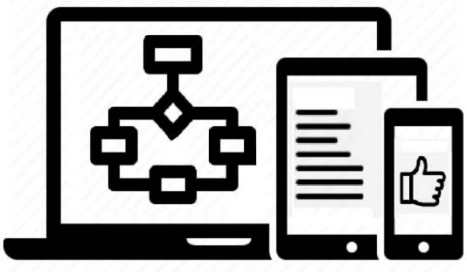
МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Программное обеспечение информационных технологий»

ПРАКТИКА НАПИСАНИЯ ПРОГРАММНОГО КОДА

Методические рекомендации к лабораторным работам для студентов направления подготовки 01.03.04 «Прикладная математика» дневной формы обучения



Могилев 2023

УДК 004.42
ББК 32.973-018
П69

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Программное обеспечение информационных технологий» «24» марта 2023 г., протокол № 8

Составитель канд. техн. наук, доц. Н. Н. Горбатенко

Рецензент Ю. С. Романович

Методические рекомендации разработаны на основе рабочей программы по дисциплине «Практика написания программного кода» для студентов направления подготовки 01.03.04 «Прикладная математика» дневной формы обучения и предназначены для использования при выполнении лабораторных работ.

Учебное издание

ПРАКТИКА НАПИСАНИЯ ПРОГРАММНОГО КОДА

Ответственный за выпуск	В. В. Кутузов
Корректор	И. В. Голубцова
Компьютерная верстка	Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 21 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

1 Основные правила выполнения лабораторных работ

1 Ознакомиться с целью предстоящей работы и условием индивидуального задания.

2 Изучить вопросы раздела «Необходимые теоретические сведения».

3 Разработать логическую модель решения задачи индивидуального задания, используя язык моделирования UML.

4 Набрать и отладить текст программы на ПК в среде Visual Studio.NET.

5 Подготовить отчёт по лабораторной работе. Отчёт оформляется на листах формата А4, он должен содержать:

– титульный лист (рисунок 1.1);

– цель работы;

– текст индивидуального задания;

– логическую модель решения задачи;

– листинг разработанной программы;

– скриншоты с результатами выполнение программы на ПК;

– выводы о проделанной работе.

Белорусско-Российский университет
Кафедра «Программное обеспечение информационных технологий»

Отчёт по лабораторной работе №__
по дисциплине «ООП»

Название темы лабораторной работы
Вариант задания –

Выполнил	Проверил
Ст. гр. ПИР-161	Преподаватель
ФИО	ФИО

Могилев 2018

Рисунок 1.1 – Структура титульного листа

6 Защитить лабораторную работу. Защита включает в себя демонстрацию работы программы преподавателю и ответы на его вопросы по теме лабораторной работы в объёме методических рекомендаций. После защиты лабораторной работы преподаватель ставит на титульном листе свою подпись и дату. Только после этого лабораторная работа считается полностью выполненной, и студент может приступить к выполнению следующей работы.

2 Лабораторная работа № 1. Язык моделирования UML. Построение диаграмм классов

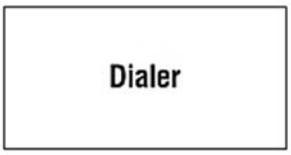
Цель работы: получить навыки построения диаграмм классов на языке моделирования UML.

2.1 Необходимые теоретические сведения

Диаграмма классов описывает классы и отражает отношения, существующие между ними.

Классы

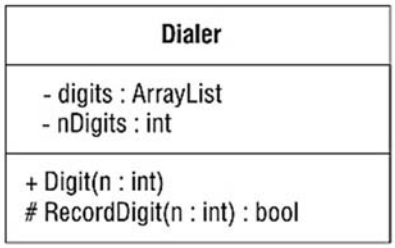
На UML-диаграммах классы изображают в виде прямоугольников, внутри которых записывают имя класса (рисунок 2.1). Код, соответствующий этой диаграмме, приведен справа от рисунка. Это наиболее распространенный способ изображения класса. На большинстве диаграмм ничего, кроме имени класса, и не нужно, чтобы понять происходящее.




```
public class Dialer
{
}
```

Рисунок 2.1 – Изображение класса `Dialer`

В прямоугольнике класса может быть несколько отделений. В верхнем записывается имя класса, в среднем – переменные-члены, в нижнем – методы. На рисунке 2.2 показаны отделения и соответствующий такому представлению код.




```
public class Dialer
{
  private ArrayList digits;
  private int nDigits;
  public void Digit(int n);
  protected bool RecordDigit(int n);
}
```

Рисунок 2.2 – Изображение класса с отделениями и соответствующий код

Обратите внимание на символ, предшествующий именам переменных и методов в прямоугольнике класса. Минусом `(-)` обозначаются закрытые `(private)` члены, решеткой `(#)` – защищенные `(protected)`, плюсом `(+)` – открытые `(public)`.

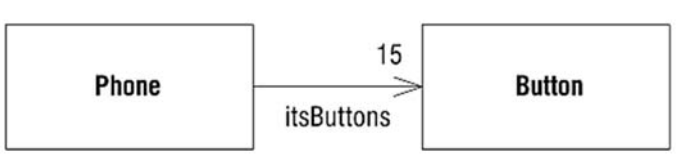
Тип переменной или аргумента метода указывается после двоеточия, стоящего за их именами, а вслед за именем метода после двоеточия указывается тип возвращаемого значения.

Иногда подобные детали полезны, но злоупотреблять ими не стоит. UML-диаграммы – не место для объявления переменных и методов. Это лучше делать в исходном коде. Использовать описанные дополнения следует лишь в том случае, когда они существенны с точки зрения назначения диаграммы.

Ассоциация

Ассоциации между классами чаще всего представляют переменные экземпляра, в которых хранятся ссылки на другие объекты. Например, на рисунке 2.3 показана ассоциация между классами `Phone` и `Button`. Направление стрелки го-

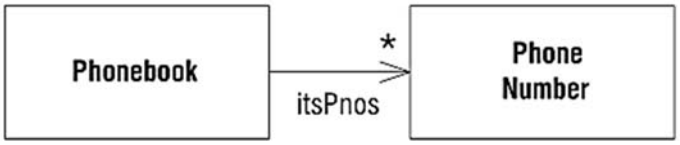
ворит о том, что в `Phone` хранится ссылка на `Button`. Рядом со стрелкой указывается имя переменной экземпляра, а число говорит о том, сколько в этой переменной может храниться ссылок.




```
public class Phone
{
  private Button
itsButtons[15];
}
```

Рисунок 2.3 – Ассоциация

На рисунке 2.3 с объектом `Phone` связано 15 объектов `Button`. На рисунке 2.4 показана ситуация, когда ограничения на количество нет. Объект `Phonebook` (Телефонная книга) может быть связан со многими объектами `PhoneNumber` (звездочкой обозначается понятие «много»). В C# такое отношение чаще всего реализуется с помощью класса `ArrayList` или иных классов коллекций.

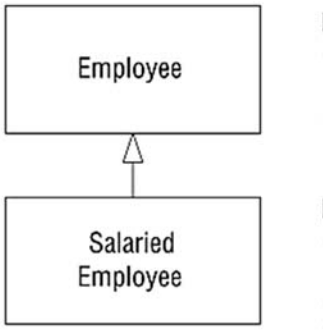



```
public class Phonebook
{
  private ArrayList itsPnos;
}
```

Рисунок 2.4 – Ассоциация один-ко-многим

Наследование

В UML нужно очень внимательно относиться к остриям стрелок. На рисунке 2.5 показана стрелка, указывающая на `Employee`, обозначает наследование. Если небрежно отнестись к рисованию стрелок, то будет непонятно, имеется ли в виду наследование или ассоциация. Чтобы не возникало путаницы, лучше располагать классы, связанные отношением наследования, по вертикали, а связанные отношением ассоциация – по горизонтали.




```
public class Employee
{
  ...
}

public class SalariedEmployee : Employee
{
  ...
}
```

Рисунок 2.5 – Наследование

В UML направления всех стрелок определяются зависимостью в исходном коде. Так как имя `Employee` упомянуто в классе `SalariedEmployee`, а не наоборот, то стрелка указывает на `Employee`. Таким образом, в UML стрелка наследования всегда направлена в сторону базового класса.

