МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО
ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
И ПРОГРАММНОЙ ИНЖЕНЕРИИ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

Методические указания к выполнению лабораторных работ

Санкт-Петербург 2020

Составители: Павлов Е.В., Пятлина Е.О.

Рецензент: проф. Шейнин Ю.Е., ГУАП.

Объектно-ориентированное проектирование информационных систем. Методические указания к выполнению лабораторных работ – СПб ГУАП, 2020 – 29 с.

В методические указания включены краткие теоретические сведения, необходимые для выполнения лабораторных работ, требования к содержанию отчетов и порядку выполнения работ, примеры выполнения.
Методические указания предназначены для выполнения лабораторных работ по дисциплине «Объектно-ориентированное проектирование информационных систем» студентами различных форм обучения по направлению 09.03.01 «Информатика и вычислительная техника».

Подготовлены кафедрой компьютерных технологий и программной инженерии.

СОДЕРЖАНИЕ

1. ЛАБОРАТОРНАЯ РАБОТА №1 «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ. РАЗРАБОТКА ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ» 3

2. ЛАБОРАТОРНАЯ РАБОТА №2 «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ. РАЗРАБОТКА ДИАГРАММЫ КЛАССОВ» 10

3. ЛАБОРАТОРНЫЕ РАБОТЫ №3 И №4 «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ. РАЗРАБОТКА ДИАГРАММ ПОСЛЕДОВАТЕЛЬНОСТИ И КОММУНИКАЦИИ» 20

4. ЛАБОРАТОРНАЯ РАБОТА №5 . ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ. РАЗРАБОТКА ДИАГРАММЫ СОСТОЯНИЙ» 25

5. ЛАБОРАТОРНЫЕ РАБОТЫА №6 « ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ.И СИСТЕМЫ. РАЗРАБОТКА ДИАГРАММЫ РАЗВЕРТЫВАНИЯ»......................26

ЛАБОРАТОРНАЯ РАБОТА №1 .
«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ**. РАЗРАБОТКА ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ»**

1.1 Цель работы

Целью данной работы является изучение способов анализа функциональных требований к информационной системе и их представление в виде диаграммы вариантов использования.

1.2 Задание на лабораторную работу

Разработать диаграмму вариантов использования (прецедентов) для информационной системы или её функционально законченной части в соответствии с вариантом задания.
На диаграмме прецедентов должны быть использованы все типы отношений (*ассоциация, обобщение, включение и расширение*).

1.3 Порядок выполнения работы

1. Получить вариант задания (тему) у преподавателя;
2. Изучить теоретический материал, изложенный в подразделе 1.4;
3. Разработать диаграмму вариантов использования (прецедентов);
4. Ознакомиться с требованиями по содержанию отчета в подразделе 1.5;
5. Написать отчет о работе.

Для выполнения ЛР можно воспользоваться любым CASE-средством или средой моделирования, которые поддерживают спецификацию UML 2.0. В прошлом году я рекомендовал ребятам Umbrello UML Modeller, так как по списку ниже оно отвечало всем необходимым требованиям, но как выяснилось позже, моя рекомендация была, мягко говоря, не очень: убогий интерфейс и неудобство работы с некоторыми видами диаграмм (но для выполнения ЛР годиться). К сожалению, у меня нет сейчас времени посмотреть подходящее CASE-средство, поэтому либо воспользуйтесь Umbrello UML Modeller, либо самостоятельно выберете подходящее для себя CASE-средство.
Ознакомиться с полным списком средств моделирования для UML можно по ссылке
https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools
Только имейте в виду, что в ЛР №3 вам потребуется генерация кода и обратное проектирование, не все CASE-средства поддерживают данные функции.

1.4 Теоретический материал

Диаграмма вариантов использования (прецедентов) представляет собой диаграмму, на которой отражены отношения, существующие между актерами и вариантами использования.
Основная задача — представить единое средство, дающее возможность заказчику, конечному пользователю и разработчику совместно обсуждать функциональность и поведение системы.

Вариант использования — это описание множества последовательных действий, включая их варианты, выполняемых системой с целью получения значимого результата для действующего лица. Вариант использования описывает типичное взаимодействие между пользователем и системой. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать в данной информационной системе.
Сценарий — это конкретная последовательность действий, иллюстрирующих поведение. Сценарии по отношению к вариантам использования — то же самое, что экземпляры по отношению к классам, поскольку сценарий — это в основном один экземпляр варианта использования.

Рисунок 1.4.1 — Пример вариантов использования

Вариант использования описывает, что делает система (подсистема, класс, или интерфейс), но не указывает, как она это делает.
Можно специфицировать поведение варианта использования, описав поток событий в текстовой форме, понятной постороннему читателю, диаграмм последовательности (ЛР №4) или в виде конечного автомата (ЛР №5), либо с помощью псевдокода. В описании должны присутствовать указание на то, как и когда вариант использования начинается и заканчивается, когда он взаимодействует с действующими лицами, какими объектами они обмениваются, а также упоминание основного и альтернативного потоков поведения.
Желательно отделять основной поток от альтернативных, поскольку вариант использования описывает не одну, а множество последовательностей, и выразить все детали интересующего вас варианта использования в виде одной последовательности невозможно. Например, в системе управления человеческими ресурсами присутствует вариант использования Hire employee (Нанять работника). Существует множество разновидностей этой основной бизнессфункции. Вы можете переманить работника из другой компании (наиболее общий сценарий), перевести сотрудника из одного подразделения в другое (что часто случается в транснациональных компаниях) или нанять иностранца (особый случай, регулируемый специальными правилами). Каждый из этих вариантов описывается своей последовательностью.

Действующее лицо (актер) представляет собой связанное множество ролей, которые исполняют пользователи вариантов использования во время взаимодействия с ними. Обычно действующее лицо представляет ту роль, которую в данной системе играет человек, аппаратное устройство или даже другая система. Действующие лица не являются частью системы, так как существуют вне ее.
Можно использовать механизм расширения UML, приписав действующему лицу стереотип, чтобы создать другую пиктограмму, больше подходящую для достижения целей проектирования.

Рисунок 1.4.2 — Пример актеров, связанных отношением обобщения

Действующие лица можно связывать с вариантами использования только при помощи ассоциаций. *Ассоциация* между действующим лицом и вариантом использования показывает, что они общаются друг с другом, возможно, посылая или принимая сообщения. На диаграмме изображается в виде сплошной линии. Если ассоциация направленная, то она показывает направление передачи сообщения.

Обобщения между вариантами использования подобны обобщениям между классами. Это означает, что дочерний вариант использования наследует поведение и суть родительского варианта использования; потомок может добавить или переопределить поведение родителя, а кроме того, быть подставленным вместо него в любом месте, где тот появляется (как родительский, так и дочерний вариант использования могут иметь конкретные экземпляры). Обобщение изображается в виде сплошной линии с треугольником на стороне родительского варианта использования.

Связь *включения* между вариантами использования означает, что базовый вариант использования в определенном месте явно включает в себя поведение некоторого другого. Включенный вариант использования не существует отдельно: он является экземпляром только внутри базового, который его содержит. Можно считать, что базовый вариант использования заимствует поведение включаемого.
Благодаря наличию связей включения удается избежать многократного описания одного и того же потока событий, поскольку общее поведение можно представить в виде отдельного варианта использования, включаемого в базовые. Связь включения являет собой пример делегирования, когда ряд обязанностей системы описывается в одном месте (во включаемом варианте использования), а остальные варианты использования при необходимости вводят эти обязанности в свой набор. Графически включение представляется пунктирной линией со стереотипом «include» и указывает на включаемый вариант использования.

Связь *расширения* между вариантами использования означает, что базовый неявно включает поведение некоторого другого в косвенно указанном месте. Базовый вариант использования способен существовать отдельно, но при некоторых условиях его поведение может быть расширено поведением другого варианта использования. Базовый вариант использования можно расширить только вызовом из определенной точки, – так называемой точки расширения (extension point). При выполнении работы вы можете не указывать точки расширения.

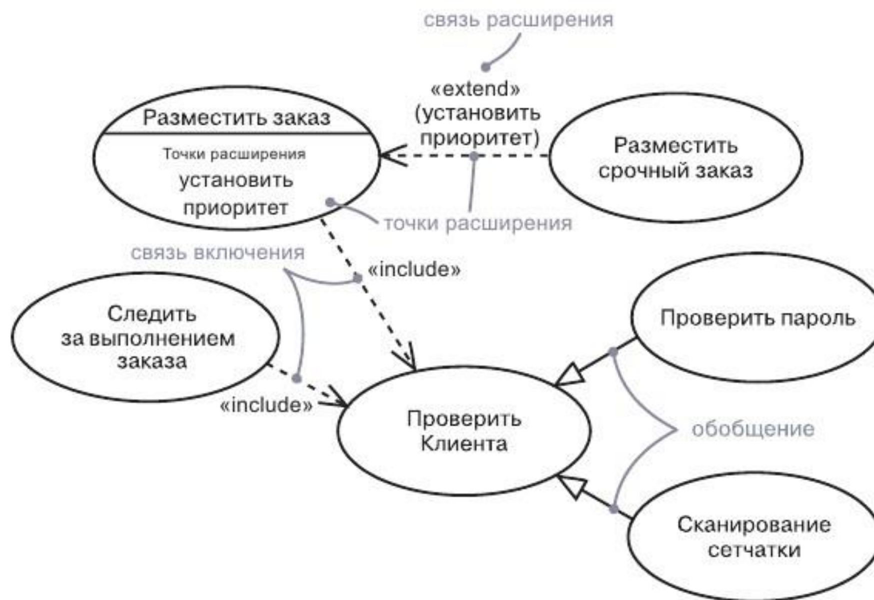


Рисунок 1.4.3 — Пример отношений между прецедентами

Связь расширения используется при моделировании тех частей вариантов использования, которые пользователь видит как необязательные. Таким образом обязательное поведение отделяется от необязательного. Также связь расширения применяется, чтобы выделить настраиваемые части реализуемой системы; следовательно, система может существовать как с различными расширениями, так и без них. Графически включение представляется пунктирной линией со стереотипом «extend» и указывает на расширяемый вариант использования.

