Министерство образования Республики Беларусь

Учреждение образования
«Полоцкий государственный университет»

К. Я. Раханов
А. О. Лукьянов
Д. М. Васильева

ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания
по выполнению лабораторных работ
для студентов специальностей
1-40 05 01 «Информационные системы и технологии (по направлениям)»,
1-40 05 01-02 «Информационные системы и технологии (в экономике)»

Текстовое электронное издание

Новополоцк
Полоцкий государственный университет
2022

1 – дополнительный титульный экран – сведения об издании

УДК 002.52/54(075.8)

Одобрено и рекомендовано к изданию методической комиссией
факультета информационных технологий
в качестве методических указаний (протокол № 11 от 27.12.2021 г.)

Кафедра вычислительных систем и сетей

РЕЦЕНЗЕНТ:
доц., канд. техн. наук, доц. каф. технологий программирования
Полоцкого государственного университета А. Ф. ОСЬКИН;
доц., канд. техн. наук, зав. каф. математики и компьютерной безопасности
Полоцкого государственного университета И. Б. БУРАЧЕНОК

2 – дополнительный титульный экран – производственно-технические сведения

Для создания текстового электронного издания «Технология разработки программного обеспечения» использованы текстовый процессор Microsoft Word и программа Adobe Acrobat XI Pro для создания и просмотра электронных публикаций в формате PDF.

Технические требования:
1 оптический диск.
Системные требования:
PC с процессором не ниже Core 2 Duo;
2 Gb RAM; свободное место на HDD 2 Mb;
Windows XP/7/8/8.1/10
привод CD-ROM/DVD-ROM;
мышь

Редактор *Т. А. Дарьянова*

Подписано к использованию 22.04.2022.
Объем издания 2,87 Мб. Заказ 272.

Издатель и полиграфическое исполнение:
учреждение образования «Полоцкий государственный университет».

Свидетельство о государственной регистрации
издателя, изготовителя, распространителя печатных изданий
№ 1/305 от 22.04.2014.

ЛП № 02330/278 от 08.05.2014.

211440, ул. Блохина, 29,
г. Новополоцк,
Тел. 8 (0214) 59-95-41, 59-95-44
http://www.psu.by

Содержание

Введение

Программное обеспечение (ПО) – множество программ, обеспечивающих работу вычислительных машин для решения с их помощью задач определенной предметной области.

Технология разработки ПО – это совокупность процессов и методов создания программного продукта с заданными характеристиками качества. К процессам и методам создания ПО относятся:

- разработка технического задания;
- разработка диаграммы вариантов использования;
- разработка диаграммы деятельности;
- разработка диаграммы взаимодействия;
- модели и генерирование кода на основе UML диаграмм;
- оптимизация программного кода;
- разработка программного обеспечения для проектируемой предметной области;
- работа с системой контроля версий.

Техническое задание является юридическим документом – как приложение включается в договор между заказчиком и исполнителем на проведение проектных работ и является его основой: определяет порядок и условия работ, в т.ч. цель, задачи, принципы, ожидаемые результаты и сроки выполнения.

Разработка диаграммы вариантов использования необходима для определения набора функций и требований к ним, действующих лиц и их взаимодействия, а также для построения базовой модели программы для ее последующей детализации в других диаграммах.

Разработка диаграммы деятельности позволяет спроектировать модель всей системы и каждого из ее блоков, а также продемонстрировать последовательность действий, выполняемых системой.

Диаграмма взаимодействия предназначена для описания взаимодействия между элементами программы и передачи сообщений между ними.

Генерация кода позволяет автоматически создавать код программы, используя за основу только UML диаграмму классов.

Диаграмма классов UML иллюстрирует структуру системы, описывая классы, их атрибуты, методы и отношения между объектами.

Методы оптимизация программного кода позволяют преобразовать код для улучшения его характеристик и повышения эффективности.

Цель выполнения лабораторных работ – формирование систематизированных знаний о жизненном цикле разработки программного обеспечения и технологиях, применяемых на различных его этапах, включая моделирование предметной области, формализацию требований, алгоритмизацию проектных решений, программную реализацию и отладку приложений.

Задачами выполнения лабораторных работ дисциплины являются:

– приобретение знаний о цели и основных задачах в области разработки программного обеспечения;

– приобретение знаний об основах моделей и методологий жизненного цикла разработки программного обеспечения;

– приобретение знаний о парадигмах программирования;

– овладение базовыми методами анализа предметной области и формализации требований к разработке программного обеспечения;

– овладение базовыми методами моделирования и алгоритмизации для анализа и разработки проектных решений;

– овладение базовыми методами написания качественного и эффективного кода;

– изучение принципов юзабилити для создания дружественных пользовательских интерфейсов;

– ознакомление со стандартами разработки программных средств и систем и областью программной инженерии.

В результате изучения и выполнения методических указаний по выполнению лабораторных работ обучающийся должен

знать:

– базовые понятия информационных технологий, основные и перспективные направления развития информационных систем и технологий;

– определение, эволюционное развитие моделей жизненного цикла разработки программного обеспечения;

– парадигмы программирования и существующие подходы к разработке программ;

– методы, технологии и средства анализа и моделирования предметной области;

– методы, технологии и средства анализа, моделирования и алгоритмизации проектных решений;

– принципы, методы и средства структурного программирования;

– принципы, методы и средства объектно-ориентированного программирования;

уметь:

– выявлять и определять существенные элементы разработки;

– выполнять анализ предметной области;

– определять и формулировать требования к разработке программного обеспечения;

– выполнять графическую интерпретацию проектных решений;

– применять современные подходы к программированию и отладке приложений;

владеть:

– современными технологиями проектирования и разработки программного обеспечения;

– навыками в составе группы специалистов разрабатывать проектную документацию к программному обеспечению;

– методами кодирования и отладки программного обеспечения для реализации проектных решений;

– современными средствами инфокоммуникаций.

Лабораторная работа 1
РАЗРАБОТКА ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Цель работы: изучить элементы диаграммы вариантов использования, освоить принципы построения диаграммы вариантов использования, создать диаграмму вариантов использования согласно индивидуальному варианту задания.

Ход работы:

1) изучить краткие теоретические сведения;
2) ответить на контрольные вопросы;
3) выполнить практическое задание;
4) оформить отчет;
5) защитить лабораторную работу.

Термины и определения

Диаграмма вариантов использования

Диаграмма вариантов использования (англ. *use case diagram*) – диаграмма, на которой изображаются функции проектируемой системы, внешние действующие лица и отношения между ними [2].

Диаграмма вариантов использования предназначена для достижения следующих целей:

– определить набор функций, действующих лиц и их взаимодействия;

– указать требования к отдельным функциям в виде набора действий;

– разработать базовую модель программы для ее последующей детализации в других диаграммах;

– упростить взаимодействие между разработчиками, заказчиками и пользователями.

Диаграмма вариантов использования может включаться в техническую или сопроводительную документацию.

Элементы диаграммы вариантов использования

Вариант использования (англ. *use case*) – описание действия программы или актера. Описываемые диаграммой действия включают в себя возможные реализации этих действий. Вариант использования изображается в виде эллипса.

Актер (англ. *actor*) представляет собой действующее лицо, которое взаимодействует с программой и использует ее функциональные возможности. Диаграмма может включать несколько актеров. Если актер находится слева, то он считается основным и инициирует вариант использования, справа – второстепенным. Второстепенный актер участвует в варианте использования, но не инициирует его. Например, второстепенному актеру может предоставляться информация о результате выполнения варианта использования.

Комментарий (англ. *comment*) в языке UML предназначен для включения в диаграмму произвольной текстовой информации в форме примечания, которое может быть присоединено к одному или нескольким элементам диаграммы. В качестве такой информации могут

быть, например, пояснения разработчика относительно назначения элементов, справочная информация об авторе и особенностях разработки отдельных элементов [3].

Связи на диаграмме вариантов использования

Ассоциация (англ. *association*) служит для обозначения взаимодействия актера с теми вариантами использования, которые он может инициировать. Ассоциация всегда является ненаправленной и изображается сплошной линией между актером и вариантом использования. Пример графического изображения ассоциации приведен на рисунке 1.1.

Рисунок 1.1. – Пример графического изображения ассоциации актера и вариантов использования

Отношение включения (англ. *include*) означает, что некоторый вариант использования содержит поведение, определенное в другом варианте использования. Пример графического изображения связи включения приведен на рисунке 1.2.

Рисунок 1.2. – Пример графического изображения связи включения

Отношение расширения (англ. *extend*) определяет взаимосвязь одного варианта использования с некоторым другим вариантом использования, функциональность или поведение которого задействуется первым не всегда, а только при выполнении некоторых дополнительных условий. Пример графического изображения связи расширения приведен на рисунке 1.3.

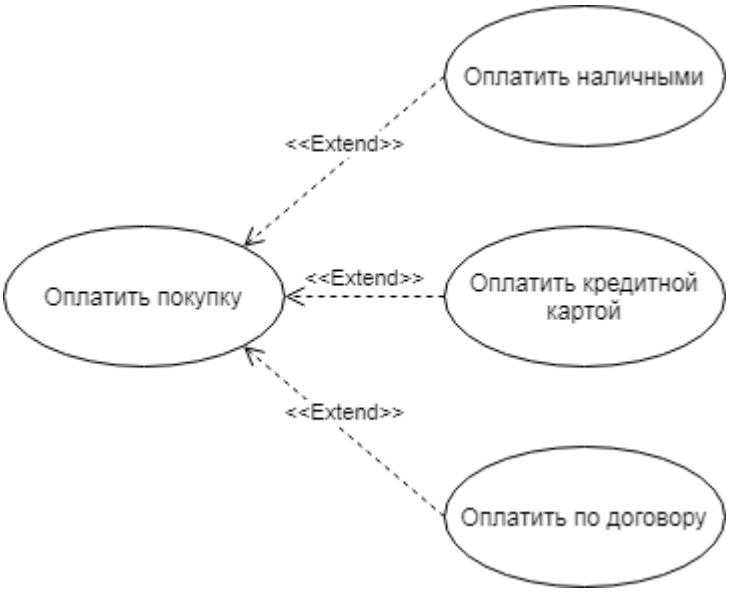


Рисунок 1.3. – Пример графического изображения связи расширения

Отношение обобщения (англ. *generalization*) предназначено для обозначения того, что один элемент диаграммы является специальным или частным случаем другого элемента [3]. Пример графического изображения связи обобщения приведен на рисунке 1.4.

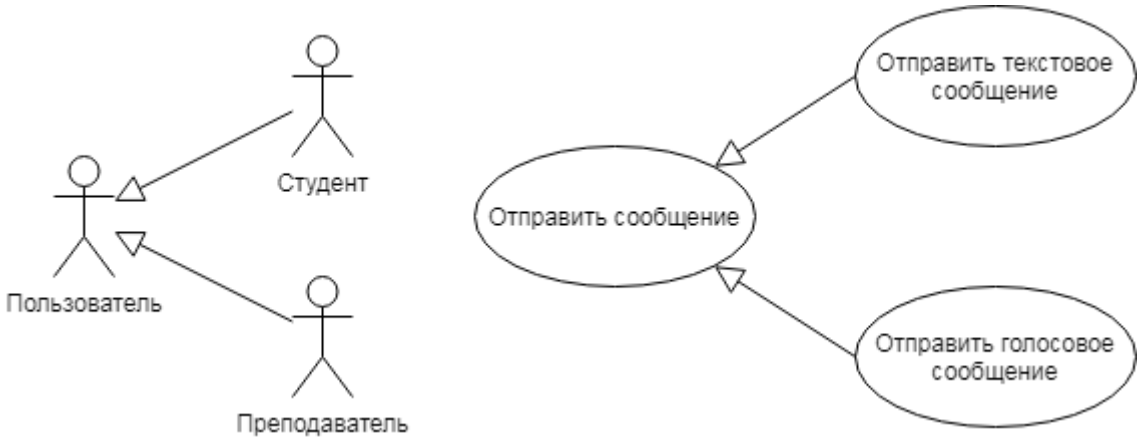


Рисунок 1.4. – Пример графического изображения связи обобщения

Для построения диаграмм вариантов использования удобно использовать специализированное программное обеспечение, рассмотрение которого приведено далее.

Группировка элементов в пакеты

Все размещенные на диаграмме графические фигуры называют элементами диаграммы. В ряде случаев элементы могут быть сгруппированы в пакеты по функциональному признаку. На диаграмме вариантов допускается применять дубликаты описанных элементов для упрощения ее восприятия.

Пакет – основной способ организации элементов модели в языке UML. Каждый пакет владеет всеми своими элементами, включенными в него. Про внутренние элементы пакета говорят, что они принадлежат пакету или входят в него. При этом каждый

элемент может принадлежать только одному пакету. В свою очередь, одни пакеты могут быть вложены в другие пакеты. В этом случае первые называются подпакетами, поскольку все элементы подпакета будут принадлежать более общему пакету. Тем самым для элементов модели задается отношение вложенности пакетов, которое представляет собой иерархию.

Для графического изображения пакетов на диаграммах применяется специальный графический символ – большой прямоугольник с небольшим прямоугольником, присоединенным к левой части верхней стороны первого. Внутри большого прямоугольника может записываться информация, относящаяся к данному пакету. Если такой информации нет, то внутри большого прямоугольника записывается имя пакета, которое должно быть уникальным в пределах рассматриваемой диаграммы. Если же такая информация имеется, то имя пакета записывается в верхнем маленьком прямоугольнике. Пример графического изображения пакетов показан на рисунке 1.5.

Рисунок 1.5. – Пример графического изображения пакетов

Программное обеспечение для построения UML-диаграмм

Для построения UML-диаграмм рекомендуется использование сервиса diagrams.net, доступного как в виде онлайн сервиса *https://app.diagrams.net/*, так и в виде настольного приложения *draw.io*.

Diagrams.net – свободно распространяемое приложение для создания диаграмм для рабочих процессов, BPM, организационных, сетевых диаграмм.

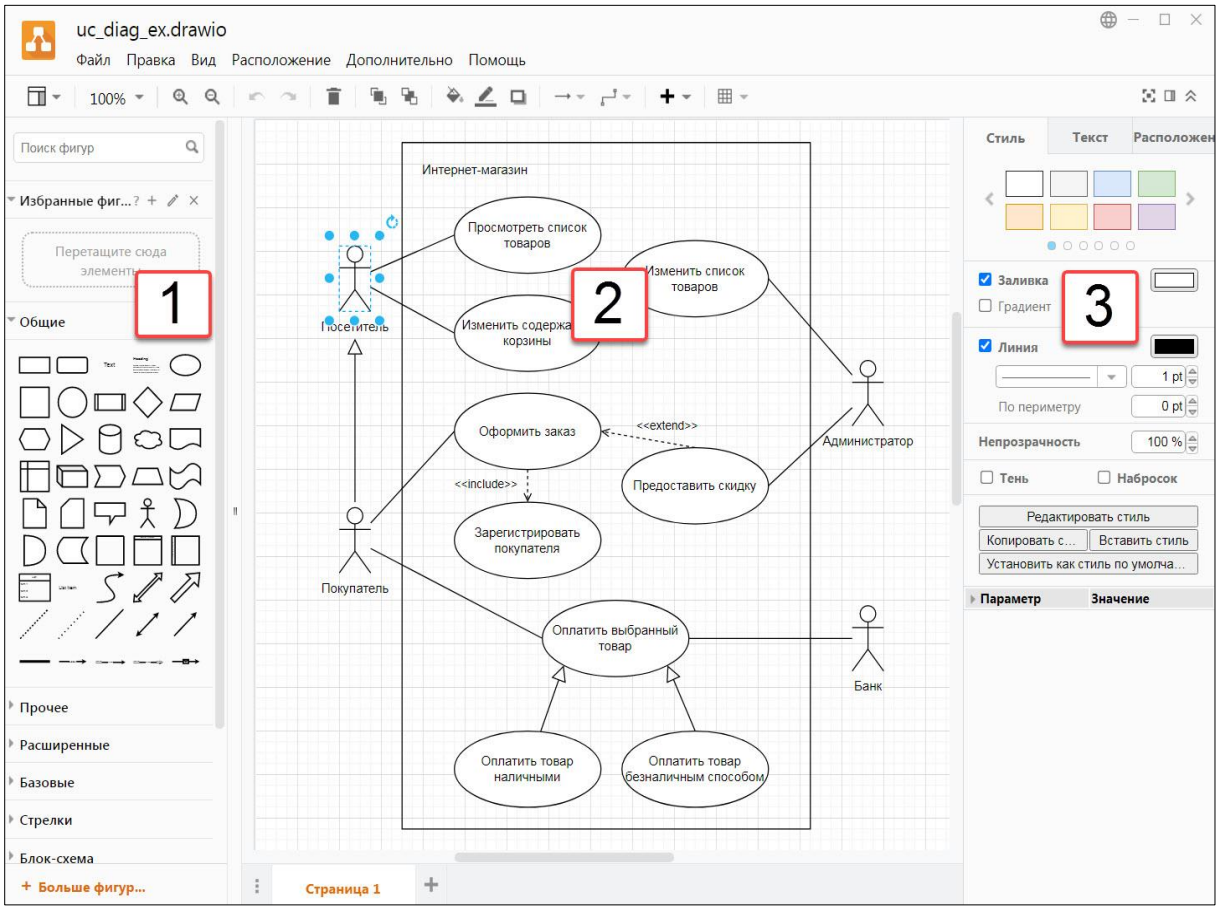
Интерфейс приложения Diagrams.net представлен на рисунке 1.6.

Для использования сервиса в облачном режиме необходимо перейти по ссылке *https://app.diagrams.net/*. Для использования настольной версии сервиса необходимо скачать соответствующие версии операционной системы файл из репозитория на GitHub по ссылке *https://github.com/jgraph/drawio-desktop/releases*.

Далее показан пример создания диаграммы в настольно варианте сервиса версии 15.7.3.

После запуска приложения отобразится диалоговое окно, показанное на рисунке 1.7, с вариантами создания новой диаграммы и использования уже существующей. Для создания новой диаграммы необходимо выбрать пункт «Создать новую диаграмму».

После этого откроется мастер создания новой диаграммы, представленный на рисунке 1.8. Здесь необходимо указать название файла с диаграммой и его тип. Также можно выбрать один из нескольких готовых шаблонов диаграмм. Укажем имя файла диаграммы, содержащее название дисциплины, свою фамилию, группу и номер лабораторной работы, разделяя эти данные символом нижнего подчеркивания. Тип файла диаграммы – «XML файл (.drawio)». Шаблон использовать не будем и выберем базовый вариант «Пустая диаграмма».



1 – элементы диаграмм; 2 – рабочая область для построения диаграммы;
3 – параметры элементов диаграммы

Рисунок 1.6. – Интерфейс приложения Diagrams.net

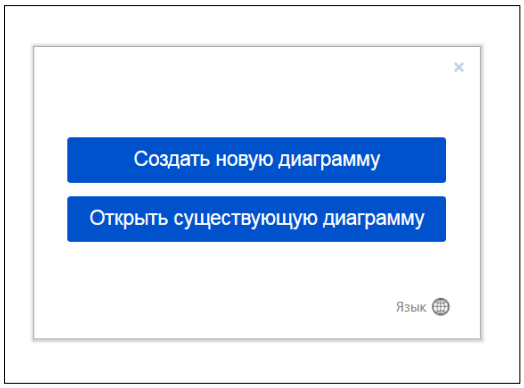


Рисунок 1.7. – Диалоговое окно с вариантами создания новой диаграммы
и использования уже существующей

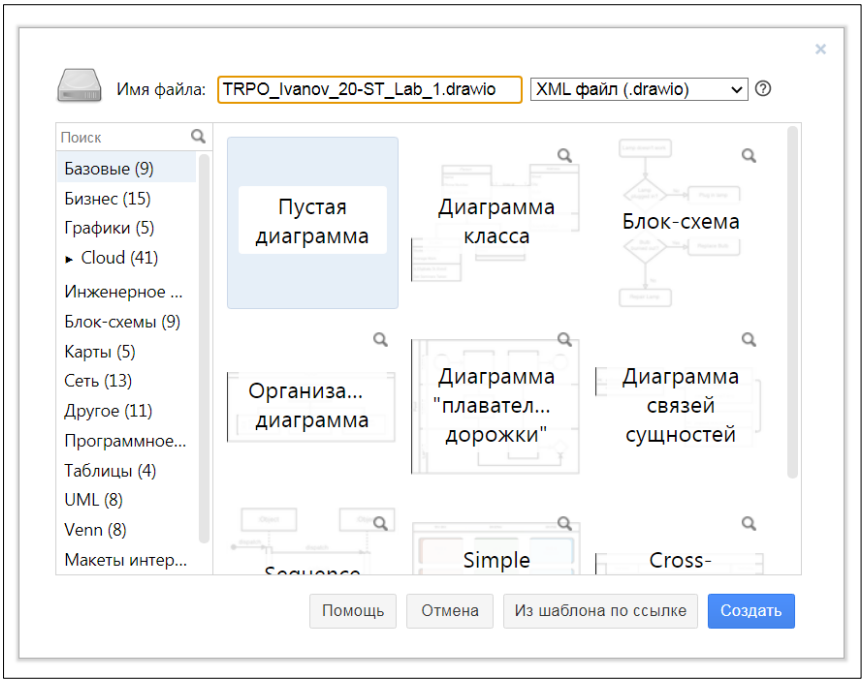


Рисунок 1.8. – Мастер создания новой диаграммы

После подтверждения создания диаграммы нажатием кнопки «Создать» в правом нижнем углу диалогового окна будет отображен интерфейс для работы с новой диаграммой, представленный на рисунке 1.9.

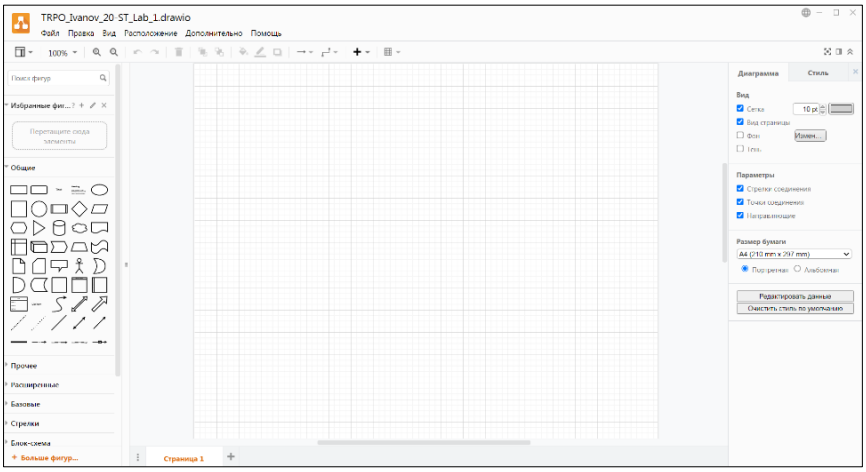


Рисунок 1.9. – Интерфейс для работы с новой диаграммой

Интерфейс для работы с диаграммой можно разделить на три части: две боковых панели и центральная рабочая область. Левая боковая панель содержит элементы для построения диаграммы. Правая боковая панель предоставляет доступ к свойствам выбранного элемента. Центральная рабочая область отражает текущий вид диаграммы. Построение диаграммы сводится к перетаскиванию необходимых элементов из левой боковой панели, расстановке их на рабочей области и соединении необходимыми связями, а также редактированию свойств отдельных элементов в правой боковой панели, если это необходимо. Процесс создания диаграммы показан на рисунке 1.10.

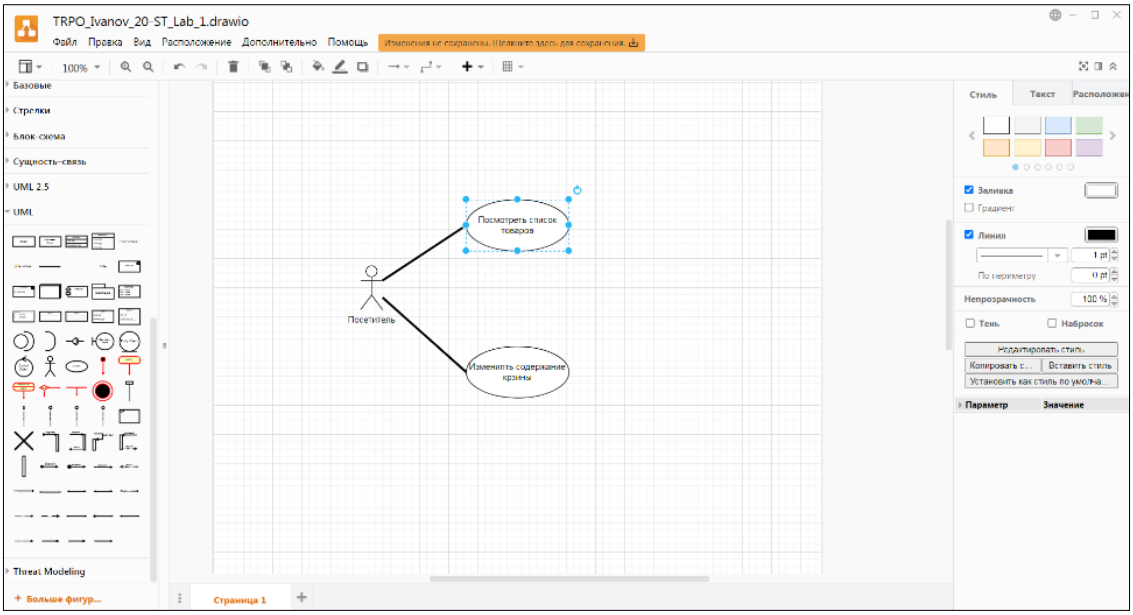


Рисунок 1.10. – Процесс создания диаграммы

После завершения создания диаграммы, а также в процессе ее создания для сохранения промежуточных результатов необходимо сохранить файл диаграммы. Для этого в верхнем меню необходимо выбрать пункт «Файл» и подпункт «Сохранить» или воспользоваться клавиатурным сокращением клавиш «Ctrl + S». После этого требуется выбрать место расположения файла в диалоговом окне и подтвердить сохранение кнопкой «Сохранить».

После этого диаграмма будет сохранена и можно продолжить работу с ней после перезапуска приложения.