В UML есть специальная нотация для того вида наследования, который существует в C# между классом и интерфейсом. Это пунктирная стрелка наследования, показанная на рисунке 2.6.

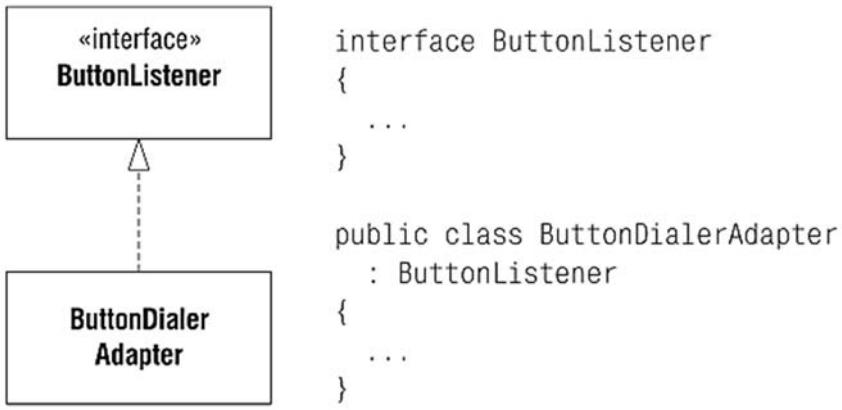


Рисунок 2.6 – Отношение «реализует»

Пример диаграммы классов

На рисунке 2.7 приведена простая диаграмма классов, входящих в систему банкомата. Эта диаграмма интересна не только тем, что на ней показано, но и тем, что не показано. Обратите внимание, что на диаграмме не показаны все интерфейсы. Важным является то, чтобы пользователь знал, какие классы нужно сделать интерфейсами, а какие реализовать. Например, из диаграммы сразу видно, что класс `WithdrawalTransaction` общается с интерфейсом `CashDispenser`. Понятно, что в системе должен быть класс, реализующий `CashDispenser`, но здесь нас не интересует его конкретное воплощение.

Заметьте также, что на рисунке 2.7 нет подробного документирования методов различных интерфейсов пользователя. Конечно, в интерфейсе `WithdrawalUI` будут и еще методы, кроме двух показанных. Но помещать их на эту диаграмму означало бы только загромоздить ее. Рекомендуется использовать репрезентативную выборку методов.

Еще отметим соглашение о расположении ассоциаций по горизонтали, а наследования – по вертикали. Это очень помогает не путать два принципиально различных вида отношений. Без такого соглашения было бы очень трудно извлечь смысл из переплетения стрелок.

Посмотрите, как в диаграмме выделены три разные зоны. Транзакции и их действия находятся слева, различные интерфейсы – справа, а реализации интерфейсов – внизу. Обратите также внимание, что соединений между группами не-

много и они расположены регулярно. В одном случае это три ассоциации, причем все направлены в одну сторону. В другом – три отношения наследования, объединенные в одну линию.

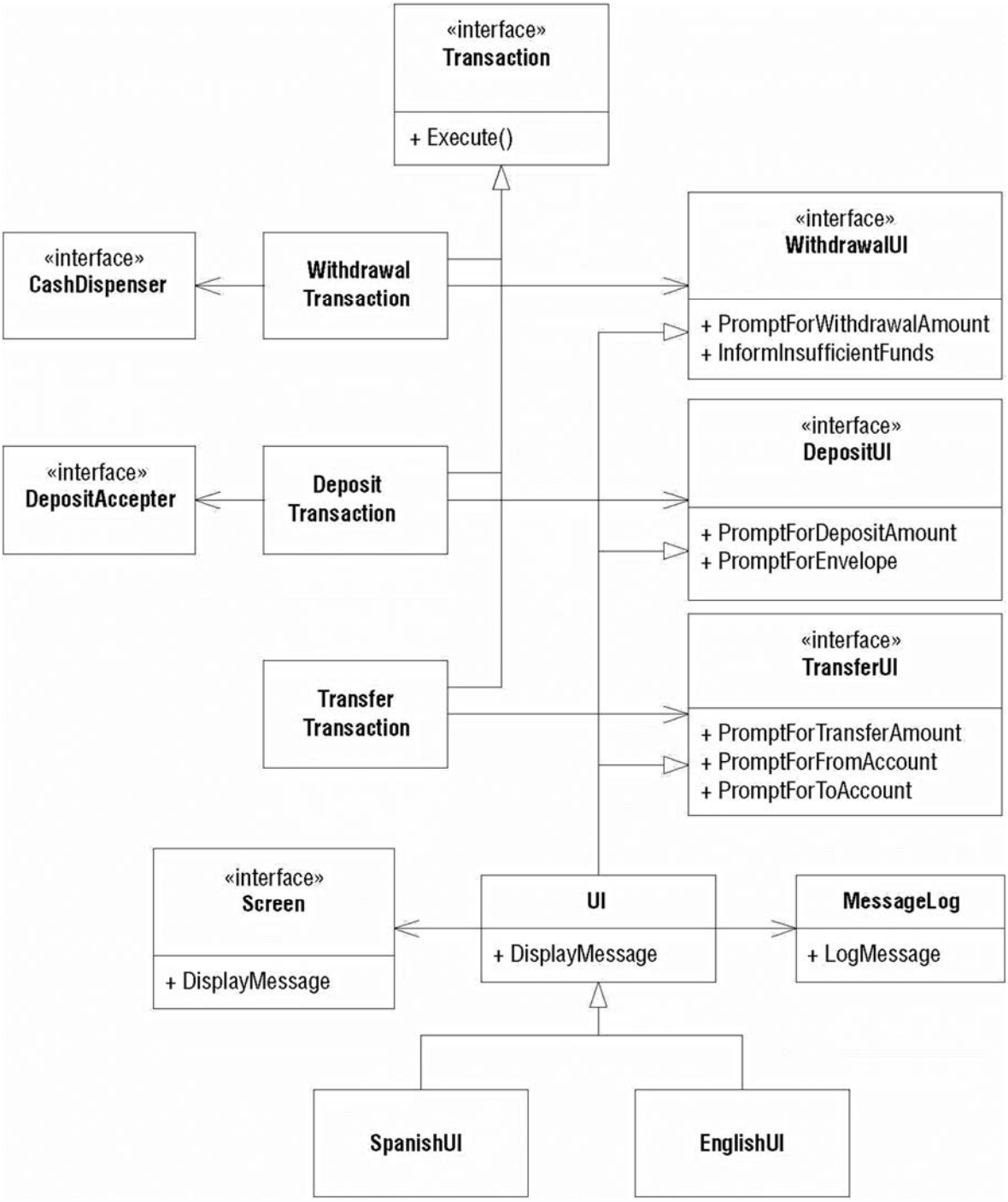


Рисунок 2.7 – Диаграмма классов банкомата

Такие группы и способ их соединения облегают чтение диаграмм. Глядя на эту диаграмму, можно увидеть следующий код.

```
public abstract class UI :
WithdrawalUI, DepositUI, TransferUI
```

```
{
  private Screen itsScreen;
  private MessageLog itsMessageLog;

  public abstract void PromptForDepositAmount();
  public abstract void PromptForWithdrawalAmount( );
  public abstract void InformInsufficientFunds();
  public abstract void PromptForEnvelope();
  public abstract void PromptForTransferAmount();
  public abstract void PromptForFromAccount();
  public abstract void PromptForToAccount();

  public void DisplayMessage(string message)
  {
    itsMessageLog.LogMessage(message);
    itsScreen.DisplayMessage{message);
  }
}
```

2.2 Индивидуальные задания

Задание

Для заданного варианта задания нарисовать UML-диаграмму классов.

