Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

Многопрофильный колледж

УТВЕРЖДАЮ
Директор
_____ / С.А. Махновский
«09» февраля 2022 г.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

по учебной дисциплине
ОП.14 Инструментальные средства разработки компьютерных систем и комплексов

для обучающихся специальности

09.02.01 Компьютерные системы и комплексы
(базовой подготовки)

Магнитогорск, 2022

ОДОБРЕНО

Предметно-цикловой комиссией
««Информатика и вычислительная техника»
Председатель И.Г. Зорина
Протокол № 5 от 19.01.2022

Методической комиссией МпК
Протокол № 4 от 09.02.2022

Разработчики:
преподаватель ФГБОУ ВО «МГТУ им. Г.И. Носова» Многопрофильный колледж
Ю.А. Мазнина
преподаватель ФГБОУ ВО «МГТУ им. Г.И. Носова» Многопрофильный колледж
С.М. Сайдильдина

Методические указания по выполнению практических работ разработаны на основе рабочей программы учебной дисциплины «Инструментальные средства разработки компьютерных систем и комплексов».

Содержание практических работ ориентировано на подготовку обучающихся к освоению профессионального(ых) модуля(ей) программы подготовки специалистов среднего звена по специальности 09.02.01 Компьютерные системы и комплексы и овладению профессиональными компетенциями.

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ 4
2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ 7
Практическая работа № 1 7
Практическая работа № 2 14
Практическая работа № 3 19
Практическая работа № 4 33
Практическая работа № 5 34
Практическая работа № 6 35
Практическая работа № 7 36
Практическая работа № 8 52
Практическая работа № 9 70
Практическая работа № 10 83
Практическая работа № 11 97
Практическая работа № 12 100

1 ВВЕДЕНИЕ

Важную часть теоретической и профессиональной практической подготовки обучающихся составляют практические занятия.

Состав и содержание практических занятий направлены на реализацию Федерального государственного образовательного стандарта среднего профессионального образования.

Ведущей дидактической целью практических занятий является формирование профессиональных практических умений (умений выполнять определенные действия, операции, необходимые в последующем в профессиональной деятельности, необходимых в последующей учебной деятельности.

В соответствии с рабочей программой учебной дисциплины «Инструментальные средства разработки компьютерных систем и комплексов» предусмотрено проведение практических занятий.

В результате их выполнения, обучающийся должен:

уметь:

- У1 проводить контроль, диагностику и восстановление работоспособности компьютерных систем и комплексов;
- У2 принимать участие в отладке и технических испытаниях компьютерных систем и комплексов;
- У3 принимать участие в инсталляции, конфигурировании и настройке операционной системы, драйверов, резидентных программ
- У4 проводить анализ предметной области; осуществлять выбор модели и средства построения информационных систем и программных средств;
- У5 проектировать и разрабатывать информационные системы и программные средства по заданным требованиям и спецификациям;
- У6 проектировать логическую и физическую схемы базы данных, используя современные case-средства;
- У7 разрабатывать графический интерфейс приложения;
- У8 осуществлять разработку кода информационных систем и программных средств на языках высокого уровня;
- У9 выполнять отладку и тестирование информационных систем и программных средств;
- У10 оформлять документацию на информационные системы и программные средства;
- У01.4. составлять резюме;
- У01.5. собирать портфолио работ и достижений
- У02.1 распознавать и анализировать профессиональную задачу и/или проблему;
- У02.2 определять этапы решения профессиональной задачи, составлять и реализовывать план действия по достижению результата;
- У02.3 использовать цифровые средства и ресурсы для генерирования новых идей и решений;
- У02.5 оценивать результаты решения задач профессиональной деятельности;
- У02.6 использовать цифровые средства и приложения для создания продукта;-
- У03.1. принимать решения в стандартной профессиональной ситуации и определять необходимые ресурсы;
- У03.2. принимать решения в нестандартной профессиональной ситуации и определять необходимые ресурсы;
- У03.3. оценивать результат и последствия своих действий (самостоятельно или с помощью наставника);
- У03.4 строить логические умозаключения на основании информации/данных, в том числе в различных цифровых средах (в том числе, оценивать результат и последствия своих действий);
- У03.5 самостоятельно определять пробелы в своих знаниях и компетенциях с использованием инструментов самооценки и цифровых оценочных средств
- У04.1. определять необходимые источники информации;

– У04.2. искать информацию в сети Интернет с использованием фильтров и ключевых слов;

– У04.3. выделять наиболее значимое в изучаемом материале и структурировать получаемую информацию;

– У04.4. выбирать оптимальный формат, способ и место хранения информации и данных с помощью цифровых инструментов;

– У04.7. оформлять результаты поиска информации

– У05.1. использовать средства информационно-коммуникационных технологий для решения профессиональных задач;

– У05.2. использовать специализированное программное обеспечение;

– У05.3. проявлять культуру информационной безопасности;

– У06.1. работать в коллективе и команде;