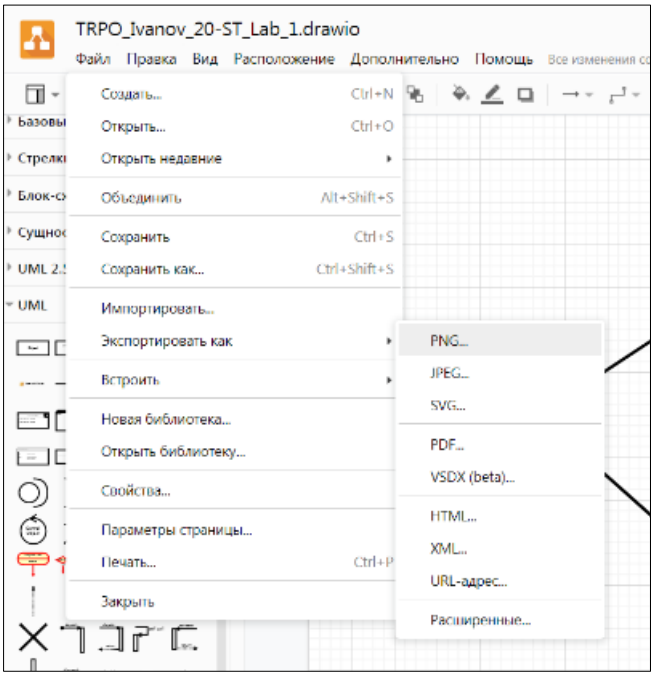


Рисунок 1.11. – Структура выбора пунктов меню для экспорта диаграммы в графический формат

Для получения графического файла с диаграммой воспользуемся функцией экспорта. Для этого в верхнем меню выбираем пункт «Файл» и в подпункте «Экспортировать как» – необходимый графический формат. Для примера проведем экспорт в формат PNG, выбрав соответствующий пункт. Структура выбора пунктов меню для экспорта диаграммы в графический формат – см. рисунок 1.11.

После выбора требуемого графического формата будут отображены настройки экспорта, показанные на рисунке 1.12. Здесь можно выбрать масштаб, формат границ и настройки фона, а также указать область экспорта.

Для примера оставим параметры экспорта по умолчанию. После этого нажатием кнопки «Экспортировать» подтвердим экспорт. Далее отобразится диалоговое окно для выбора расположения файла графического представления диаграммы и его имени.

Выбрав требуемое расположения файла и его имя, завершим экспорт нажатием кнопки «Сохранить». После этого будет создан графический файл, содержащий созданную диаграмму.

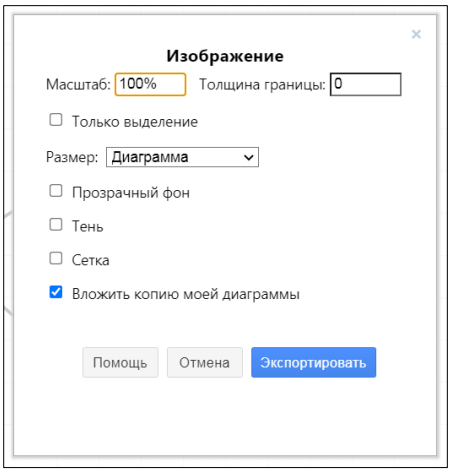


Рисунок 1.12. – Настройки экспорта диаграммы

Пример диаграммы вариантов использования

На рисунке 1.13 приведена диаграмма вариантов использования для системы продажи товаров в интернет-магазине. Эта диаграмма может быть использована при разработке и реализации интернет-проектов.

В качестве основного актера описываемой системы можно рассматривать посетителя интернет-магазина. Он может просматривать список товаров интернет-магазина, помещать товар в виртуальную корзину и изменять содержимое этой корзины. Посетитель может стать покупателем, если он принимает решение об оформлении заказа на покупку выбранных им товаров. В качестве базового варианта использования данной модели может служить вариант использования *«Оформить заказ»*.

В качестве других актеров рассматриваемой системы могут выступать администратор интернет-магазина и банк. При этом администратор может изменять список товаров и задавать условия для предоставления скидки, а банк – принимать оплату за выбранный покупателем товар.

Поскольку при оформлении заказа на покупку товара необходима регистрация покупателя, и эта функциональность выполняется всегда, она может быть выделена в отдельный вариант использования, который будет связан с базовым отношением включения. С другой стороны, при оформлении заказа постоянному покупателю может быть предоставлена специальная скидка. Это требование может быть также представлено в качестве отдельного варианта использования *«Предоставить скидку»*, который будет связан с базовым отношением расширения.

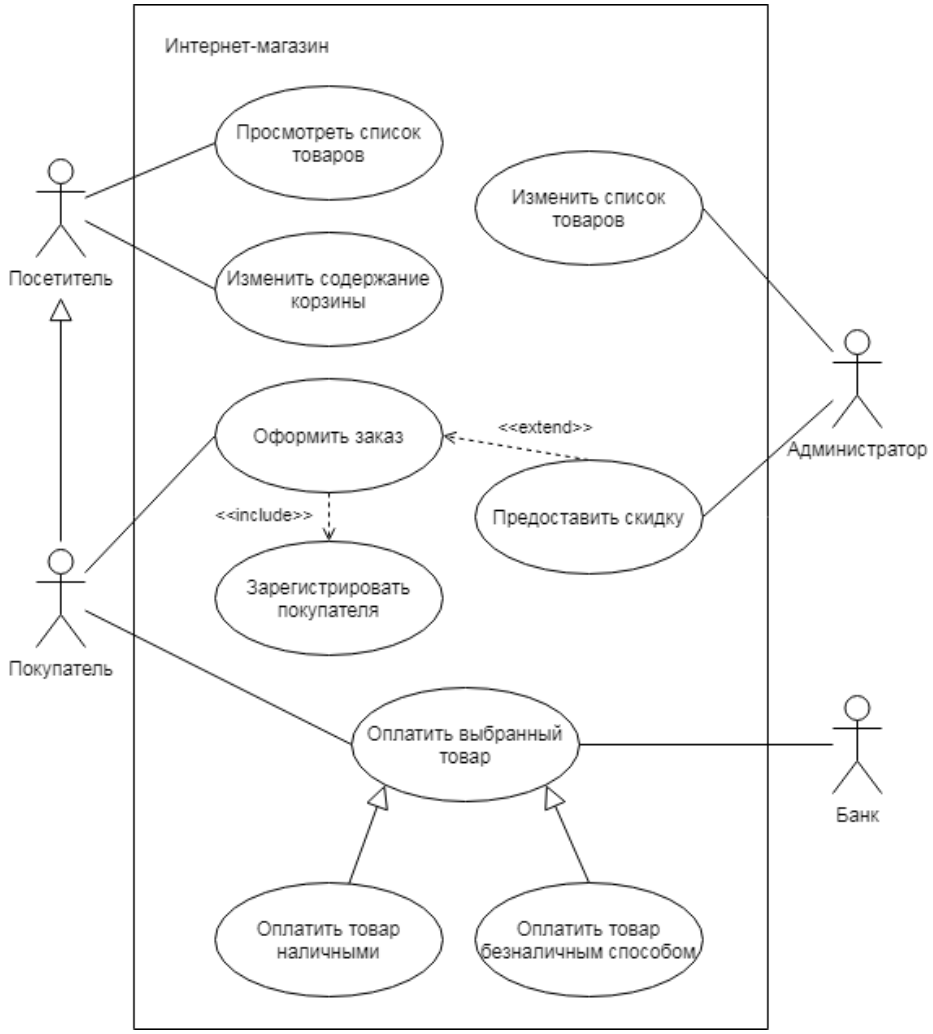


Рисунок 1.13. – Диаграмма вариантов использования для интернет-магазина

Контрольные вопросы

1. Поясните понятие «Диаграмма вариантов использования».
2. Перечислите и охарактеризуйте элементы диаграммы вариантов использования.
3. Назовите цели применения диаграммы вариантов использования.
4. Перечислите и охарактеризуйте возможные типы связей на диаграмме вариантов использования.
5. Назовите второстепенные элементы для диаграммы вариантов использования.

Задание

Разработать и описать диаграмму вариантов использования по индивидуальному заданию, представленному в приложении А. Варианты индивидуальных заданий выдаются преподавателем.

Разработанная диаграмма вариантов использования должна охватывать всю предметную область информационной системы, указанной в индивидуальном задании. При реализации диаграммы необходимо использовать все элементы, применимые для диаграмм вариантов использования.

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Индивидуальное задание.
4. Разработанная диаграмма вариантов использования.
5. Описание диаграммы вариантов использования, ее элементов и связей.
6. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab1-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab1-10-IT-Ivanov.zip

Литература

1. Новиков, Ф.А. Моделирование на UML. Теория, практика, видеокурс / Ф.А. Новиков, Д.Ю. Иванов. – СПб. : Проф. лит., Наука и Техника, 2010. – 640 с.

2. Буч, Гр. Язык UML. Руководство пользователя / Грейди Буч, Джеймс Рамбо, Айвар Джекобсон ; пер. с англ. Н. Мухин. – 2-е изд. – М. : ДМК Пресс, 2006. – 496 с. : ил.

3. Леоненков, А.В. Самоучитель UML / А.В. Леоненков. – СПб..: БХВ-Петербург, 2004. – 432 с..: ил.

4. Рамбо, Дж. UML: специальный справочник / Джеймс Рамбо, Айвар Джекобсон, Грейди Буч. – СПб. : Питер, 2002. – 656 с. : ил.

5. Леоненков, А.В. Самоучитель UML / А.В. Леоненков. – 2-е изд. – СПб. : БХВ-Петербург, 2007. – 576 с. : ил.

6. Буч, Гр. Введение в UML от создателей языка / Грейди Буч, Джеймс Рамбо, Айвар Джекобсон ; пер. с англ. Н. Мухин. – 2-е изд. – М. : ДМК Пресс, 2010. – 496 с. : ил.

7. Бабич, А.В. UML: Первое знакомство / А.В. Бабич. – М. : Интуит, 2016. – 209 с.

Лабораторная работа 2
РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ

Цель работы: ознакомится со структурой, порядком оформления и назначением технического задания. Разработать техническое задание для индивидуального варианта предметной области.

Ход работы:

1) изучить краткие теоретические сведения;
2) ответить на контрольные вопросы;
3) выполнить практическое задание;
4) оформить отчет;
5) защитить лабораторную работу.

Термины и определения

Техническое задание (ТЗ) как термин в области информационных технологий – юридически значимый документ, содержащий исчерпывающую информацию, необходимую для постановки задач исполнителям на разработку, внедрение или интеграцию программного продукта, информационной системы, сайта, портала либо прочего ИТ-сервиса.

Техническое задание является юридическим документом и как приложение включается в договор между заказчиком и исполнителем на проведение проектных работ и является его основой: определяет порядок и условия работ, в т.ч. цель, задачи, принципы, ожидаемые результаты и сроки выполнения. Т.е. должны быть объективные критерии, по которым можно определить, сделан ли тот или иной пункт работ или нет. Все изменения, дополнения и уточнения формулировок ТЗ обязательно согласуются с заказчиком и им утверждаются. Это важно, т.к. в случае обнаружения в процессе решения проектной задачи неточностей или ошибочности исходных данных возникает необходимость определения степени вины каждой из сторон-участниц разработки, распределения понесенных в связи с этим убытков.

Техническое задание позволяет:

исполнителю понять суть задачи, показать заказчику «технический облик» будущего изделия, программного изделия или автоматизированной системы; спланировать выполнение проекта и работать по намеченному плану; отказаться от выполнения работ, не указанных в ТЗ;

заказчику осознать, что именно ему нужно; требовать от исполнителя соответствия продукта всем условиям, оговоренным в ТЗ;

обеим сторонам – представить готовый продукт; выполнить попунктную проверку готового продукта (приемочное тестирование – проведение испытаний); избежать ошибок, связанных с изменением требований (на всех стадиях и этапах создания, за исключением испытаний).

В зависимости от ожиданий заказчика существует три альтернативы для выбора шаблона ТЗ. Если заказчик требует оформления документации в соответствии с государственным стандартом, выбор делается в сторону стандарта ГОСТ 34.602-8 – Информационная технология – Комплекс стандартов на автоматизированные системы – Техническое

задание на создание автоматизированной системы (или 19.201-78 – Единая система программной документации – Техническое задание – Требования к содержанию и оформлению). Подготовка технического задания по ГОСТ 34.602-89 требует значительных временных затрат.

Если поставлены сжатые сроки подготовки ТЗ и заказчик не требует оформления документации в соответствии с государственным стандартом, то можно использовать шаблон технического задания по стандарту IEEE Std 830. Стандарт IEEE Std 830 предполагает, что детальные требования могут быть обширными и не существует оптимальной структуры для всех систем. По этой причине стандартом рекомендуется обеспечивать такое структурирование детальных требований, которое делает их оптимальными для понимания.

Существует и третья альтернатива для выбора шаблона ТЗ, когда заказчик предлагает использовать принятый в компании корпоративный шаблон для описания требований к информационным системам.

В ТЗ можно включить:

- введение;
- основания для разработки;
- назначение разработки;
- требования к программе или программному изделию;
- требования к программной документации;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- приложения.

Содержание разделов

1. В разделе «Введение» указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

2. В разделе «Основания для разработки» должны быть указаны:

- документ (документы), на основании которых ведется разработка;
- организация, утвердившая этот документ, и дата его утверждения;
- наименование и (или) условное обозначение темы разработки.

3. В разделе «Назначение разработки» должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

4. Раздел «Требования к программе или программному изделию» должен содержать следующие подразделы:

- требования к функциональным характеристикам;
- требования к надежности;
- условия эксплуатации;
- требования к составу и параметрам технических средств;
- требования к информационной и программной совместимости;
- требования к маркировке и упаковке;
- требования к транспортированию и хранению;
- специальные требования.

4.1. В подразделе «Требования к функциональным характеристикам» должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т.п.

4.2. В подразделе «Требования к надежности» должны быть указаны требования к обеспечению надежного функционирования (обеспечения устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т.п.).

4.3. В подразделе «Условия эксплуатации» должны быть указаны условия эксплуатации (температура окружающего воздуха, относительная влажность и т.п. для выбранных типов носителей данных), при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала.

4.4. В подразделе «Требования к составу и параметрам технических средств» указывают необходимый состав технических средств с указанием их основных технических характеристик.

4.5. В подразделе «Требования к информационной и программной совместимости» должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования и программным средствам, используемым программой.

При необходимости должна обеспечиваться защита информации и программ.

4.6. В подразделе «Требования к маркировке и упаковке» в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки.

4.7. В подразделе «Требования к транспортированию и хранению» должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях.

5. В разделе «Требования к программной документации» должен быть указан предварительный состав программной документации и (при необходимости) специальные требования к ней.

6. В разделе «Технико-экономические показатели» должны быть указаны: ориентировочная экономическая эффективность, предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

7. В разделе «Стадии и этапы разработки» устанавливают необходимые стадии разработки, этапы и содержание работ (перечень программных документов, которые должны быть разработаны, согласованы и утверждены), а также, как правило, сроки разработки и определяют исполнителей.

8. В разделе «Порядок контроля и приемки» должны быть указаны виды испытаний и общие требования к приемке работы.

9. В приложениях к техническому заданию (при необходимости) приводят:

– перечень научно-исследовательских и других работ, обосновывающих разработку;

– схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые могут быть использованы при разработке;

– другие источники разработки.

Далее приведен пример оформления технического задания на разработку информационной системы кинотеатра, выполненный по ГОСТ 19.201-78.

Пример оформления технического задания

Техническое задание

1 Введение

1.1 Наименование программы

Наименование программы – информационно-справочная система «Кинотеатр».

1.2 Краткая характеристика области применения

Информационно-справочная система «Кинотеатр» предназначена для организации продажи билетов в кинотеатрах, состоящих из нескольких залов. Для каждого зала в кинотеатре должен быть предусмотрен только один оператор-кассир, выполняющий продажу и возврат билетов посетителям.

2 Основания для разработки

Основанием для разработки является Договор №111. Договор утвержден Директором ООО «Кино» Ивановым Иваном Ивановичем, именуемым в дальнейшем Заказчиком, и Петровым Петром Петровичем, именуемым в дальнейшем Исполнителем.

Наименование темы разработки – «Разработка информационно-справочной системы «Кинотеатр».

3 Назначение разработки

Программа будет использоваться в кинотеатре двумя группами пользователей: оператор-кассир и посетитель.

3.1 Функциональное назначение

Для посетителя кинотеатра программа предоставляет возможность просмотра текущей заполненности зала (отображение свободных и занятых мест).

Для оператора-кассира программа позволяет помечать места в зале как «занятые» (при продаже билетов) или «свободные» (в случае возврата билетов).

3.2 Эксплуатационное назначение

Программа должна эксплуатироваться в зале ожидания кинотеатра. Запущенная с правами посетителя, она может транслироваться на большие мониторы (для посетителей). С правами кассира программа запускается на компьютере кассира.

4 Требования к программе или программному изделию

4.1 Требования к функциональным характеристикам

4.1.1 Требования к составу выполняемых функций

После запуска программы пользователю должна отображаться форма ввода логина и пароля, показанная на рисунке 1.

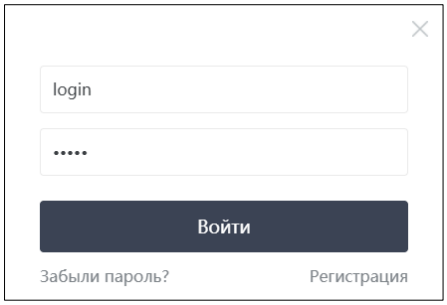


Рисунок 1. – Форма ввода логина и пароля

В системе существует два типа пользователей – кассир и посетитель. Программа должна проверять тип пользователя и открывать соответствующий интерфейс.

Для посетителя кинотеатра программа должна предоставлять следующие возможности:

– просмотр расписания фильмов;
– просмотр заполненности зала для конкретного проката фильма.

При просмотре расписания должна выводиться таблица, каждая строка которой описывает прокат фильма и содержит следующую информацию:

– дата и время проката;
– название фильма;
– возрастные ограничения.

Макет окна просмотра прокатов для посетителя показан на рисунке 2.

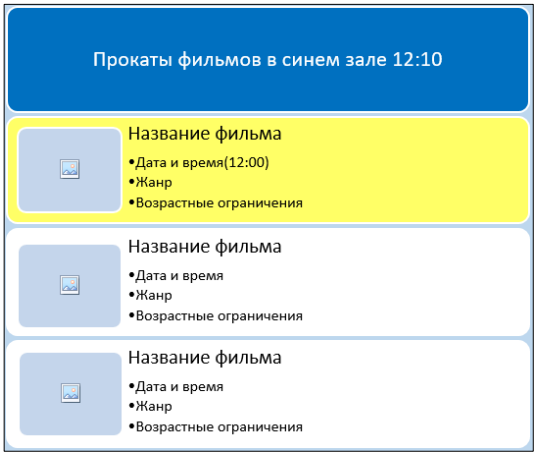


Рисунок 2. – Макет окна просмотра прокатов для посетителя

В верхней строке должно отображаться название зала и текущее время. Уже начатые сеансы должны быть помечены желтым цветом (на них еще можно купить билеты, с опозданием). После завершения проката строка таблицы должна автоматически удаляться (отображаются только текущие и будущие прокаты).

При просмотре заполненности зала, посетителю должна выводиться схема кинотеатра, на которой показано:

– положение экрана;
– ряды, состоящие из мест;
– свободные места (выделены белым цветом) и занятые (выделены красным).

Макет схемы зала приведен на рисунке 3.

Рисунок 3. – Макет схемы зала

Для оператора-кассира программа должна предоставлять все функции, предоставляемые посетителю, а также возможности:

– выбора группы из свободных или занятых мест;
– пометки выбранных мест как «занятых» или «свободных»;
– изменение расписания проката фильмов.

Окно расписания проката для оператора, помимо таблицы, должно содержать кнопки «Добавить» и «Удалить», как показано на рисунке 4.

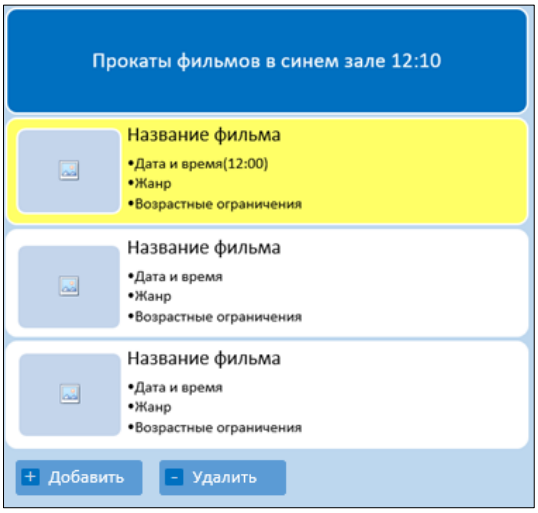


Рисунок 4. – Макет окна расписания проката для оператора

Для удаления сеанса оператор выбирает строку таблицы и нажимает кнопку «Удалить». Удалить можно только прокат, на который нет проданных билетов.

Для изменения информации о прокате оператор выполняет двойной клик мышью по изменяемому полю – после этого вводит в поле новое значение. Изменять можно только прокаты, показ которых еще не начат.

Для добавления поля оператор нажимает кнопку «Добавить», в конец таблицы добавляется новая строка с пустыми полями. После заполнения текущего поля оператор

может нажимать кнопку Tab для перехода на следующее поле. После нажатия кнопки Tab в последней колонке проверяется корректность введенных данных и выполняется сохранение информации (добавление в базу). Если введены некорректные значения, то соответствующее поле таблицы в интерфейсе оператора выделяется красным цветом.

Все изменения, выполняемые оператором-кассиром, должны отражаться на схеме для посетителей.

4.1.2 Требования к организации входных и выходных данных

Данные о прокатах фильмов и пользователях хранятся в базе данных. Ввод данных в базу (логины и пароли) осуществляет администратор, выполняющий поставку программного обеспечения заказчику.

После установки программы ввод данных в систему проводит только кассир, проверка данных выполняется на стороне клиента:

- дата и время должны быть записаны в формате: «ДД.ММ.ГГГГ ЧЧ:ММ»;
- название – последовательность не более чем из 200 любых символов;
- жанр – последовательность не более чем из 200 любых символов;
- возрастные ограничения – «+».

4.1.3 Требования к временным характеристикам

После изменения кассиром данных, находящихся в базе данных, новая информация на клиентах отображается не позднее, чем через 5 с.

4.2 Требования к надежности

Вероятность безотказной работы системы должна составлять не менее 99% при условии исправности сети (связи приложений оператора и посетителя с базой данных).

4.2.1 Требования к обеспечению надежного (устойчивого) функционирования программы

В связи с тем, что в базе данных хранятся данные о совершенных клиентами покупках, базу данных необходимо резервировать.

Надежное (устойчивое) функционирование программы должно быть обеспечено выполнением заказчиком совокупности организационно-технических мероприятий, перечень которых приведен ниже:

- организация бесперебойного питания технических средств;
- использование лицензионного программного обеспечения.

4.2.2 Время восстановления после отказа

Время восстановления после отказа, вызванного сбоем электропитания технических средств, нефатальным сбоем операционной системы, не должно превышать 10 мин при условии соблюдения условий эксплуатации технических и программных средств.

Время восстановления после отказа, вызванного неисправностью технических средств, фатальным сбоем операционной системы, не должно превышать времени, требуемого на устранение неисправностей технических средств и переустановки программных средств.

4.2.3 Отказы из-за некорректных действий оператора

Отказы программы возможны вследствие некорректных действий оператора (пользователя) при взаимодействии с операционной системой. Во избежание возникновения отказов программы по указанной выше причине следует обеспечить работу пользователя без предоставления ему административных привилегий.

4.3 Условия эксплуатации

Программа (клиент) запускается на компьютере оператора-кассира и компьютере, доступном посетителям кинотеатра. База данных находится на стороннем компьютере. Должна существовать устойчивая связь по сети между клиентами и базой данных.

Окно программы должно быть открыто на весь экран, не должно быть возможности закрыть, свернуть приложение или запустить любое стороннее программное обеспечение. Запуск программы должен осуществляться сразу после старта операционной системы.

4.3.1 Климатические условия эксплуатации

Требования к климатическим условиям не предъявляются.

4.3.2 Требования к видам обслуживания

Программа не требует проведения каких-либо видов обслуживания.

4.3.3 Требования к численности и квалификации персонала

При установке и настройке системы необходим системный администратор. В процессе эксплуатации с программой работают оператор-кассир и посетитель кинотеатра.

Системный администратор должен иметь высшее профильное образование и сертификаты компании-производителя операционной системы.

Пользователь программы (оператор) должен обладать практическими навыками работы с графическим пользовательским интерфейсом операционной системы.

К квалификации посетителя кинотеатра специальные требования не предъявляются.

4.4 Требования к составу и параметрам технических средств

Компьютер для установки информационно-справочной системы, включающий в себя:

- процессор с тактовой частотой, не менее 2400 МГц;
- оперативную память объемом, не менее 6 ГБ;
- видеокарту с объемом видеопамяти, не менее 2 ГБ;
- монитор;
- мышь;
- клавиатуру.

4.5 Требования к информационной и программной совместимости

Требования к информационной и программной совместимости не предъявляются.

4.6 Требование к маркировке и упаковке

Программное изделие передается по сети Интернет в виде архива – загружается с официального сайта производителя. Специальные требования к маркировке не предъявляются. Для проверки подлинности программного обеспечения рекомендуется проверять контрольные суммы загруженных файлов со значениями, указанными на официальном сайте.

4.7 Требования к транспортированию и хранению

Требования к транспортированию и хранению не предъявляются.

4.8 Специальные требования

Программа должна обеспечивать взаимодействие с пользователем посредством графического пользовательского интерфейса, разработанного согласно рекомендациям компании-производителя операционной системы.

5 Требования к программной документации

Состав программной документации:

- техническое задание;
- программа и методика испытаний;
- руководство системного программиста;
- руководство оператора;
- руководство программиста;
- ведомость эксплуатационных документов.

6 Технико-экономические показатели

Программа «Кинотеатр» пригодна для небольших кинотеатров, не рассматривающих возможность продажи билетов через Internet.

Функциональность программы совпадает с аналогами, установленными в других кинотеатрах.

В связи с тем что из года в год кинотеатров не становится значительно больше, а количество маленьких кинотеатров даже снижается, не стоит ожидать роста годовой потребности. Однако в случае бесплатного распространения программы потребность в ней может быть весьма высокой. Экономический эффект при этом может быть обеспечен за счет платной установки системы и присутствующей в ней рекламе.

7 Стадии и этапы разработки

Разработка должна быть проведена в три стадии:

- техническое задание;
- технический и рабочий проекты;
- внедрение.

На стадии «Техническое задание» должен быть выполнен этап разработки, согласования и утверждения настоящего технического задания.

На стадии «Технический и рабочий проекты» должны быть выполнены перечисленные ниже этапы работ:

– разработка программы;
– разработка программной документации;
– испытания программы.

На стадии «Внедрение» должен быть выполнен этап разработки «Подготовка и передача программы».

Согласно Договору, Исполнитель обязан разработать и установить информационно-справочную систему «Кинотеатр» на оборудовании Заказчика не позднее 01.01.2021, предоставить исходные коды и документацию к разработанной системе не позднее 01.12.2020.

8 Порядок контроля и приемки

Приемо-сдаточные испытания программы должны проводиться согласно разработанной исполнителем и согласованной заказчиком «Программе и методике испытаний».