1 Система Факультатив. Преподаватель объявляет запись на Курс. Студент записывается на Курс, обучается и по окончании Преподаватель выставляет Оценку, которая сохраняется в Архиве Студентов. Преподавателей и Курсов при обучении может быть несколько.

2 Система Платежи. Клиент имеет Счет в банке и Кредитную Карту (КК). Клиент может оплатить Заказ, сделать платеж на другой Счет, заблокировать КК и аннулировать Счет. Администратор может заблокировать КК за превышение кредита.

3 Система Больница. Пациенту назначается лечащий Врач. Врач может сделать назначение Пациенту (процедуры, лекарства, операции). Медсестра или другой Врач выполняют назначение. Пациент может быть выписан из Больницы по окончании лечения, при нарушении режима или при иных обстоятельствах.

Контрольные вопросы

1 Для чего переназначена диаграмма классов?

2 Перечислите основные графические символы, используемые при построении диаграммы классов.

3 Как на диаграмме классов представляются отношения наследования и агрегации?

3 Лабораторная работа № 2. Язык моделирования UML. Построение диаграмм взаимодействия

Цель работы: получить навыки построения диаграмм взаимодействия объектов классов на языке моделирования UML.

3.1 Необходимые теоретические сведения

Диаграммы языка UML, отражающие взаимодействие объектов друг с другом, получили название диаграмм взаимодействий. Наиболее распространенным видом диаграмм взаимодействий является диаграмма последовательностей, представленная на рисунке 3.1.

Диаграммы последовательностей следует читать сверху вниз:

– каждый прямоугольник в верхней части диаграммы представляет конкретный объект. Большинство прямоугольников содержит имена классов, причем обратите внимание, что в некоторых случаях перед именем класса стоит двоеточие. Другие прямоугольники содержат такие имена, как `shape1:Square;`;

– прямоугольники в верхней части диаграммы содержат имя класса (справа от двоеточия) и, возможно, имя объекта (указывается перед двоеточием);

– вертикальные линии представляют жизненный путь объектов. К сожалению, большинство программ автоматизированной подготовки UML-диаграмм не поддерживают этот аспект и рисуют линии от верхнего края и до конца листа, оставляя неясными действительные сроки существования объекта;

– обмен сообщениями между объектами отображается с помощью горизонтальных линий, проведенных между соответствующими вертикальными линиями;

– иногда возвращаемые значения и/или объекты должны быть указаны подробно, а иногда и так понятно, что они возвращаются.

Обратимся к рисунку 3.1. Слева вверху показана подпрограмма `Main`, которая посылает объекту `ShapeDB` (который не поименован) сообщение с требованием извлечь сведения о фигурах `(getShapes)`. Получив этот запрос, объект `ShapeDB` выполняет следующее:

– создает экземпляр класса `Collection`;

– создает экземпляр класса `Square`;

– добавляет объект класса `Square` в объект класса `Collection`;

– создает экземпляр класса `Circle`;

– добавляет объект класса `Circle` в объект класса `Collection`;

– возвращает указатель на объект класса `Collection` в вызывающую подпрограмму `(Main)`.

Чтобы выяснить дальнейшие действия программы, достаточно прочитать в указанной выше манере оставшуюся часть диаграммы. Данная диаграмма называется диаграммой последовательностей, поскольку она отражает последовательность выполняемых операций.

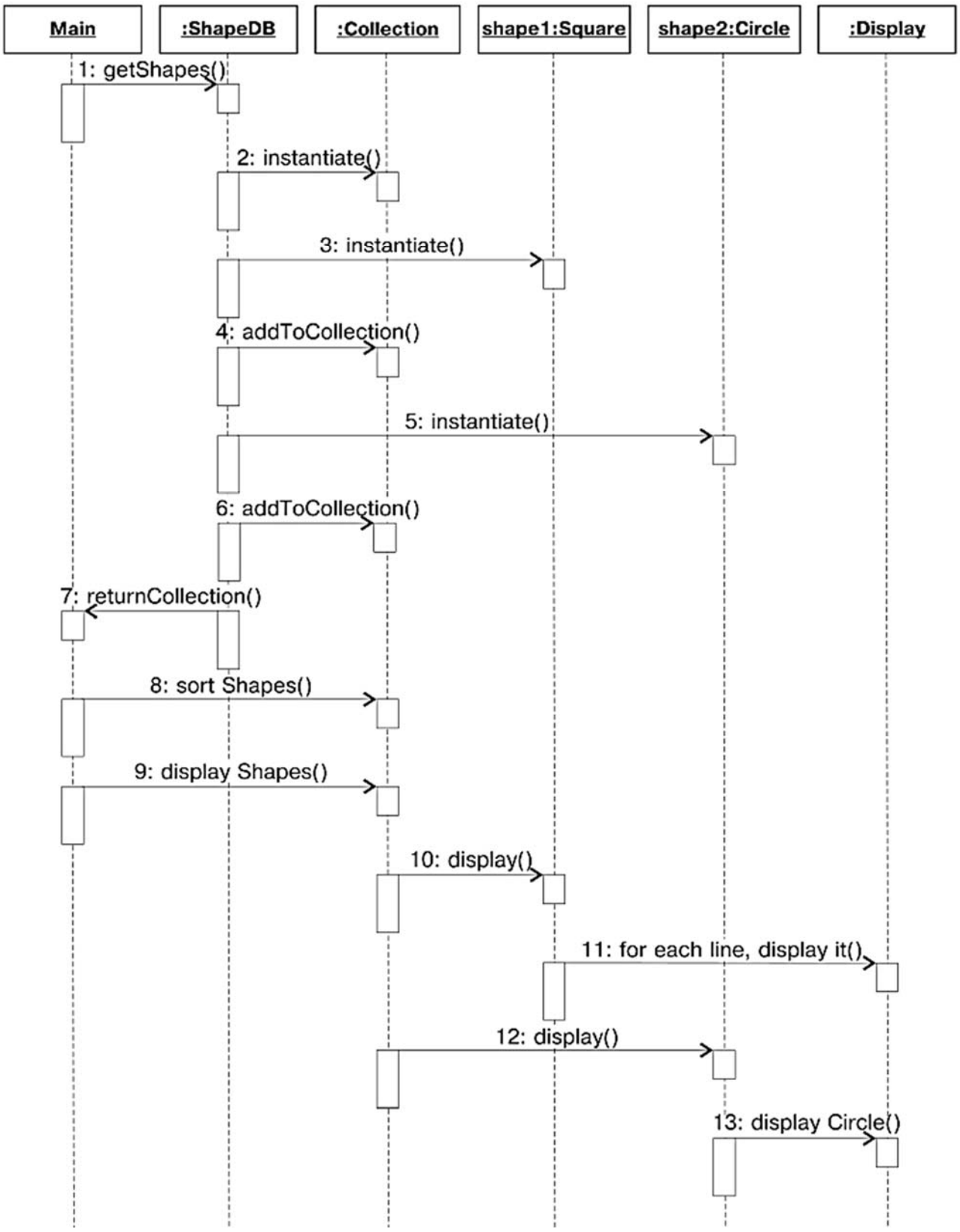


Рисунок 3.1 – Диаграмма последовательностей для программы обработки фигур

3.2 Индивидуальные задания

Задание

Нарисовать диаграмму взаимодействия объектов программной системы, реализованной при выполнении лабораторной работы № 1.

Контрольные вопросы

1 Для чего предназначена диаграмма взаимодействия?

2 Объясните последовательность построения диаграммы взаимодействия объектов программной системы.

3 На примере построенной диаграммы последовательностей расскажите порядок объектов программы.

4 Лабораторная работа № 3. Разработка программ с использованием принципа единственной обязанности (SRP) и принципа открытости/закрытости (OCP)

Цель работы: получить навыки разработки программ с использованием принципа единственной обязанности (SRP) и принципа открытости/закрытости (OCP).

4.1 Необходимые теоретические сведения

Принцип единственной обязанности

Принцип единственной обязанности обозначает, что каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

Автор этого принципа Р. С. Мартин [2] определяет ответственность как причину изменения и заключает, что классы должны иметь одну и только одну причину для изменений. Например, представим себе класс, который составляет и печатает отчёт. Такой класс может измениться по двум причинам:

1) может измениться само содержимое отчёта;
2) может измениться формат отчёта.

