцепление для распределённых серверных объектов.