Ход проведения приемо-сдаточных испытаний заказчик и исполнитель документируют в протоколе испытаний.

На основании протокола испытаний исполнитель совместно с заказчиком подписывают акт приемки-сдачи программы в эксплуатацию.

Контрольные вопросы

1. Дайте определение понятию «техническое задание».
2. Охарактеризуйте техническое задание как юридический документ.
3. Назовите возможности теченского задания.
4. Для чего необходимо разрабатывать техническое задание?
5. Опишите структуру технического задания.
6. Назовите и охарактеризуйте шаблоны для технического задания.
7. Назовите и охарактеризуйте основные разделы технического задания.

Задание

На основе варианта индивидуального задания из прошлой лабораторной работы сформулировать техническое задание, которое будет включать: этапы разработки, необходимую функциональность, разделение ресурсов проекта, сроки выполнения этапов. Варианты индивидуальных заданий представлены в приложении А.

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Описание индивидуального варианта задания.
4. Разработанное техническое задание.
5. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab2-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab2-10-IT-Ivanov.zip

Литература

1. Трегубов, С.И. Основы конструирования электронных средств: техническое задание : учеб. пособие / С.И. Трегубов, А.А. Левицкий. – Красноярск : СФУ, 2020. – 178 с. : табл.

2. Красносельский, С.А. Основы проектирования : учеб. пособие / С.А. Красносельский. – М. : Директ-Медиа, 2014. – 232 с.

3. Расчет и конструирование элементов оборудования : учеб. пособие / Е.А. Соловьев [и др.]. – Красноярск : СФУ, 2019. – 186 с. : ил., граф., схемы.

4. Хабибуллин, И.Ш. Программирование на языке высокого уровня. C/C++ / И.Ш. Хабибуллин. – СПб. : БХВ-Петербург, 2006. – 512 с.

5. Зараменских, Е.П. Информационные системы: управление жизненным циклом : учеб. и практикум / Е.П. Зараменских. – М. : Юрайт, 2019. – 431 с.

6. Горев, А.Э. Информационные технологии в профессиональной деятельности / А.Э. Горев. – 2-е изд., перераб. и доп. – М. : Юрайт, 2020. – 289 с. – (Профессиональное образование).

Лабораторная работа 3
РАЗРАБОТКА ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ

Цель работы: ознакомиться с проектированием деятельности на основе языка UML и разработать диаграмму деятельности для программного обеспечения.

Ход работы:

1) изучить краткие теоретические сведения;
2) ответить на контрольные вопросы;
3) разработать диаграмму деятельности в соответствии с вариантом индивидуального задания;
4) оформить отчет;
5) защитить лабораторную работу.

Термины и определения

Поток (поток управления) – последовательность программных инструкций (данных), которые выполняются программой для реализации какой-либо задачи. В одной программе может выполняться один поток или несколько. При одновременном вызове нескольких потоков в одной программе потоки могут выполняться параллельно.

Поток объектов/данных – один из видов потока управления, который выполняет определенные преобразования входного потока объектов/данных и передает результаты обработки информации в виде потоков объектов/данных для других блоков на диаграмме.

Узел – обозначение элементов диаграммы деятельности.

Спецификация – условие или правило для определенного узла, только при выполнении/соответствию которого можно перейти к следующему узлу.

Дуга – направленная связь, которая имеет в качестве источника и цели некоторые узлы. При этом дуга, источник и цель дуги должны относиться к одной деятельности.

Деятельность – совокупность последовательного и параллельного выполнения узлов, соединенных между собой дугами, которые идут от выходов одного узла ко входам другого. Также деятельность может быть представлена совокупностью более мелких деятельностей.

Маркер – абстрактная точка, указывающая на место выполнения процесса. На диаграмме маркер не отображается, данный термин необходим для удобства описания динамического процесса. Маркер переходит от одного узла к другому. Если маркер пустой (не содержит никакой дополнительной информации), тогда он называется маркером управления.

Булево выражение – логическое выражение, результатом вычисления которой является «истина» или «ложь». Например: «2+2=4 вычисление верно?» Варианты ответа: «Да» – истинный / «Нет» – ложный.

Сторожевое условие – логическое условие, представляющее собой булево выражение.

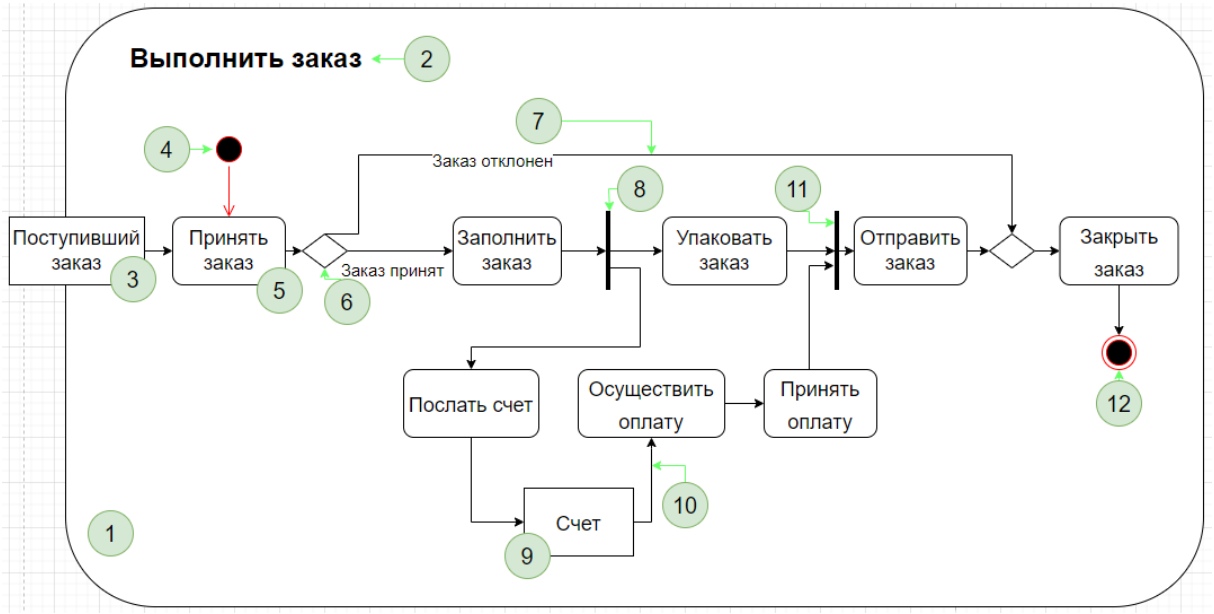
Начальный узел – узел, в котором при вызове деятельности начинается поток.

Узел соединения – узел, который синхронизирует несколько потоков.

Диаграмма деятельности (англ. *Activity diagram*) – диаграмма UML, которая демонстрирует логику и последовательность действий при выполнении программы.

На диаграмме выполнение деятельности представляется совокупностью последовательного и параллельного выполнения узлов (в т.ч. действий), соединенных между собой дугами.

Упрощенный пример диаграммы деятельности для выполнения заказа представлен на рисунке 3.1, на которой изображены и подписаны графические элементы и потоки (связи) между ними.



1 – деятельность; 2 – название деятельности; 3 – узел данных предыдущей деятельности; 4 – начальный узел (узел начала деятельности); 5 – действие (узел деятельности); 6 – узел решения; 7 – поток управления (дуга деятельности); 8 – узел объединения; 9 – узел данных(объектов); 10 – поток данных (объектов); 11 –узел разделения; 12 – конечный узел (узел завершения)

Рисунок 3.1. – Пример диаграммы деятельности «Выполнение заказа»

Рассмотрим основные графические обозначения и элементы диаграммы деятельности:

1. ▭ Прямоугольники с закругленными углами – деятельности и действия. **Узел деятельности** представляет собой саму деятельность, если диаграмма деятельности состоит из более чем одной деятельности, и действие, если диаграмма деятельности представлена одной деятельностью с подробным описанием действий выполняемых в ней. Узел деятельности имеет имя, которое может повторяться в пределах одной диаграммы с целью изображения многократного выполнения одного и того же действия или деятельности. На диаграмме узлы деятельности обычно связаны между собой дугами деятельности.

2. Сплошная линия со стрелкой – **дуга деятельности**, которая может быть как с именем, так и без (рисунок 3.2).

Существует две разновидности дуги деятельности.

Поток управления – дуга, которая связывает между собой два узла деятельности и по которой передаются только маркеры управления [1].

а – дуга без имени; *б* – дуга с именем

Рисунок 3.2. – Обозначение на диаграмме дуги деятельности

В качестве примера можно рассмотреть фрагмент диаграммы, где осуществляется передача управления от действия «*Заполнить заказ*» к действию «*Отправить счет*». Дуга деятельности между ними является потоком управления, который указывает на то, что по завершении действия «*Заполнить заказ*» будет вызвано действие «*Отправить счет*» (рисунок 3.3).

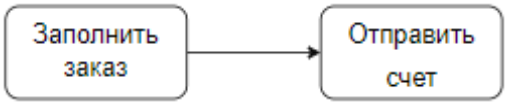


Рисунок 3.3. – Дуга потока управления

Поток объектов представляется в форме дуги деятельности, по которой передаются только объекты или данные [1]. В качестве примера можно рассмотреть фрагмент диаграммы деятельности передачи данных «*Заказ*» от действия «*Оформить заказ*» к действию «*Принять заказ*», представленный на рисунке 3.4. Здесь обе линии со стрелками являются дугами потока данных.

Рисунок 3.4. – Дуги потока объектов

3. ◇ Ромбы – решения. **Узел решения** предназначен для определения правила ветвления различных вариантов дальнейшего развития сценария. В точку ветвления входит ровно один переход, а выходит – два или более. Для каждого исходящего перехода задается булевское выражение, которое вычисляется только один раз при входе в точку ветвления.

Дуги, входящие в узел решения и выходящие из него, должны быть все или потоками управления, или потоками объектов [1].

В качестве примера можно привести фрагмент диаграммы деятельности, на которой изображен узел решения, следующий за действием «*Сообщить об отсутствии товара*». Ветвление основывается на том, согласился ли клиент оставить заявку на товар или отказался. В первом случае управление передастся действию «*Оставить заявку на товар*», во втором – действию «*Отказаться от товара*» (рисунок 3.5).

Каждая из выходящих дуг должна иметь некоторое сторожевое условие, которое при оценивании должно возвращать значение «истина» или «ложь». При оценивании всех сторожевых условий узла решения только одно из них может принимать значение «истина». Это условие будет соответствовать единственной дуге, по которой маркер проследует далее.

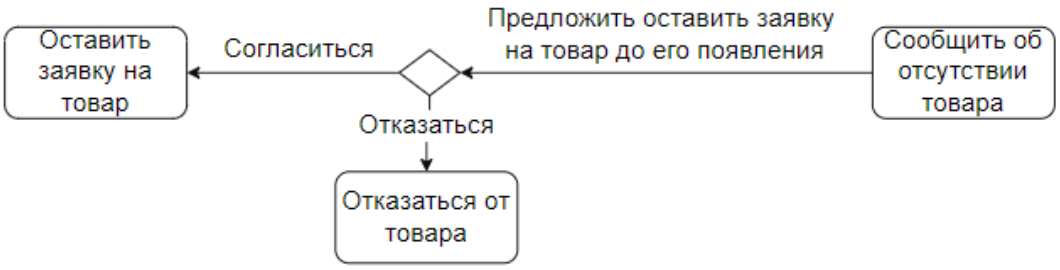


Рисунок 3.5. – Фрагмент диаграммы с узлом решения

Для некоторой выходящей из узла решения дуги может быть задано сторожевое условие с ключевым словом «или». Это сторожевое условие является допустимым для любого маркера, если этот маркер не принят ни одной другой дугой, выходящей из этого узла решения [1]. Например, для действия *«Принять заказ»* задано две дуги: *«Заказ отклонен»* и *«Заказ принят»*. Для дуги *«Заказ отклонен»* может быть задано сторожевое условие *«Заказа нет на складе»* или *«Заказ не корректно оформлен»* (рисунок 3.6).

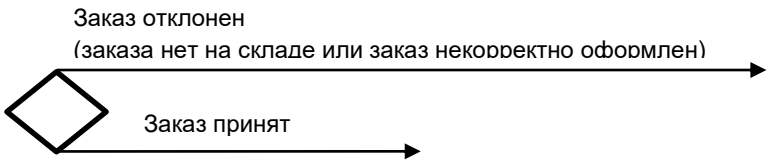


Рисунок 3.6. – Пример дуги со сторожевым условием
и частица «или»

Существуют случаи, когда функциональность узла решения объединяется с узлом соединения. Тогда на вход узла поступает несколько дуг, а на выход одна и более, и каждая из выходящих дуг должна иметь некоторое сторожевое условие (рисунок 3.7).

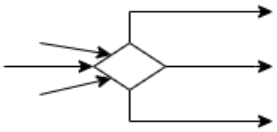


Рисунок 3.7. – Изображение узла решения, объеденного
с функционалом узла соединения

4.  Широкие полосы – начало (разветвление) и окончание (схождение) ветвления действий. **Узел объединения (соединения)** имеет два и более входящих узла и один исходящий (рисунок 3.8, *а*). **Узел разделения** является узлом управления, который разделяет поток на несколько параллельно выполняемых потоков (рисунок 3.8, *б*).

Узел соединения служит для объединения двух и более потоков, при этом дуги, входящие в узел соединения и выходящие из него, должны быть все или потоками управления, или потоками объектов [1]. На диаграмме узел соединения изображается в виде горизонтального или вертикального жирного отрезка линии.

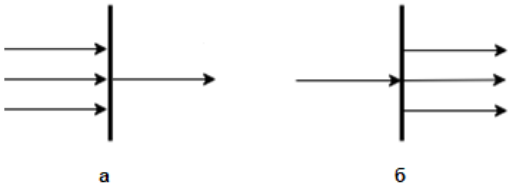


Рисунок 3.8. – Изображение узла объединения (*а*) и узла разделения (*б*)

В качестве примера можно рассмотреть фрагмент диаграммы деятельности, на котором изображен узел соединения для синхронизации окончания выполнения действий «*Просмотр конкретного товара*» и «*Добавить товар в корзину*», по завершению которых маркер управления будет передан действию «*Просмотр каталога товаров*» (рисунок 3.9).

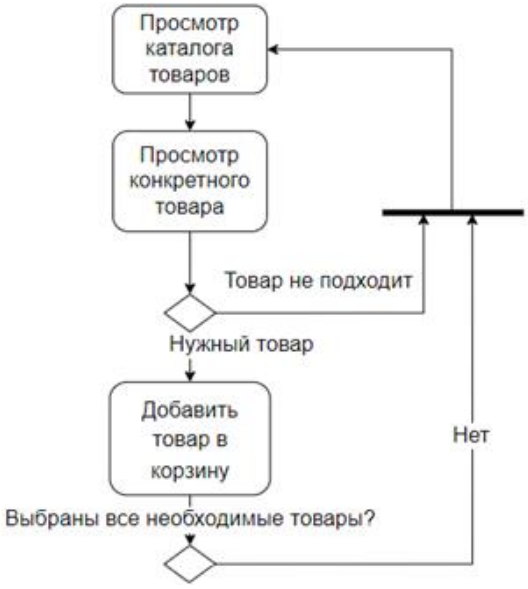
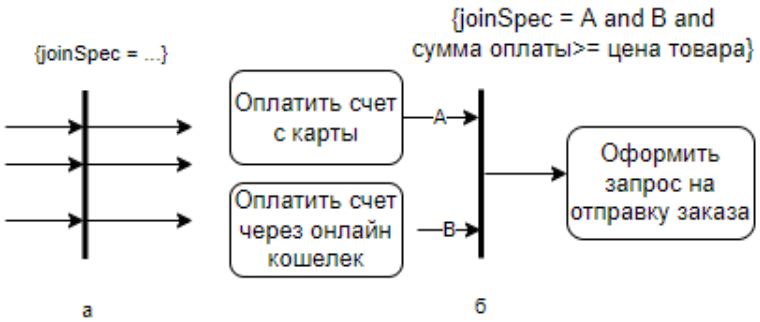


Рисунок 3.9. – Изображение узла соединения

Рядом с узлом соединения может быть изображена дополнительная спецификация (рисунок 3.10, *а*). Спецификация узла соединения может содержать имена входящих дуг, чтобы ссылаться на маркер, который мог бы быть предложен по этой дуге во время оценивания логического условия [1].



а – способ представления; *б* – пример заполненной спецификации

Рисунок 3.10. – Примеры изображения узла соединения с дополнительной спецификацией

В качестве примера выполнения узла соединения рассмотрим приведенный ниже фрагмент диаграммы деятельности. Этот фрагмент гарантирует, что действие «*Оформить запрос на отправку заказа*» будет выполнено только в том случае, если сумма оплаты соответствует цене товара (см. рисунок 3.10, *б*).

Спецификация узла соединения вызывается всякий раз, когда по любой входящей дуге проходит новый маркер. Вызов логического условия не прерывается никакими другими новыми маркерами, предлагаемыми в ходе проверки на соответствие условию спецификации. Любые маркеры, предлагаемые выходящей дуге, должны быть приняты или отклонены до того, как другие маркеры будут предложены выходящей дуге. Если маркеры не соответствуют условию, они не допускаются к выходящей дуге [1].

Существуют случаи, когда узел соединения используют не для того, чтобы объединить несколько потоков в один, а наоборот, сделать из одного потока несколько, и такой узел называется узлом разделения. На вход узла поступает одна дуга, а на выход несколько. Узел разделения используется в том случае, когда необходимо выполнить несколько действий одновременно. Например, на рисунке 3.11 оплата будет выполняться частично с карты и частично с кошелька.

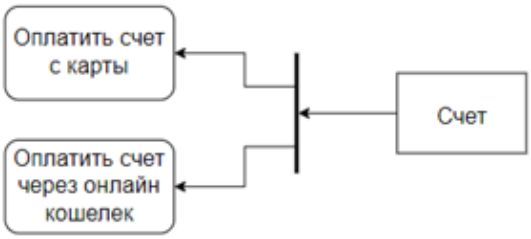
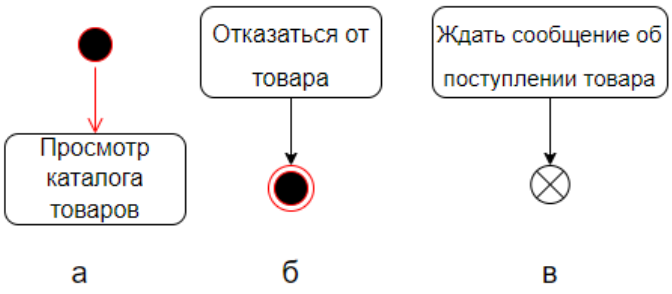


Рисунок 3.11. – Объединение узлов объедения и разделения

Узел разделения имеет одну входящую дугу и несколько выходящих дуг. Дуги, входящие в узел разделения и выходящие из него, должны быть все или потоками управления, или потоками объектов [2].

1. Черный круг – начальный узел. **Начальный узел** деятельности является узлом, в котором начинается поток при вызове данной деятельности.



а – начального узла; *б* – узла финала деятельности; *в* – финала потока

Рисунок 3.12. – Графические символы для изображения

Начальный узел является начальной точкой в выполнении деятельности. Когда деятельность вызывается, то в начальном узле размещается маркер управления, который предлагается всем выходящим из него дугам. Начальный узел не может иметь входящих дуг, а в качестве источника может иметь только дуги управления [1].

На диаграмме начальный узел изображается в виде черного круга. В качестве примера можно рассмотреть фрагмент диаграммы некоторой деятельности, при вызове которой начальный узел отправляет управление в действие «*Просмотр каталога товаров*» (см. рисунок 3.12, *а*).

2.  Черный круг с обводкой – окончание процесса (конечный узел). **Конечный узел –** узел управления, который завершает все потоки данной диаграммы деятельности. На диаграмме может быть более одного конечного узла.

Узлы финала деятельности не имеют выходящих дуг, поэтому он только принимает все маркеры, которые предлагаются на входящих дугах. Маркер, достигающий финального узла деятельности, прекращает эту деятельность. В частности, он останавливает все выполняющиеся действия в этой деятельности с выходными параметрами [1].

В качестве примера изображения конечного узла на диаграмме приведен фрагмент диаграммы некоторой деятельности, на котором после выполнения действия «*Отказаться от товара*» все маркеры данной деятельности прекращаются (см. рисунок 3.12, *б*).

3.  Белый круг с крестом – окончание потока. Конечный узел потока означает окончание одного потока управления, не завершая содержащей его деятельности. **Узел финала потока** является финальным узлом, который завершает отдельный поток управления или поток объектов, не прекращая содержащей его деятельности [2].

Финал потока уничтожает любые маркеры, которые достигают его. Он не оказывает влияния на другие потоки в этой деятельности. В качестве примера можно привести фрагмент окончания потока «*Ждать сообщение об поступлении товара*», который завершается после передачи управления узлу финала потока (см. рисунок 3.12, *в*). При этом деятельность, которая содержит данный фрагмент, не останавливается и может выполняться в других потоках [1].

Разбиение деятельности предназначено для группировки действий, которые относятся к одной деятельности и имеют некоторую общую характеристику.

Отдельное разбиение деятельности изображается в форме двух горизонтальных или вертикальных параллельных линий с именем разбиения, помещенным в прямоугольник на одном из его концов (рисунок 3.13). Любые узлы и дуги между этими линиями рассматриваются как принадлежащие этому разбиению. Это изображение по–прежнему называют разбиением с использованием нотации дорожки [3].

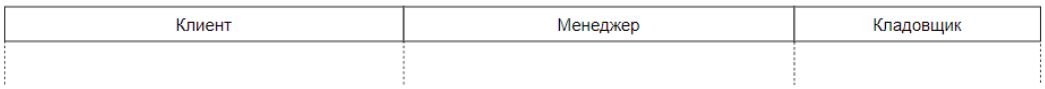


Рисунок 3.13. – Варианты изображения разбиений деятельности

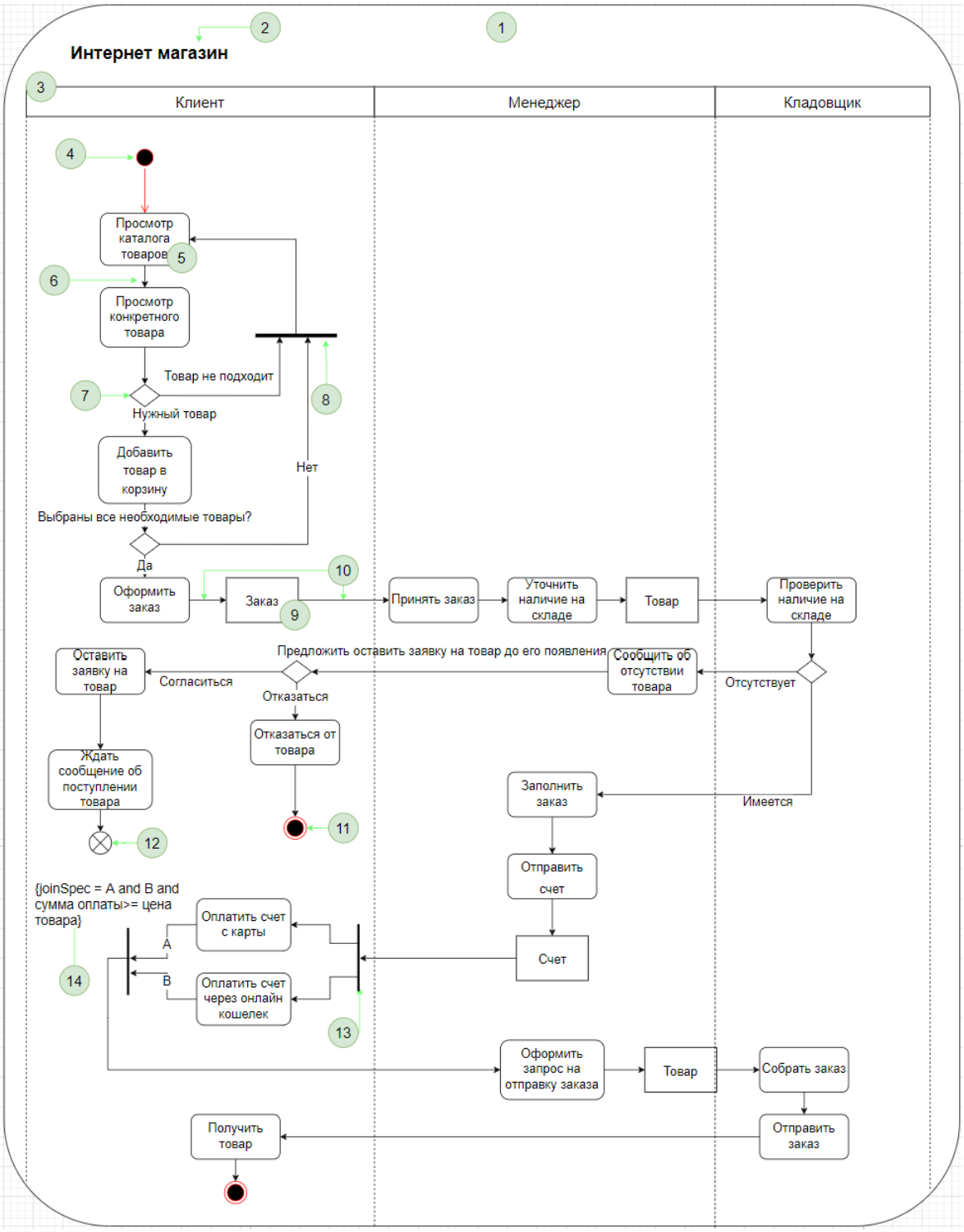
При использовании разбиений на диаграмме деятельности следует руководствоваться следующими правилами [1]:

– любой узел или дуга деятельности не может одновременно принадлежать двум или более разбиениям;

– разбиения не влияют на потоки маркеров;

– разбиения одного уровня вложенности должны быть представлены частями внутренней структуры одного и того же классификатора.

Пример диаграммы деятельности «Интернет-магазин» представлен на рисунке 3.14, на котором изображены графические элементы и потоки между ними.



1 – деятельность; 2 – название деятельности; 3 – разбиение деятельности; 4 – начальный узел (узел начала деятельности); 5 – узел деятельности; 6 – поток управления; 7 – узел решения; 8 – узел объединения; 9 – узел данных (объектов); 10 – поток данных (объектов); 11 – конечный узел (узел завершения); 12 – узел завершения потока; 13 – узел разделения; 14 – сторожевое условие

Рисунок 3.14. – Пример диаграммы деятельности «Интернет-магазина»:

Контрольные вопросы

1. Для чего используется диаграмма деятельности?
2. Дайте определение понятию «деятельность».
3. Назовите и охарактеризуйте основные элементы диаграммы деятельности.
4. Назовите и охарактеризуйте разновидности дуг деятельности.
5. Назовите правила, которыми следует руководствоваться при использовании разбиений на диаграмме деятельности.