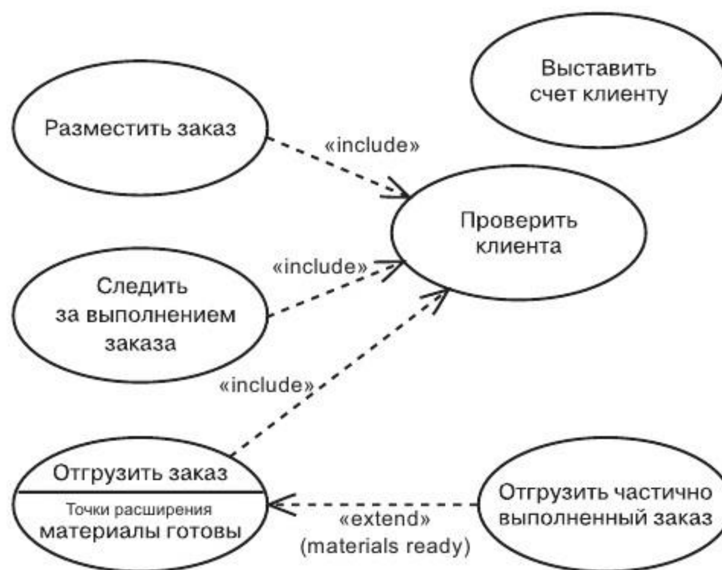


Рисунок 1.4.4 — Пример отношений между прецедентами

Организация вариантов использования, предусматривающая извлечение общего поведения (через связь включения) и разделение вариаций (через связь расширения).

Диаграммы вариантов использования применяются главным образом при бизнес- анализе для моделирования видов работ, выполняемых организацией, и для моделирования функциональных требований к ПО при его проектировании и разработке.

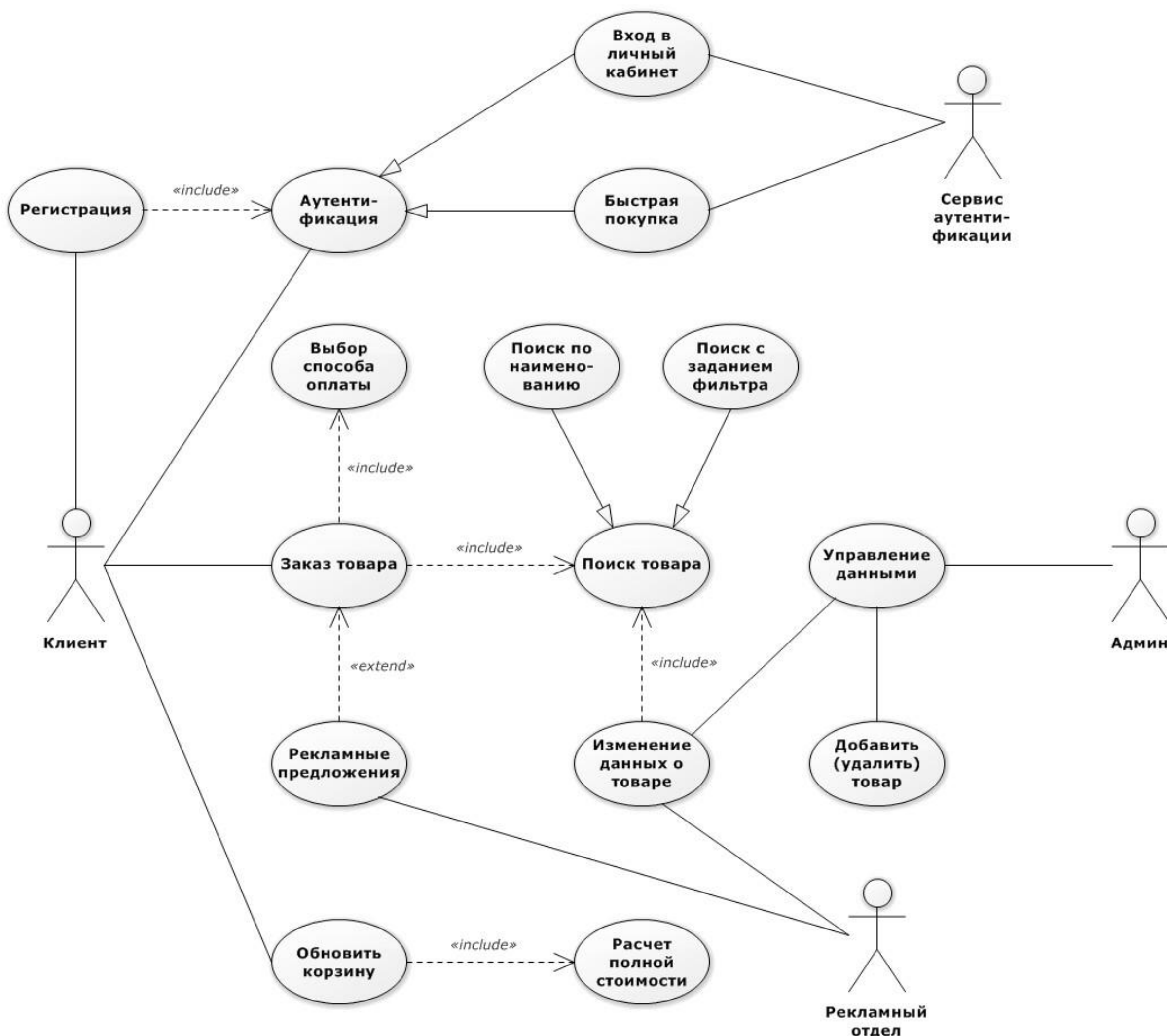


Рис. 1.4.5 — Пример диаграммы прецедентов для ИС «Интернет-магазин»

Обратите внимание, что инициация всего функционала, связанного с аутентификацией пользователя (как при заходе в личный кабинет, так и при осуществлении быстрой покупки) закреплена за отельным актером (сервисом аутентификации). На само деле зря этот грешный сервис аутентификации включил в диаграмму, так как он теперь в каждом отчете будет гостить.

1.5 Содержание отчета

Отчет о работе должен содержать:

1. Титульный лист
2. Цель работы
3. Задание на лабораторную работу и вариант
4. Диаграмма вариантов использования
5. Выводы по работе
6. Список использованных источников

Пример вывода:

В результате выполнения данной лабораторной работы были изучены способы анализа функциональных требований и поведения информационной системы на основе диаграммы вариантов использования.

Разработанная диаграмма охватывает основные функции системы «Интернет магазин» с точки зрения пользователя (или клиента) и менеджера (администратора сайта).

При осуществлении заказа пользователь может получать рекламные предложения, данная функция настраивается под различные виды товаров и может быть отключена администратором, поэтому в данном случае имеет место отношение «extend».

Для разработки данной диаграммы использовалось бесплатное для некоммерческого использование CASE-средство Software Ideas Modeler.

2. **ЛАБОРАТОРНАЯ РАБОТА №2.**
«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ. **РАЗРАБОТКА ДИАГРАММЫ КЛАССОВ»**

2.1 Цель работы

Целью данной работы является изучение способов построения модели предметной области информационной системы и разработка диаграммы классов.

2.2 Задание на лабораторную работу

Разработать диаграмму классов для информационной системы или ее функционально законченной части в соответствии с вариантом задания.
На диаграмме классов должны быть использованы минимум *три вида отношений*, также в явном виде должны быть указаны кратность ассоциаций, уровни доступа к атрибутам и методам классов (public, private, protected). Список атрибутов и методов должен описывать назначение каждого отдельного класса. В противном случае необходимо ввести в диаграмму комментарии для описания роли, которую конкретный класс выполняет в разрабатываемой системе.

2.3 Порядок выполнения работы

1. Изучить теоретический материал, изложенный в подразделе 2.4;
2. Разработать диаграмму классов;
3. Ознакомиться с требованиями по содержанию отчета в подразделе 2.5;
4. Написать отчет о работе.

При выполнении данной ЛР убедитесь, что выбранная вами среда моделирования или CASE-средство поддерживает генерацию кода и реверс-инжиниринг для диаграммы классов[1] (ЛР №3).