– У06.2. выбирать цифровые средства общения в соответствии с целью взаимодействия и индивидуальными особенностями (в том числе культурными) собеседника;

– У06.4. использовать цифровые средства общения при взаимодействии с другими людьми, в том числе для организации совместной деятельности

– У.07.1. распределять обязанности в команде;

– У07.2. выбирать оптимальные способы, приемы и методы решения профессиональных задач коллективом исполнителей;

– У07.4. анализировать достигнутые результаты работы команды

– У08.3.находить информацию в целях самообразования и обучения при помощи цифровых инструментов;

– У08.5. выбирать цифровые средства в целях саморазвития

– У09.1. находить и анализировать информацию в области инноваций в профессиональной деятельности;

– У09.3. владеть актуальными методами работы в профессиональной и смежных сферах.

Содержание практических и лабораторных занятий ориентировано на подготовку обучающихся к освоению профессионального модуля программы подготовки специалистов среднего звена по специальности и овладению ***профессиональными компетенциями:***

ПК 3.3 – Производить тестирование, определение параметров и отладку микропроцессорных систем;

ПК 3.4 – Разрабатывать, внедрять и адаптировать прикладное программное обеспечение.

А также формированию ***общих компетенций***:

ОК 01 – Понимать сущность и социальную значимость своей будущей профессии, проявлять к ней устойчивый интерес;

ОК 02 – Организовывать собственную деятельность, выбирать типовые методы и способы выполнения профессиональных задач, оценивать их эффективность и качество;

ОК 03 – Принимать решения в стандартных и нестандартных ситуациях и нести за них ответственность;

ОК 04 – Осуществлять поиск и использование информации, необходимой для эффективного выполнения профессиональных задач, профессионального и личностного развития;

ОК 05 – Использовать информационно-коммуникационные технологии в профессиональной деятельности;

ОК 06 – Работать в коллективе и команде, эффективно общаться с коллегами, руководством, потребителями;

ОК 07 – Брать на себя ответственность за работу членов команды (подчиненных), результат выполнения заданий;

ОК 08 – Самостоятельно определять задачи профессионального и личностного развития, заниматься самообразованием, осознанно планировать повышение квалификации;

ОК 09 – Ориентироваться в условиях частой смены технологий в профессиональной деятельности.

Выполнение обучающихся практических и/или лабораторных работ по учебной дисциплине «Инструментальные средства разработки компьютерных систем и комплексов» направлено на: *(выбрать):*

- *обобщение, систематизацию, углубление, закрепление, развитие и детализацию полученных теоретических знаний по конкретным темам учебной дисциплины;*
- *формирование умений применять полученные знания на практике, реализацию единства интеллектуальной и практической деятельности;*
- *развитие интеллектуальных умений у будущих специалистов: аналитических, проектировочных, конструктивных и др.;*
- *выработку при решении поставленных задач профессионально значимых качеств, таких как самостоятельность, ответственность, точность, творческая инициатива.*

Практические занятия проводятся после соответствующей темы, которая обеспечивает наличие знаний, необходимых для ее выполнения.

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Тема 1.1 Инструментальные средства этапа проектирования компьютерных систем и комплексов

Практическая работа № 1
Анализ предметной области

Цель: Проанализировать и описать заданную предметную область, сформировать требования к информационной системе, распределить роли в группе разработчиков.

Выполнив задания, Вы будете:

уметь:

– У3. анализировать проектную и техническую документацию;

– У4. использовать специализированные графические средства построения и анализа архитектуры программных продуктов.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:

Лабораторное занятие направлено на ознакомление с процессом описания информационной системы и получение навыков по использованию основных методов анализа ИС.

Рекомендуемые предметные области для разработки информационной системы:

1. Разработка программного комплекса «Управление кредитами в банке».
2. Разработка программного комплекса «Отделение колледжа».
3. Разработка программного комплекса «Обслуживание банкомата».
4. Разработка программного комплекса «Управление гостиницей».
5. Разработка программного комплекса «Магазин по продаже автозапчастей».
6. Разработка программного комплекса «Строительная фирма».
7. Разработка программного комплекса «Управление библиотечным фондом».
8. Разработка программного комплекса «АРМ работника склада»
9. Разработка программного комплекса «АРМ администратора ателье по ремонту оргтехники»
10. Разработка программного комплекса «АРМ администратора автосалона».
11. Разработка программного комплекса «АРМ администратора ресторана».
12. Разработка программного комплекса «АРМ администратора фитнес-клуба».
13. Разработка программного комплекса «АРМ администратора аэропорта».
14. Разработка программного комплекса «АРМ работника отдела кадров».
15. Разработка программного комплекса «АРМ администратора спорткомплекса».

Вы также можете предложить преподавателю свой вариант предметной области.

Требования к результатам выполнения лабораторной работы:

1. наличие описания информационной системы;
2. проведение анализа осуществимости выполнения проекта;
3. наличие заключения о возможности реализации проекта.