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Разработанная диаграмма деятельности.
4. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab3-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab3-10-IT-Ivanov.zip

Литература

1. Леоненков, А.В. Самоучитель UML / А.В. Леоненков. – 2-е изд. – СПб. : БХВ-Петербург, 2007. – 576 с. : ил.

2. Рамбо, Дж. UML 2.0. Объектно-ориентированное моделирование и разработка / Джеймс Рамбо, Майкл Блаха. – 2-е изд. – СПб: Питер, 2021. – 544 с. : ил.

3. Буч, Гр. Язык UML. Руководство пользователя / Грейди Буч, Джеймс Рамбо, Айвар Джекобсон ; пер. с англ. Н. Мухин. – 2-е изд. – М. : ДМК Пресс, 2006. – 496 с. : ил.

7. Зараменских, Е.П. Информационные системы: управление жизненным циклом : учеб. и практикум / Е.П. Зараменских. – М. : Юрайт, 2019. – 431 с.

Лабораторная работа 4
РАЗРАБОТКА ДИАГРАММЫ ВЗАИМОДЕЙСТВИЯ

Цель работы: ознакомиться с методом проектирования взаимодействия на основе языка UML и научиться разрабатывать диаграмму взаимодействия для программного обеспечения.

Ход работы:

1) изучить краткие теоретические сведения;
2) ответить на контрольные вопросы;
3) разработать диаграмму взаимодействия в соответствии с вариантом индивидуального задания;
4) оформить отчет;
5) защитить лабораторную работу.

Термины и определения

Класс – абстрактное описание сущностей, которые относятся к одному общему типу. Класс описывает действия и характеристики сущностей. Одним классом описывается только один тип сущностей. Если сущность будет обладать дополнительным действием или характеристикой, то создается отдельный класс. Например, для сущности человека (класс Person) может быть создано описание, которое содержит рост и вес, а сущность студента (класс Student) кроме роста и веса может содержать номер курса и факультета. Описание класса является абстрактным, т.к. оно содержит не все действия или характеристики, а только те, которые необходимы для реализации поставленной задачи.

Объект – созданный экземпляр сущности на основании класса. Например, для класса Person может быть создано несколько объектов, которые содержат собственные рост и вес (Маша: 165 см, 50 кг; Петя: 120 см, 22 кг).

Линия жизни – служит для обозначения периода времени, в течение которого участник (объект) существует в системе и может участвовать во всех ее взаимодействиях [5].

Стереотипы – механизм, позволяющий разделять классы или сообщения на категории [3].

Сообщение – элемент модели, предназначенный для представления отдельной коммуникации между линиями жизни некоторого взаимодействия [3].

Фрейм – рамка и ярлык с меткого оператора взаимодействия и названием диаграммы.

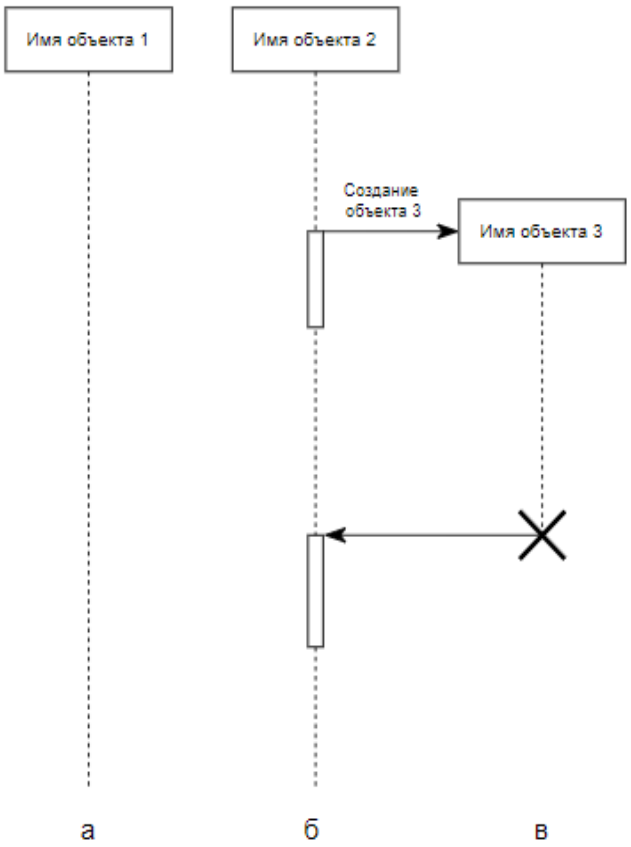
Диаграмма взаимодействия предназначена для описания взаимодействия между элементами программы и передачи сообщений между ними. Как правило, диаграмма описывает какой-то основной сценарий работы программы.

Реализация взаимодействия изображается посредством сообщений, которые передаются между различными линиями жизни. Сообщения изображаются в виде стрелок различной формы и образуют некоторый порядок относительно времени своей передачи. При этом сообщения, расположенные на диаграмме взаимодействия выше, передаются раньше тех, которые расположены ниже [3].

Основными элементами диаграммы взаимодействия являются объекты, классы, программные компоненты, системы (подсистемы), обладающие поведением. Элементы изображаются слева направо таким образом, чтобы элемент, расположенный сверху слева, был элементом, который инициирует взаимодействие. Каждый элемент на диаграмме взаимодействия не может существовать без линии жизни.

Линия жизни демонстрирует время существования в системе одного конкретного ее участника. На диаграмме линия жизни изображается в виде вертикальных пунктирных линий. Если объект существует в системе постоянно, то его линия жизни продолжается от самой верхней части диаграммы до самой нижней. Пример изображения линии жизни представлен на рисунке 4.1, *а*.

В процессе функционирования программ одни объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия, или в состоянии пассивного ожидания сообщений от других объектов. Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название фокуса управления (рисунок 4.1, *б*) [2].



а – линия жизни; *б* – фокус управления на линии жизни; *в* – уничтожение объекта

Рисунок 4.1 – Изображения линий жизни и фокусов управления объектов:

Отдельные объекты, выполнив свою роль в системе, могут быть уничтожены (разрушены), чтобы освободить занимаемые ими ресурсы (см. рисунок 4.1, *в*). Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML используется специальный символ в форме латинской

буквы X. Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий [3].

Фокус управления изображается в форме вытянутого узкого прямоугольника, верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а ее нижняя сторона – окончание фокуса управления (окончание активности). Этот прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию жизни, если на всем ее протяжении он является активным.

С другой стороны, периоды активности объекта могут чередоваться с периодами его пассивности или ожидания. В этом случае у такого объекта имеются несколько фокусов управления. Важно понимать, что получить фокус управления может только существующий объект, у которого в этот момент имеется линия жизни. Если же некоторый объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него может быть создан другой экземпляр этого же класса, который, строго говоря, будет являться другим объектом [3].

Актер, или **внешний пользователь,** в большинстве случаев выступает инициатором взаимодействия в системе. Когда актер является инициатором, то он изображается на диаграмме взаимодействия самым первым объектом слева со своим фокусом управления (рисунок 4.2, *а*). Чаще всего актер и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. При этом сам актер может иметь собственное имя либо оставаться анонимным [6].

Также, не редки случаи, когда объект может вызвать себя самостоятельно. Такое сообщение является рефлексивным (рисунок 4.2, *б*).

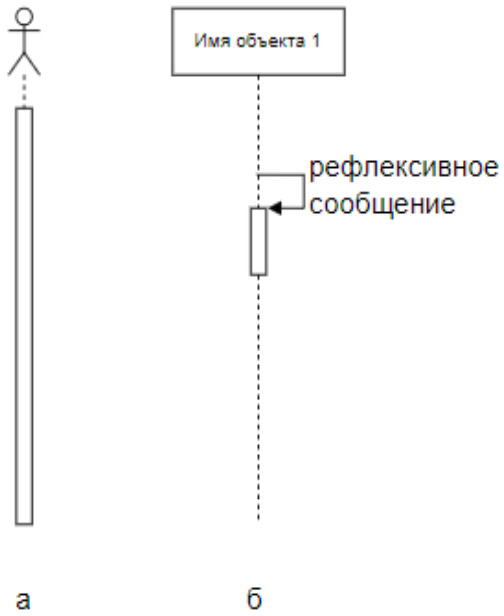


Рисунок 4.2 – Графическое изображение актера (*а*) и рефлексивного сообщения (*б*) на диаграмме взаимодействия

На диаграммах взаимодействия допустимо использование стандартных **стереотипов класса** [3]:

–  актер – экземпляр участника системы;

–  граничный класс – вид класса, отделяющий внутреннюю структуру системы от внешней среды (экранная форма, пользовательский интерфейс, устройство ввода-вывода).

–  управляющий класс – активный элемент, который используются для выполнения некоторых операций над объектами (программный компонент, модуль, обработчик);

–  класс-сущность – обычно применяется для обозначения классов, которые хранят некую информацию о бизнес-объектах (соответствует таблице или элементу базы данных).

Также, одним из основных понятий диаграммы взаимодействия, является **Сообщение**.

Прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено [3]. Сообщение – это вызов методов конкретных объектов. Для корректного вызова и выполнения метода в сообщении необходимо указать, какие данные будут переданы, и определить, что вернется в ответ.

При реализации диаграммы взаимодействия используют следующие виды сообщений [3]:

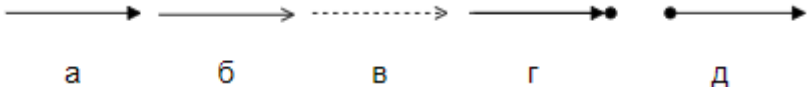
– синхронное сообщение (**synchCall**) – соответствует синхронному вызову операции и подразумевает ожидание ответа от объекта получателя. Пока ответ не поступит, никаких действий в программе не производится (рисунок 4.3, *а*);

– асинхронное сообщение (**asynchCall**) – которое соответствует асинхронному вызову операции и подразумевает, что объект может продолжать работу, не ожидая ответа (рисунок 4.3, *б*);

– ответное сообщение (**reply**) – ответное сообщение от вызванного метода. Данный вид сообщения показывается на диаграмме по мере необходимости или когда возвращаемые им данные несут смысловую нагрузку (рисунок 4.3, *в*);

– потерянное сообщение (**lost**) – сообщение, не имеющее адресата сообщения, т.е. для него существует событие передачи и отсутствует событие приема (рисунок 4.3, *г*);

– найденное сообщение (**found**) – сообщение, не имеющее инициатора сообщения, т.е. для него существует событие приема и отсутствует событие передачи (рисунок 4.3, *д*) [4].

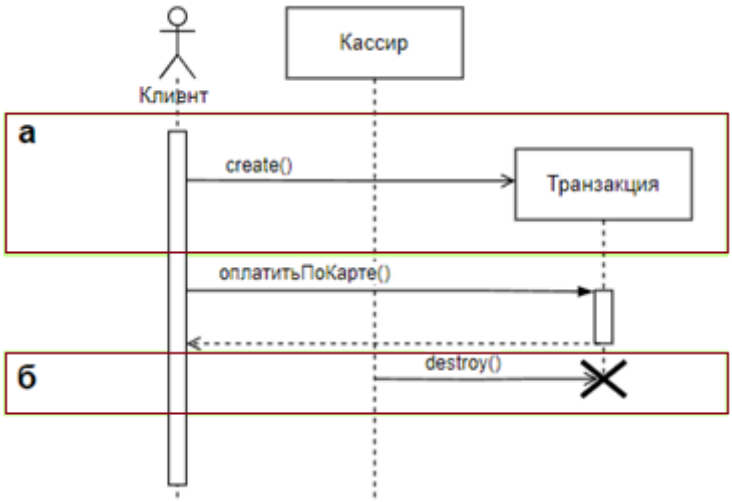


а – синхронное сообщение; *б* – асинхронное сообщение или асинхронный сигнал;
в – ответное сообщение; *г* – потерянное сообщение; *д* – найденное сообщение

Рисунок 4.3. – Графическое изображение различных видов сообщений:

Сообщения также имеют ряд стереотипов. Наиболее часто используемые стереотипы – это **create** и **destroy.**

Сообщение со стереотипом **create** вызывает в классе метод, который создает экземпляр класса – объект. На диаграмме взаимодействия необязательно показывать с самого начала все объекты, участвующие во взаимодействии. При использовании сообщения со стереотипом create создаваемый объект отображается на уровне конца сообщения [3]. Пример представлен на рисунке 4.4, *а*.



а – создание объекта; *б* – уничтожение объекта

Рисунок 4.4. – Графическое изображение различных стереотипов сообщений:

Сообщение со стереотипом **destroy** используется для уничтожения экземпляра класса, при этом в конце линии жизни объекта изображается знак X. Пример представлен на рисунке 4.4, *б*.

Для расширения сценариев с помощью ветвления и распараллеливания линий сообщений, а также создания цикличности используют **фреймы** взаимодействия (рисунок 4.5) **и операторы** взаимодействия.

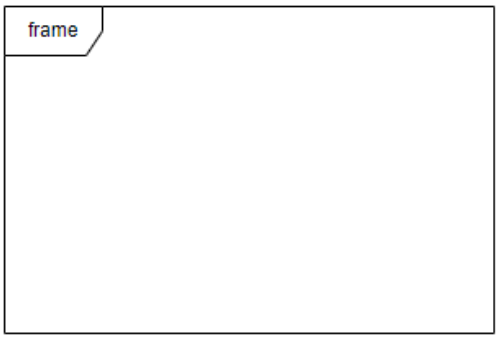


Рисунок 4.5. – Графическое изображение фрейма

Отдельные фрагменты диаграммы взаимодействия можно выделить с помощью фрейма [3]. Фрейм должен содержать оператор взаимодействия, а также, в некоторых случаях, название выделенного участка диаграммы либо название всей диаграммы. UML содержит следующие операторы:

– **alt** (alternative) **–** какое-то количество фрагментов, из которых выполняется тот фрагмент, результат которого совпадает с истинным условием (рисунок 4.6);

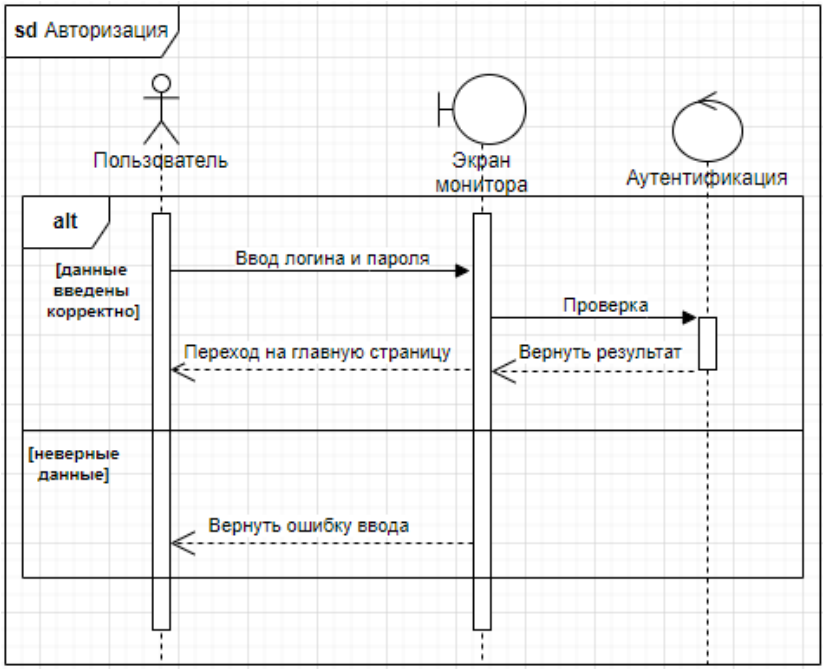


Рисунок 4.6. – Изображение фрейма с оператором alt

– **opt** (optional) **–** фрагмент диаграммы, выполнение которого не обязательно. Срабатывает только в случае, если условие истинно. Эквивалентно **alt** с одним видом условия. Пример представлен на рисунке 4.7, в данном случае истинным значением являются неверные данные. При вводе неверного логина или пароля аутентификация предоставляет пользователю повторный ввод данных;

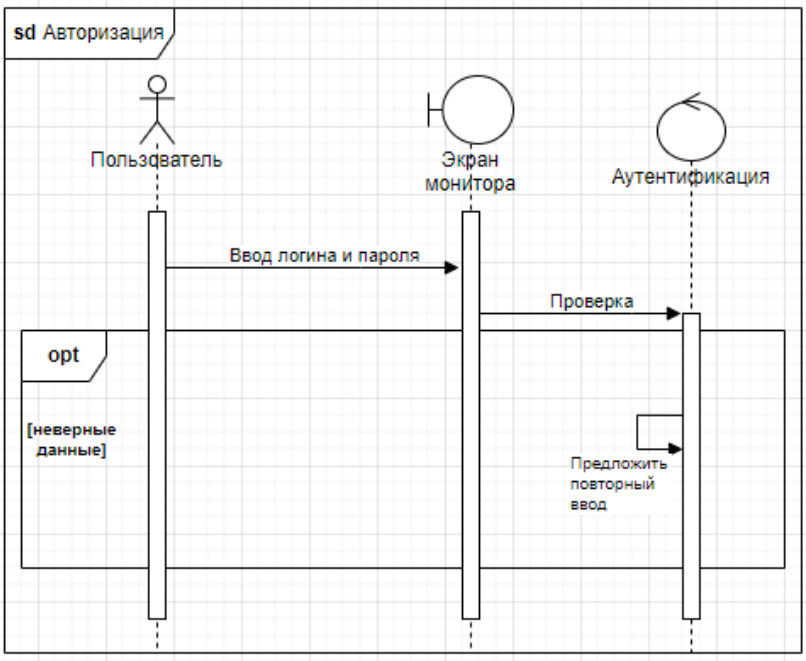


Рисунок 4.7. – Изображение фрейма с оператором opt

– **break** (break) – описывает сценарий прерывания работы фрейма согласно заданному сторожевому условию. При выполнении сторожевого условия break выполняется вместо оставшейся части фрагмента взаимодействия [4]. Пример представлен на рисунке 4.8. При вводе неверного пароля транзакция завершается;

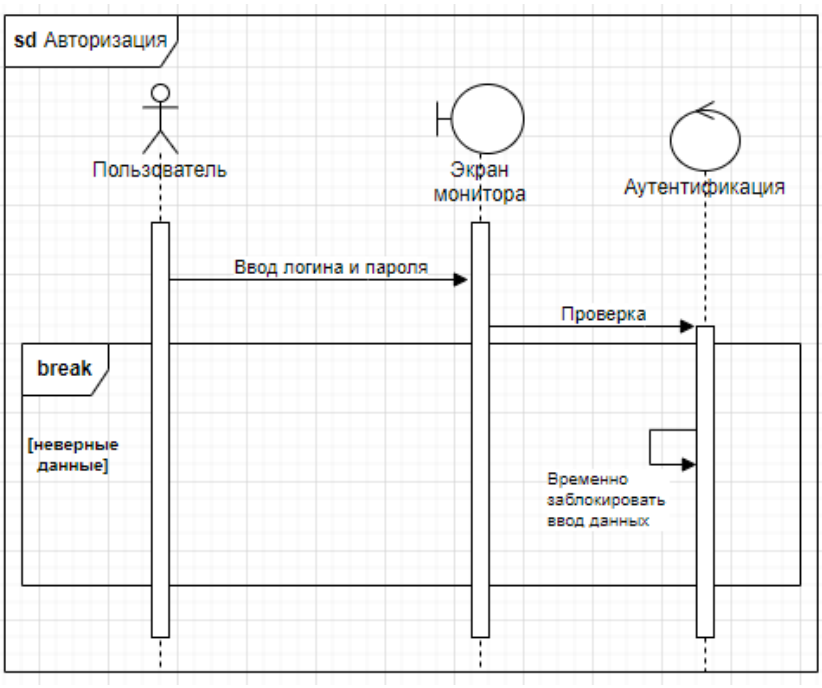


Рисунок 4.8. – Изображение фрейма с оператором break

– **loop** (loop) – цикл; последовательность, содержащаяся в операторе loop, может выполняться несколько раз. В скобках указывается минимальное и максимальное количество повторений цикла [6]. Пример представлен на рисунке 4.9. В данном случае необходимо как минимум 1 раз ввести логин и пароль. Допускается 3 попытки ввода данных. В случае некорректного ввода значений вызывается оператор loop и выполняется очередная итерация цикла, иначе будет осуществлен выход из цикла.

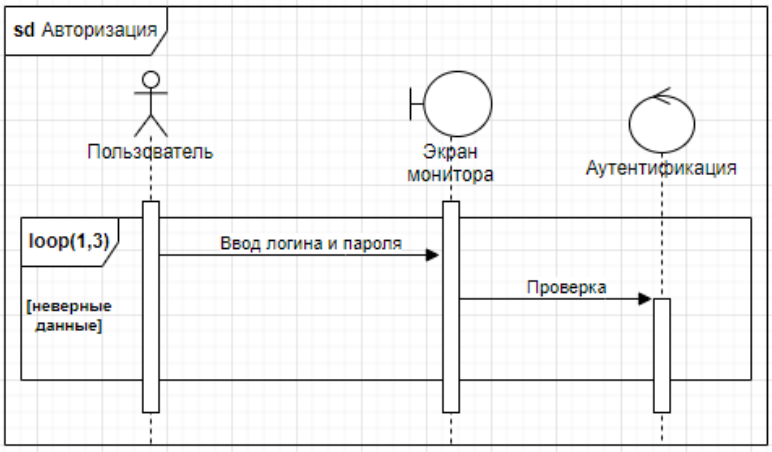


Рисунок 4.9. – Изображение фрейма с оператором loop

– sd (sequence diagram) – используется для очерчивания всей диаграммы взаимодействия, если это необходимо (например, обозначение названия диаграммы) [4]. Примеры использования фрейма с параметром sd представлены на рисунках 4.6–4.10.

Контрольные вопросы

1. Назовите назначение диаграммы взаимодействия.
2. Дайте определение понятиям линии жизни и фокусу управления.
3. Назовите основные назначения стандартных стереотипов класса.
4. Назовите и охарактеризуйте основные виды сообщений.
5. Назовите и охарактеризуйте основные стереотипы сообщений.
6. При помощи чего можно выделить отдельные фрагменты диаграммы взаимодействия?

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Разработанная диаграмма взаимодействия.
4. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab4-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab4-10-IT-Ivanov.zip

Литература

1. Информационные технологии управления : учеб. пособие / под ред. Ю.М. Черкасова. – М. : ИНФРА-М, 2001. – 216 с.

Новиков, Ф.А. Моделирование на UML. Теория, практика, видеокурс / Ф.А. Новиков, Д.Ю. Иванов. – СПб. : Проф. лит., Наука и Техника, 2010. – 640 с.

4. Леоненков, А.В. Самоучитель UML / А.В. Леоненков. – 2-е изд. – СПб. : БХВ-Петербург, 2007. – 576 с. : ил.

2. Фаулер, М. UML в кратком изложении / М. Фаулер, К. Скотт. – М. : Мир, 1999. – 192 с.

3. Малышева, Е.Н. Проектирование информационных систем. Раздел 5. Индустриальное проектирование информационных систем. Объектно-ориентированная case-технология проектирования информационных систем / Е.Н. Малышева. – Кемерово : Кемер. гос. ин-т культуры, 2009. – 70 с.

4. Леоненков, А.В. Объектно-ориентированный анализ и проектирование с использованием uml и ibm rational rose : учеб. пособие / А.В. Леоненков. – М. : Интуит, Ай Пи Ар Медиа, 2020. – 320 с.

5. Буч, Гр. UML. Руководство пользователя / Грейди Буч, Джеймс Рамбо, Айвар Джекобсон. – М. : ДМК, 2000. – 432 с. : ил.

5. Фаулер, М. UML. Основы : [пер. с англ.] / М. Фаулер, К. Скотт. – СПб: Символ-Плюс, 2002. – 192 с.

6. Арлоу, Дж. UML 2.0 и унифицированный процесс. Практический объектноориентированный анализ и проектирование : [пер. с англ.] / Дж. Арлоу, И. Нейштадт. – 2е изд. – СПб. : СимволПлюс, 2007. – 624 с. : ил.

Лабораторная работа 5
МОДЕЛИ И ГЕНЕРИРОВАНИЕ КОДА НА ОСНОВЕ UML-ДИАГРАММ

Цель работы: изучить принципы построения диаграммы классов и разработать диаграмму классов согласно варианту задания.

Ход работы:

1) изучить теоретические сведения;
2) ответить на контрольные вопросы;
3) разработать диаграмму классов в соответствии с вариантом индивидуального задания при помощи diagrams.net и Visual Studio с генерацией кода на языке C# или C++;
4) оформить отчет;
5) защитить лабораторную работу.

Термины и определения

Класс – абстрактное описание сущностей, которые относятся к одному общему типу. Класс описывает действия и характеристики сущностей. Одним классом описывается только один тип сущностей. Если сущность будет обладать дополнительным действием или характеристикой, то создается отдельный класс. Например, для сущности человека (класс *Person*) может быть создано описание, которое содержит такие параметры, как рост и вес, а сущность студента (класс *Student*), кроме роста и веса, может содержать номер курса и факультета. Описание класса является абстрактным потому, что оно не содержит все действия или характеристики, а содержит только те, которые необходимы для реализации поставленной задачи.

Объект – созданный экземпляр сущности на основании класса. Например, для класса *Person* может быть создано несколько объектов, которые содержат собственные рост и вес (Маша: 165 см, 50 кг; Петя: 120 см, 22 кг).

Атрибут (поле, свойство) – именованная характеристика класса, которая имеет определенный тип. Атрибуту устанавливается значение заданного типа. Пример: атрибут – вес, тип – дробный (вещественный), значение – 67.5 кг.

Метод – функция или процедура, принадлежащая какому-либо классу или объекту. Метод состоит из некоторого количества внутренних операций для выполнения определенного действия и может иметь набор входных параметров.

Параметр – передаваемые между объектами данные определенного типа или класса.

Тип – ограничения на принимаемые значения или набор возможных значений для передаваемых данных, атрибутов, методов, переменных [1].

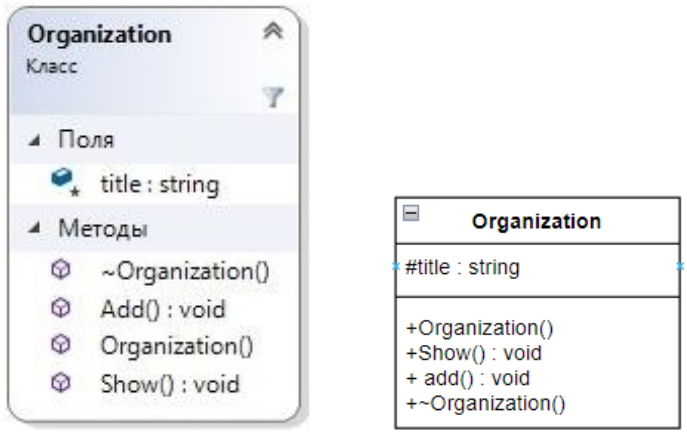
Переменная – внутренний объект, который используется только внутри класса. Из вне класса получить доступ к переменной или задать для нее значение нельзя.