Логично, что оба аспекта этих причин на самом деле являются двумя разными ответственностями. SRP говорит, что в таком случае нужно разделить класс на два новых класса, для которых будет характерна только одна ответственность. Причина, почему нужно сохранять направленность классов на единственную цель, в том, что это делает классы более здоровыми. Что касается класса, приведённого выше, если произошло изменение в процессе составления отчёта – есть большая вероятность, что в негодность придёт код, отвечающий за печать. Более подробную информацию можно найти в [2, с. 154–158].

Принцип открытости/закрытости

Принцип открытости/закрытости устанавливает следующее положение: программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения.

Принцип открытости/закрытости означает, что программные сущности должны быть:

– открыты для расширения: означает, что поведение сущности может быть расширено путём создания новых типов сущностей;

– закрыты для изменения: в результате расширения поведения сущности не должны вноситься изменения в код, который эти сущности использует.

Это особенно значимо в производственной среде, когда изменения в исходном коде потребуют проведение пересмотра кода, модульного тестирования и других подобных процедур, чтобы получить право на использования его в программном продукте. Код, подчиняющийся данному принципу, не изменяется при расширении и поэтому не требует таких трудозатрат. Более подробную информацию можно найти в [2, с. 160–171].

4.2 Индивидуальные задания

Задание

Разработать программу решения задачи из лабораторной работы № 1 в виде консольного приложения с использованием принципа единственной обязанности (SRP) и принципа открытости/закрытости (OCP).

Контрольные вопросы

1 Объясните назначение принципов объектно-ориентированного проектирования SOLID.

2 Перечислите принципы, входящие в SOLID.

3 Сформулируйте принцип единственной обязанности. Объясните основную идею, лежащую в основе этого принципа, приведите пример использования.

4 Сформулируйте принцип открытости/закрытости. Объясняете основную идею, лежащую в основе этого принципа, приведите пример использования.

5 Лабораторная работа № 4. Разработка программ с использованием принципа подстановки Лисков (LSP), принципа разделения интерфейсов (ISP) и принципа инверсии (DIP)

Цель работы: получить навыки разработки программ с использованием принципа подстановки Лисков (LSP), принципа разделения интерфейсов (ISP) и принципа инверсии (DIP).

5.1 Необходимые теоретические сведения

Принцип подстановки Лисков

Принцип подстановки Лисков представляет собой некоторое руководство по созданию иерархий наследования. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 г., выглядело следующим образом. Если для каждого объекта `O1` типа `S` существует объект `O2` типа `T` такой, что для любой программы `P`, определенной в терминах `T`, поведение `P` не изменяется при замене `O2` на `O1`, то `S` является подтипом `T`.

То есть иными словами класс `S` может считаться подклассом `T`, если замена объектов `T` на объекты `S` не приведет к изменению работы программы.

В общем случае данный принцип можно сформулировать так: должна быть возможность вместо базового типа подставить любой его подтип.

Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма. Более подробную информацию можно найти в [2, с. 173–189].

Принцип разделения интерфейсов

Принцип разделения интерфейсов относится к тем случаям, когда классы имеют «жирный интерфейс», т. е. слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Таким образом, интерфейс получатся слишком избыточен или «жирным».

Принцип разделения интерфейсов можно сформулировать так: клиенты не должны вынужденно зависеть от методов, которыми не пользуются.

При нарушении этого принципа клиент, использующий некоторый интерфейс со всеми его методами, зависит от методов, которыми не пользуется, и поэтому оказывается восприимчив к изменениям в этих методах. В итоге приходим к жесткой зависимости между различными частями интерфейса, которые могут быть не связаны при его реализации.

В этом случае интерфейс класса разделяется на отдельные части, которые составляют раздельные интерфейсы. Затем эти интерфейсы независимо друг от друга могут применяться и изменяться. В итоге применение принципа разделения

интерфейсов делает систему слабосвязанной, и тем самым ее легче модифицировать и обновлять. Более подробную информацию можно найти в [2, с. 200–212].

Принцип инверсии зависимостей

Данный принцип гласит, что, во-первых, классы высокого уровня не должны зависеть от низкоуровневых классов. При этом оба должны зависеть от абстракций. Во-вторых, абстракции не должны зависеть от деталей, но детали должны зависеть от абстракций. Классы высокого уровня реализуют бизнес-правила или логику в системе (приложении). Низкоуровневые классы занимаются более подробными операциями, другими словами, они могут заниматься записью информации в базу данных или передачей сообщений в операционную систему или службы и т. п.

Говорят, что высокоуровневый класс, который имеет зависимость от классов низкого уровня или какого-либо другого класса и много знает о других классах, с которыми он взаимодействует, тесно связан. Когда класс явно знает о дизайне и реализации другого класса, возникает риск того, что изменения в одном классе нарушат другой класс.

Поэтому необходимо держать эти высокоуровневые и низкоуровневые классы слабо связанными, насколько можно. Чтобы сделать это, нужно сделать их зависимыми от абстракций, а не друг от друга.

Более подробную информацию можно найти в [2, с. 191–199].

5.2 Индивидуальные задания

Задание

Разработать программу решения задачи в соответствии с выданным вариантом в виде консольного приложения с использованием принципа подстановки Лисков (LSP), принципа разделения интерфейсов (ISP) и принципа инверсии (DIP).

Варианты заданий

1 Студент, преподаватель, заведующий кафедрой, персона.
2 Небоскрёб, дача, коттедж, жилое здание.
3 Организация, страховая компания, нефтегазовая компания, завод.
4 Журнал, книга, печатное издание, учебник.
5 Тест, экзамен, выпускной экзамен, испытание.
6 Строительное сооружение, театр, производственный корпус, гостиница.
7 Игрушка, телевизор, товар, молоко.
8 Квитанция, накладная, документ, счёт.
9 Автомобиль, поезд, самолёт, транспортное средство.
10 Республика, монархия, королевство, государство.
11 Корабль, пароход, парусник, корвет.

12 Двигатель, бензиновый двигатель, дизельный двигатель, реактивный двигатель.

13 Деталь, узел, механизм, изделие.

14 Млекопитающее, парнокопытное, птица, животное.

15 Выпускник вуза, Бакалавр, Магистр, Инженер.

Контрольные вопросы

1 Сформулируйте принцип подстановки Лисков. Объясняете основную идею, лежащую в основе этого принципа, приведите пример использования.

2 Сформулируйте принцип разделения интерфейсов. Объясняете основную идею, лежащую в основе этого принципа, приведите пример использования.

3 Сформулируйте принцип инверсии. Объясняете основную идею, лежащую в основе этого принципа, приведите пример использования.

6 Лабораторная работа № 5. 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.

6.1 Необходимые теоретические сведения

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 C# 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 6.1).