В первую очередь необходимо определить источники информации. Это могут быть менеджеры отделов, где система будет использоваться, разработчики программного обеспечения, знакомые с типом будущей системы, технологи, конечные пользователи и т.д.

В ходе анализа осуществимости создания информационной системы целесообразно ответить на вопросы:

- Что произойдет с организацией, если система не будет введена в эксплуатацию?
- Какие текущие проблемы существуют в организации и как новая система поможет их решить?
- Каким образом система будет способствовать целям бизнеса?

- Требует ли разработка системы технологии, которая до этого не использовалась в организации?

Результатом анализа осуществимости создания информационной системы является заключение о возможности реализации проекта. В нем даются рекомендации относительно начала или продолжения разработки системы. Могут быть предложены ориентировочные суммы бюджета или его изменения, графика работ по созданию системы или предъявлены более высокие требования к системе.

Документ, описывающий предметную область, должен содержать рекомендации относительно разработки системы, базовые предложения по объёму требуемого бюджета и времени, графику работ, числу разработчиков, требуемому программному обеспечению, и может включать следующие разделы:

- Введение
- Организация выполнения проекта
- Анализ рисков и др.

После обработки собранной информации готовится отчет по анализу осуществимости создания системы.

Порядок выполнения работы:

1. Изучить теоретический материал по теме.
2. Составить подробное описание информационной системы.
3. На основании описания системы провести анализ осуществимости создания информационной системы..
4. Распределить роли в группе (руководитель проекта, системный аналитик, разработчик, разработчик, тестер-разработчик).
5. Составить документ, описывающий предметную область.
6. Составить отчет о проделанной работе.

Ход работы:

Краткие теоретические сведения
Общие сведения о разработке программного обеспечения

Проблемы управления программными проектами впервые проявились в 60-х - начале 70-х годов, когда провалились многие большие проекты по разработке программных продуктов. Были зафиксированы задержки в создании ПО, оно было ненадежным, затраты на разработку в несколько раз превосходили первоначальные оценки, созданные программные системы часто имели низкие показатели производительности. Причины провалов коренились в тех подходах, которые использовались в управлении проектами. Применяемая методика была основана на опыте управления техническими проектами и оказалась неэффективной при разработке программного обеспечения.

Важно понимать разницу между профессиональной разработкой ПО и любительским программированием. Необходимость управления программными проектами вытекает из того факта, что процесс создания профессионального ПО всегда является субъектом бюджетной политики организации, где оно разрабатывается, и имеет временные ограничения. Работа руководителя программного проекта по большому счету заключается в том, чтобы гарантировать выполнение этих бюджетных и временных ограничений с учетом бизнес-целей организации относительно разрабатываемого ПО.

Руководители проектов призваны спланировать все этапы разработки программного продукта. Они также должны контролировать ход выполнения работ и соблюдения всех требуемых стандартов. Постоянный контроль за ходом выполнения работ необходим для того, чтобы процесс разработки не выходил за временные и бюджетные ограничения. Хорошее управление не гарантирует успешного завершения проекта, но плохое управление обязательно приведет к его провалу. Это может выразиться в задержке сроков сдачи готового ПО, в превышении сметной стоимости проекта и в несоответствии готового ПО спецификации требований.

Процесс разработки ПО существенно отличается от процессов реализации технических проектов, что порождает определенные сложности в управлении программными проектами:

1. *Программный продукт нематериален.* Программное обеспечение нематериально, его нельзя увидеть или потрогать. Руководитель программного проекта не видит процесс "роста" разрабатываемого ПО. Он может полагаться только на документацию, которая фиксирует процесс разработки программного продукта.

2. *Не существует стандартных процессов разработки ПО.* На сегодняшний день не существует четкой зависимости между процессом создания ПО и типом создаваемого программного продукта. Другие технические дисциплины имеют длительную историю, процессы разработки технических изделий многократно опробованы и проверены. Процессы создания большинства технических систем хорошо изучены. Изучением же процессов создания ПО специалисты занимаются только последнее время. Поэтому пока нельзя точно предсказать, на каком этапе процесса разработки ПО могут возникнуть проблемы, угрожающие всему программному проекту.

3. *Большие программные проекты - это часто "одноразовые" проекты.* Большие программные проекты, как правило, значительно отличаются от проектов, реализованных ранее. Поэтому, чтобы уменьшить неопределенность в планировании проекта, руководители проектов должны обладать очень большим практическим опытом. Но постоянные технологические изменения в компьютерной технике и коммуникационном оборудовании обесценивают предыдущий опыт. Знания и навыки, накопленные опытом, могут не востребоваться в новом проекте.

Перечисленные отличия могут привести к тому, что реализация проекта выйдет из временного графика или превысит бюджетные ассигнования. Программные системы зачастую оказываются новинками как в "идеологическом", так и в техническом плане. Поэтому, предвидя возможные проблемы в реализации программного проекта, следует всегда помнить, что многим из них свойственно выходить за рамки временных и бюджетных ограничений.