Конструктор – метод класса, который вызывается при создании объекта и инициализирует начальные значения атрибутов и переменных объекта. Конструктор может содержать входные параметры.

Диаграмма классов – диаграмма, на которой изображены классы (типы) программы, различные связи, существующие между ними, а также, атрибуты и методы классов.

На диаграмме класс изображается в виде прямоугольника, разделенного на три блока (рисунок 5.1):

– имя класса;
– атрибуты (поля) класса;
– операции (методы) класса.



а *б*

а – в Visual Studio; *б* – в diagrams.net

Рисунок 5.1. – Изображение класса на диаграмме классов

Каждый класс должен иметь уникальное имя, состоящее из любого количества букв (чаще всего латинского алфавита), цифр и знаков препинания (кроме двоеточия и точки).

Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым.

Имя атрибута должно быть уникально в пределах класса. За именем атрибута может следовать его тип и значение по умолчанию [1].

Базовые типы атрибутов и методов:

– int – целое число;
– float – вещественное число ординарной точности с плавающей точкой;
– double – вещественное число двойной точности с плавающей точкой;
– bool – логический тип;
– char – один символ в кодировке ASCII;
– string – символьная строка;
– datetime – дата и время;
– void – тип без значения.

Класс может иметь любое число методов либо не иметь ни одного. Часто вызов метода объекта изменяет его атрибуты. Графически методы представлены в нижнем блоке описания класса [1]. Допускается указание только имен методов и иногда типов. Чаще всего методы именуются короткими глаголами, описывающими некое поведение класса, которому принадлежит метод. Обычно каждое слово в имени метода пишется с заглавной буквы, например, Add или Show.

Все атрибуты и операции имеют один из нижеперечисленных типов видимости, которые также можно указать на диаграмме. Существуют следующие **типы видимости**:

— **-** – private (приватный, данный атрибут или метод может использоваться только тем классом, которому принадлежит);

— **#** – protected (защищенный, данный атрибут или метод может использоваться классом, которому принадлежит, и его наследниками);

— **+** – public (общий, данный атрибут или метод может использоваться всеми классами программы).

Видимость для полей и методов указывается в виде левого символа в строке с именем соответствующего элемента.

Классы соединяются друг с другом при помощи **связей** [1]:

— ассоциация;

— агрегация;

— наследование.

Ассоциация – это типизированные отношения между классами, которое указывает на логическую связь между ними [7].

Каждый конец ассоциации обладает кратностью, которая показывает, сколько объектов, расположенных с соответствующего конца ассоциации, может участвовать в данном отношении [2]. В примере на рисунке 5.2 каждый *Товар* имеет какое угодно *Количество на складе*, но это *Количество на складе* обязательно принадлежит какому-то *Товару*.

Рисунок 5.2. – Изображение связи ассоциации

Ассоциации может быть присвоено имя. В качестве имени обычно выбирается глагол или глагольное словосочетание, сообщающие смысл и назначение связи. Также на концах ассоциации под кратностью может указываться имя роли, т.е. какую роль выполняют объекты, находящиеся с данного конца ассоциации (пример родитель – наследник) [5].

Агрегация – это частный случай ассоциации, описывающий объекты, состоящие из частей [4]. Агрегация в UML представляется в виде прямой с ромбом на конце (рисунок 5.3).

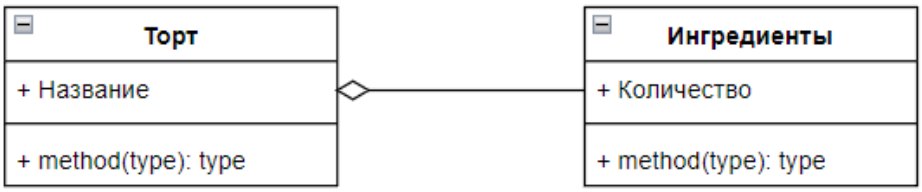


Рисунок 5.3. – Изображение связи агрегации

Ромб на связи указывает, какой класс является агрегирующим (т.е. «состоящим из»); класс с противоположного конца – агрегированным (т.е. те самые «части»).

Композиция – это частный случай агрегация, характеризующийся двумя дополнительными ограничениями. Составляющая часть может принадлежать не более чем одному агрегату, и существует до тех пор, пор существует этот агрегат [4]. Композиция изображается так же, как агрегация, только ромбик закрашен (рисунок 5.4).

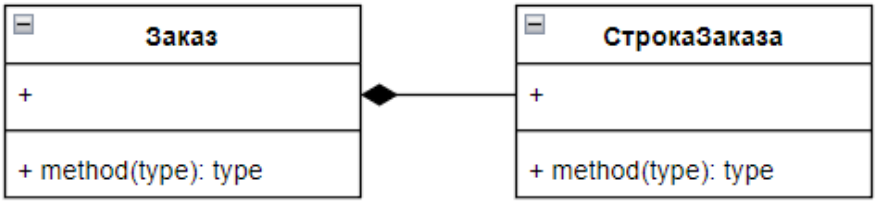


Рисунок 5.4. – Изображение связи композиции

Важно понимать разницу между агрегацией и композицией: при агрегации объекты-части могут существовать сами по себе, при композиции – нет [5]. Пример агрегации: торт – ингредиенты, пример композиции: дом – комната.

Наследование – это механизм разделения и повторного использования одного и того же класса в разных объектах. Класс потомок наследует методы и свойства родительского класса. При создании производного класса на основе базового (одного или нескольких) возникает иерархия наследования [6] (рисунок 5.5).

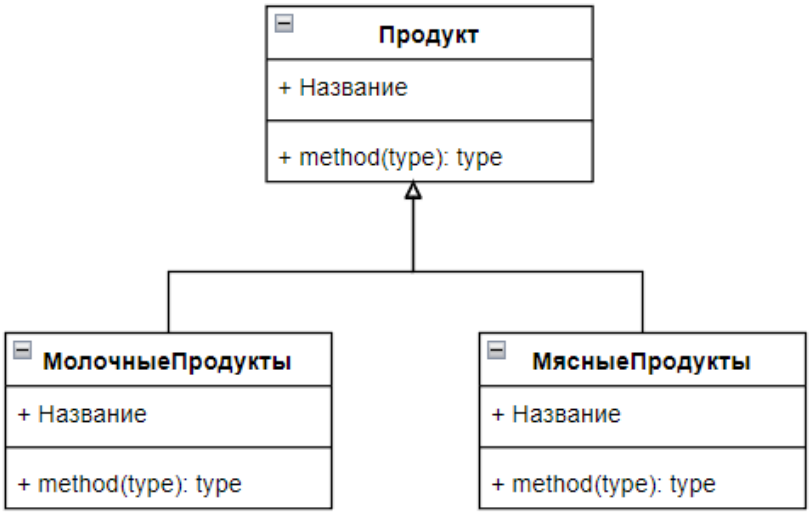


Рисунок 5.5. – Изображение связи наследования

На рисунке 5.6 приведен пример готовой диаграммы классов. Основной сущностью в системе будет являться конкретный товар. Какое-то количество конкретного товара хранится на складе. На складе работают сотрудники, которые работают с заказами, заполняют накладные и загружают/выгружают товары со склада. Поскольку, *Заказ* и *Накладная* являются однотипными классами и имеют одинаковые атрибуты, то их можно объединить при помощи родительского класса *Документ*.

Диаграмма классов, написанная в среде наподобие diagrams.net, легко воспринимаема. С ее помощью легче понять логику программы, выявить ее недостатки и исправить их, но при этом для генерации кода она будет сложна и трудозатратна.

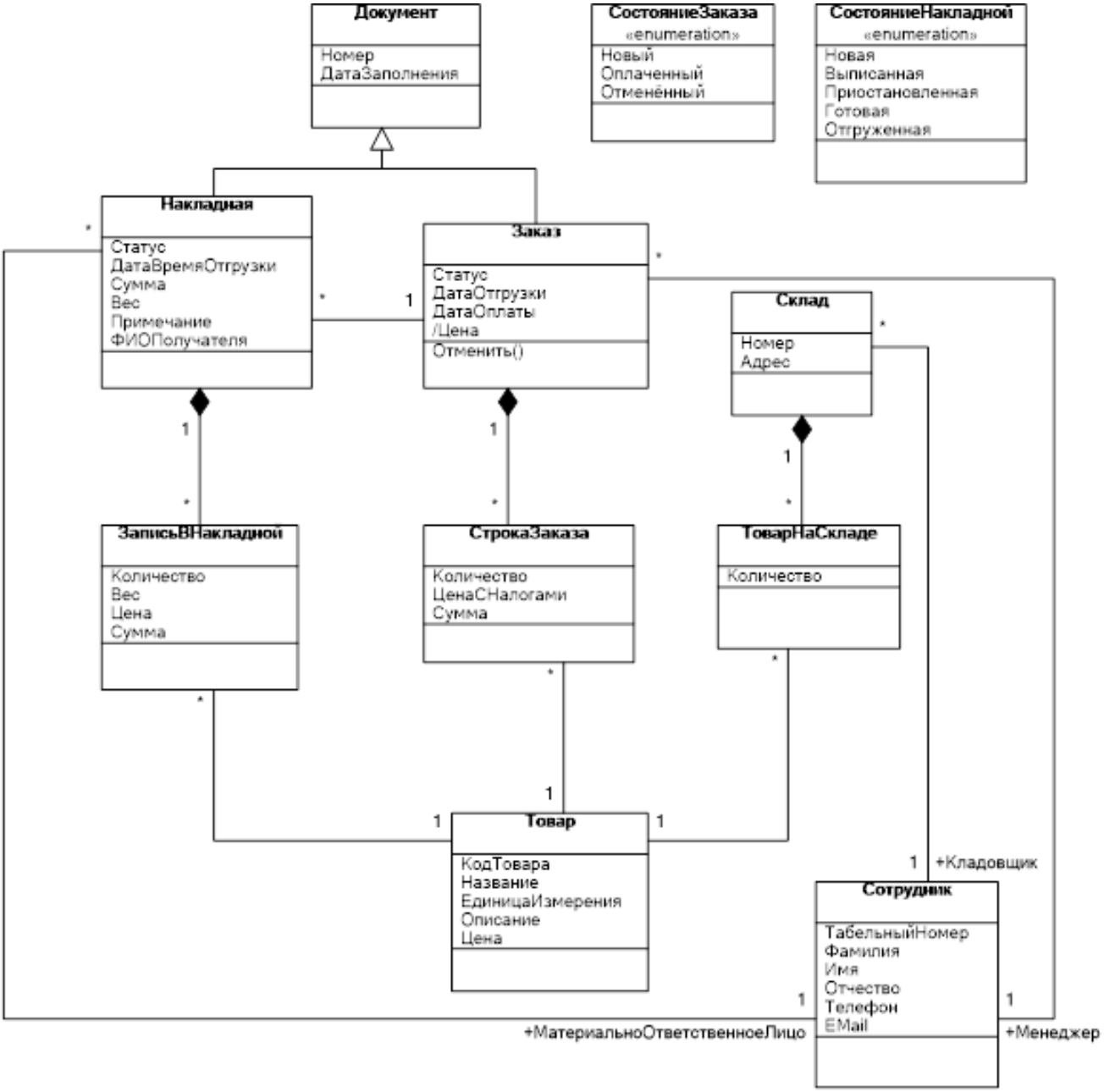


Рисунок 5.6. – Пример диаграммы классов реализованной при помощи diagrams.net

Проектирование диаграммы классов

В качестве примера рассмотрим создание диаграммы классов для системы учета оценок студентов в ведомостях по дисциплине. Ведомость – итоговый документ, который содержит список итоговых оценок по курсу для студентов группы. В системе учета участвуют следующие сущности: *Студент*, *Преподаватель*, *Ведомость*, *Отметка*.

В сущности *Студент* выделяются следующие характеристики:

– фамилия;
– имя;
– отчество;
– курс;
– номер зачетки,
– номер группы.

В сущности *Преподаватель* можно выделить следующие характеристики:

– фамилия;
– имя;
– отчество,
– должность.

В сущности *Ведомость*:

– название предмета;
– преподаватель, который ведет предмет;
– список отметок.

В сущности *Отметка*:

– балл;
– студент;
– дата выставления отметки.

Для работы с системой учета сущности журнала необходимо обладать следующими действиями:

1. Добавление студента и выставление отметки.
2. Печать итогового списка студентов с выставленными отметками.

Из представленного списка характеристик видно, что преподаватель и студент имеют одинаковые характеристики (фамилия, имя, отчество), которые можно выделить в отдельную общую сущность – *Персона*. Кроме общих характеристик персона может готовить к печати свое полное имя (ФИО).

Сущность *Отметка* имеет агрегацию со *Студент*, а сущность *Ведомость* имеет агрегацию с *Преподаватель* и ассоциацию с *Отметка*. Ассоциация ведомости с отметками содержит количество отметок, равное количеству студентов в группе.

На рисунке 5.7 приведена спроектированная диаграмма классов.

На диаграмме представлены пять классов (Person, Student, Tutor, Grade, Statement). Каждый класс описывает сущность, которая задействована в системе учета.

Класс Person является базовым классом для классов-наследников Student и Tutor, то есть Person является родительским классом, а Student и Tutor – наследуют переменные, свойства и методы родительского класса.

Класс Person содержит поля (свойства) FirstName, MiddleName, LastName и метод GetFIO. Свойства FirstName, MiddleName, LastName имеют строковый тип (string) и модификатор доступа procted, чтобы в классах-наследниках к ним мог быть выполнен доступ. Метод GetFIO будет склеивать строки FirstName, MiddleName, LastName через пробел и возвращать результирующие строковое значение.

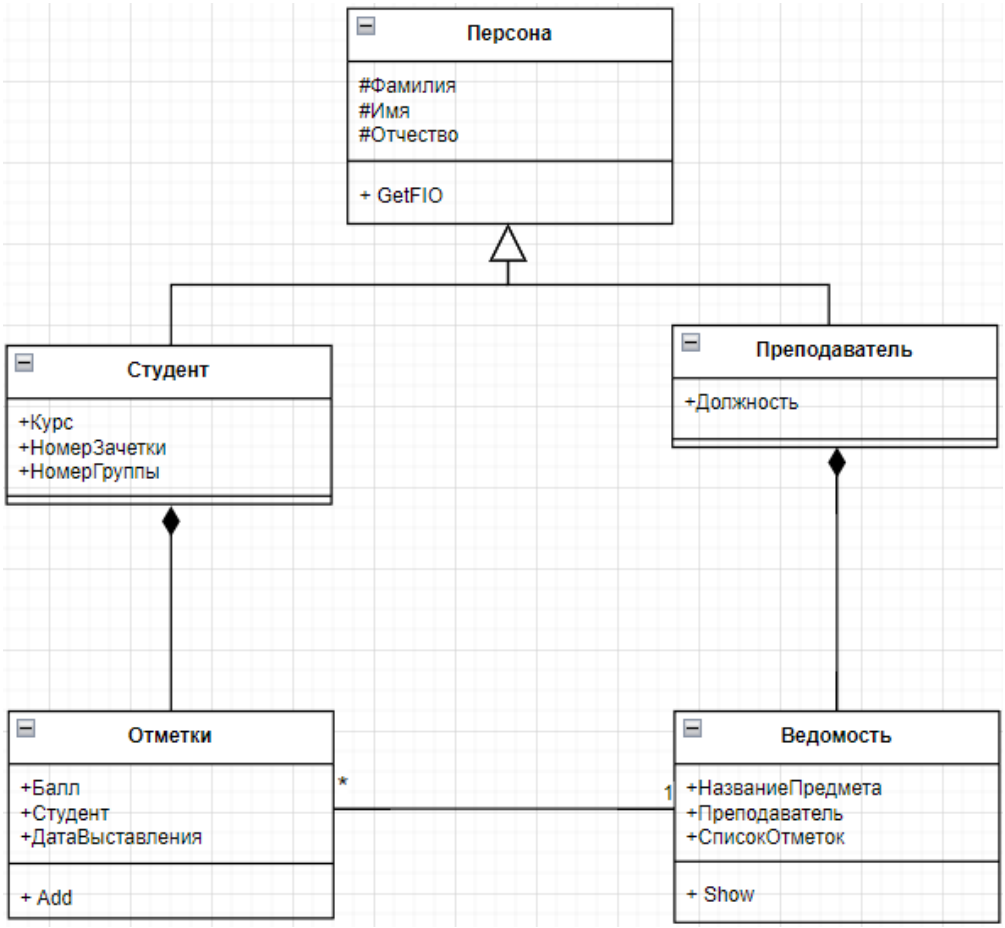


Рисунок 5.7. – Диаграмма классов

Создание диаграммы классов в Visual Studio

Для генерации кода разрабатывается диаграмма классов в Visual Studio.

Visual Studio – среда разработки программного обеспечения от Microsoft, в настоящее время является одной из самых популярных в мире. Visual Studio позволяет разрабатывать на языках программирования C#, C++, Java, Python, JavaScript, TypeScript, Node.js и др. под операционными система Windows, MacOS, Linux.

Рассмотрим создание диаграммы классов в Visual Studio. В данной среде диаграмма создается при помощи конструктора диаграммы классов.

Для начала необходимо создать консольное приложения, для этого необходимо запустить Visual Studio и выбрать пункт «Создать новый проект» (рисунок 5.8).

После чего откроется меню создания проекта, где необходимо выбрать тип создаваемого приложения (в нашем случае это консольное приложение) и язык программирования (рисунок 5.9).

Последним этапом будет настройка проекта. Данное окно откроется сразу после нажатия на кнопку «Далее» в меню создания проекта. При настройке проекта указывается его имя, место расположения и имя решения (рисунок 5.10).

Пример только что созданного проекта изображен на рисунке 5.11.

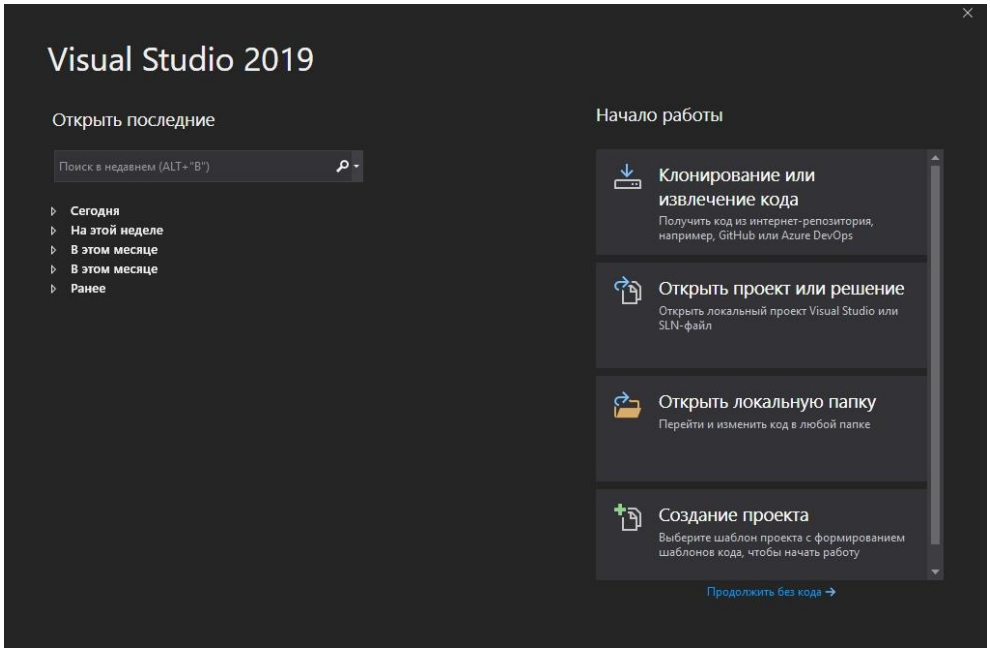


Рисунок 5.8. – Запущенная среда разработки Visual Studio

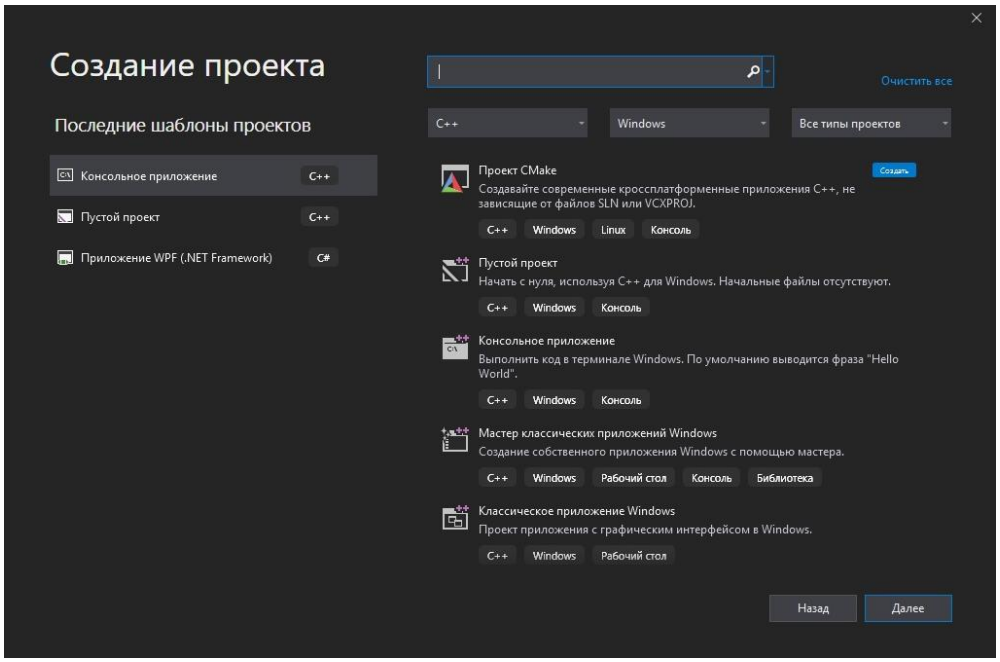


Рисунок 5.9. – Меню создания проекта

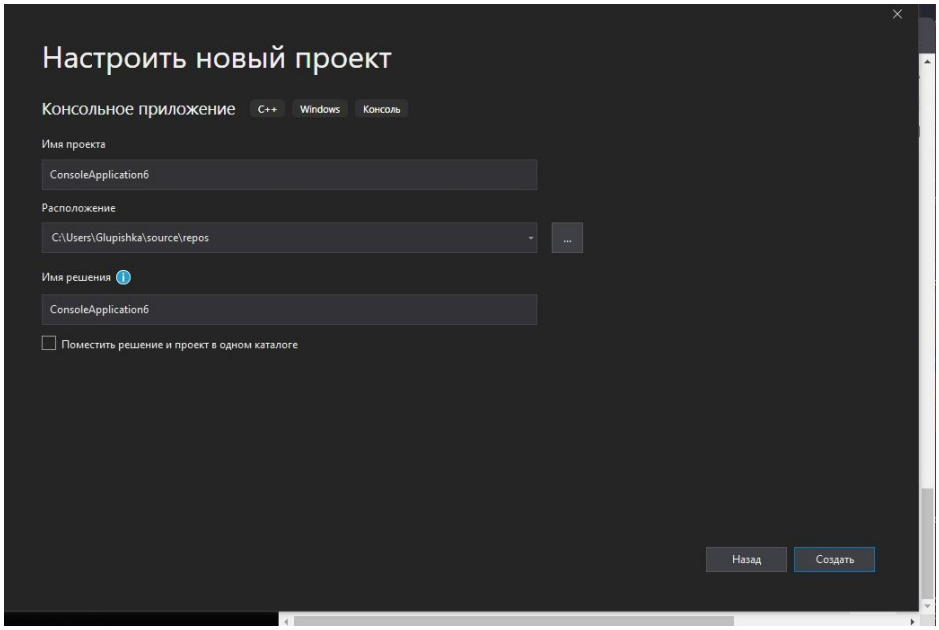


Рисунок 5.10 – Настройки проекта

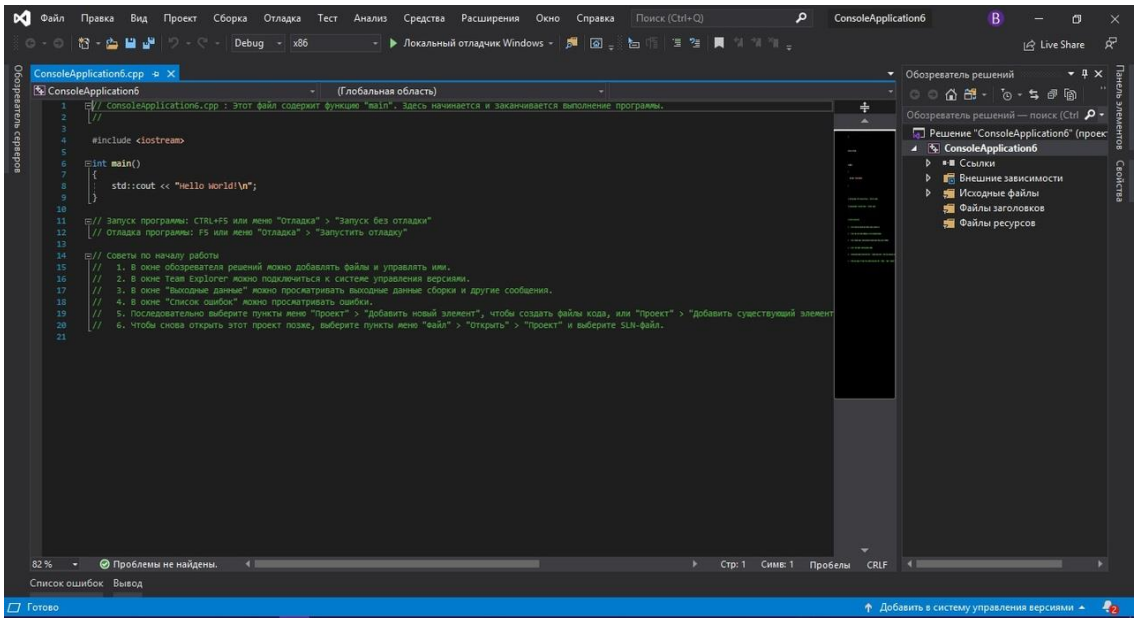


Рисунок 5.11 – Созданный проект

На созданном проекте необходимо открыть конструктор диаграммы классов. Для этого кликаем правой кнопкой мыши на сам проект в обозревателе решений (рисунок 5.12), наводим курсор на «Представление» и выбираем «Перейти к диаграмме классов» (рисунок 5.13). После чего откроется конструктор диаграммы классов.