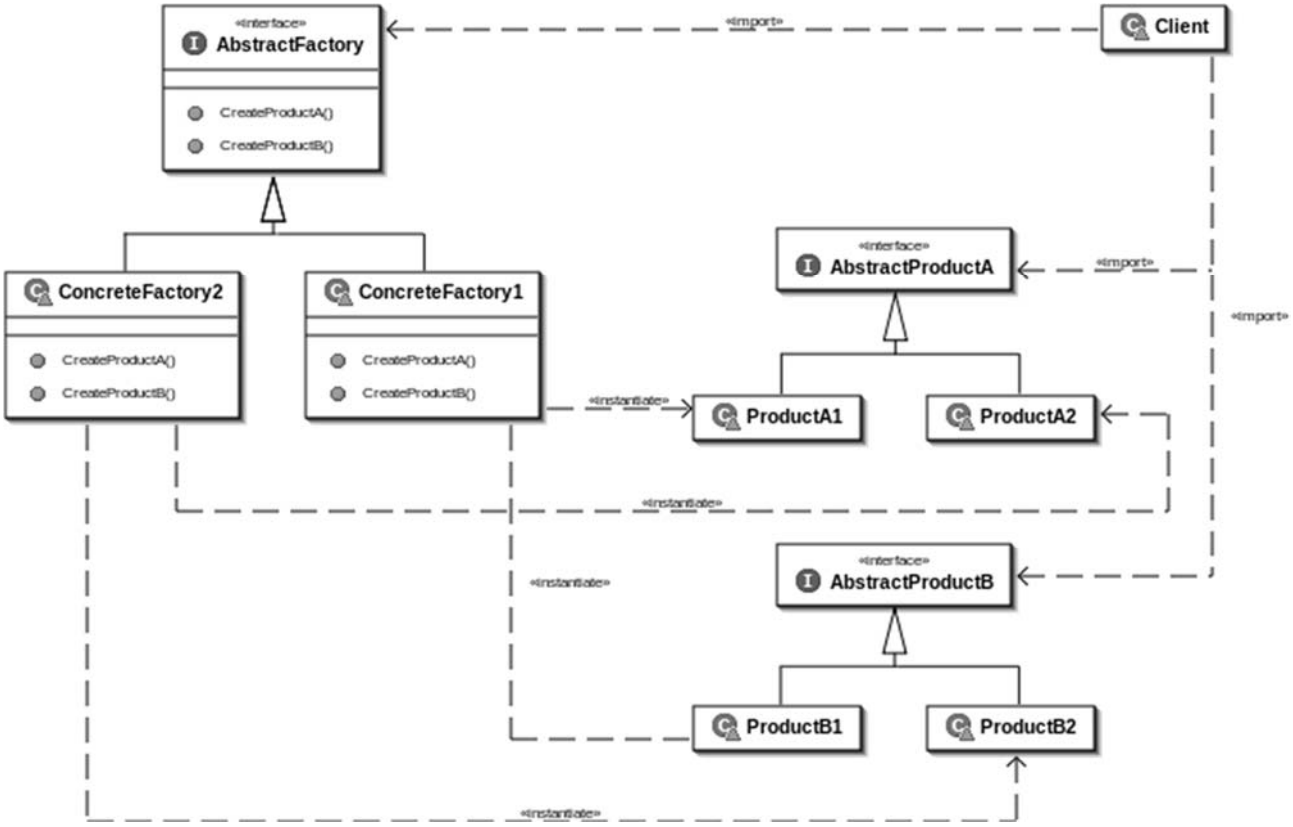


Рисунок 6.1 – UML-диаграмма шаблона Абстрактный продукт

Далее 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()
    { }
}
```

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

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.

Участники шаблона фабричный метод (рисунок 6.2):

– `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`.

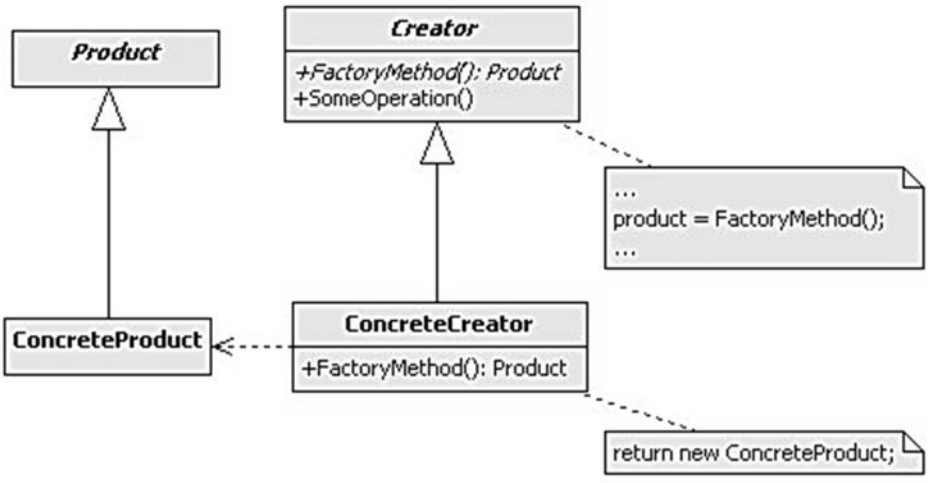


Рисунок 6.2 – UML-диаграмма шаблона Фабричный метод

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(); }
}
```

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 6.3.

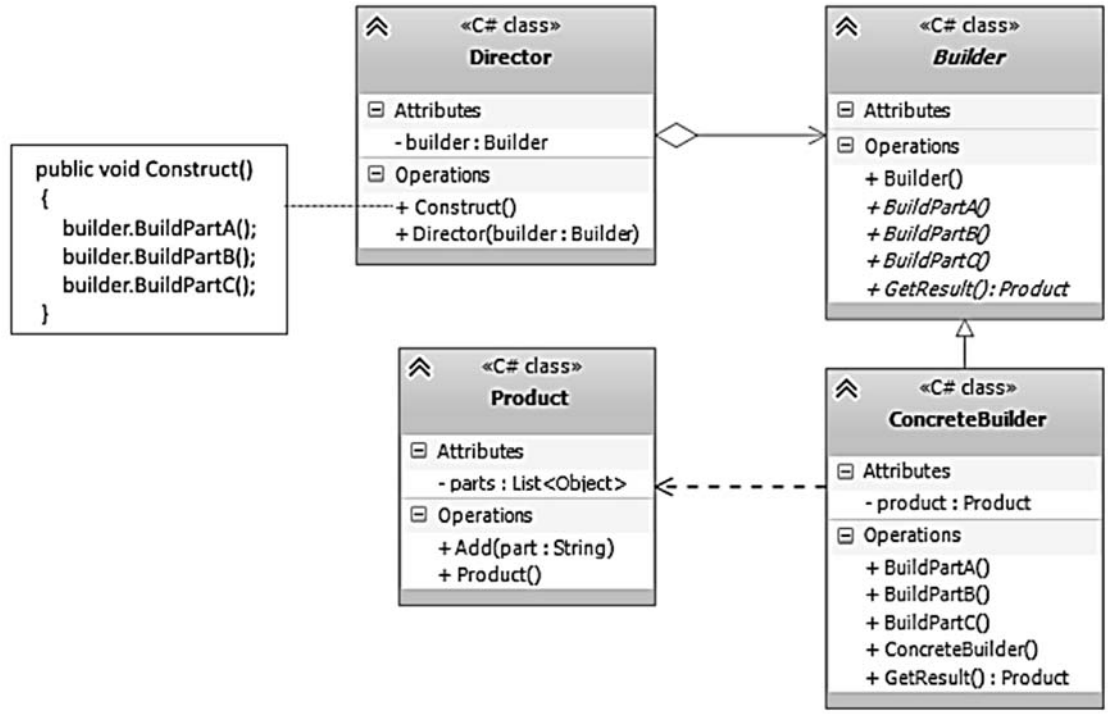


Рисунок 6.3 – UML-диаграмма шаблона Строитель

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`.

6.2 Индивидуальные задания

Задание

Для указанного варианта задания необходимо нарисовать UML-диаграмму классов реализуемой программы и разработать консольное приложение на языке C#.

1 Паттерн `Builder`. Имеется текст статьи в формате TXT. Статья состоит из заголовка, фамилий авторов, самого текста статьи и хеш-кода текста статьи. Написать приложение, позволяющее конвертировать документ в формате TXT в документ формата XML. Необходимо также проверять корректность хеш-кода статьи.

2 Паттерн `Abstract Factory`. Разработать систему Кинопрокат. Пользователь может выбрать определённую киноленту, при заказе киноленты указывается язык звуковой дорожки, который совпадает с языком файла субтитров. Система должна поставлять фильм с требуемыми характеристиками, причём при смене языка звуковой дорожки должен меняться и язык файла субтитров и наоборот.

3 Паттерн `Factory Method`. Фигуры игры «тетрис». Реализовать процесс случайного выбора фигуры из конечного набора фигур. Предусмотреть появление супер-фигур с большим числом клеток, чем обычные.