Процесс управления разработкой программного обеспечения

Невозможно описать и стандартизировать все работы, выполняемые в проекте по созданию ПО. Эти работы весьма существенно зависят от организации, где выполняется разработка ПО, и от типа создаваемого программного продукта. Но всегда можно выделить следующие:

- Написание предложений по созданию ПО.
- Планирование и составление графика работ по созданию ПО.
- Оценивание стоимости проекта.
- Подбор персонала.
- Контроль за ходом выполнения работ.
- Написание отчетов и представлений.

Первая стадия программного проекта может состоять из написания предложений по реализации этого проекта. Предложения должны содержать описание целей проектов и способов их достижения. Они также обычно включают в себя оценки финансовых и временных затрат на выполнение проекта. При необходимости здесь могут приводиться обоснования для передачи проекта на выполнение сторонней организации или команде разработчиков.

Написание предложений — очень ответственная работа, так как для многих организаций вопрос о том, будет ли проект выполняться самой организацией или разрабатываться по контракту сторонней компанией, является критическим. Не существует каких-либо рекомендаций по написанию предложений, многое здесь зависит от опыт.

На этапе планирования проекта определяются процессы, этапы и полученные на каждом из них результаты, которые должны привести к выполнению проекта. Реализация этого плана приведет к достижению целей проекта. Определение стоимости проекта напрямую связано с его планированием, поскольку здесь оцениваются ресурсы, требующиеся для выполнения плана.

Контроль за ходом выполнения работ (мониторинг проекта) — это непрерывный процесс, продолжающийся в течение всего срока реализации проекта. Руководитель должен постоянно отслеживать ход реализации проекта и сравнивать фактические и плановые показатели выполнения работ с их стоимостью. Хотя многие организации имеют механизмы формального

мониторинга работ, опытный руководитель может составить ясную картину о стадии развитии проекта просто путем неформального общения с разработчиками.

Неформальный мониторинг часто помогает обнаружить потенциальные проблемы, которые в явном виде могут обнаружиться позднее. Например, ежедневное обсуждение хода выполнения работ может выявить отдельные недоработки в создаваемом программном продукте. Вместо ожидания отчетов, в которых будет отражен факт "пробуксовки" графика работ, можно обсудить со специалистами намечающиеся программистские проблемы и не допустить срыва графика работ.

В течение реализации проекта обычно происходит несколько формальных контрольных проверок хода выполнения работ по созданию ПО. Такие проверки должны дать общую картину хода реализации проекта в целом и показать, насколько уже разработанная часть ПО соответствует целям проекта.

Время выполнения больших программных проектов может занимать несколько лет. В течение этого времени цели и намерения организации, заказавшей программный проект, могут существенно измениться. Может оказаться, что разрабатываемый программный продукт стал уже ненужным либо исходные требования к создаваемому ПО просто устарели и их необходимо кардинально менять. В такой ситуации руководство организации-разработчика может принять решение о прекращении разработки ПО или об изменении проекта в целом с тем, чтобы учесть изменившиеся цели и намерения организации-заказчика.

Руководители проектов обычно обязаны сами подбирать исполнителей для своих проектов. В идеальном случае профессиональный уровень исполнителей должен соответствовать той работе, которую они будут выполнять в ходе реализации проекта. Однако во многих случаях руководители должны полагаться на команду разработчиков, которая далека от идеальной. Такая ситуация может быть вызвана следующими причинами:

1. Бюджет проекта не позволяет привлечь высококвалифицированный персонал. В таком случае за меньшую плату привлекаются менее квалифицированные специалисты.
2. Бывают ситуации, когда невозможно найти специалистов необходимой квалификации как в самой организации-разработчике, так и вне ее. Например, в организации "лучшие люди" могут быть уже заняты в других проектах.
3. Организация хочет повысить профессиональный уровень своих работников. В этом случае она может привлечь к участию в проекте неопытных или недостаточно квалифицированных работников, чтобы они приобрели необходимый опыт и поучились у более опытных специалистов.

Таким образом, почти всегда подбор специалистов для выполнения проекта имеет определенные ограничения и не является свободным. Вместе с тем необходимо, чтобы хотя бы несколько членов группы разработчиков имели квалификацию и опыт, достаточные для работы над данным проектом. В противном случае невозможно избежать ошибок в разработке ПО.

Руководитель проекта обычно обязан посылать отчеты о ходе его выполнения как заказчику, так и подрядным организациям. Это должны быть краткие документы, основанные на информации, извлекаемой из подробных' отчетов о проекте. В этих отчетах должна быть та информация, которая позволяет четко оценить степень готовности создаваемого программного продукта.