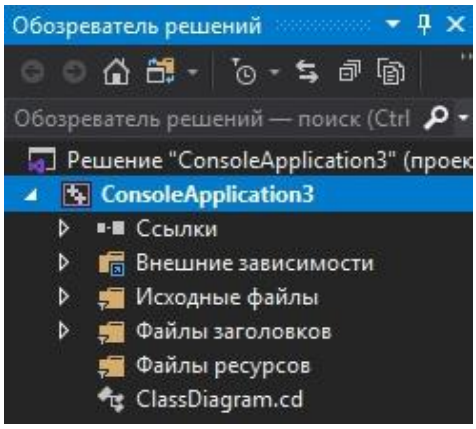


Рисунок 5.12. – Обозреватель решений

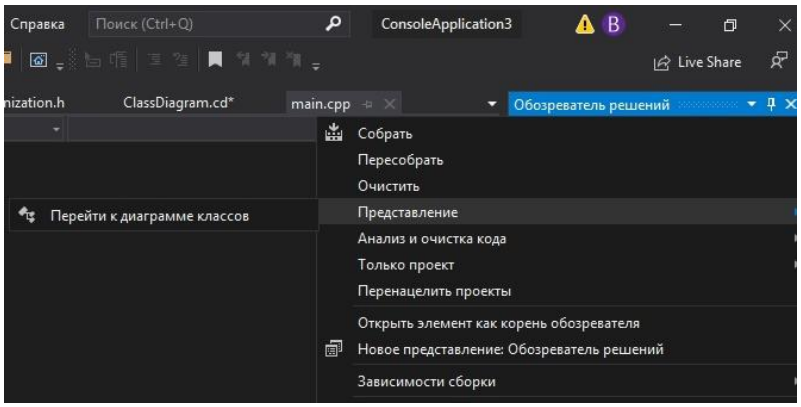


Рисунок 5.13. – Открытие диаграммы классов

На рисунке 5.14 представлен открытый конструктор диаграммы классов. В центре расположена рабочая область, здесь будет располагаться диаграмма. Справа представлена панель элементов.

Чтобы добавить новый класс, следует перетянуть необходимый объект из панели элементов в центр экрана. Для примера создадим новый класс. После того как элемент класса был перенесен в центр экрана, открывается диалоговое окно конструктора класса (рисунок 5.15).

На экране появляется пустой класс, имеющий только название и тип доступа. Для того чтобы добавить в класс новые методы или поля, необходимо кликнуть на него правой кнопкой мыши и навести на «Добавить». Во всплывающем меню предоставляется выбор элементов класса (рисунок 5.16).

После выбора одного из элементов он появится внутри класса, и тогда можно будет его отредактировать (рисунок 5.17).

Для созданного класса будет автоматически генерироваться исходный код, содержащий методы и поля и др. (рисунок 5.18).

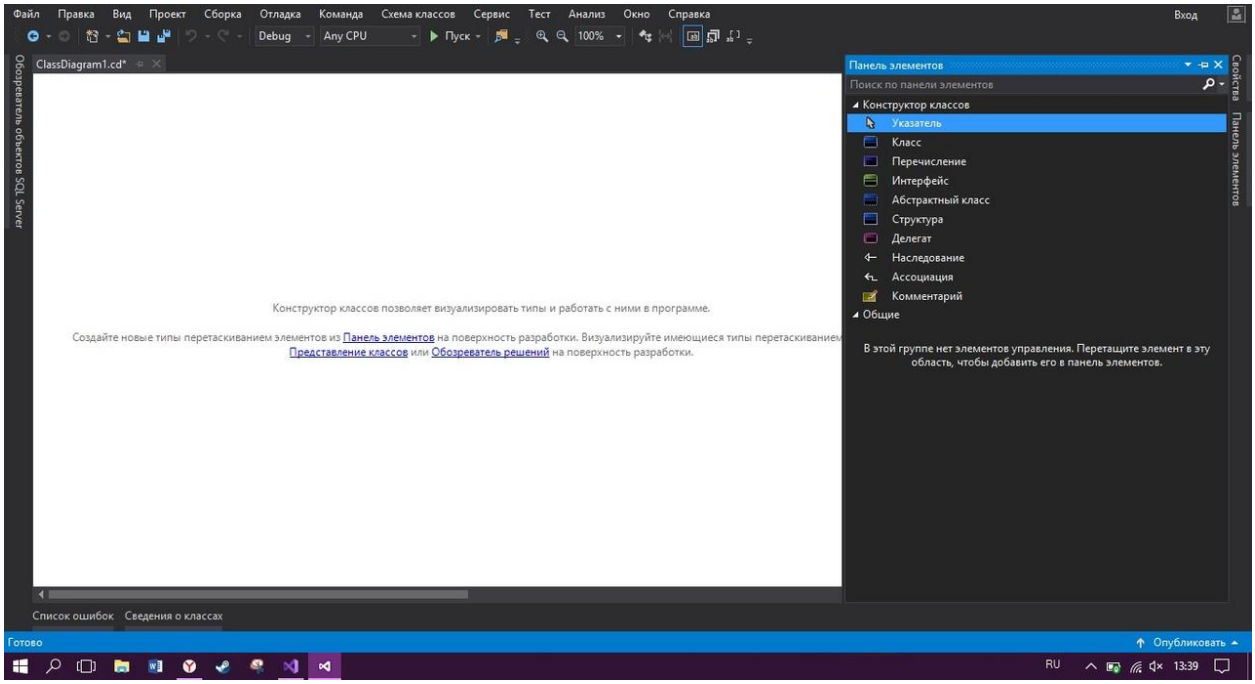


Рисунок 5.14. – Конструктор диаграммы классов

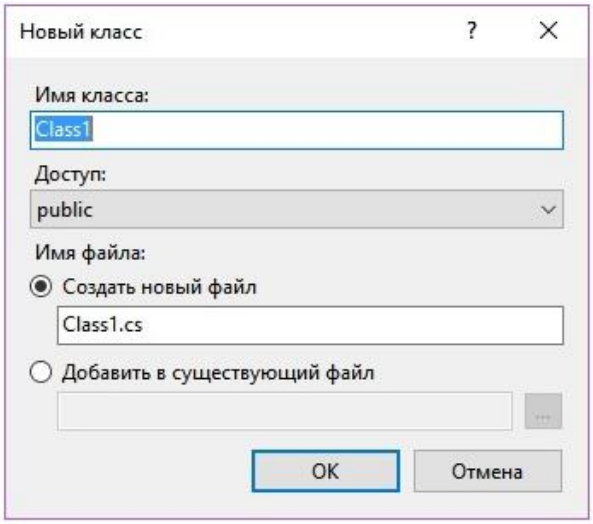


Рисунок 5.15. – Добавление нового класса

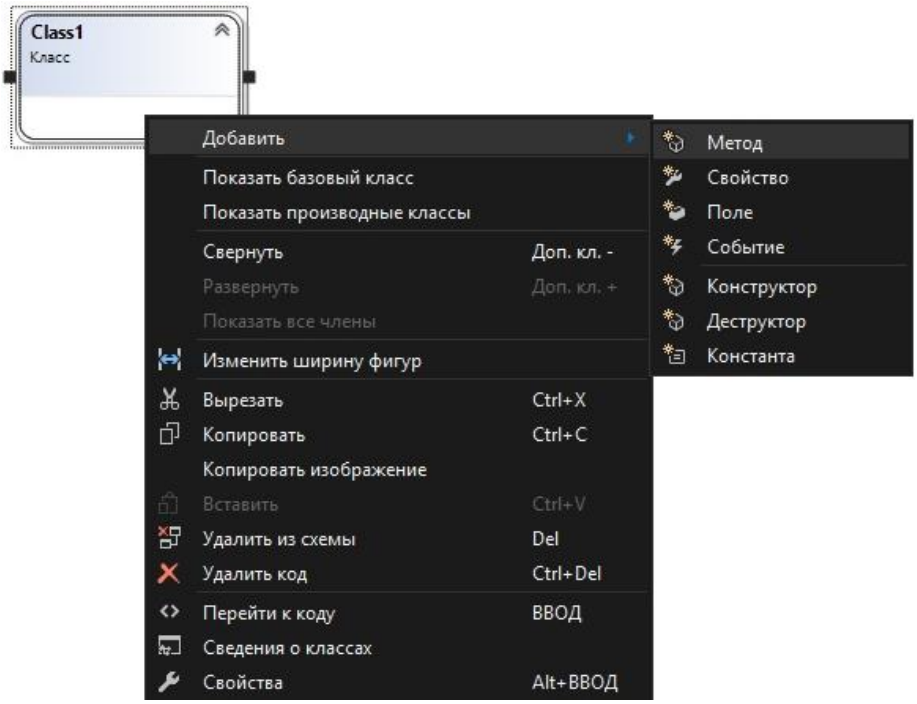


Рисунок 5.16. – Добавление элементов класса

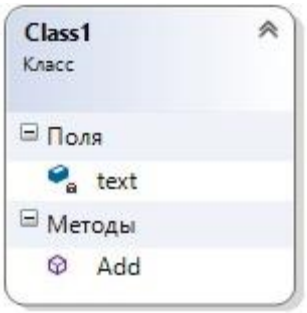


Рисунок 5.17. – Класс с полем и методом

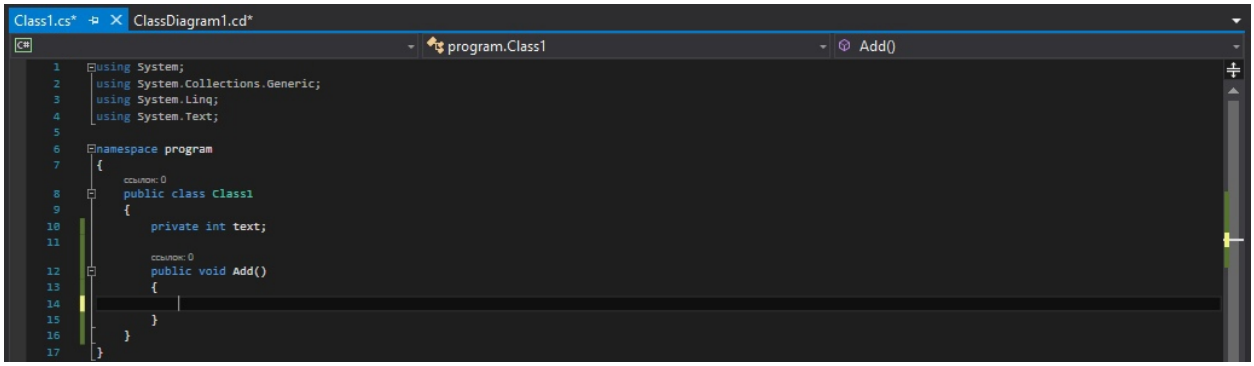


Рисунок 5.18. – Сгенерированный код

На рисунке 5.19 представлена диаграмма классов, созданная в Visual Studio, которая была продемонстрирована на рисунке 5.7.

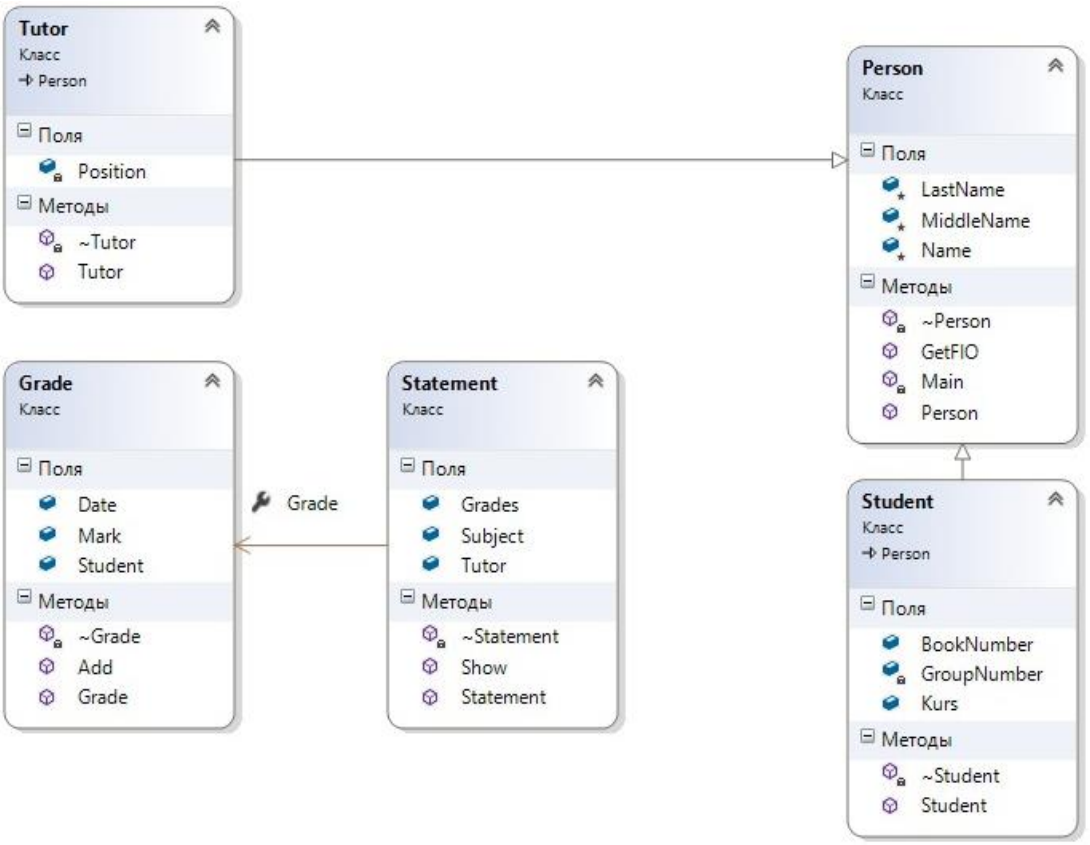


Рисунок 5.19. – Пример диаграммы классов реализованной при помощи Visual Studio

В листингах 5.1–5.5 представлен сгенерированный код данной диаграммы.

Рассмотрим сгенерированный код подробнее.

Начиная со строк 1–5, располагается директива using, которая разрешает использование типов в пространстве имен, поэтому уточнение использования типа в этом пространстве имен не требуется. Со строки 7 начинается пространство имен данного проекта. Оно используется для логической группировки классов, методов и переменных. В строке 9 объявляется класс Person. Строки 11–13 содержат поля (переменные) данного класса. В строке 15 объявляется конструктор данного класса; это специальный метод класса, который предназначен для инициализации элементов класса некоторыми начальными значениями. В строке 20 объявляется деструктор, служащий для уничтожения элементов класса.

Существует *несколько правил при использовании конструктора и деструктора*:

1. Конструктор и деструктор всегда имеют тип доступа public.
2. Тип данных возвращаемого значения не указывается, в т.ч. void.
3. Деструктору нельзя передавать никаких параметров.
4. Имя конструктора и деструктора соответствуют имени класса, за исключением одного, у деструктора имеется приставка ~ .
5. В классе может быть несколько конструкторов. Имена всех конструкторов будут одинаковыми. Программа различает конструкторы по передаваемым параметрам. Если

конструктор не имеет параметров, то он является пустым, и является конструктором по умолчанию.

6. В классе может быть объявлен только один деструктор.

Листинг 5.1. – Код класса Person

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.  using System.Threading.Tasks;
6.
7.  namespace metoda
8.  {
9.    public class Person
10.   {
11.     protected string Name;
12.     protected string LastName;
13.     protected string MiddleName;
14.
15.     public Person()
16.     {
17.       throw new System.NotImplementedException();
18.     }
19.
20.     ~ Person()
21.     {
22.       throw new System.NotImplementedException();
23.     }
24.
25.     public void GetFIO()
26.     {
27.       throw new System.NotImplementedException();
28.     }
29.
30.     static void Main(string[] args){}
31.   }
32. }
```

В строке 25 и 28 объявлен метод класса GetFIO. Строки 17, 22 и 27 содержат системное исключение «NotImplementedException()», которое означает, что метод не реализован. Если такой метод будет вызван, то метод сгенерирует исключение NotImplementedException.

Последующие листинги практически идентичны листингу главного класса, а главный он, поскольку все остальные классы являются его наследниками, на что указывают строки 8 в листингах 5.2 и 5.3.

Листинг 5.2. – Код класса Student

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace metoda
{
 public class Student : Person
  {
   private string TypeOfInsurance;

   public Student()
    {
     throw new System.NotImplementedException();
    }

   ~ Student()
    {
     throw new System.NotImplementedException();
    }
 }
}
```

Листинг 5.3. – Код класса Tutor

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace metoda
{
 public class Tutor : Person
  {
   private string Position;

   public Tutor()
    {
     throw new System.NotImplementedException();
    }

   ~ Tutor()
    {
     throw new System.NotImplementedException();
```

```
20.     }
21.   }
22. }
```

Листинг 5.4. – Код класса Grade

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.  using System.Threading.Tasks;
6.
7.  namespace metoda
8.  {
9.    public class Grade
10.   {
11.     protected string Student;
12.     protected int Mark;
13.     protected int Date;
14.
15.     public Grade()
16.     {
17.       throw new System.NotImplementedException();
18.     }
19.
20.     ~ Grade()
21.     {
22.       throw new System.NotImplementedException();
23.     }
24.
25.     public void Add()
26.     {
27.       throw new System.NotImplementedException();
28.     }
29.   }
30. }
```

Листинг 5.5. – Код класса Statement

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.  using System.Threading.Tasks;
```

```
6.
7.  namespace metoda
8.  {
9.    public class Statement
10.   {
11.     protected string Subject;
12.     protected int Grades;
13.     protected string Tutor;
14.
15.     public Statement()
16.     {
17.       throw new System.NotImplementedException();
18.     }
19.
20.     ~ Statement()
21.     {
22.       throw new System.NotImplementedException();
23.     }
24.
25.     public void Show()
26.     {
27.       throw new System.NotImplementedException();
28.     }
29.   }
30. }
```

Контрольные вопросы

1. Назовите назначение диаграммы классов.
2. Дайте определение понятиям атрибута и операции, переменной, параметра.
3. Назовите базовые типы атрибутов и методов.
4. Назовите и охарактеризуйте типы видимости атрибутов и методов.
5. Назовите и охарактеризуйте основные связи, существующие между классами.

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Разработанная диаграмма классов.
4. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab5-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab5-10-IT-Ivanov.zip

Литература

1. Буч, Гр. Введение в UML от создателей языка / Грейди Буч, Ивар Якобсон, Джеймс Рамбо. – М. : ДМК-Пресс, 2010. – 496 с. : ил.

2. Новиков, Ф.А. Моделирование на UML. Теория, практика, видеокурс / Ф.А. Новиков, Д.Ю. Иванов. – СПб. : Проф. лит., Наука и Техника, 2010. – 640 с.

3. Леоненков, А.В. Самоучитель UML / А.В. Леоненков. – 2-е изд. – СПб. : БХВ-Петербург, 2007. – 576 с. : ил.

4. Рамбо, Дж. UML 2.0. Объектно-ориентированное моделирование и разработка / Джеймс Рамбо, Майкл Блаха. – 2-е изд. – СПб: Питер, 2021. – 544 с.

5. Буч, Гр. Язык UML. Руководство пользователя / Грейди Буч, Джеймс Рамбо, Айвар Джекобсон ; пер. с англ. Н. Мухин. – 2-е изд. – М. : ДМК Пресс, 2006. – 496 с. : ил.

6. Гома, Хассан. UML. Проектирование систем реального времени, распределенных и параллельных приложений / Хассан Гома. – М. : ДМК-Пресс, 2007. – 700 с. : ил.

7. Хританков, А.С. Проектирование на UML / А.С. Хританков, В.А. Полежаев, А.И. Андрианов. – М. : Берлин : Directmedia, 2018. – 240 с.

Лабораторная работа 6
ОПТИМИЗАЦИЯ ПРОГРАММНОГО КОДА

Цель работы: изучить компоненты класса, реализовать функциональные блоки сгенерированных шаблонов.

Ход работы:

1) изучить теоретические сведения;
2) ответить на контрольные вопросы;
3) выполнить индивидуальное задание;
4) оформить отчет;
5) защитить лабораторную работу.

Термины и определения

Очень часто определяемые классы и другие типы в .NET не существуют сами по себе, а заключаются в специальные контейнеры – **пространства имен**. Пространства имен позволяют организовать код программы в логические блоки, объединить и отделить от остального кода некоторую функциональность, которая связана одной идеей или выполняет некоторую задачу.

Для определения пространства имен применяется ключевое слово *namespace*, после которого идет название пространства имен.

Методы класса содержат набор инструкций, которые выполняют определенные действия. Таким образом, метод – это именованный блок кода, который выполняет определенные действия. Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров, если таковые имеются.

Конструктор выполняет инициализацию объекта. При этом если в классе определяются собственные конструкторы, то он лишается конструктора по умолчанию.

В коде конструктор представляет метод, который называется по имени класса и может иметь параметры, но для него не надо определять возвращаемый тип.

Класс может хранить некоторые данные. Для хранения данных в классе применяются **поля**.

Переменная – определенная область памяти, в которую можно записать различные данные. Как правило, этой области памяти присваивается имя.

Поля класса являются переменными любого типа, которые объявлены непосредственно в классе. В отличие от переменных, определенных в методах, поля класса могут иметь модификаторы, которые указываются перед полем.

Свойства класса – привилегированные компоненты C#. Свойства при обращении к ним ведут себя как поля. Однако в отличие от полей свойства реализуются с помощью методов доступа, которые определяют инструкции, выполняемые при обращении к свойству или при его назначении.

Помимо полей класс может определять для хранения данных **константы**. В отличие от полей их значение устанавливается один раз непосредственно при их объявлении и впоследствии не может быть изменено. При этом константы хранят данные, которые относятся не к одному объекту, а ко всему классу в целом. И для обращения к константам используется не имя объекта, а имя класса.

Исключительные ситуации (исключения) – ошибки, возникающие при выполнении программы, которые трудно (а иногда и вовсе невозможно) предусмотреть или предвидеть. Например, при передаче файла по сети может неожиданно оборваться сетевое подключение. В языке C# разработчикам предоставлены возможности для обработки таких ситуаций с помощью конструкции *try … catch … finally*.

Далее рассмотрим описанные элементы класса на примере.

Пример реализации классов

Реализация программного кода классов выполняется с целью решения задачи предметной области согласно варианту задания (приложении А). Ранее в ходе выполнения лабораторной работы 5 («Модели и генерирование кода на основе UML-диаграмм») были сгенерированы классы, описанные диаграммой классов. Далее сгенерированные классы необходимо описать (реализовать).

Сгенерированные в среде разработки классы содержат неописанные (нереализованные) методы и конструкторы. Нереализованные методы не содержат полезного программного кода, а вместо этого генерируют исключение *NotImplementedException*. Исключение генерируется каждый раз при вызове нереализованного метода и позволяет разработчику не пропустить не описанные фрагменты кода.

В качестве примера для реализации классов будем рассматривать информационную систему управления ведомостью, которая описывалась лабораторной работе 5. Реализуемые классы должны выполнять следующую задачу: формировать ведомость по заданной учебной дисциплине. В процессе создания ведомости указывается дисциплина и преподаватель. После этого пользователь может добавлять выставляемые оценки с указанием студента и даты. Также есть возможность удаления ошибочно внесенных оценок и формирования итоговой ведомости в удобном для восприятия виде с выводом на экран.

Для реализации указанных возможностей системы выделяются следующие сущности и классы для них: Человек – *Person*, Преподаватель – *Tutor*, Студент – *Student*, Оценка – *Grade* и Ведомость – *Statement*. Каждая сущность имеет свойства и методы, реализующие необходимую функциональность.

Сгенерированный класс *Person*, содержащий вызовы исключения *NotImplementedException*, представлен в листинге 6.1.

Листинг 6.1. – Сгенерированный класс Person

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.  using System.Threading.Tasks;
6.
7.  namespace metoda
8.  {
9.    public class Person
10.   {
```

```
11.     protected string Name;
12.     protected string LastName;
13.     protected string MiddleName;
14.
15.     public Person()
16.     {
17.       throw new System.NotImplementedException();
18.     }
19.
20.     ~ Person()
21.     {
22.       throw new System.NotImplementedException();
23.     }
24.
25.     public void GetFIO()
26.     {
27.       throw new System.NotImplementedException();
28.     }
29.
30.     static void Main(string[] args){}
31.   }
32. }
```

Участки класса с кодом *throw new System.NotImplementedException()* заменяем на полезную логику. Пример реализованного класса *Person* представлен в листинге 6.2.

Листинг 6.2. – Реализованный класс Person

```
1.   namespace TrpoLab
2.   {
3.     class Person
4.     {
5.       public string FirstName { get; set; }
6.       public string LastName { get; set; }
7.       public string MiddleName { get; set; }
8.
9.       public Person(string lastName, string firstName,
     string middleName)
10.     {
11.       FirstName = firstName;
12.       LastName = lastName;
13.       MiddleName = middleName;
14.     }
15.
16.     public string GetGIO()
17.     {
```

```
18.         return LastName + " " + FirstName + " " +
 MiddleName;
19.     }
20.   }
21. }
```

В приведенном коде класса Person можно выделить следующие элементы.

В строке 1 задано пространство имен *TrpoLab* с использованием ключевого слова *namespace*. В строке 3 указано название класса *Person* с использованием ключевого слова *class*. Далее в строках 5, 6 и 7 описаны три строковых свойства для Имени, Фамилии и Отчества с модификатором доступа *public*. Полное определение свойства содержит два блока: *get* и *set*. В блоке *get* выполняются действия по получению значения свойства. В этом блоке с помощью оператора *return* возвращается некоторое значение. В блоке *set* устанавливается значение свойства. В этом блоке с помощью параметра *value* можно получить значение, которое передано свойству. Блоки *get* и *set* еще называются акссесорами или методами доступа к значению свойства, а также геттером и сеттером. В .NET реализованы автоматические свойства. Они имеют сокращенное объявление *{ get; set; }*.

Строки с 9 по 14 содержат реализацию конструктора класса. Для создания объекта класса *Person* необходимо передать в конструктор три параметра: фамилию, имя и отчество. В строках с 11 по 13 происходит присвоение переданных в конструктор значений свойствам созданного объекта класса.

Строки с 16 по 19 содержат реализацию метода *GetFIO*, предназначенного для отображения полной информации о человеке, а именно фамилии, имени и отчества разделенных пробелом. Данный метод не принимает параметров и возвращает результат в виде строкового значения. Строка 18 начинается с ключевого слова *return*. *Return* означает, что выполнение метода заканчивается и метод возвращает значение, описанное после ключевого слова *return*.

Далее рассмотрим сущность Преподаватель, реализованную в классе *Tutor*, код которого представлен в листинге 6.3.