2.4 Теоретический материал

Поскольку объектно-ориентированное программирование у вас в этом же семестре, то соответственно такие понятия, как класс и объект для многих из вас могут быть незнакомы, поэтому напишу пару слов об ООП...
Фундаментальным понятием в объектно-ориентированном подходе является понятие абстракции. Процесс абстрагирования подразумевает выделение наиболее значимых признаков, исключая из рассмотрения не значимые. Собственно *абстракция*, это набор наиболее существенных признаков.

Как правило, принято разделять структурный подход и объектно-ориентированный. Основой структурного подхода в программировании является идея разбиения (декомпозиция) программы на подпрограммы, которые чаще всего называют функциями. В основе же объектно-ориентированного подхода лежит идея разбиения программы на объекты, каждый из которых является экземпляром определенного класса, а классы в свою очередь образуют иерархию с наследованием свойств.

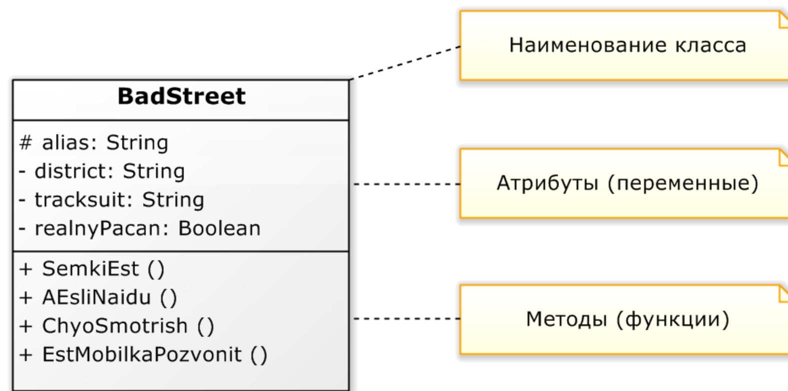
Ключевыми понятия в объектно-ориентированном подходе являются понятия *класса* и *объекта*. Думаю, вы знаете, что такое тип данных — это множество всех допустимых значений, которые могут принимать данные этого типа. Например, Integer для целых чисел или Float для чисел с плавающей запятой. Класс по сути является таким же типом данных, точнее класс — это *абстрактный тип данных*.

Допустим, вы разрабатываете приложение, например, игру, в которой студент *учится* в унивеситете. Вам было бы гораздо удобнее, если бы вы могли создать переменные, представляющие собой аудитории, преподователей, студентов, дисциплины, etc. И чем ближе эти переменные соответствуют реальности, тем легче написать соответствующую программу. Соответственно у вас студент или аудитория будут представлены отдельными классами (то есть абстрактными типами данных), а объектами или экземплярами этих классов будут отдельные их представители. Слова «экземпляр класса» и «объект» являются синонимами. Например, для класса «Аудитория» объектом (или экземпляром класса) будет аудитория 24- 03, а аудитория 24-05 — другой объект этого же класса. Для класса «Студент», каждый отдельный студент будет являться экземпляром данного класса.

Соответственно у каждого класса есть набор атрибутов (или переменных) и методов (или функций), которые вы описали отдельным классом BadStreet.

```
Class BadStreet {
protected:
bool alias;
private:
        string district;
        string tracksuit;
        string realnyPacan;
public:
        void  SemkiEst();
        void AEsliNaidu();
        void ChyoSmotrish();
        void EstMobilkaPozvonit();
}
```

В UML класс изображается следующим образом:



Процедура создания объекта класса идентична процедуре объявления переменной одного из базовых типов. Мы указываем имя нашего класса, а затем наименование нового объекта.

```
BadStreet Harchok; // создание экземпляра (объекта) класса
Harchok.EstMobilkaPozvonit(); // вызов метода, чтобы отжать смартфон
Harchok.tracksuit = "Podvalniy Adidas"; // инициализация атрибута
```

В объектно-ориентированном программировании выделяют следующие три основных принципа:

- инкапсуляция;
- наследование;
- полиморфизм.

1) *Инкапсуляция* — это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе, и скрыть детали реализации от программиста. Инкапсуляция, как правило, обеспечивается закрытым уровнем доступа к атрибутам класса.

В структурном программировании переменная, объявленная в теле функции имеет локальную область видимости, и соответственно она не видна за пределами данной функции и к ней нельзя обратиться кроме как внутри функции, где она была объявлена. В объектно-ориентированном подходе конфигурация области видимости осуществляется с помощью уровней доступа для классификаторов (элементов) класса, различают следующие модификаторы доступа: *public*, *private*, *protected*.

Public – означает, что мы можем обратиться к элементу класса из любой точки программы (в UML обозначается через символ «+»).

Private – обратиться к элементам класса, которые помечены данным модификатором, можно только внутри самого класса (UML-обозначение: «-»).

Protected – обратиться к элементам класса можно внутри самого класса и его потомков (UML-обозначение: «#»).

Если мы попытаемся обратиться к атрибутам класса, помеченные как private и protected в произвольном месте программы (например, в функции Main), как в примере выше:

```
Harchok.tracksuit = "Podvalniy Adidas",
```

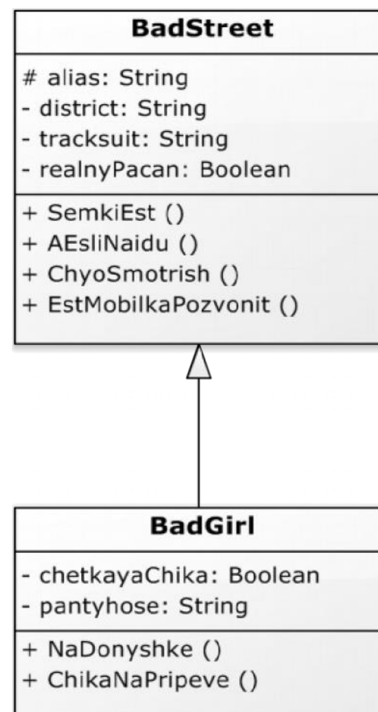
то компилятор радостно сообщит нам об ошибке.

Еще в UML предусмотрен специальный модификатор доступа для диаграммы пакетов — Package – означает, что класс виден только для тех классов, которые объявлены в том же пакете (UML-обозначение: «~»).
Согласно общей стратегии использования классов атрибуты класса следует оставлять закрытыми. Благодаря этому достигается инкапсуляция данных внутри класса. А вот большая часть методов (если не все), как правило, общедоступна. Разумеется, по мере необходимости мы можем ослаблять данные ограничения.

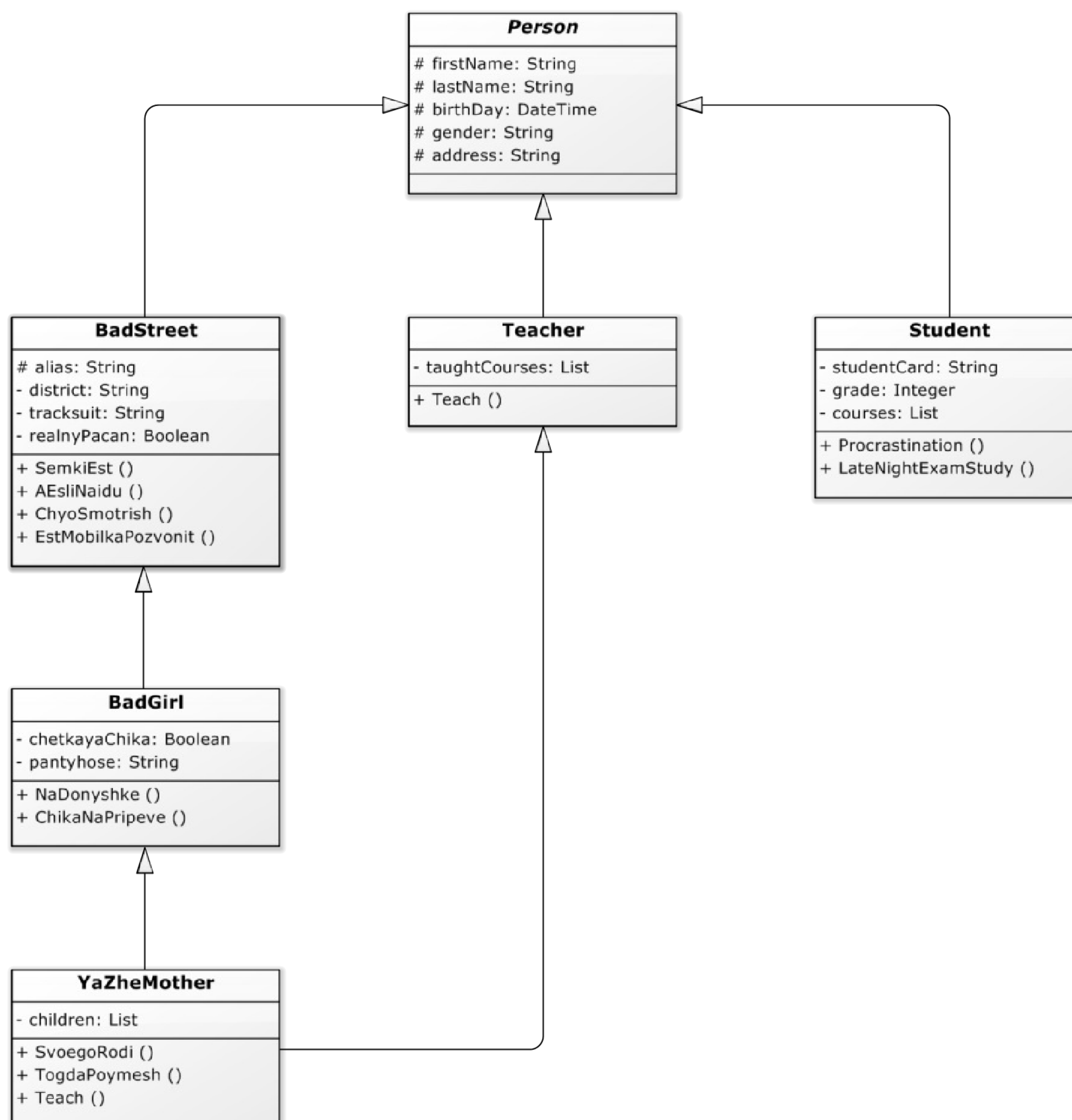
Таким образом, инкапсуляция представляет собой механизм ограничения доступа одних компонентов программы к другим. Иногда инкапсуляцию связывают с сокрытием данных — в каком именно смысле? Например, для того, чтобы пользоваться микроволновкой, вам не нужно знать принципов работы магнетрона, или чтобы использовать смартфон, нет никакой необходимости разбираться в устройстве интегральных схем. Микроволновка и смартфон это полностью инкапсулированные объекты, их внутреннее устройство скрыто от вас и вам не нужно его знать, для того чтобы использовать их. Это же относиться и к методам классов — для программиста методы должны быть полностью инкапсулированными объектами, программист не должен задумываться над их внутренним устройством, он должен знать, как использовать данные методы для решения поставленных перед ним задач. Например, если у вас есть библиотека PDO (для интерфейса доступа к БД в PHP), вам не нужно знать, как реализованы методы для подключения к БД, подготовки и осуществления запросов, инициализации и отката транзакций, вам нужно знать, как использовать данные методы.