В рамках курса «Технология разработки программного обеспечения» выделены следующие роли в группе по разработке ПО:

- Руководитель – общее руководство проектом, написание документации, общение с заказчиком ПО
- Системный аналитик – разработка требований (составление технического задания, проекта программного обеспечения)
- Тестер – составление плана тестирования и аттестации готового ПО (продукта), составление сценария тестирования, базовый пример, проведение мероприятий по плану тестирования
- Разработчик – моделирование компонент программного обеспечения, кодирование

Планирование проекта разработки программного обеспечения

Эффективное управление программным проектом напрямую зависит от правильного планирования работ, необходимых для его выполнения. План помогает руководителю предвидеть проблемы, которые могут возникнуть на каких-либо этапах создания ПО, и разработать превентивные меры для их предупреждения или решения. План, разработанный на начальном этапе проекта, рассматривается всеми его участниками как руководящий документ, выполнение которого должно привести к успешному завершению проекта. Этот первоначальный план должен максимально подробно описывать все этапы реализации проекта.

Процесс планирования начинается, исходя из описания системы, с определения проектных ограничений (временные ограничения, возможности наличного персонала, бюджетные ограничения и т.д.). Эти ограничения должны определяться параллельно с оцениванием проектных параметров, таких как структура и размер проекта, а также распределением функций среди исполнителей. Затем определяются этапы разработки и то, какие результаты документация, прототипы, подсистемы или версии программного продукта) должны быть получены по окончании этих этапов. Далее начинается циклическая часть планирования. Сначала разрабатывается график работ по выполнению проекта или дается разрешение на продолжение использования ранее созданного графика. После этого проводится контроль выполнения работ и отмечаются расхождения между реальным и плановым ходом работ.

Далее, по мере поступления новой информации о ходе выполнения проекта, возможен пересмотр первоначальных оценок параметров проекта. Это, в свою очередь, может привести к изменению графика работ. Если в результате этих изменений нарушаются сроки завершения проекта, должны быть пересмотрены (и согласованы с заказчиком ПО) проектные ограничения.

Конечно, большинство руководителей проектов не думают, что реализация их проектов пройдет гладко, без всяких проблем. Желательно описать возможные проблемы еще до того, как они проявят себя в ходе выполнения проекта. Поэтому лучше составлять "пессимистические" графики работ, чем "оптимистические". Но, конечно, невозможно построить план, учитывающий все, в том числе случайные, проблемы и задержки выполнения проекта, поэтому и возникает необходимость периодического пересмотра проектных ограничений и этапов создания программного продукта.

План проекта должен четко показать ресурсы, необходимые для реализации проекта, разделение работ на этапы и временной график выполнения этих этапов. В некоторых организациях план проекта составляется как единый документ, содержащий все виды планов, описанных выше. В других случаях план проекта описывает только технологический процесс создания ПО. В таком плане обязательно присутствуют ссылки на планы других видов, но они разрабатываются отдельно от плана проекта.

Детализация планов проектов очень разнится в зависимости от типа разрабатываемого программного продукта и организации-разработчика. Но в любом случае большинство планов содержат следующие разделы.

1. *Введение*. Краткое описание целей проекта и проектных ограничений (бюджетных, временных и т.д.), которые важны для управления проектом.
2. *Организация выполнения проекта*. Описание способа подбора команды разработчиков и распределение обязанностей между членами команды.
3. *Анализ рисков*. Описание возможных проектных рисков, вероятности их проявления и стратегий, направленных на их уменьшение.
4. *Аппаратные и программные ресурсы, необходимые для реализации проекта*. Перечень аппаратных средств и программного обеспечения, необходимого для разработки программного продукта. Если аппаратные средства требуется закупать, приводится их стоимость совместно с графиком закупки и поставки.
5. *Разбиение работ на этапы*. Процесс реализации проекта разбивается на отдельные процессы, определяются этапы выполнения проекта, приводится описание результатов ("выходов") каждого этапа и контрольные отметки.
6. *График работ*. В этом графике отображаются зависимости между отдельными процессами (этапами) разработки ПО, оценки времени их выполнения и распределение членов команды разработчиков по отдельным этапам.

7. *Механизмы мониторинга и контроля за ходом выполнения проекта*. Описываются предоставляемые руководителем отчеты о ходе выполнения работ, сроки их предоставления, а также механизмы мониторинга всего проекта.

План должен регулярно пересматриваться в процессе реализации проекта. Одни части плана, например график работ, изменяются часто, другие более стабильны. Для внесения изменений в план требуется специальная организация документопотока, позволяющая отслеживать эти изменения.

Общие сведения о требованиях к информационным системам

Проблемы, которые приходится решать специалистам в процессе создания программного обеспечения, очень сложны. Природа этих проблем не всегда ясна, особенно если разрабатываемая программная система инновационная. В частности, трудно чётко описать те действия, которые должна выполнять система. Описание функциональных возможностей и ограничений, накладываемых на систему, называется требованиями к этой системе, а сам процесс формирования, анализа, документирования и проверки этих функциональных возможностей и ограничений – разработкой требований.

Требования подразделяются на пользовательские и системные. Пользовательские требования – это описание на естественном языке (плюс поясняющие диаграммы) функций, выполняемых системой, и ограничений, накладываемых на неё. Системные требования – это описание особенностей системы (архитектура системы, требования к параметрам оборудования и т.д.), необходимых для эффективной реализации требований пользователя.