Листинг 6.3. – Реализованный класс Tutor

```
1.  namespace TrpoLab
2.  {
3.    class Tutor : Person
4.    {
5.      public string Position { get; set; }
6.
7.      public Tutor(string lastName = "", string firstName =
    "", string middleName = "", string position = "")
8.          : base(lastName, firstName, middleName)
9.      {
10.         Position = position;
11.     }
12.
13.     public string About()
```

```
14.     {
15.         return GetGIO() + ", должность: " + Position;
16.     }
17.   }
18. }
```

В приведенном коде класса Tutor можно выделить следующие элементы.

В строке 1 задано пространство имен *TrpoLab* с использованием ключевого слова *namespace*. В строке 3 указано название класса *Tutor* с использованием ключевого слова class, а также указано наследование от класса Person с использование символа двоеточие и последующим указанием родительского класса. Далее в строке 5 описано строковое свойство *Position* для Должности с модификатором доступа *public*.

Строки с 7 по 11 содержат реализацию конструктора класса. Для создания объекта класса *Tutor* необходимо передать в конструктор четыре параметра: фамилию, имя, отчество и должность. Здесь также следует отметить использование конструктора родительского Person с помощью ключевого слова *base*, которому передаются параметры Фамилия, Имя и Отчество.

Метод для отображения информации о преподавателе *About* реализован в строках 13–16. Данные метод не принимает параметров и возвращает строковое значение, содержащее ФИО преподавателя с указанием его должности.

Следующим рассмотрим сущность Студент, реализованную в классе *Student*, код которого представлен в листинге 6.4.

Листинг 6.4. – Реализованный класс Student

```
1.  namespace TrpoLab
2.  {
3.    class Student : Person
4.    {
5.      public ushort Year { get; set; }
6.      public ulong GradebookNumber { get; set; }
7.      public string Group { get; set; }
8.
9.      public Student(string lastName = "", string firstName
    = "", string middleName = "", ushort year = 0,
   ulong gradebookNumber = 0, string group = "")
10.         : base(lastName, firstName, middleName)
11.     {
12.         Year = year;
13.         GradebookNumber = gradebookNumber;
14.         Group = group;
15.     }
16.
17.     public string Info()
18.     {
```

```
19.         return Year + ", " + Group + ", " +
 GradebookNumber;
20.     }
21.   }
22. }
```

В приведенном коде класса *Student* можно выделить следующие элементы.

В 1-й строке задано пространство имен *TrpoLab* с использованием ключевого слова *namespace*. В 3-й строке указано название класса *Student* с использованием ключевого слова *class*, а также указано наследование от класса *Person* с использование символа двоеточие и последующим указанием родительского класса. Далее в строках 5, 6 и 7 описаны два числовых свойства для Курса и Номера зачетной книжки и строковое свойство Группа с модификаторами доступа *public*.

Строки с 9 по 15 содержат реализацию конструктора класса. Для создания объекта класса *Student* необходимо передать в конструктор 6 параметров: фамилию, имя, отчество, курс, номер зачетной книжки и группу. Здесь также следует отметить использование конструктора родительского *Person* с помощью ключевого слова *base*, которому передаются параметры Фамилия, Имя и Отчество.

Метод для отображения информации о студенте *Info* реализован с 17 по 20 строки. Данные метод не принимает параметров и возвращает строковое значение, содержащее курс, группу и номер зачетной книжки студента разделенные пробелами.

Далее рассмотрим сущность Оценка, реализованную в классе *Grade*, код которого представлен в листинге 6.5.

Листинг 6.5. – Реализованный класс Grade

```
1.   namespace TrpoLab
2.   {
3.     class Grade
4.     {
5.       public byte Mark { set; get; }
6.       public string DateOfIssue { set; get; }
7.       public Student Student { set; get; }
8.
9.       public Grade(byte mark = 0, string dateOfIssue = "",
    Student student = null)
10.      {
11.        Mark = mark;
12.        DateOfIssue = dateOfIssue;
13.        Student = student;
14.      }
15.
16.      public string Print()
17.      {
18.        return Student.GetGIO() + " (" + Student.Info() +
     ")| " + Mark + " | " + DateOfIssue;
```

```
19.      }
20.    }
21.  }
```

В приведенном коде класса *Grade* можно выделить следующие элементы.

В строке 1 задано пространство имен *TrpoLab* с использованием ключевого слова *namespace*. В строке 3 указано название класса *Grade* с использованием ключевого слова *class*. Далее в строках 5, 6 и 7 описаны три свойства с модификатором доступа *public*: числовое свойство для отметки, строковое свойство для даты выставления и свойство типа *Student* для информации о студенте.

Строки с 9 по 14 содержат реализацию конструктора класса. Для создания объекта класса *Person* необходимо передать в конструктор 3 параметра: отметка, дата и студент.

Строки с 16 по 19 содержат реализацию метода *Print*, предназначенного для отображения полной информации об оценке, а именно ФИО студента, информацию о нем, отметку и дату ее выставления. Данный метод не принимает параметров и возвращает результат в виде строкового значения.

Также рассмотрим сущность Ведомость, реализованную в классе *Statement*, код которого представлен в листинге 6.6.

Листинг 6.6. – Реализованный класс Statement

```
1.  using System;
2.  using System.Collections.Generic;
3.
4.  namespace TrpoLab
5.  {
6.    class Statement
7.    {
8.      protected string Subject;
9.      protected Tutor Tutor;
10.     protected List<Grade> Grades;
11.
12.     public Statement(string subject = "",
    Tutor tutor = null)
13.     {
14.       Subject = subject;
15.       Tutor = tutor;
16.       Grades = new List<Grade>();
17.     }
18.
19.     public void Add(Grade grade)
20.     {
21.       Grades.Add(grade);
22.     }
23.
24.     public void Erase(int item)
```

```
25.     {
26.       Grades.RemoveAt(item - 1);
27.     }
28.
29.     public void Clear()
30.     {
31.       Grades.Clear();
32.     }
33.
34.     public int GetCount()
35.     {
36.       return Grades.Count;
37.     }
38.
39.     public void Print()
40.     {
41.       if(Grades.Count > 0)
42.       {
43.         int index = 1;
44.         Console.WriteLine("------------------------------
    ------------------------------------------");
45.         Console.WriteLine("Ведомость");
46.         Console.WriteLine("Дисциплина: " + Subject);
47.         Console.WriteLine("Преподаватель: " +
    Tutor.About());
48.         Console.WriteLine("№ п/п | ФИО студента (курс,
    группа, номер зачетной книжки) | Оценка | Дата");
49.         foreach (Grade grade in Grades)
50.         {
51.           Console.WriteLine(index + " | " +
    grade.Print());
52.           index++;
53.         }
54.         Console.WriteLine("------------------------------
     ------------------------------------------");
55.       }
56.       else
57.       {
58.         Console.WriteLine("Ведомость пуста!");
59.       }
60.     }
61.   }
62. }
```

В приведенном коде класса *Statement* можно выделить следующие элементы.

В первых двух строках осуществляется подключение пространств имен *System* и *System.Collections.Generic* с использованием ключевого слова *using*.

System.Collections.Generic подключается к проекту, чтобы иметь возможность использовать класс *List<T>*. Класс *List* позволяет управлять списком объектов. В качестве основных возможностей коллекции являются методы добавления элемента (*Add*), удаления элемента (*Remove*), очистки списка (*Clear*), сортировки элементов (*Sort*), а также свойства *Count* для получения количества элементов списка. При объявлении коллекции указывается класс объекта, элементы которого она хранит. Указывается класс элементов коллекции через треугольные скобки *<T>* (пример: *List<Grade>*).

Для вывода информации на консоль используется встроенный метод *Console.WriteLine* в пространстве имен *System*. Для вывода необходимой информации в консоль следует передать ее в метод *Console.WriteLine*. Например, Следующий код *Console.WriteLine("Ведомость пуста!");* отобразит в консоли надпись: «Ведомость пуста!», показанную на рисунке 6.1.

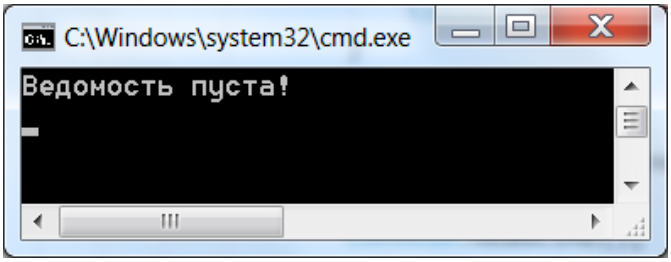


Рисунок 6.1. – Консольный вывод командой *Console.WriteLine*

В строке 4 задано пространство имен *TrpoLab* с использованием ключевого слова *namespace*. В строке 6 указано название класса *Statement* с использованием ключевого слова *class*.

Далее в строках 8, 9 и 10 описаны три свойства с модификатором доступа *protected*: строковое свойство для названия дисциплины (*Subject*), свойство типа *Tutor* для информации о преподавателе (*Tutor*) и свойство типа *List<Grade>* для информации об оценках (*Grades*).

Строки с 12 по 17 содержат реализацию конструктора класса. Для создания объекта класса *Statement* необходимо передать в конструктор два параметра: название дисциплины и преподаватель.

Далее реализованы методы, необходимые для работы со списком оценок: *Add* для добавление новой оценки, *Erase* для удаления существующей оценки из списка, *Clear* для очистки списка оценок и *GetCount* для получения количества оценок в списке.

Строки с 39 по 60 содержат реализацию метода *Print*, предназначенного для отображения ведомости с информацией о дисциплине, преподавателе и внесенных оценках. Данный метод не принимает параметров и не возвращает результата, выводя всю информацию в консоль.

В строке 41 используется условный оператор *if*, который реализует ветвление кода в зависимости от условия. В данном случае если количество элементов в списке больше нуля, то отображается содержимое ведомости (строки 43 – 54), иначе отображается сообщение о том, что ведомость пуста (строка 58).

В строке 43 объявляется переменная *index* типа *int*, которая содержит порядковый номер записи в ведомости. Для перехода к следующему порядковому номеру в строке 52 используется операция инкремента *++*. Эта операция увеличивает значение переменной *index* на 1.

В строках с 44 по 48 выводится заголовок ведомости в консоль.

В строке 49 используется оператор цикла *foreach*, который позволяется выполнять действия со всеми объектами списка. Это необходимо для вывода информации о каждой записи в списке.

В строке 54 выводится окончание ведомости.

В результате на представленном примере были реализованы сгенерированные классы, которые содержат внутреннюю логику работы самого класса. Взаимодействие между классами будет реализовываться в лабораторной работе 7 («Разработка программного обеспечения для проектируемой предметной области»).

Контрольные вопросы

1. Поясните понятие и назначение полей класса.
2. Поясните понятие и назначение методов класса.
3. Охарактеризуйте особенности констант класса.
4. Для чего применяются пространства имен?
5. Для чего применяются конструкторы классов?

Задание

Реализовать методы классов в соответствии с их функциями и определить области видимости для отдельных элементов согласно индивидуальному варианту задания из предыдущей лабораторной работы. Варианты индивидуальных заданий представлены в приложении А.

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Листинги кода классов с комментариями.
4. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab6-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab6-10-IT-Ivanov.zip

Лабораторная работа 7
РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ПРОЕКТИРУЕМОЙ ПРЕДМЕТНОЙ ОБЛАСТИ

Цель работы: изучить принципы проектирования ПО для конкретной предметной области, разработать ПО согласно индивидуальному заданию.

Ход работы:

1) изучить теоретическую часть;
2) ответить на контрольные вопросы;
3) выполнить индивидуальное задание;
4) оформить отчет;
5) защитить лабораторную работу.

Реализация логики взаимодействия между классами

Реализация бизнес-логики задания осуществляется путем описания взаимодействия между классами. Классы взаимодействуют между собой тогда, когда в параметры конструктора или методов передаются объекты других классов [1]. Например, чтобы в ведомость добавить выставленную отметку, необходимо создать экземпляр класса *Grade* и передать его в метод *Add* экземпляра класса *Statement* (листинг 7.1).

Листинг 7.1. – Реализация добавления отметки в ведомость

```
1.  Tutor tutor = new Tutor(“Petrov”, “Petr”, “Petrovich”, “professor”);
2.  Statement statement = new Statement(“Programming”, tutor);
3.  Student student = new Student();
4.  student.LastName = “Ivanov”;
5.  student.FirstName = “Ivan”;
6.  student.MiddleName = “Ivanovich”;
7.  student.Year = 1;
8.  student.GradebookNumber = 123456;
9.  student.Group = “20-IT”;
10. Grade grade = new Grade
11. {
12.   Student = student
13. };
14. grade.Mark = 10;
15. grade.DateOfIssue = “10.10.2020”;
16. statement.Add(grade);
```

В приведенном листинге описана последовательность действия для создания и добавления оценки в ведомость.

Для этого в строке 1 описывается преподаватель с помощью экземпляра класса *Tutor* с указанием ФИО и должности преподавателя в конструкторе. Далее в строке 2

создается ведомость в виде экземпляра класса *Statement*, в конструктор которой передается название дисциплины и сведения о преподавателе в виде объекта класса *Tutor*. В строке 3 создается экземпляр класса *Student*.

В строках с 4 по 9 задаются свойства объекта *student* с использованием статических данных. В строках с 10 по 13 создается экземпляр класса *Grade* для описание выставляемой отметки. При его создании описывается свойство *Student*. В строках 14 и 15 задаются свойства объекта *grade* с использованием статических данных. В 16 строке выполняется добавление отметки (экземпляра класса *Grade*) в ведомость (экземпляр класса *Statement*) с использованием метода ведомости *Add*, принимающего в виде параметра отметку.

Реализация текстового интерактивного интерфейса

Чтобы создавать новые объекты и получать данные для создаваемых объектов от пользователя, необходимо реализовать интерфейс взаимодействия с пользователем. Наиболее простой для реализации интерфейс – консольный ввод данных (интерфейс командной строки).

Вывод данных на консоль осуществляется с помощью методов *Write* и *WriteLine* класса *Console*.

Console.Write выводит в консоль текстовую информацию, переданную в качестве параметра без добавления дополнительных спецсимволов.

Console.WriteLine аналогична команде *Console.Write* за исключением того, что добавляет в конце спецсимвол окончания строки.

Пример формирования текстового меню с помощью описанных команд приведен в листинге 7.2.

Листинг 7.2. – Формирование текстового меню командами вывода на консоль

```
1. Console.WriteLine("1 – Очистить ведомость");
2. Console.WriteLine("2 - Внести оценку в ведомость");
3. Console.WriteLine("3 - Удалить оценку из ведомости");
4. Console.WriteLine("4 – Отобразить ведомость");
5. Console.WriteLine("5 - Выход");
6. Console.Write("> ");
```

Ввод данных осуществляется методом *ReadLine* класса *Console*. *Console.ReadLine* осуществляет считывание строкового значения, введенного пользователем в консоль, после нажатия клавиши Ввод. Пример считывания данных в процессе внесения сведений о студенте представлен в листинге 7.3.

Листинг 7.3. – Считывание данных в консоли

```
1.  Student student = new Student();
2.  student.LastName = Console.ReadLine();
3.  student.FirstName = Console.ReadLine();
4.  student.MiddleName = Console.ReadLine();
5.  student.Group = Console.ReadLine();
```

В приведенном листинге в строке 1 создается экземпляр класса *Student* для описания учащегося. Далее в строках с 2 по 5 осуществляется консольный ввод свойств созданного объекта с использованием метода *ReadLine*.

Для интерактивного взаимодействия с пользователем могут применяться циклы, которые позволяют считывать данные до тех пор, пока не пользователь не введет корректное значение. Для примера в листинге 7.4 представлена проверка корректности ввода курса студента с учетом того, что значение должно быть целочисленным.

Листинг 7.4. – Проверка вводимых данных

```
1.  string tmpStr = "";
2.  ushort tmpValUshort = 0;
3.  bool correctInput = false;
4.  do
5.  {
6.    Console.Write("Введите курс студента: ");
7.    tmpStr = Console.ReadLine();
8.    correctInput = ushort.TryParse(tmpStr,out tmpValUshort);
9.    if (!correctInput)
10.   {
11.     Console.WriteLine("Ошибка ввода!\r\nПовторите ввод");
12.   }
13.   else
14.   {
15.     student.Year = tmpValUshort;
16.   }
17. }
18. while (!correctInput);
```

В приведенном листинге используется цикл с постусловием *do-while*. Это обеспечивает проверку корректности введенных данных после пользовательского ввода в консоль.

В строках с 1 по 3 объявлены вспомогательные переменные. Строковая переменная *tmpStr* используется для промежуточного хранения вводимых данных в символьном виде. Переменная *tmpValUshort* имеет беззнаковый целочисленный тип и предназначена для промежуточного хранения результата преобразования введенных данных в числовое значение. Переменная *correctInput* содержит логическое значение результата преобразования. В случае успешного выполнения преобразования из строки в число она принимает значение истина (*true*), иначе ложь (*false*).

Строки 4 и 18 содержат управляющие конструкции цикла. В строке 18 выполняется проверка условия продолжения из цикла. В строке 6 отображается приглашение для ввода курса студента с использованием метода консольного вывода *Console.WriteLine*. В строке 7 выполняется считывание введенной пользователем строки в промежуточную переменную с помощь метода *Console.ReadLine*. В строке 8 используется метод *TryParse*, принимающий в качестве аргументов строковое значение и переменную для записи числового значения. В результате выполнения возвращает

логическое значение и при успешном преобразовании записывает полученное значение в выходную переменную.

Строки с 9 по 16 реализуют проверку результата преобразования с помощью условного оператора if-else. В случае ложности переменной *correctInput* в строке 11 отображается сообщение об ошибке введенных данных и приглашение к повторному вводу. Иначе в строке 15 числовое значение преобразования *tmpValUshort* сохраняется в свойстве *Year* экземпляра класса студента *student*. Результат выполнения проверки представлен на рисунке 7.1.

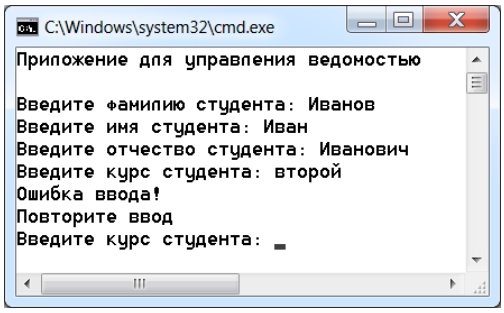


Рисунок 7.1. – Выполнение проверки вводимых данных

Навигация пользователя между действиями осуществляется с помощью текстового интерактивного меню, где цифрой обозначается действие, которое будет совершать пользователь. Такая реализация навигации по текстовому меню представлена в листинге 7.5.

Листинг 7.5. – Реализация навигации по текстовому меню

```
1.  byte selectItem = 0;
2.  while(selectItem != 5)
3.  {
4.    switch (selectItem)
5.    {
6.      case 0:
7.        Console.WriteLine("Выберите действие:");
8.        //...
9.        break;
10.     case 1:
11.       Console.Write("Ведомость очищена");
12.       //...
13.       break;
14.     case 2:
15.       Console.Write("Введите фамилию студента: ");
16.       //...
17.       break;
18.     case 3:
19.       Console.Write("Введите номер удаляемой записи: ");
20.       //...
```

```
21.       break;
22.     case 4:
23.       Console.Write("Нажмите \"Ввод\" для продолжения");
24.       //...
25.       break;
26.     default:
27.       selectItem = 0;
28.       break;
29.   }
30. }
```

Результат сформированного текстового меню представлен на рисунке 7.2.

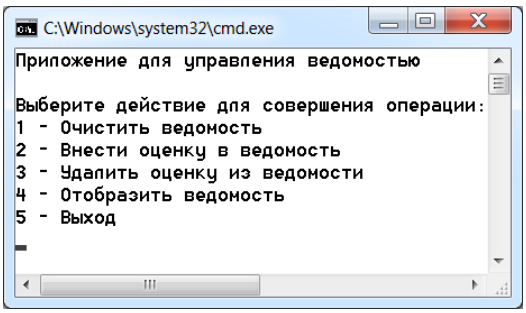


Рисунок 7.2. – Отображение текстового меню

Полный листинг программы приведен в приложении Б. Демонстрация логики работы программы приводится на снимках экрана (рисунки 7.3–7.5).

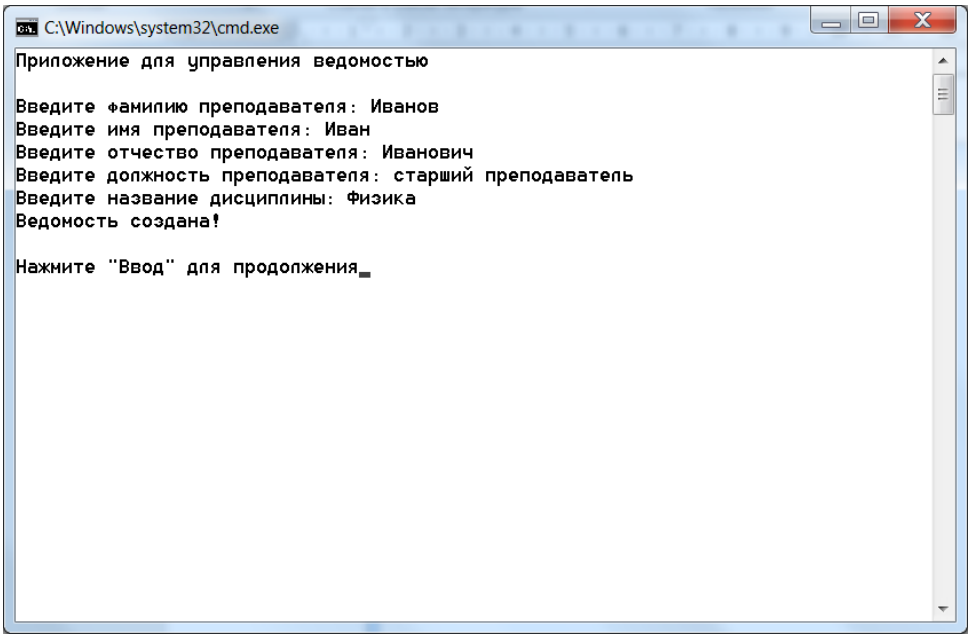


Рисунок 7.3. – Меню создания ведомости

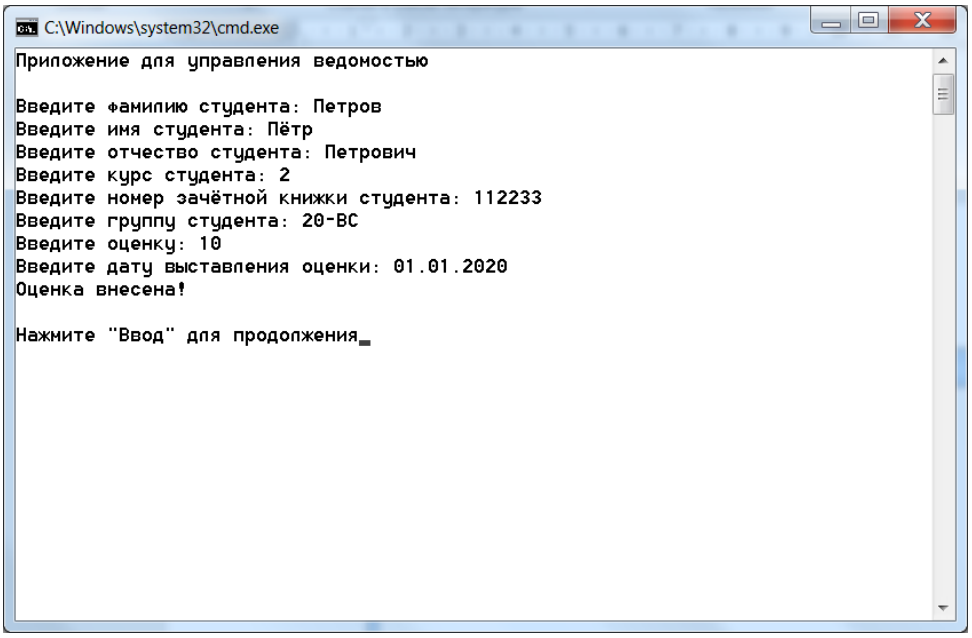


Рисунок 7.4. – Меню добавления оценки

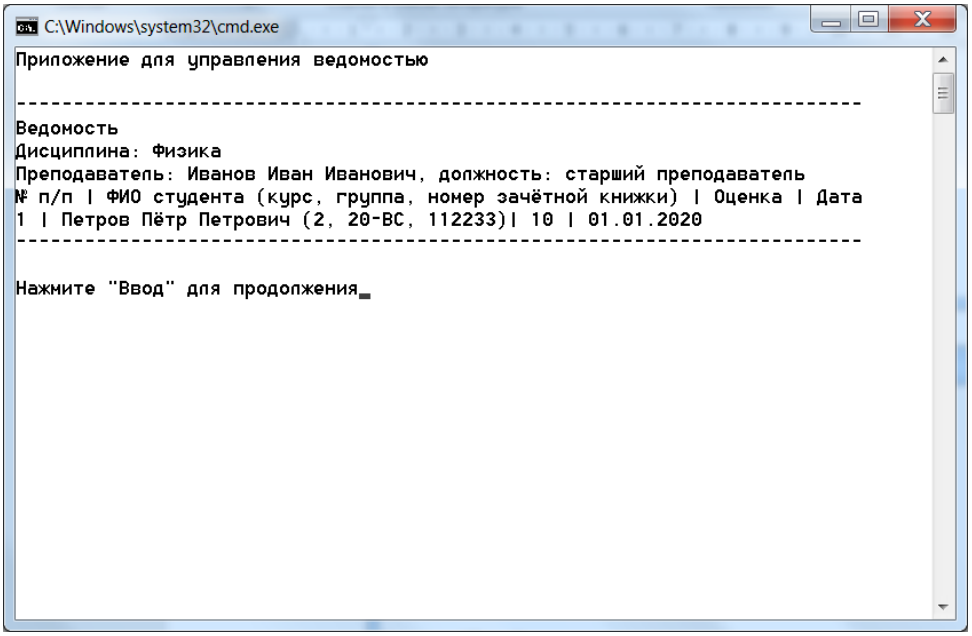


Рисунок 7.5. – Отображение ведомости

Контрольные вопросы

1. Для чего предназначается пользовательский интерфейс?
2. Перечислите и охарактеризуйте разновидности пользовательского интерфейса.
3. Какой оператор используется для построения интерактивного текстового меню?
4. Какие операторы используются для проверки корректности вводимых данных?
5. Перечислите и охарактеризуйте команды для взаимодействия с консолью.

Задание

Реализовать приложение с интерактивным текстовым меню согласно индивидуальному варианту задания из предыдущей лабораторной работы. Варианты индивидуальных заданий представлены в приложении А

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Листинги кода классов с комментариями.
4. Скриншоты выполнения программы.
5. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab7-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab7-10-IT-Ivanov.zip

Литература

1. Loeliger, Jon. Version Control with Git / Jon Loeliger, Matthew McCullough. – 2nd ed. – O'Reilly Media, Inc., 2012. – 456 p.