2) *Наследование* — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Иными словами, это когда объект приобретает свойства другого объекта и при этом добавляет к ним черты, характерные только для него. Класс, от которого производится наследование, называется родительским, а новый класс — дочерним, потомком или производным.

Например, класс BadStreet определен у нас для лиц мужского пола, но в нашем приложении наравне с реальными юношами есть и девушки. Соответственно мы можем на его основе создать класс для девушек, который унаследует поведение родительского класса. Таким образом, класс BadGirl встраивает (наследует) компоненты класса BadStreet в дополнение к своим собственным. В классе-наследнике компоненты родителя не повторяются: они автоматически доступны благодаря наследованию в соответствии с модификаторами доступа (то есть отсутствует дублирование кода). В нем будут указаны лишь компоненты, специфичные для наследника.
Наследование в UML изображается в виде полой стрелки, которая указывает на родителя.



Потомок — это прямой или непрямой наследник; классы BadStreet, Teacher и Student являются прямыми наследниками класса Person, а классы BadGirl и YaZheMother непрямыми (косвенными).



Класс вправе иметь произвольное число родителей, в случае если родителей 2 или более, говорят о множественном наследовании (например, у класса YaZheMother два родителя — BadGirl и Teacher, так как в данном классе возникает необходимость в функциях наставления children и особенно посторонних людей, предпочтительно других девушек, у которых нет children).
Во многих языках программирования наследование от нескольких классов запрещено, например в Java, C# или PHP, это связано с тем, что множественное наследование сопряжено с рядом технических проблем (например, ромбовидное наследование, когда некоторый класс многократно становится наследником одного и того же класса, проходя по разным ветвям наследования; в примере выше класс YaZheMother два раза является наследником класса Person). Разумеется, то что оно запрещено, не значит, что его нельзя реализовать в указанных языках, можно, но только иным способом, через реализацию интерфейсов, что позволяет решить ряд проблем, связанных с классической реализацией множественного наследования.

В целом рекомендуется отказаться от множественного наследования в пользу одиночного. Несмотря на то, что оно призвано упростить код, по факту множественное наследование его только усложняет. Но есть моменты, где оно необходимо, например, при моделирования систем (по типу дома на колесах, очевидно, что это вид транспорта и жилище одновременно) или для создания повторно используемых библиотек.

3. *Полиморфизм* — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Иными словами, это способность функций (методов) обрабатывать данные разных типов (вообще это не совсем точно, но зато просто звучит). На практике полиморфизм реализуется с помощью абстрактных классов и виртуальных методов, если попытаться объяснить полиморфизм простыми словами, то, например, у нас есть класс Teacher, для которого определен метод
«Использовать мел». Полиморфизм указывает, какую версию метода вызвать для текущего объекта. То есть преподаватель может использовать мел для того, чтобы написать что-то на доске, а может использовать для того, чтобы швырнуть его в спящего на лекции студента. То есть вариант использования конкретного метода, будет определяться в процессе выполнения программы. Например, для класса YaZheMother происходит переопределение метода Teach(), соответственно вызов конкретной версии метода будет зависит от набора параметров в процессе выполнения программы.
Правда, полиморфизм бывает разный: параметрический, подтипов, etc. Вообще полиморфизм особо не относится к данной дисциплине (в рамках текущей программы) и в ЛР вы его не коснетесь, поэтому не буду ничего расписывать про перегрузку, абстрактные классы, виртуальные методы и прочее. В курсе объектно-ориентированного программирования вам об этом расскажут, по крайней мере, я на это надеюсь :)

Вернемся к языку UML.

Диаграмма классов – диаграмма, демонстрирующая классы системы, их атрибуты, методы и взаимосвязи между ними.
Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.
Класс – это описание множества объектов с одинаковыми атрибутами, операциями, связями и семантикой.

Класс на диаграмме показывается в виде прямоугольника, разделенного на 3 области. В верхней содержится название класса, в средней – описание атрибутов (свойств), в нижней – названия операций (методов) – услуг, предоставляемых объектами этого класса.

Для каждого атрибута или метода класса задается видимость (visibility). Это свойство, указывающее на возможность использования данного атрибута или операции другими классификаторами. В UML можно специфицировать четыре уровня видимости:
1. public (открытый). Любой внешний классификатор, которому виден данный, может использовать это свойство. Обозначается символом + перед именем атрибута или операции.
2. protected (защищенный). Любой наследник классификатора может использовать данное свойство. Обозначается символом # (решетка) перед именем атрибута или операции.
3. private (закрытый). Данное свойство может использовать только сам классификатор. Обозначается символом – (дефис) перед именем атрибута или операции.

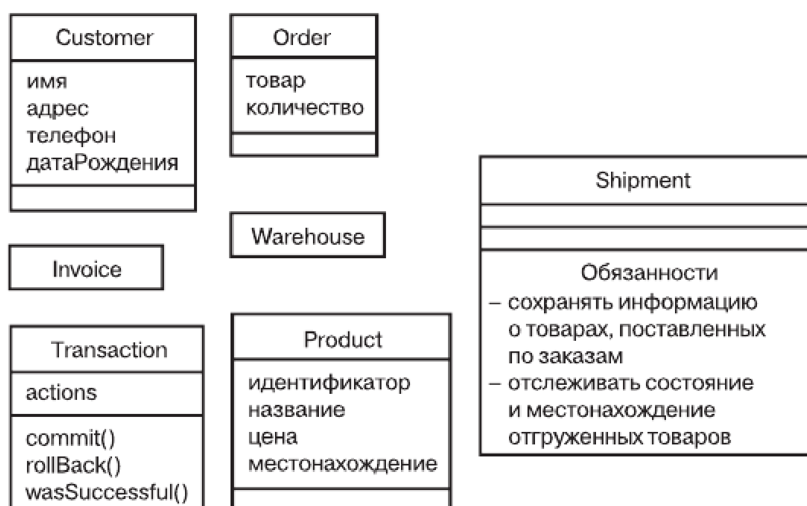


Рисунок 2.4.1 — Примеры графического представления классов в UML

Данное значение позволяет реализовать свойство инкапсуляции данных. Например, объявив все атрибуты класса закрытыми, можно полностью скрыть от внешнего мира его данные, гарантируя отсутствие несанкционированного доступа к ним. Это позволяет сократить число ошибок в программе. При этом любые изменения в составе атрибутов класса никак не скажутся на остальной части ПС.

4. package (пакетный). Только классификаторы, объявленные в том же пакете, могут использовать данное свойство. Например, пакетная видимость для классов показывает, что класс видим только для других классов, объявленных в том же самом пакете, но скрыт для классов, объявленных в других пакетах. Обозначается символом ~ (тильда) перед именем атрибута или операции.

Если вы не снабжаете свойство символом видимости явным образом, то по умолчанию в UML принимается значение public.