Первые шаги по разработке требований к информационным системам – анализ осуществимости

Разработка требований — это процесс, включающий мероприятия, необходимые для создания и утверждения документа, содержащего спецификацию системных требований. Для новых программных систем процесс разработки требований должен начинаться с анализа осуществимости. Началом такого анализа является общее описание системы и ее назначения, а результатом анализа — отчет, в котором должна быть четкая рекомендация, продолжать или нет процесс разработки требований проектируемой системы. Другими словами, анализ осуществимости должен осветить следующие вопросы.

1. Отвечает ли система общим и бизнес-целям организации-заказчика и организации-разработчика?
2. Можно ли реализовать систему, используя существующие на данный момент технологии и не выходя за пределы заданной стоимости?
3. Можно ли объединить систему с другими системами, которые уже эксплуатируются?

Критическим является вопрос, будет ли система соответствовать целям организации. Если система не соответствует этим целям, она не представляет никакой ценности для организации. В то же время многие организации разрабатывают системы, не соответствующие их целям, либо не совсем ясно понимая эти цели, либо под влиянием политических или общественных факторов.

Форма представления результата:

1. Цель работы
2. Введение. Краткое описание целей проекта и проектных ограничений (бюджетных, временных и т.д.), которые важны для управления проектом
3. Описание информационной системы (ПО) - наличие заключения о возможности реализации проекта, содержащего рекомендации относительно разработки системы, базовые предложения по объёму требуемого бюджета, числу разработчиков, времени и требуемому программному обеспечению
4. Анализ осуществимости (согласно требованиям к результатам выполнения), указать возможные проблемы и пути их решения.
5. Роли участников группы разработки ПО.
6. Программно-аппаратные средства, используемые при выполнении работы.

7. Заключение (выводы)
8. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент учел и рассмотрел особенности предметной области и грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент учел и рассмотрел особенности предметной области и грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент не полностью учел и рассмотрел особенности предметной области и представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, студент не учел особенности предметной области, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 1.1 Инструментальные средства этапа проектирования компьютерных систем и комплексов

Практическая работа № 2
Моделирование программного обеспечения средствами UML: диаграмма вариантов использования, диаграмма последовательности, диаграмма коммуникации, диаграмма состояний, диаграмма классов

Цель: Изучить основные принципы моделирования программного обеспечения посредством диаграмм UML: диаграммы вариантов использования, диаграммы последовательности, диаграммы коммуникации, диаграммы состояний, диаграммы классов. Составить пакет диаграмм для заданной предметной области.

Выполнив работу, Вы будете:
уметь:
- У3. анализировать проектную и техническую документацию;
- У4. использовать специализированные графические средства построения и анализа архитектуры программных продуктов.

Материальное обеспечение:
Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:
Выполните построение диаграмм UML для заданной преподавателем предметной области.
Требования к построению моделей:
- модель должна отражать весь указанный в описании предметной области функционал.

Порядок выполнения работы:
1. Изучить предлагаемый теоретический материал по теме.
2. Разработать диаграмму вариантов использования и диаграмму состояний и последовательности с использованием языка UML для заданной предметной области в соответствии с требованиями.
7. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения:
Диаграммы вариантов использования

Понятие варианта использования (use case) впервые ввел Ивар Якобсон и придал ему такую значимость, что в настоящее время вариант использования превратился в основной элемент разработки и планирования проекта.

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать. На языке UML вариант использования изображают следующим образом:

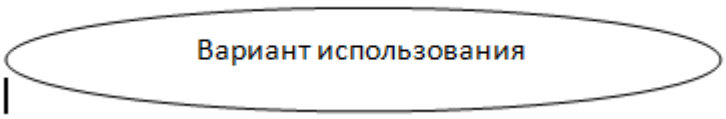

Рисунок 1 – Вариант использования

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой,

которой необходима некоторая информация от данной системы. Показывать на диаграмме действующих лиц следует только в том случае, когда им действительно необходимы некоторые варианты использования. На языке UML действующие лица представляют в виде фигур:

Рисунок 2 – Действующее лицо (актер)

Действующие лица делятся на три основных типа:

- пользователи;
- системы;
- другие системы, взаимодействующие с данной;
- время.

Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе.

Связи между вариантами использования и действующими лицами

В языке UML на диаграммах вариантов использования поддерживается несколько типов связей между элементами диаграммы. Это связи коммуникации (communication), включения (include), расширения (extend) и обобщения (generalization).

Связь коммуникации – это связь между вариантом использования и действующим лицом. На языке UML связи коммуникации показывают с помощью однонаправленной ассоциации (сплошной линии).

Рисунок 3– Пример связи коммуникации

Связь включения применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования. С помощью таких связей обычно моделируют многократно используемую функциональность.