Контрольные вопросы

1 Каково назначение шаблона Строитель?
2 Какая проблема решается с помощью шаблона Строитель?
3 В чём заключается решение, предлагаемое в шаблоне Строитель?
4 Для чего предназначен шаблон Абстрактная фабрика?
5 Какое решение предлагается в шаблоне Абстрактная фабрика?
6 Какую проблему позволяет решить шаблон Фабричный метод?

7 Лабораторная работа № 6. Разработка программ с использованием структурных паттернов

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

7.1 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:

1) клиент обращается с запросом к объекту-адаптеру `Adapte`r, вызывая его метод `Request()` через целевой интерфейс `ITarget`.

2) адаптер `Adapter` преобразует запрос в один или несколько вызовов к адаптируемому объекту `Adapted`.

3) 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 7.1);

– адаптер класса использует для адаптации наследование адаптируемому классу (рисунок 7.2).

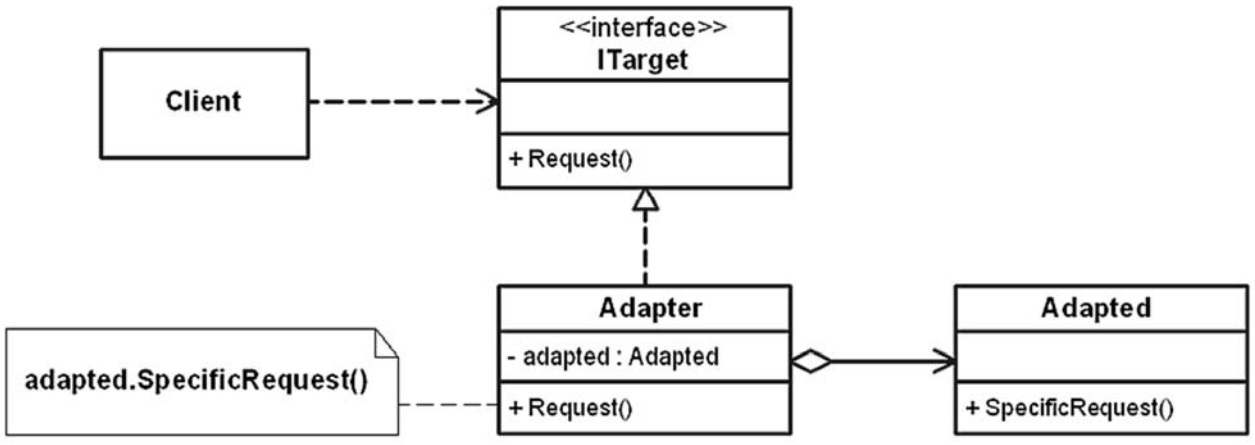


Рисунок 7.1 – Диаграмма классов шаблона Адаптер объекта

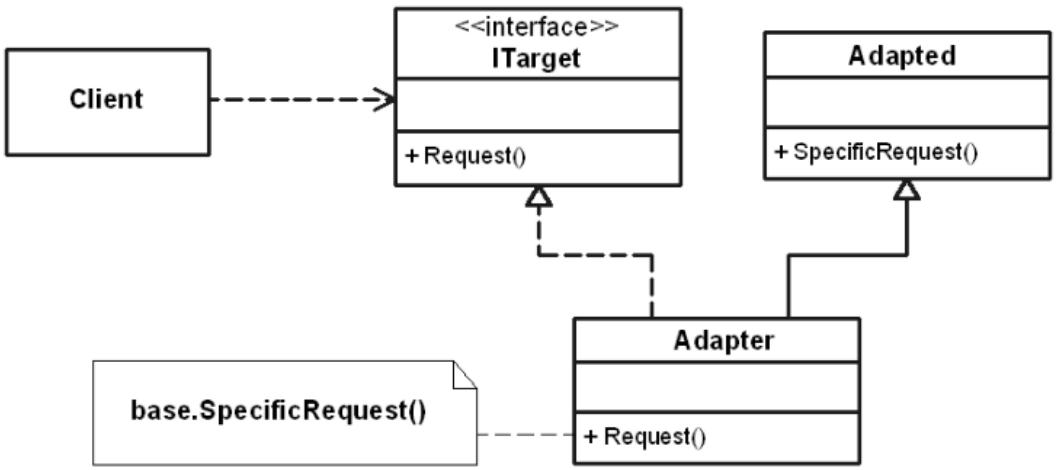


Рисунок 7.2 – Диаграмма классов шаблона Адаптер класса

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 `Adap*ter` 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 шаблона Адаптер на языке C#

Простейший пример реализации шаблона Адаптер объекта на языке C# представлен в листинге 7.1.

Листинг 7.1 – Пример реализации шаблона Адаптер объекта на языке C#

```
// 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 (листинг 7.2).

Листинг 7.2 – Исходный код класса Adapter для шаблона Адаптер класса

```
// 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 7.3).

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, т. е. 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.

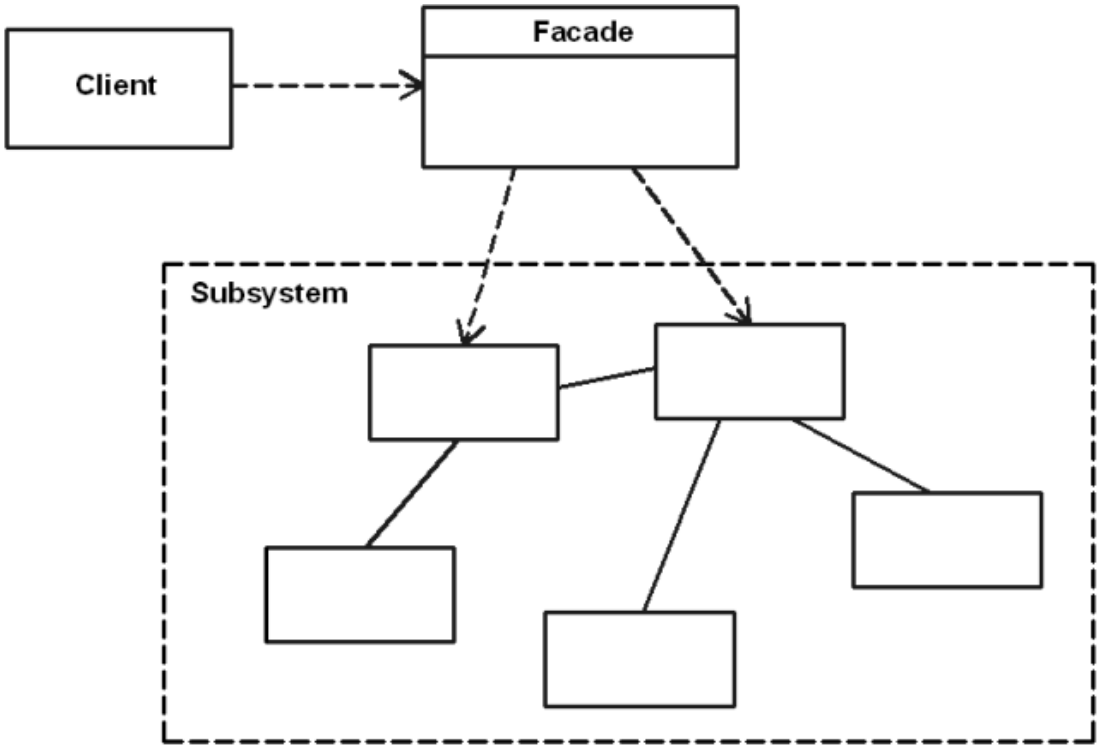


Рисунок 7.3 – Диаграмма классов шаблона Фасад

Простой пример реализации шаблона Фасад на языке C# представлен в листинге 7.3.

Листинг 7.3 – Пример реализации шаблона Фасад на языке C#

```
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();
      }
}
```

7.2 Индивидуальные задания

Задание

1 Изучить пример проектирования программной системы с использованием паттерна Адаптер [3, с. 117–123].