Ассоциация — это структурная связь, указывающая, что объекты одной сущности соединяются с объектами другой. Так, имея ассоциацию между двумя классами, вы можете соединить объекты одного класса с объектами другого. Вполне допустимо, чтобы оба конца ассоциации соединяли один и тот же класс – иными словами, один объект класса может связываться с другим объектом того же класса. Ассоциация, связывающая два класса, называется бинарной.

Ассоциации может быть присвоено имя, описывающее природу отношения. Обычно имя ассоциации не указывается, если только вы не хотите явно задать для нее ролевые имена или в вашей модели настолько много ассоциаций, что возникает необходимость ссылаться на них и отличать друг от друга. Имя будет особенно полезным, если между одними и теми же классами существует несколько различных ассоциаций.

Имея простую, без дополнений, ассоциацию между двумя классами, вы можете осуществлять навигацию от объектов одного вида к объектам другого. Если только не указано иное, навигация по ассоциации двунаправлена. Иногда требуется ограничить ее лишь одним направлением. Например, при моделировании служб операционной системы существует ассоциацией между объектами User (Пользователь) и Password (Пароль). Для данного объекта User понадобится искать соответствующий ему объект Password, но по объекту Password нет смысла находить соответствующий объект User. Вы можете явно задать направление навигации, снабдив ассоциацию дополнением в виде стрелки, указывающей в нужную сторону.

Указание направления обхода не обязательно означает, что вы не можете попасть от объекта, находящегося на одном конце ассоциации, к объекту на другом ее конце. Скорее, навигация констатирует «знание» одного объекта о другом.

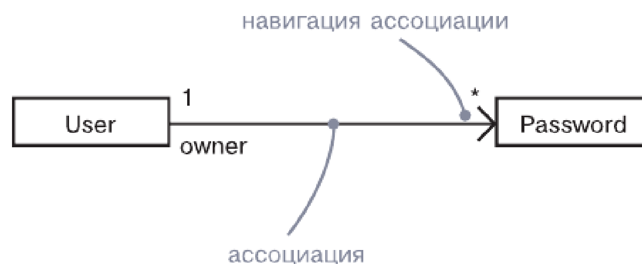


Рисунок 2.4.2 — Пример навигации ассоциации

Часто при моделировании бывает важно указать, сколько объектов может быть связано посредством одного экземпляра ассоциации. Это число называется кратностью (multiplicity) роли ассоциации и записывается либо как выражение, значением которого является диапазон значений, либо в явном виде. Указывая кратность на одном конце ассоциации, вы тем самым говорите, что на этом конце именно столько объектов должно соответствовать каждому объекту на противоположном конце. Кратность можно задать равной единице (1), можно указать диапазон: «ноль или единица» (0..1), «много» (0..*), «единица или больше» (1..*). Разрешается также указывать определенное число (например, 3). С помощью списка можно задать и более сложные кратности, например 0 . . 1, 3..4, 6..*, что означает «любое число объектов, кроме 2 и 5».

На диаграммах классов обычно показываются ассоциации и обобщения.

Обобщение — это связь между сущностью общего характера (называемой суперклассом, или родителем) и более специфичной сущностью (называемой подклассом, дочерним классом или потомком). Объекты дочернего класса могут быть использованы как переменные или параметры типа его родителя, но не наоборот. Другими словами, дочерняя сущность может быть подставлена там, где объявлена родительская. Дочерняя сущность наследует свойства родителя, а именно его атрибуты и операции. Часто, хотя и не всегда, потомок имеет дополнительные атрибуты и операции помимо родительских. Реализация операции в дочернем классе замещает реализацию той же операции родителя – это явление называется полиморфизмом. Одинаковые операции должны иметь одинаковую сигнатуру (имя и параметры). Изображается сплошной линией с треугольников со стороны родителя
О классе, у которого есть только один родитель, говорят, что он использует одиночное наследование, в отличие от класса, у которого более чем один родитель (множественное наследование).

Зависимость — это связь, которая устанавливает, что одна сущность, например класс Window (Окно), использует информацию и сервис (операцию либо услугу), представляемые другой сущностью, например классом Event (Событие), но не обязательно – наоборот. Зависимость изображается в виде пунктирной линии со стрелкой, направленной на зависимую сущность. Выбирайте зависимость, когда вам нужно показать, что одна сущность использует другую.

Агрегация. Простая ассоциация между двумя классами представляет структурную связь между равноправными элементами: оба класса концептуально находятся на одном уровне – ни один из них не может считаться важнее другого. Рано или поздно вам понадобится смоделировать связь «целое-часть», в которой один класс представляет крупную сущность (целое), содержащую в себе более мелкие (части). Этот тип связей, основанных на отношениях обладания, называется *агрегацией* и подразумевает, что объект-целое обладает объектами-частями. По сути, агрегация – это особый вид ассоциации, поэтому изображается она линией простой ассоциации, к которой добавлен пустой ромб со стороны объекта-целого

Композиция — это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Части с нефиксированной множественностью могут быть созданы после самого композита, но однажды созданные, живут и умирают вместе с ним. Кроме того, такие части могут быть явно удалены перед «смертью» композита. Иными словами композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено. Это означает, что в композитной агрегации объект может быть одновременно частью только одного композита. Графически представляется сплошной линией с закрашенным ромбом на стороне целого.

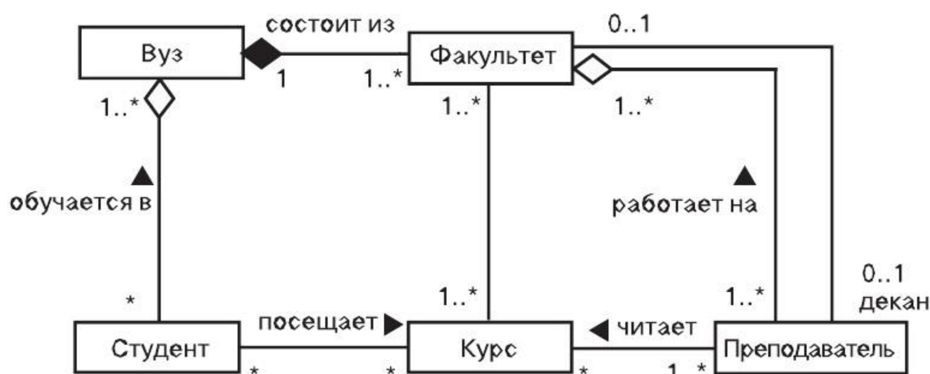


Рисунок 2.4.3 — Пример отношений агрегации и композиции

Реализация — отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком). Графические изображается как и обобщение, но с пунктирной линией. Треугольник должен быть со стороны поставщика. Семантически реализация представляет нечто среднее между зависимостью и обобщением, поэтому ее графическая нотация представляет собой сочетание нотаций зависимости и обобщения.

В основном вы будете применять реализацию для описания связи между интерфейсом и классом (или же компонентом, который предоставляет операцию либо сервис для него). Интерфейс – это набор операций, используемых для спецификации сервиса класса или компонента. Таким образом, интерфейс описывает контракт, который класс или компонент обязаны исполнять.