Связь расширения применяется при описании изменений в нормальном поведении системы. Она позволяет варианту использования только при необходимости использовать функциональные возможности другого.

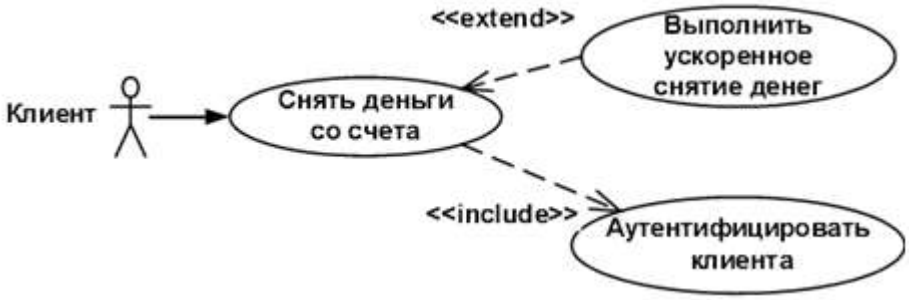

Рисунок 4 – Пример связи включения и расширения

С помощью связи обобщения показывают, что у нескольких действующих лиц имеются общие черты.

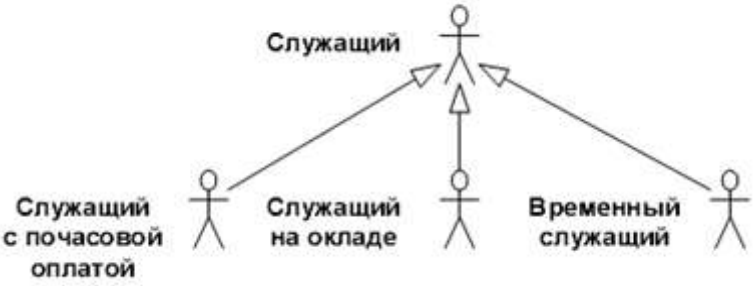

Рисунок 5 – Пример связи обобщения

Диаграммы состояний

Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий.

Существует много форм диаграмм состояний, незначительно отличающихся друг от друга семантикой.

На диаграмме имеются два специальных состояния – начальное (start) и конечное (stop). Начальное состояние выделено черной точкой, оно соответствует состоянию объекта, когда он только что был создан. Конечное состояние обозначается черной точкой в белом кружке, оно соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний может быть одно и только одно начальное состояние. В то же время, может быть столько конечных состояний, сколько вам нужно, или их может не быть вообще. Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы. Процессы, происходящие, когда объект находится в определенном состоянии, называются действиями (actions).

С состоянием можно связывать данные пяти типов.

1. Деятельность

Деятельностью (activity) называется поведение, реализуемое объектом, пока он находится в данном состоянии. Деятельность – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово do (делать) и двоеточие.

2. Входное действие

Входным действием (entry action) называется поведение, которое выполняется, когда объект переходит в данное состояние. Данное действие осуществляется не после того, как объект перешел в это состояние, а, скорее, как часть этого перехода. В отличие от деятельности, входное действие рассматривается как непрерываемое. Входное действие также показывают внутри состояния, ему предшествует слово entry (вход) и двоеточие.

3. Выходное действие

Выходное действие (exit action) подобно входному. Однако, оно осуществляется как составная часть процесса выхода из данного состояния. Оно является частью процесса такого перехода. Как и входное, выходное действие является непрерываемым.

Выходное действие изображают внутри состояния, ему предшествует слово exit (выход) и двоеточие.

Поведение объекта во время деятельности, при входных и выходных действиях может включать отправку события другому объекту. В этом случае описанию деятельности, входного действия или выходного действия предшествует знак « ^ ».

Соответствующая строка на диаграмме выглядит как

Do: ^Цель.Событие (Аргументы)

Здесь Цель – это объект, получающий событие, Событие – это посылаемое сообщение, а Аргументы являются параметрами посылаемого сообщения.

Деятельность может также выполняться в результате получения объектом некоторого события. При получении некоторого события выполняется определенная деятельность.

Переходом (Transition) называется перемещение из одного состояния в другое. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями.

На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся последующим.

Переходы могут быть рефлексивными. Объект может перейти в то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии.

У перехода существует несколько спецификаций. Они включают события, аргументы, ограждающие условия, действия и посылаемые события.

4. События

Событие (event) – это то, что вызывает переход из одного состояния в другое. Событие размещают на диаграмме вдоль линии перехода.

На диаграмме для отображения события можно использовать как имя операции, так и обычную фразу.

Большинство переходов должны иметь события, так как именно они, прежде всего, заставляют переход осуществиться. Тем не менее, бывают и автоматические переходы, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со скоростью, позволяющей осуществиться входным действиям, деятельности и выходным действиям.

Ограждающие условия

Ограждающие условия (guard conditions) определяют, когда переход может, а когда не может осуществиться. В противном случае переход не осуществится.

Ограждающие условия изображают на диаграмме вдоль линии перехода после имени события, заключая их в квадратные скобки.