2 Изучить пример проектирования программной системы с использованием паттерна Фасад [3, с. 123–133].

3 Разработать библиотеку классов, которая содержит указанные классы (таблица 7.1), задействованные в шаблоне Адаптер. Для адаптера объектов атрибуты класса должны быть реализованы как автоматические свойства, а для шаблона Адаптер классов – как защищённые поля.

Таблица 7.1 – Варианты заданий для разработки приложения с использованием шаблона Адаптер

Номер варианта	Тип адаптера	Требуемый интерфейс	Адаптируемый класс
1	Объектов	`+CalculateDp(T0:int,dT:int):double` – определить изменение давления при заданной начальной температуре `T0` и изменении температуры `dT`; `+ModifMass(dm:double):void` – изменить массу газа в баллоне на величину `dm`; `+GetData():string` – возвращает строку с данными об объекте	Баллон с газом. *Атрибуты*: `Volume: double` – объём баллона, м^3; `Mass: double` – масса газа, кг; `Molar: double` – молярная масса газа, кг/моль. *Операции*: `+GetPressure(T : int) :double` – определить давление в баллоне при заданной температуре газа `T`; `+AmountOfMatter(): double` – определить количество вещества; `+ToString():string` – возвращает строку с данными об объекте

Окончание таблицы 7.1

Номер варианта	Тип адаптера	Требуемый интерфейс	Адаптируемый класс
2	Классов	`+ModifVolume(dV:double):void` – изменить объём баллона на величину `dV`; `+GetDp(T0:int, T1:int): double` – определить изменение давления при изменении температуры с `T0` до `T1`; `+Passport():string` – возвращает строку с данными об объекте	Баллон с газом. *Атрибуты*: `Volume: double` – объём баллона, м^3; `Mass: double` – масса газа, кг; `Molar: double` – молярная масса газа, кг/моль. *Операции*: `+GetPressure(T : int) :double` – определить давление в баллоне при заданной температуре газа `T`; `+AmountOfMatter(): double` – определить количество вещества; `+ToString():string` – возвращает строку с данными об объекте

4 Разработать библиотеку классов, которая должна содержать класс-фасад и заданный набор классов (таблица 7.2). В фасаде необходимо задать ссылки на другие классы библиотеки.

5 Добавить в решение консольное приложение, которое для реализованных шаблонов играет роль клиента. Продемонстрировать в консольном приложении работу шаблонов проектирования.

Таблица 7.2 – Варианты заданий для разработки приложения с использованием шаблона Фасад

Номер варианта	Данные для разработки приложения на основе шаблона Фасад
1	*Расчёт страхового взноса за недвижимость*. Классы (типы недвижимости): квартира, таун-хаус, коттедж. Параметры: срок страхования, жилплощадь (м^2), число проживающих, год постройки здания, износ здания (%)
2	*Расчёт ежедневной нормы потребления килокалорий*. Классы (Тип телосложения): Астеник, Нормостеник, Гиперстеник. Параметры: Рост, Вес, Возраст, Пол, Группа физической активности (низкая, средняя и высокая активность)
3	*Расчёт стоимости туристической путевки*. Классы (виды путевок): пляжный отдых, экскурсия, горные лыжи. Параметры: длительность, страна, гостиница (число звезд), рацион питания (двухразовый, трехразовый, всё включено)

Контрольные вопросы

1 Какие шаблоны проектирования называют структурными?

2 Каково назначение структурного шаблона Адаптер?

3 Каким образом осуществляется взаимодействие клиента с адаптируемым классом в шаблоне Адаптер?

4 В чём заключается различие между Адаптером объектов и Адаптером классов?

5 Какая проблема решается с помощью шаблона Фасад?

6 В чём заключается решение, предлагаемое в шаблоне Фасад?

7 К каким последствиям приводит использование шаблона Фасад?

8 Лабораторная работа № 7. Разработка программ с использованием паттернов поведения

Цель работы: научиться разрабатывать программы с использованием таких таких поведенческих шаблонов проектирования, как Стратегия, Состояние и Шаблонный метод.

8.1 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, событий 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 ● и ◉;

– 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 8.1.

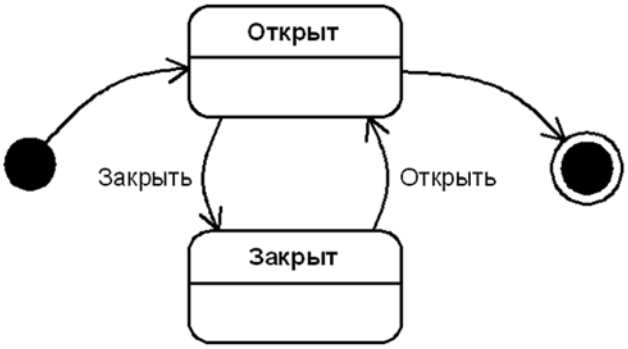


Рисунок 8.1 – Простая диаграмма конечных автоматов UML

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 8.2.

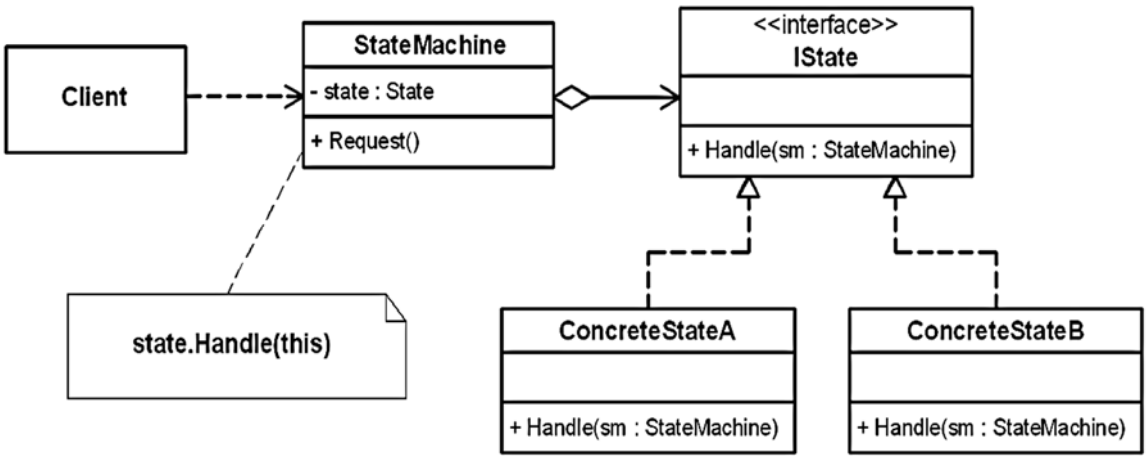


Рисунок 8.2 – Диаграмма классов шаблона Состояние

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 на языке C# приведён в листинге 8.1.

Листинг 8.1 – Пример реализации шаблона Состояние на языке C#

```
// Представляет состояния
interface IState
{
      void Handle(StateMachine sm);
}
// Представляет конкретные состояния A
class ConcreteStateA : IState
{
      public void Handle(StateMachine sm)
      {
          Console.Write("Объект    {0}    в    сост.    A",
          sm.ToString());
      }
}
// Представляет конкретные состояния B
class ConcreteStateB : IState
{
      public void Handle(StateMachine sm)
      {
          Console.Write("Объект    {0}    в    сост.    B",
          sm.ToString());
      }
}
// 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 8.3.

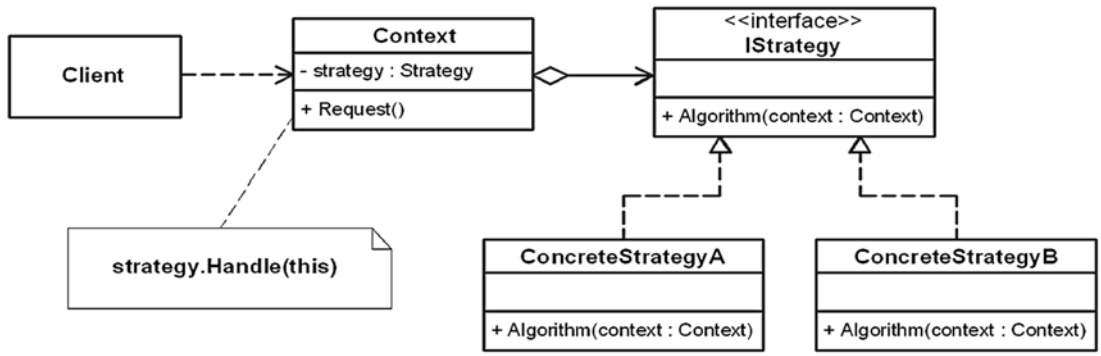


Рисунок 8.3 – Диаграмма классов шаблона Стратегия

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.

Реализация.

Простой пример реализации шаблона Стратегия на языке C# приведён в листинге 8.2.

Листинг 8.2 – Пример реализации шаблона Стратегия на языке C#

```
// 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 8.4.