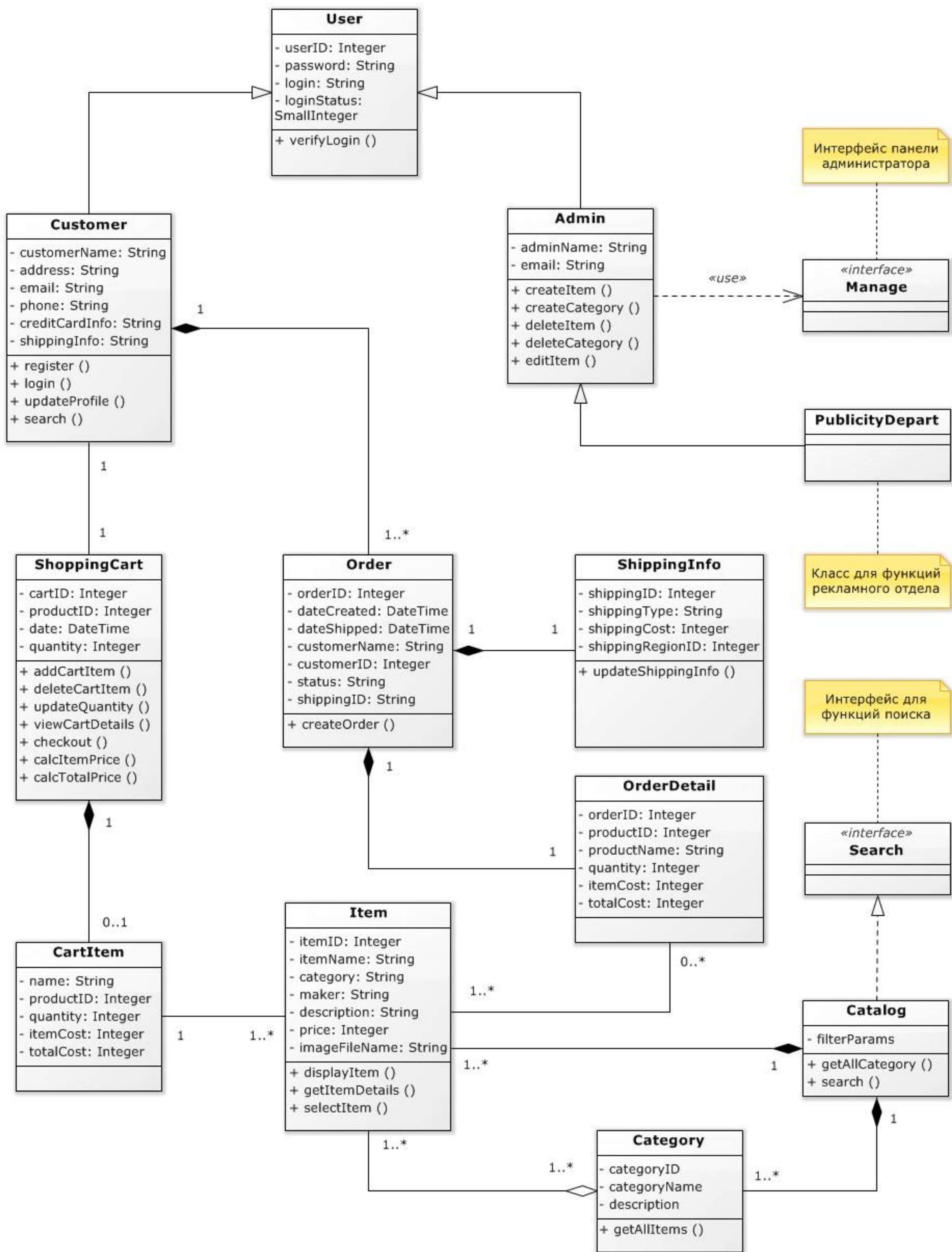


Рисунок 2.4.4 — Диаграмма классов для ИС «Интернет-магазин»

2.5 Содержание отчета

Отчет о работе должен содержать:

1. Титульный лист
2. Цель работы
3. Задание на лабораторную работу и вариант
4. Диаграмма классов
5. Выводы по работе
6. Список использованных источников

Пример вывода:

В результате выполнения данной лабораторной работы были изучены способы представления предметной области информационной системы и разработана диаграмма классов для системы «Интернет-магазин».
Разработанная диаграмма описывает основные классы системы, атрибуты и методы классов. Некоторые классы содержат неполный набор параметров, это связано с тем, что полный перечень всех атрибутов и методов, которые содержат классы системы, не является востребованным на данном этапе проектирования и уменьшит читаемость диаграммы.
На данной диаграмме использованы различные виды отношений, что в действительности является избыточным шагом, так как многие из представленных отношений могут быть заменены на ассоциации без ущерба для самой модели.

3. ЛАБОРАТОРНЫЕ РАБОТЫ №3 И №4 «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ. РАЗРАБОТКА ДИАГРАММ ПОСЛЕДОВАТЕЛЬНОСТИ И КОММУНИКАЦИИ»

3.1. Цель работы

Целью данной работы является изучение способов описания взаимодействия между объектами и их жизненного цикла проектируемой информационной системы в виде диаграмм последовательности и коммуникации.

3.2. Задание на лабораторную работу

Разработать диаграммы последовательности, описывающие взаимодействие объектов для прецедентов из ЛР №1 и осуществить ручное или автоматическое преобразование диаграмм последовательности в диаграммы коммуникации.

3.3. Порядок выполнения работы

1. Изучить теоретический материал, изложенный в подразделе 4.4;
2. Разработать диаграммы последовательности для ВИ из ЛР №1;
3. Преобразовать диаграммы последовательности в диаграммы коммуникации в соответствии с заданием;
4. Ознакомиться с требованиями по содержанию отчета в подразделе 2.5;
5. Написать отчет о работе.

Некоторые CASE-средства позволяют выполнить автоматическое преобразование диаграмм

последовательности в диаграммы коммуникации[2].
В UML 1.0 диаграммы коммуникации носили наименование диаграмм кооперации, в UML 2.0 для диаграмм кооперации определен иной вид и назначение.

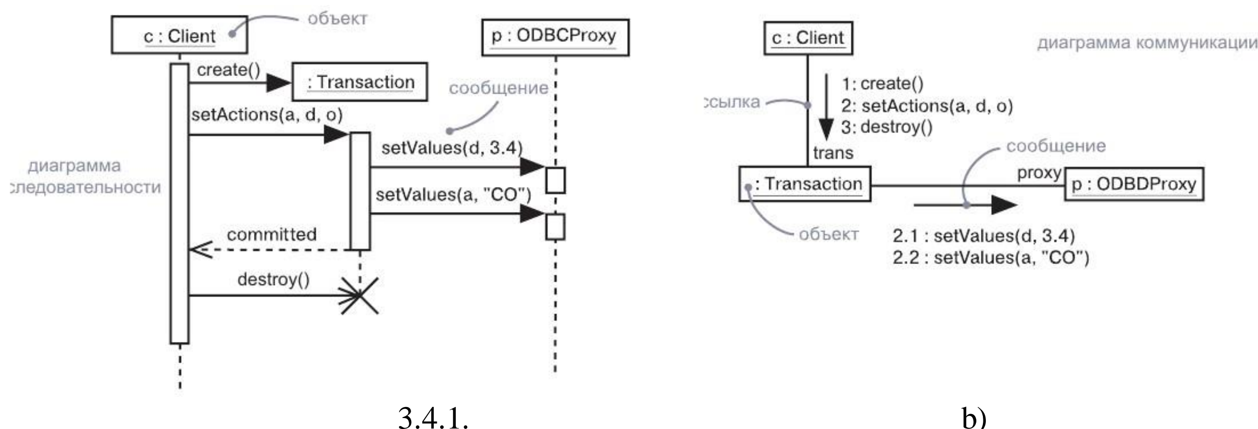
Каждая диаграмма последовательности должна описывать логику работы отдельного варианта использования при определенном сценарии работы.
Для одного и того же варианта использования также может быть определено несколько диаграмм последовательности, описывающих взаимодействие объектов при различных условиях (сценариях) работы, например,

Вариант использования:	покупка товара;
Основной поток (сценарий):	покупка товара при регистрации на сайте;
Альтернативный поток (сценарий):	покупка товара без регистрации (быстрая покупка).

2 Umbrello UML Modeller не поддерживает автоматическое преобразование диаграмм взаимодействия. Помню, что Rational Rose поддерживал данное преобразование (если не ошибаюсь F5 или Ctrl + F5), но не думаю, что стоит заморачиваться с этим.

3.4. Теоретический материал

Диаграмма последовательности – это диаграмма взаимодействия, которая подчеркивает временной порядок сообщений. Изображается как таблица, в которой представлены объекты, расположенные вдоль оси X, и сообщения, упорядоченные по ходу времени – вдоль оси Y. Диаграмма коммуникации – это диаграмма взаимодействия, которая выделяет структурную организацию объектов, отправляющих и принимающих сообщения. Графически представляет собой набор дуг и вершин.



3.4.1. b)

Рисунок 3.4.1 — Пример диаграмм взаимодействия
a) — диаграмма последовательности b) — диаграмма коммуникации

Диаграммы последовательности отличаются от диаграмм коммуникации двумя признаками: Во-первых, это линии жизни (lifelines). Линия жизни объекта – вертикальная пунктирная линия, символизирующая существование объекта в течение некоторого периода времени. Большинство объектов, представленных на диаграмме взаимодействия, существуют в течение всего взаимодействия, поэтому все они выровнены по верхней границе диаграммы, а линии их жизни проведены от верха до низа.
Объекты могут быть созданы в процессе взаимодействия. Их время жизни начинается с получения сообщения create, направленного к прямоугольнику объекта в начале жизненного пути. Равным образом в процессе взаимодействия объекты могут уничтожаться. Их линия жизни заканчивается при получении сообщения destroy, что графически отмечено большим символом X.
Во-вторых, это наличие фокуса управления. Фокус управления (focus of control) – высокий узкий прямоугольник, показывающий период времени, в течение которого объект выполняет действие – как непосредственно, так и с помощью зависимой процедуры. Верхняя грань прямоугольника выровнена по началу действия, а нижняя – по его завершению и может быть отмечена сообщением возврата, которое обычно отображается пунктирной линией.

Вы можете показать вложенность фокуса управления, вызванную рекурсией, вызовом собственной операции либо возвратом вызова из другого объекта, наложив другой фокус управления чуть правее родительского (таким образом можно изобразить сколько угодно уровней вложения). Если нужно особенно точно показать расположение фокуса управления, оттените часть прямоугольника, обозначающего период времени, в течение которого на самом деле работает метод объекта и управление не передается другому объекту.