Лабораторная работа 8
РАБОТА С СИСТЕМОЙ КОНТРОЛЯ ВЕРСИЙ

Цель работы: изучить работу с системой контроля версий git, загрузить полученные коды в репозиторий.

Ход работы:
1) изучить теоретическую часть;
2) ответить на контрольные вопросы;
3) выполнить индивидуальное задание;
4) оформить отчет.

Термины и определения

Репозиторий (англ. *repository*) – центральное хранилище, которое содержит версии файлов исходных кодов.

Версия файла (англ. *revision*) – состояние файла в определенный момент времени. Репозиторий предоставляет возможность хранить неограниченное число версий одного и того же файла.

Актуальная версия файла – обычно это самая последняя версия файла, размещенного в репозитории.

Рабочая версия файла (англ. *working copy*) – версия файла, с которой в текущий момент ведется работа и которая не загружена в репозиторий.

Загрузка (англ. *upload*) – размещение файла в репозитории. В процессе загрузки в репозиторий помещается рабочая версия файла.

Выгрузка (англ. *checkout*) – получение файла из репозитория. В процессе выгрузки осуществляется получение из репозитория необходимой версии файла.

Синхронизация (англ. *update, sync*) – последовательное копирование всех изменений из удаленного репозитория на локальную машину и из локального репозитория в удаленный.

Ветвь (англ. *branch*) – процесс разработки, независимый от других. Ветвь представляет собой копию части (как правило, одного каталога) хранилища, в которую можно вносить свои изменения, не влияющие на другие ветви. Документы в разных ветвях имеют одинаковую историю до точки ветвления и разные – после нее.

Ствол (англ. *trunk, mainline*) – основная ветвь разработки проекта. Политика работы со стволом может отличаться от проекта к проекту, но в целом она такова: большинство изменений вносится в ствол. Если требуется значительное изменение, способное привести к нестабильности, создается ветвь, которая сливается со стволом, когда нововведение будет в достаточной мере испытано. Перед выпуском очередной версии создается отдельная ветвь, в которую вносятся только исправления.

Метка (англ. *tag, label*) – метка, которую можно присвоить определенной версии документа. Метка представляет собой символическое имя для группы документов, причем метка описывает не только набор имен файлов, но и ревизию каждого файла.

Конфликт – ситуация, когда несколько пользователей сделали изменения одного и того же участка документа. Конфликт обнаруживается, когда один пользователь опубликовал свои изменения, а второй пытается опубликовать и система сама не может корректно слить конфликтующие изменения. Второму пользователю нужно самому разрешить.

Коммит (англ. *commit)* – это основной объект в управлении контроля версий. Он содержит все изменения за время этого коммита. Коммиты связаны между с собой как односвязный список. Так, имеется первый коммит. Когда создается второй коммит, то он (второй) знает, что идет после первого. И таким образом можно отследить всю историю изменений. Также у коммита есть дополнительная информация, так называемые метаданные: уникальный идентификатор коммита, по которому можно его найти; имя автора коммита, который создал его; дата создания коммита; комментарий, который описывает, что было сделано во время этого коммита.

Система управления версиями (*система контроля версий, СКВ,* англ. *Version Control System, VCS*) – программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведется работа с большим количеством непрерывно изменяющихся электронных документов. В частности, системы управления версиями применяются в системах автоматизированного проектирования, обычно в составе систем управления данными об изделии [1].

Виды систем контроля версий:

– локальные системы контроля версий;
– централизованные системы контроля версий;
– децентрализованные системы контроля версий.

Локальные системы контроля версий

Наиболее простым решением в качестве метода контроля версий является копирование файлов в отдельный каталог (возможно, каталог с отметкой по времени). Данный подход очень распространен из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в каком каталоге находились, и случайно изменить не тот файл или скопировать не те файлы, которые необходимо.

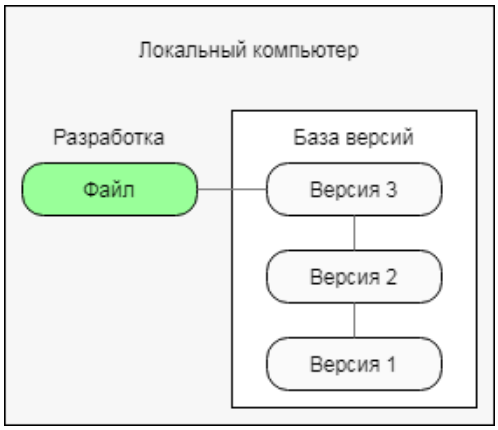


Рисунок 8.1. – Упрощенная схема локальной системы контроля версий

Для того чтобы решить эту проблему, программисты давно разработали локальные СКВ с простой базой данных, которая хранит записи обо всех изменениях в файлах, осуществляя тем самым контроль ревизий [2].

Упрощенная схема локальной системы контроля версий – см. рисунок 8.1.

Одной из популярных СКВ была система RCS. Эта СКВ хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени. Удобна в использовании единственным пользователем на одном компьютере.

Централизованные системы контроля версий

Следующая серьезная проблема, с которой сталкиваются, – это необходимость взаимодействия разработчика с другими разработчиками. Для того чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет [3].

На рисунке 8.2 представлена упрощенная схема централизованной системы контроля версий.

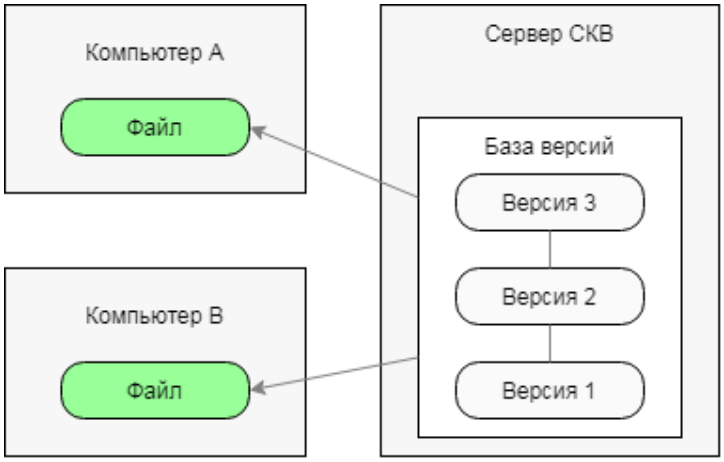


Рисунок 8.2. – Упрощенная схема
централизованной системы контроля версий

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определенной степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Однако данный подход тоже имеет серьезные минусы. Самый очевидный из них – это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жесткий диск, на котором хранится центральная база данный, поврежден, а своевременные резервные копии отсутствуют, будет утеряна вся история проекта, не считая единичных снимков

репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, есть риск лишиться всего.

Распределенные системы контроля версий

Озвученные ранее проблемы решают распределенные системы контроля версий (РСКВ). В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определенный момент времени) – они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, откажет, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полной резервной копией всех данных [4].

На рисунке 8.3 представлена упрощенная схема распределенной системы контроля версий.

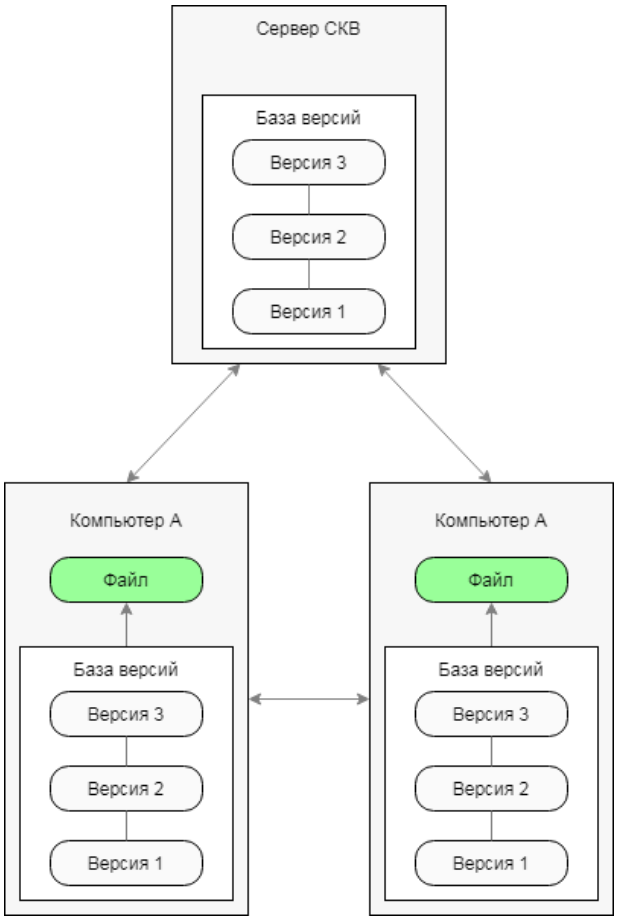


Рисунок 8.3. – Упрощенная схема распределенной системы контроля версий

Кроме того, многие РСКВ могут одновременно взаимодействовать с несколькими удаленными репозиториями, что дает возможность работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

В настоящее время git является наиболее популярной СКВ благодаря удобству использования и открытости исходных кодов. Далее рассмотрим его основные консольные команды.

Основные команды git

```
git init | git init [folder]
```

Эта команда используется для инициализации пустого репозитория из папки, в которой используется эта команда, или по пути к папке, – оба способа верны. Команда используется при запуске нового проекта или если необходимо инициализировать git-репозиторий внутри существующего проекта.

```
git clone [repo URL] [folder]
```

Эта команда используется для копирования существующего репозитория в указанную папку на вашем компьютере. Она может использоваться только с URL-репозитория в качестве параметра, и тогда он скопирует репозиторий в папку, из которой использована команда.

Если необходимо скопировать репозиторий в другое место на компьютере, следует добавить путь к папке в качестве второго параметра.

```
git add [directory | file]
```

Эта команда добавляет изменение из рабочего каталога в раздел проиндексированных файлов. Она добавляет необходимость включения изменения в конкретном файле в следующий коммит. При этом данная команда не оказывает существенного влияния на репозиторий, т.к. изменения регистрируются в нем только после выполнения команды *git commit*.

```
git commit -m "[message]"
```

Эта команда используется для фиксации всех поэтапных изменений. Изменяя параметр *-m* на *-am*, можно сразу добавлять и фиксировать изменения.

```
git push
```

Это команда, которая отправляет изменения в исходную ветку.

```
git status
```

Команда используется для проверки состояния измененных файлов и показывает, какие файлы помещены, не установлены и не отслежены.

```
git log
```

Команда используется для отображения истории коммита в формате по умолчанию.

```
git diff
```

Эта команда показывает все неустановленные различия между индексом и текущим каталогом. Она может использоваться с *-staged* для отображения различий между промежуточными файлами и самыми последними версиями. Другой вариант – использовать команду с именем файла для отображения различий между файлом и последним коммитом.

```
git pull
```

Эта команда используется для получения изменений из исходной ветки и объединяет их с локальной веткой.

```
git fetch
```

Эта команда извлекает самые последние изменения из ветки источника, но не объединяет.

```
git branch
```

Эта команда отображает список всех веток в репозитории. Она также может создать несуществующую ветвь, если будет добавлено имя ветки в качестве параметра.

```
git branch -d [branchname]
```

Использование флага *-d* удалит ветку с указанным именем.

```
git checkout [branchname]
```

Эта команда переключается на ветку с именем *[branchnamed]*. Если будет добавлен флаг *-b* перед именем ветки, он перейдет к новой ветке, которая будет создана автоматически.

```
git merge [branchname]
```

Команда объединяет ветку с указанным именем в текущую ветку.

```
git revert [commit]
```

Эта команда создает новый коммит, который отменяет изменения, внесенные в указанный коммит, и применяет его к текущей ветке.

```
git reset [filename]
```

Команда удаленно указывает файл из промежуточной ветки и оставляет рабочий каталог без изменений.

```
git config -global user.email [user_email]
git config -global user.name [user_name]
```

Данные команды используются для настройки текущей электронной почты и имени пользователя.

```
git config --global --edit
```

Эта команда позволяет редактировать настройки пользователя в текстовом редакторе.

Примерная последовательность команд для начала работы с Git

```
# создание репозитория в текущей директории
git init
# создание файла readme.md
touch readme.md
# добавление файл в индекс
git add readme.md
# создание коммита
git commit -m "Начало"
# добавление предварительно созданного пустого удаленного репозитория
git remote add origin https://github.com:uaer/loc.git
# отправка данные из локального репозитория в удаленный (в ветку master)
git push -u origin master
```

Контрольные вопросы

1. Для чего используются системы контроля версий?
2. Чем отличаются централизованные и децентрализованные СКВ? Приведите примеры СКВ каждого вида.
3. Поясните процесс синхронизации с общим хранилищем в децентрализованной СКВ.
4. Опишите порядок работы с общим хранилищем в централизованной СКВ.
5. Что такое и для чего могут применяться ветви?

Задание

Необходимо настроить использование системы контроля версий git на локальном компьютере. Создать дополнительную ветку *dev* на основе основной ветки *master*, в которую необходимо добавлять все новые изменения.

Создать файл с описанием загруженного проекта *readme.md*, в котором будет содержаться описание разрабатываемого проекта. Затем закоммитить изменения в readme файле в git репозиторий. Варианты индивидуальных заданий представлены в приложении А.

Содержание отчета

1. Титульный лист (Ф.И.О., группа, название лабораторной работы).
2. Цель работы.
3. Описание проделанной работы.
4. Снимок экрана с историей коммитов.
5. Выводы.

Отчет упаковать в ZIP-архив с названием по шаблону:
TRPO-Lab8-«группа, аббревиатура на латинице»-«Фамилия на латинице».
Пример: TRPO-Lab8-10-IT-Ivanov.zip

Литература

1. Chacon, Scott. Pro Git / Scott Chacon, Ben Straub. – 901 Grayson Street Suite 204 Berkely, CA, United States, 2021. – 456 p.

2. Loeliger, Jon. Version Control with Git / Jon Loeliger, Matthew McCullough. – 2nd ed. – O'Reilly Media, Inc., 2012. – 456 p.

3. Pilato, C. Michael. Version Control with Subversion / C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick. – 2nd ed. – O'Reilly Media, Inc., 2008. – 407 p.

4. Чакон, С. Git для профессионального программиста / С. Чакон, Б. Штрауб. – СПб. : Питер, 2016. – 496 с. : ил.

Приложение А

Варианты индивидуальных заданий

1. Система *Факультатив*. Существует перечень *Курсов*, за каждым из которых закреплен один *Преподаватель*. *Студент* записывается на один или несколько *Курсов*. По окончании обучения *Преподаватель* выставляет *Студенту Оценку* и добавляет отзыв.

2. Система *Приемная комиссия*. *Абитуриент* регистрируется на один из *Факультетов* с фиксированным планом набора, вводит баллы по соответствующим *Предметам* и аттестату. Результаты *Администратором* регистрируются в *Ведомости*. Система подсчитывает сумму баллов и определяет *Абитуриентов*, зачисленных в учебное заведение.

3. Система *Больница*. *Врач* определяет диагноз, делает назначение *Пациенту* (процедуры, лекарства, операции). *Назначение* может выполнить *Медсестра* (процедуры, лекарства) или *Врач* (любое назначение). *Пациент* может быть выписан из *Больницы*, при этом фиксируется окончательный диагноз.

4. Система *Турагентство*. *Заказчик* выбирает и оплачивает *Тур* (отдых, экскурсия, шоппинг). *Турагент* определяет тур как «горящий», размеры скидок постоянным клиентам.

5. Система *Телефонная станция*. *Администратор* осуществляет подключение *Абонентов*. *Абонент* может выбрать одну или несколько из предоставляемых *Услуг*. *Абонент* оплачивает *Счет* за разговоры и *Услуги*. *Администратор* может просмотреть список неоплаченных *Счетов* и заблокировать *Абонента*.

6. Система *Автобаза*. *Диспетчер* распределяет *Заявки* на *Рейсы* между *Водителями*, за каждым из которых закреплен свой *Автомобиль*. На *Рейс* может быть назначен *Автомобиль*, находящийся в исправном состоянии и характеристики которого соответствуют *Заявке*. *Водитель* делает отметку о выполнении *Рейса* и состоянии *Автомобиля*.

7. Система *Авиакомпания*. *Авиакомпания* имеет список рейсов. *Диспетчер* формирует летную *Бригаду* (пилоты, штурман, радист, стюардессы) на *Рейс*. *Администратор* управляет списком рейсов.

8. Система *Заказ гостиницы*. *Клиент* заполняет *Заявку*, указывая количество мест в номере, класс апартаментов и время пребывания. *Администратор* просматривает поступившую *Заявку*, выделяет наиболее подходящий из доступных *Номеров*, после чего система выставляет *Счет Клиенту*.

9. Система *Жилищно-коммунальные услуги*. *Квартиросъемщик* отправляет *Заявку*, в которой указывает род работ, масштаб, и желаемое время выполнения. *Диспетчер* формирует соответствующую *Бригаду* и регистрирует ее в *Плане работ*.

10. Система *Прокат автомобилей*. *Клиент* выбирает *Автомобиль* из списка доступных. Заполняет форму *Заказа*, указывая паспортные данные, срок аренды. *Клиент* оплачивает *Заказ*. *Администратор* регистрирует возврат автомобиля. В случае повреждения *Автомобиля*, *Администратор* вносит информацию и выставляет счет за ремонт. *Администратор* может отклонить *Заявку*, указав причины отказа.

11. Система *Тестирование*. *Тьютор* создает *Тест* из нескольких *Вопросов* закрытого типа (выбор одного или более вариантов из N предложенных) по определенному *Предмету*. *Студент* просматривает список доступных *Тестов*, отвечает на *Вопросы*.

12. Система *Ресторан*. *Клиент* осуществляет заказ из *Меню*. *Администратор* подтверждает *Заказ* и отправляет его на кухню для исполнения. *Администратор* выставляет *Счет*. *Клиент* производит его оплату.

13. Система *Команда разработчиков*. *Заказчик* представляет *Техническое Задание (ТЗ)*, в котором перечислен перечень *Работ* с указанием квалификации и количества требуемых специалистов. Менеджер рассматривает *ТЗ* и оформляет *Проект*, назначая на него незанятых *Разработчиков* требуемой квалификации, после чего рассчитывается стоимость *Проекта* и *Заказчику* выставляется *Счет*. Разработчик имеет возможность отметить количество часов, затраченных на работу над проектом.

14. Система *Железнодорожная касса*. *Пассажир* делает *Заявку* на билет до необходимой ему станции назначения, время и дату поездки. Система осуществляет поиск подходящего *Поезда*. *Пассажир* делает выбор *Поезда* и получает *Счет* на оплату. *Администратор* управляет списком зарегистрированных пассажиров.

15. Система *Городской транспорт*. На *Маршрут* назначаются *Автобус*, *Троллейбус* или *Трамвай*. Транспортные средства должны двигаться с определенным для каждого *Маршрута* интервалом или расписанием.

Система *Интернет-магазин*. *Администратор* осуществляет ведение каталога *Товаров*. *Клиент* делает и оплачивает *Заказ* на *Товары*. *Администратор* может занести неплательщиков в «черный список».

Приложение Б

Реализация текстового интерактивного меню

Листинг Б.1. – Класс Program

```
using System;

namespace TrpoLab
{
  class Program
  {
    static void Main(string[] args)
    {
      Statement statement = null;
      Tutor tutor = null;
      Student student = null;
      Grade grade = null;
      byte selectItem = 0;
      string tmpStr = "";
      ushort tmpValUshort = 0;
      ulong tmpValUlong = 0;
      byte tmpValByte = 0;
      int delItem = 0;
      bool correctInput = false;

      while(selectItem != 5)
      {
        Console.Clear();
        Console.WriteLine("Приложение для управления ведомостью\r\n");
        switch (selectItem)
        {
          case 0:
            Console.WriteLine("Выберите действие для совершения операции:");
            if(statement == null)
            {
              Console.WriteLine("1 - Создать ведомость");
            }
            else
            {
              Console.WriteLine("1 – Очистить ведомость");
```

```
52.             Console.WriteLine("2 - Внести оценку в
    ведомость");
53.             Console.WriteLine("3 - Удалить оценку из
    ведомости");
54.             Console.WriteLine("4 - Отобразить
    ведомость");
55.           }
56.           Console.WriteLine("5 - Выход");
57.           tmpStr = Console.ReadLine();
58.           correctInput = byte.TryParse(tmpStr, out
  selectItem);
59.           if(selectItem > 5 || !correctInput)
60.           {
61.             Console.WriteLine("Ошибка ввода!\r\n\r\n
   Нажмите \"Ввод\" для продолжения");
62.             Console.ReadLine();
63.             selectItem = 0;
64.           }
65.           break;
66.         case 1:
67.           if(statement == null)
68.           {
69.             tutor = new Tutor();
70.             Console.Write("Введите фамилию
    преподавателя: ");
71.             tmpStr = Console.ReadLine();
72.             tutor.LastName = tmpStr;
73.             Console.Write("Введите имя
  преподавателя: ");
74.             tmpStr = Console.ReadLine();
75.             tutor.FirstName = tmpStr;
76.             Console.Write("Введите отчество
    преподавателя: ");
77.             tmpStr = Console.ReadLine();
78.             tutor.MiddleName = tmpStr;
79.             Console.Write("Введите должность
  преподавателя: ");
80.             tmpStr = Console.ReadLine();
81.             tutor.Position = tmpStr;
82.             Console.Write("Введите название
    дисциплины: ");
83.             tmpStr = Console.ReadLine();
84.             statement = new Statement(tmpStr, tutor);
```

```
85.              Console.Write("Ведомость создана!\r\n\r\n
     Нажмите \"Ввод\" для продолжения");
86.            }
87.            else
88.            {
89.              statement.Clear();
90.              Console.Write("Ведомость очищена!\r\n\r\n
     Нажмите \"Ввод\" для продолжения");
91.            }
92.            Console.ReadLine();
93.            selectItem = 0;
94.            break;
95.          case 2:
96.            if(statement != null)
97.            {
98.              student = new Student();
99.              Console.Write("Введите фамилию
    студента: ");
100.             tmpStr = Console.ReadLine();
101.             student.LastName = tmpStr;
102.             Console.Write("Введите имя студента: ");
103.             tmpStr = Console.ReadLine();
104.             student.FirstName = tmpStr;
105.             Console.Write("Введите отчество
     студента: ");
106.             tmpStr = Console.ReadLine();
107.             student.MiddleName = tmpStr;
108.             do
109.             {
110.               Console.Write("Введите курс студента: ");
111.               tmpStr = Console.ReadLine();
112.               correctInput = ushort.TryParse(tmpStr,
    out tmpValUshort);
113.               if (!correctInput)
114.               {
115.                 Console.WriteLine("Ошибка ввода!\r\n
    Повторите ввод");
116.               }
117.               else
118.               {
119.                 student.Year = tmpValUshort;
120.               }
121.             }
122.             while (!correctInput);
```

```
123.            do
124.            {
125.              Console.Write("Введите номер зачетной
      книжки студента: ");
126.              tmpStr = Console.ReadLine();
127.              correctInput = ulong.TryParse(tmpStr,
                 out tmpValUlong);
128.              if (!correctInput)
129.              {
130.                Console.WriteLine("Ошибка ввода!\r\n
     Повторите ввод");
131.              }
132.              else
133.              {
134.                student.GradebookNumber = tmpValUlong;
135.              }
136.            }
137.            while (!correctInput);
138.            Console.Write("Введите группу студента: ");
139.            tmpStr = Console.ReadLine();
140.            student.Group = tmpStr;
141.            grade = new Grade
142.            {
143.              Student = student
144.            };
145.            do
146.            {
147.              Console.Write("Введите оценку: ");
148.              tmpStr = Console.ReadLine();
149.              correctInput = byte.TryParse(tmpStr,
      out tmpValByte);
150.              if (!correctInput)
151.              {
152.                Console.WriteLine("Ошибка ввода!\r\n
      Повторите ввод");
153.              }
154.              else
155.              {
156.                grade.Mark = tmpValByte;
157.              }
158.            }
159.            while (!correctInput);
160.            Console.Write("Введите дату выставления
     оценки: ");
```

```
161.              tmpStr = Console.ReadLine();
162.              grade.DateOfIssue = tmpStr;
163.              statement.Add(grade);
164.              Console.Write("Оценка внесена!\r\n\r\n
        Нажмите \"Ввод\" для продолжения");
165.              Console.ReadLine();
166.              selectItem = 0;
167.            }
168.            else
169.            {
170.              Console.WriteLine("Ошибка ввода!\r\n\r\n
        Нажмите \"Ввод\" для продолжения");
171.              Console.ReadLine();
172.              selectItem = 0;
173.            }
174.            break;
175.          case 3:
176.            if(statement != null)
177.            {
178.              if (statement.GetCount() > 0)
179.              {
180.                do
181.                {
182.                  Console.Write("Введите номер записи
        для удаления: ");
183.                  tmpStr = Console.ReadLine();
184.                  correctInput = int.TryParse(tmpStr,
        out delItem);
185.                  if (!correctInput)
186.                  {
187.                    Console.WriteLine("Ошибка ввода!\r\n
        Повторите ввод");
188.                  }
189.                  else
190.                  {
191.                    if (delItem > statement.GetCount() ||
        delItem == 0)
192.                    {
193.                      Console.WriteLine("Выбранного
        элемента не существует!\r\n\r\n
        Нажмите \"Ввод\" для
        продолжения");
194.                      Console.ReadLine();
195.                      selectItem = 0;
```

```
196.                 }
197.                 else
198.                 {
199.                   statement.Erase(delItem);
200.                   Console.WriteLine("Выбранный
       элемент удален!\r\n\r\n
       Нажмите \"Ввод\" для
       продолжения");
201.                     Console.ReadLine();
202.                     selectItem = 0;
203.                 }
204.               }
205.             }
206.             while (!correctInput);
207.           }
208.           else
209.           {
210.             Console.WriteLine("Ведомость пуста!
        \r\n\r\nНажмите \"Ввод\" для
        продолжения");
211.             Console.ReadLine();
212.             selectItem = 0;
213.           }
214.         }
215.         else
216.         {
217.           Console.WriteLine("Ошибка ввода!\r\n\r\n
       Нажмите \"Ввод\" для продолжения");
218.                 Console.ReadLine();
219.                 selectItem = 0;
220.         }
221.         break;
222.       case 4:
223.         if(statement != null)
224.         {
225.           statement.Print();
226.           Console.Write("\r\nНажмите \"Ввод\"
      для продолжения");
227.         }
228.         else
229.         {
230.           Console.WriteLine("Ошибка ввода!\r\n\r\n
        Нажмите \"Ввод\" для продолжения");
231.         }
```

```
232.             Console.ReadLine();
233.             selectItem = 0;
234.             break;
235.             default:
236.             selectItem = 0;
237.             break;
238.         }
239.       }
240.     }
241.   }
242. }
```