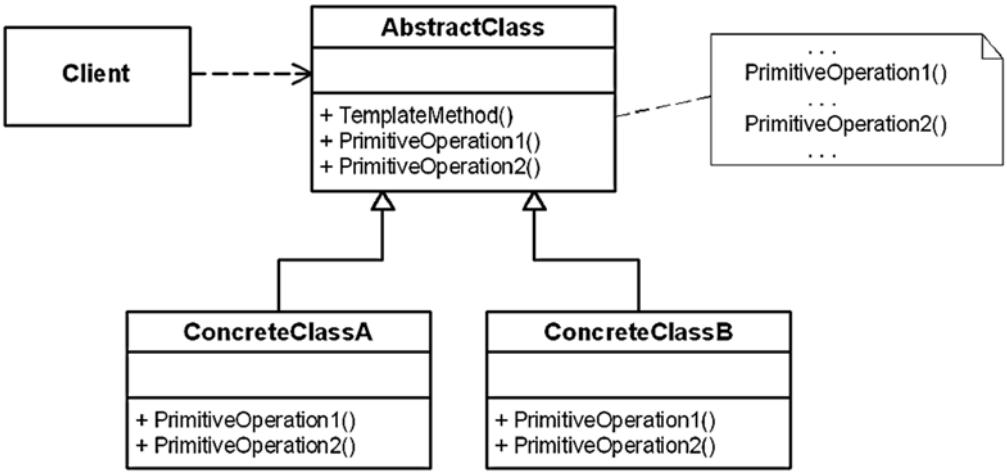


Рисунок 8.4 – Диаграмма классов шаблона Шаблонный метод

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 в листинге 8.3.

Листинг 8.3 – Пример реализации шаблона Шаблонный метод на языке C#

```
// 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));
        }
}
```

8.2 Индивидуальные задания

Задание

1 Изучить пример проектирования программной системы с использованием паттерна Состояние [3, с. 143–154].

2 Разработать диаграмму конечных автоматов для заданного класса (таблица 8.1). Описать в форме таблицы варианты реакции экземпляра класса на операции, вызываемые в указанных состояниях.

3 Разработать библиотеку классов, включающую необходимые классы для реализации шаблона Состояние (класс Конечный автомат, интерфейс Состояние, классы Конкретные состояния).

4 Разработать приложение Windows Forms для управления состояниями экземпляров класса Конечный автомат.

Таблица 8.1 – Варианты заданий для разработки приложения с использованием шаблона Состояние

Номер варианта	Классы, их атрибуты и операции	Состояния
1	*Телефон*. *Атрибуты*: номер, баланс, вероятность поступления звонка. *Операции*: позвонить, ответить на звонок, завершить разговор, пополнить баланс	Ожидание, Звонок, Разговор, Заблокирован (баланс отрицательный)
2	*Банкомат*. *Атрибуты*: ID, общая сумма денег в банкомате, вероятность отсутствия связи с банком. *Операции*: ввести PIN-код, внять заданную сумму, завершить работу, загрузить деньги в банкомат	Ожидание, Аутентификация пользователя, Выполнение операций, Заблокирован (денег нет)
3	*Грузовой лифт*. *Атрибуты*: текущий этаж, грузоподъёмность, вероятность отключения электроэнергии. *Операции*: вызвать на заданный этаж, загрузить, разгрузить, восстановить подачу энергии	Покой, Движение, Перегружен, Нет питания

Контрольные вопросы

1 Каково назначение поведенческих шаблонов проектирования?

2 Какие основные элементы используются на диаграммах конечных автоматов UML?

3 Для чего предназначен поведенческий шаблон Состояние?

4 Какое решение предлагается в шаблоне Состояние?

5 Что понимают под стратегией в шаблоне Стратегия?

6 Какое решение предлагается в шаблоне Стратегия?

7 Какую проблему позволяет решить шаблон Шаблонный метод?

9 Лабораторная работа № 8. Разработка программ с использованием шаблонов GRASP

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

9.1 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 небольшого 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.

9.2 Индивидуальные задания

Задание

1 Изучить пример проектирования программной системы «Железнодорожные перевозки» [3, с. 88–109].

2 На основе заданных классов предметной области, а также их обязанностей (таблица 9.1) разработать диаграмму классов UML, в которой применены такие шаблоны GRASP, как `Information Expert` и `Creator`.

3 Создать проект библиотеки классов, в которой должны присутствовать модули классов, отмеченных на диаграмме классов. Реализовать атрибуты и операции классов, а также отношения между ними с помощью средств языка C#.

4 Добавить в решение проект консольного приложения и связать его с полученной библиотекой классов. Продемонстрировать в консольном приложении выполнение классами требуемых обязанностей.

5 Дополнительно добавить в решение проект приложения Windows Forms. Реализовать в приложении представление данных об объектах и возможность добавления новых объектов.

6 Обеспечить возможность сохранения данных путём сериализации объектов в XML-документ.

Таблица 9.1 – Индивидуальные задания

Номер варианта	Класс	Обязанность
1	Гостиница. Гостиничный номер. Клиент	*Знать*: стоимость одного дня проживания; число дней проживания. *Делать*: добавить клиента в номер; удалить клиента из номера; определить общее число клиентов; определить суммарную плату за проживание
2	Цех. Станок. Рабочий (может обслуживать несколько станков)	*Знать*: какие станки, обслуживаются рабочим; надбавка за обслуживание более чем одного станка. *Делать*: добавить рабочего в цех; добавить рабочему станок; определить зарплату (зависит от числа обслуживаемых станков); определить суммарные расходы на зарплату

Окончание таблицы 9.1

Номер варианта	Классы	Обязанности
3	Магазин. Товар. Продавец-консультант	*Знать*: какие товары, проданы сотрудником; бонус за продажу единицы товара. *Делать*: добавить товар в магазин; продать товар; определить зарплату (зависит от числа проданных товаров); определить общую выручку

Контрольные вопросы

1 В чём заключается подход проектирования на основе обязанностей (RDD)?

2 Какие выделяют виды обязанностей объекта?

3 Какие выделяют основные шаблоны GRASP?

4 Какую проблему решает шаблон Expert и в чём заключается это решение?

5 Какие классы должны отвечать за создание объектов согласно шаблону Creator?

6 К каким проблемам приводит использование в системе классов с высокой степенью связывания?

7 Что понимают под функциональным сцеплением в объектно-ориентированным проектировании?

8 Чем плохо использование классов с низкой степенью сцепления?

Список литературы

1 **Шаллоуей, А.** Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию / А. Шаллоуей. – Москва: Вильямс, 2002. – 288 с.

2 **Мартин, Р.** Принципы, паттерны и методики гибкой разработки на языке C#: пер. с англ. / Р. Мартин, М. Мартин. – Санкт-Петербург: Символ-Плюс, 2011. – 768 с.

3 Архитектура информационных систем: лабораторный практикум для студентов направления 09.03.02 «Информационные системы и технологии» очной формы обучения / Сост. Д. Е. Турчин. – Кемерово: КузГТУ, 2015. – 246 с.

4 Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. – Санкт-Петербург: Питер, 2016. – 366 с.