Основное содержимое диаграммы последовательности – сообщения. Они изображаются стрелками, направленными от одной линии жизни к другой. Стрелка указывает на приемник сообщения. Если таковое асинхронно, то стрелка рисуется «уголком», а если синхронно (вызов), то закрашенным треугольником. Ответ на синхронное сообщение (возврат из вызова) показывается пунктирной стрелкой «уголком». Сообщение возврата может быть опущено, поскольку каждый вызов неявно подразумевает возврат, но иногда удобно таким образом продемонстрировать возвращаемое значение.
На диаграмме последовательности вы можете не делать различия в графической нотации синхронных (на которые следует ответ) и асинхронных сообщений.
Чтобы моделировать потоки управления, упорядоченные по времени применяются диаграммы последовательности. Моделирование потока управления, упорядоченного по времени, выделяет передачи сообщений в хронологическом порядке, что, в частности, удобно для визуализации динамического поведения в контексте сценариев вариантов использования. Диаграммы последовательности лучше выполняют задачу визуализации простых итераций и ветвления, чем диаграммы коммуникаций.

Чтобы моделировать потоки управления по организации. Для этого применяются диаграммы коммуникации. Моделирование потока управления по организации выделяет структурные связи между экземплярами во взаимодействии наряду с сообщениями, которые могут между ними передаваться.

Определив структуру варианта использования, необходимо описать поведение каждого из них. Обычно нужно составить одну или несколько диаграмм последовательности для каждого основного сценария, затем – диаграммы последовательности для вариаций сценариев и, наконец, хотя бы одну диаграмму последовательности, иллюстрирующую каждый из возможных типов ошибок или исключений. Обработка ошибок – часть варианта использования, поэтому она должна быть запланирована наряду с нормальным поведением.

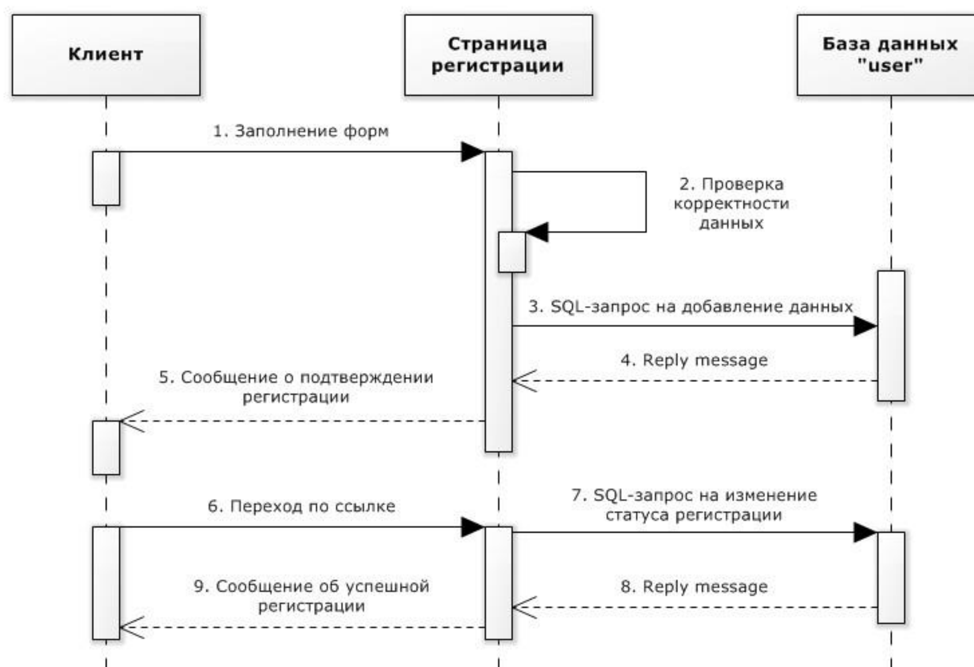


Рисунок 3.4.2 — Диаграмма последовательности для ВИ «Регистрация»

Рисунок 3.4.3 — Диаграмма коммуникации для ВИ «Регистрация»

3.5. Содержание отчета

Отчет о работе должен содержать:

1. Титульный лист
2. Цель работы
3. Задание на лабораторную работу и вариант
4. Диаграммы последовательности или коммуникации (3 штуки)
5. Выводы по работе
6. Список использованных источников (2-4 источника)

Пример вывода:

В результате выполнения данной лабораторной работы были получены навыки построения концептуальной модели проектируемой системы, описания взаимодействия объектов и их жизненного цикла.
Были построены 7 диаграмм последовательности, 5 для основного потока и 2 для альтернативного, в частности:

- ошибка при выборе способа оплаты;
- ошибка при регистрации (некорректные данные).

Для прецедентов «Регистрация», «Заказ товара» и «Выбор способа оплаты» были также составлены диаграммы коммуникации, данные диаграммы используют ту же информацию, что и диаграммы последовательности, но представляют ее в ином виде, делая акцент на взаимодействии между объектами.

4. **ЛАБОРАТОРНАЯ РАБОТА №5.** «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ. **РАЗРАБОТКА ДИАГРАММ СОСТОЯНИЙ»**

4.1 Цель работы

Целью данной работы является изучение способов описания исполняемого поведения системы в виде диаграмм состояний.

4.2 Задание на лабораторную работу

Разработать не менее трех диаграмм состояния, описывающих состояния отдельных объектов (компонентов) системы.
Диаграммы состояний должны показывать состояния объектов классов из ЛР №2.

4.3 Порядок выполнения работы

1. Изучить теоретический материал, изложенный в подразделе 5.4;
2. Разработать диаграммы состояний;
3. Ознакомиться с требованиями по содержанию отчета в подразделе 5.5;
4. Написать отчет о работе.

4.4 Теоретический материал

Диаграммы состояний предназначены для представления жизненного цикла объекта в виде конечного автомата. Каждое состояние – это период жизни объекта, когда он удовлетворяет определенным условиям. Некоторое событие может привести переходу объекта в другое состояние. При переходе может выполняться действие, предписанное данному переходу. Состояния на диаграмме показываются в виде прямоугольников со скругленными углами, а события – стрелками. Диаграмма состояний обычно связывается с классом, поскольку все объекты класса имеют одинаковое поведение. При функциональном моделировании диаграммы состояний создаются для объектов, имеющих сложное поведение, показывая логику их работы.

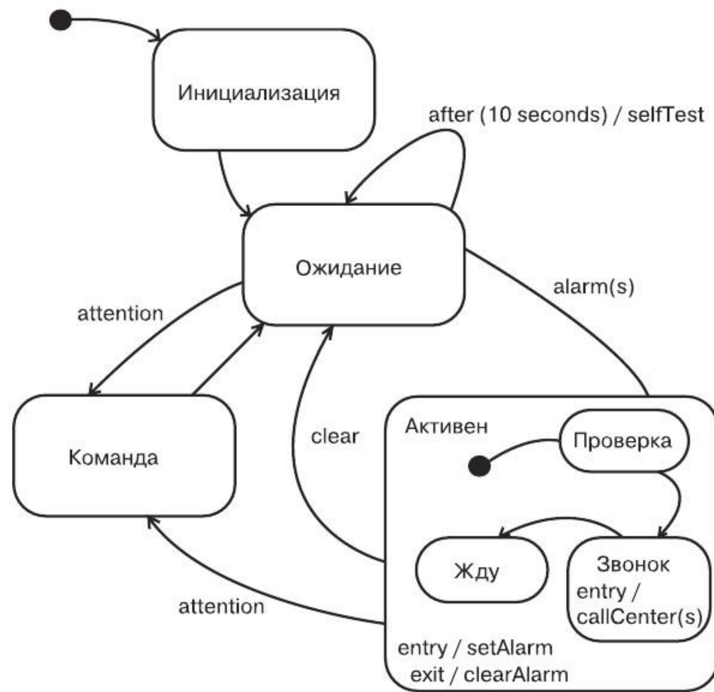


Рисунок 5.4.3 — Пример диаграммы состояния для жизненного цикла объекта

В диаграмме состояний используются следующие условные обозначения:
- Круг, обозначающий начальное состояние.
- Окружность с маленьким кругом внутри, обозначающая конечное состояния (если есть).
- Скругленный прямоугольник, обозначающий состояние. Верхушка прямоугольника содержит название состояния. Также в состоянии могут указываться активности (действия), которые происходят в данном состоянии. Действия помечаются стереотипами entry (действия, выполняемые при заходе в данное состояние), do (действия, выполняемые в процессе данного состояний) и exit (действия, выполняемые при выходе из данного состояния).
- Стрелка, обозначающая переход. Название события (если есть), вызывающего переход, отмечается рядом со стрелкой.
- Толстая горизонтальная линия с либо множеством входящих линий и одной выходящей, либо одной входящей линией и множеством выходящих. Это обозначает объединение и разветвление соответственно (в диаграмме видов деятельности параллельные процессы).

На диаграмме состояний могут быть также показаны композитные (сложные) состояния, которые включают в себя другие состояния.

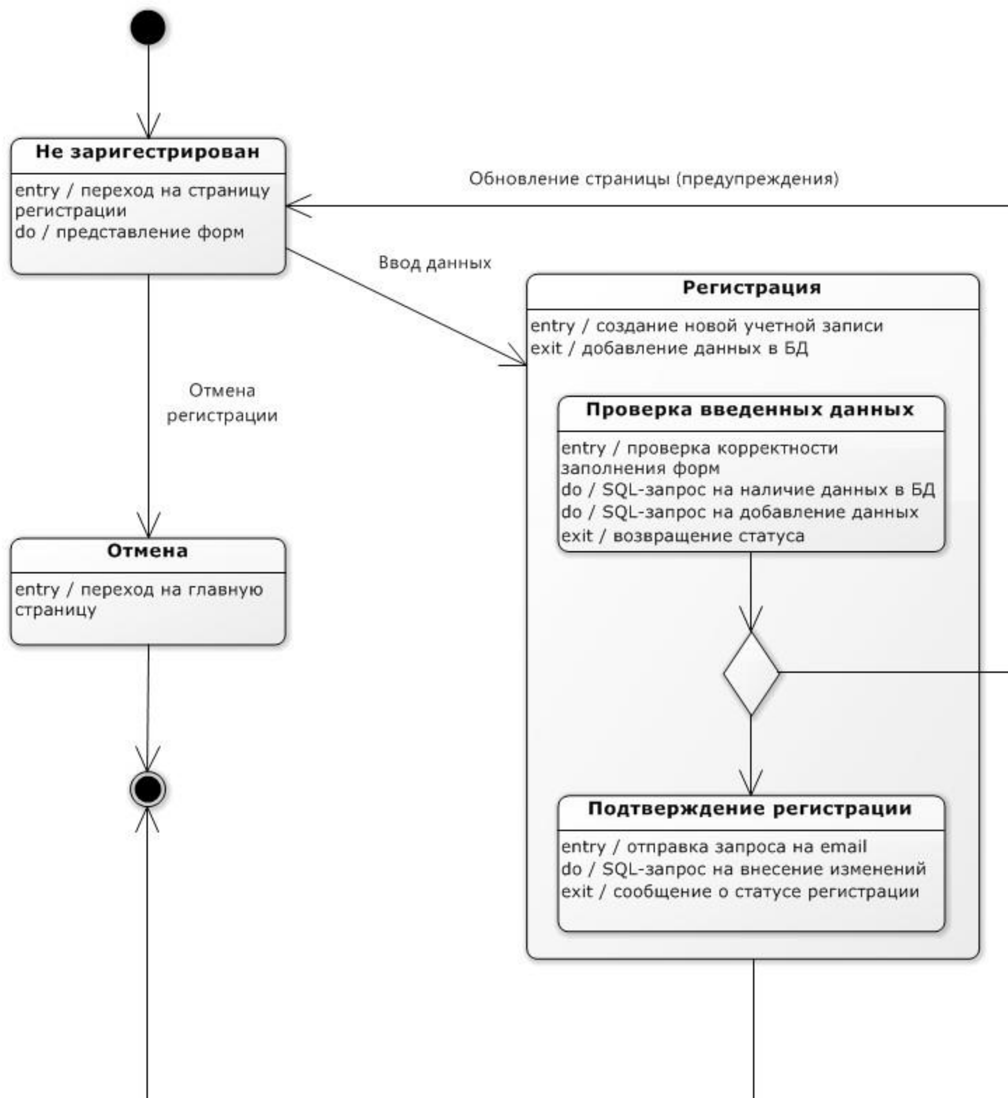


Рисунок 5.4.4 — Диаграмма состояний для пользователя при регистрации (в данном случае сессия пользователя выступает в роли объекта)

4.5 Содержание отчета

Отчет о работе должен содержать:

1. Титульный лист
2. Цель работы
3. Задание на лабораторную работу и вариант
4. Диаграммы состояний (3 штуки)
5. Выводы по работе
6. Список использованных источников (2-4 источника)

Пример вывода:

В результате выполнения данной лабораторной работы были получены навыки описания исполняемого поведения системы в виде диаграмм состояний.

5. ЛАБОРАТОРНАЯ РАБОТА №6.

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ СИСТЕМЫ.

РАЗРАБОТКА ДИАГРАММ КОМПОНЕНТОВ И РАЗВЕРТЫВАНИЯ»

6.1 Цель работы

Целью данной работы является изучение способов описания аппаратных и программных компонентов системы в виде диаграммы развертывания.

6.2 Задание на лабораторную работу

Разработать диаграмму развертывания для заданной информационной системы или ее функционально законченной части.

Для оценки «хорошо» не менее 3 узлов на диаграмме развертывания;
Для оценки «отлично» не менее 5 узлов.

6.3 Порядок выполнения работы

1. Изучить теоретический материал, изложенный в подразделе 6.4;
2. Разработать диаграмму развертывания;
3. Ознакомиться с требованиями по содержанию отчета в подразделе 6.5;
4. Написать отчет о работе.

6.4 Теоретический материал

На диаграмме развертывания показывают узлы (аппаратные компоненты) информационной системы (например, сервер базы данных, web-сервер, сервер приложения, принтер, коммутатор etc.), какие программные компоненты работают на каждом узле (например, веб-приложение, база данных), связанные с ними артефакты (классы, файлы, библиотеки) и связи между ними. Над связями обычно указывают протоколы, интерфейсы передачи данных, порты etc. (например, TCP/IP, HTTP, HTTP:80, JDBC, Wireless etc.).

Артефакты используются для моделирования таких физических сущностей, которые могут располагаться на узле, – например, исполняемых программ, библиотек, таблиц, файлов и документов. Обычно артефакт представляет собой физическую группировку таких логических элементов, как классы, интерфейсы и кооперации.

Узел – это физический элемент, который существует во время выполнения и представляет вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а зачастую также и процессором. Иными словами, узел — это аппаратное обеспечение системы. Узлы используются для моделирования топологии аппаратных средств, на которых работает система. Как правило, узел – это процессор или устройство, на котором могут быть размещены артефакты.

Во многих отношениях узлы подобны артефактам. Те и другие наделены именами, могут участвовать в связях зависимости, обобщения и ассоциации, бывают вложенными, могут иметь экземпляры и вступать во взаимодействия. Однако между ними есть и существенные различия:

- артефакты принимают участие в работе системы, а узлы – это сущности, на которых работают артефакты;
- артефакты представляют физическую упаковку логических элементов, узлы представляют физическое размещение артефактов.
Первое из этих отличий самое важное. Суть его проста: узлы исполняют артефакты, артефакты работают на узлах.
Второе различие предполагает наличие некоторой связи между классами, артефактами и узлами.
Есть несколько типов систем, для которых диаграммы размещения не нужны. Если вы разрабатываете программу, исполняемую на одной машине и обращающуюся только к стандартным устройствам на этой же машине, управление которыми полностью возложено на операционную систему (возьмем для примера клавиатуру, дисплей и модем персонального компьютера), то диаграммы размещения можно игнорировать. Но если разрабатываемая программа обращается к устройствам, которыми операционная система обычно не управляет, или эта программа физически размещена на разных процессорах, то диаграмма размещения поможет выявить отношения между программными и аппаратными средствами.

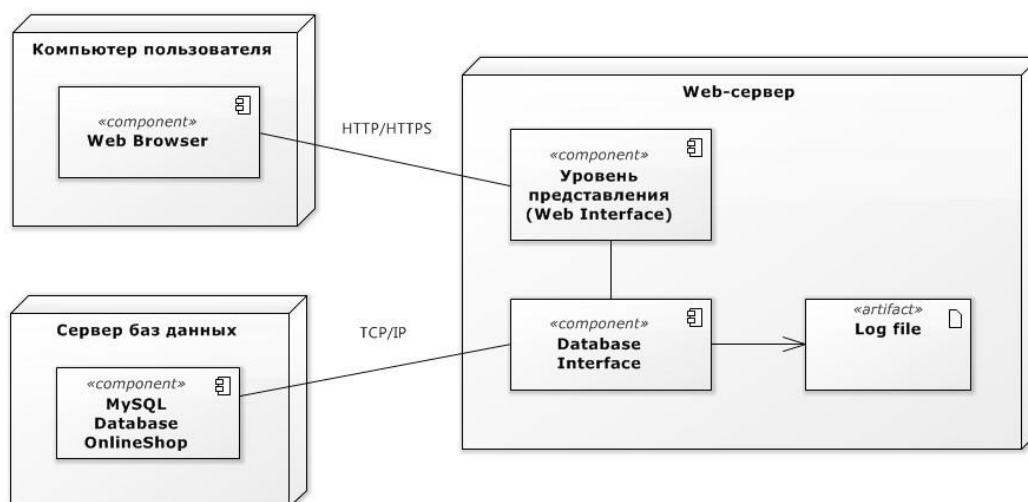


Рисунок 6.1 — Диаграмма развертывания для ИС «Интернет-магазин»

6.5 Содержание отчета

Отчет о работе должен содержать:

1. Титульный лист
2. Цель работы
3. Задание на лабораторную работу и вариант
4. Диаграмма развертывания
5. Выводы по работе
6. Список использованных источников (2-4 источника)

Пример вывода:

В результате выполнения данной лабораторной работы были получены навыки графического описания спецификации оборудования и компонентов системы.
В данной системе предполагается наличие 3 узлов: веб-сервера, где будет непосредственно располагаться Интернет-магазин, сервера базы данных и компьютера пользователя.