Ограждающие условия задавать необязательно. Однако если существует несколько автоматических переходов из состояния, необходимо определить для них взаимно исключающие ограждающие условия. Это поможет читателю диаграммы понять, какой путь перехода будет автоматически выбран.

Действие

Действием (action), как уже говорилось, является непрерываемое поведение, осуществляющееся как часть перехода. Входные и выходные действия показывают внутри состояний, поскольку они определяют, что происходит, когда объект входит или выходит из него. Большую часть действий, однако, изображают вдоль линии перехода, так как они не должны осуществляться при входе или выходе из состояния.

Действие рисуют вдоль линии перехода после имени события, ему предшествует косая черта.

Событие или действие могут быть поведением внутри объекта, а могут представлять собой сообщение, посылаемое другому объекту. Если событие или действие посылается другому объекту, перед ним на диаграмме помещают знак « ^ ».

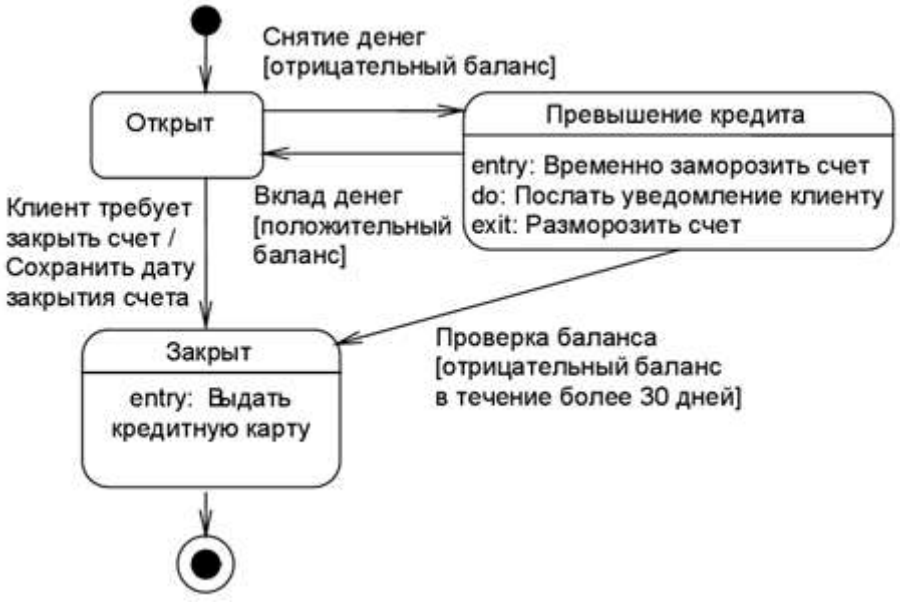

Рисунок 1 – Пример диаграммы состояний

Диаграммы состояний не надо создавать для каждого класса, они применяются только в сложных случаях. Если объект класса может существовать в нескольких состояниях и в каждом из них ведет себя по-разному, для него может потребоваться такая диаграмма.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.1 Инструментальные средства этапа разработки баз данных компьютерных систем и комплексов

Практическая работа № 3
Проектирование реляционной базы данных

Цель: Спроектировать логическую и физическую схемы базы данных, используя современные case-средства.

Выполнив работу, Вы будете:

уметь:

- У6. проектировать логическую и физическую схемы базы данных, используя современные case-средства.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, среда программирования Visual Studio 2019, система управления базами данных Microsoft SQL Server (Express).

Задание:

1. Изучите рекомендуемую литературу.
2. Выделите основные характеристики проекта и представить их в виде ментальной карты.
3. Проведите сравнение проектной и операционной видов деятельности.
4. Выделите функции менеджера проектов. Определить компетенции (soft и hard skills), которыми должен обладать менеджер проектов. Представить функции, компетенции и их взаимосвязь графически.

Ход работы:

1. Знакомство с основными положениями EDM-модели

Вам предстоит разработать WCF-приложение, или по-другому, сервис-ориентированное приложение (WCF – это Windows Communication Foundatione). Взаимодействие с источником данных WCF-приложения базируется на модели «сущность-связь» – ***Entity Data Model (EDM)***. Модель *EDM* представляет набор основных понятий, которые описывают структуру данных независимо от формы хранения. Описываемые в модели *EDM* данные могут иметь различную структуру: реляционную, текстовые файлы, файлы XML, электронные таблицы и отчеты. Модель *EDM* описывает структуры данных на основе сущностей и связей, которые являются независимыми от схем хранения. В результате такого подхода форма хранения данных отделена от приложения и не влияет на его разработку. Это обеспечивается тем, что сущности и связи описывают структуру данных так, как она используется в приложении. Модель *EDM* является концептуальной моделью, которая описывает представление структуры данных в виде **сущностей** и **связей.**

Модель ***EDM*** использует три основных понятия для описания структуры данных:

- тип сущности;
- тип ассоциации;
- свойство.

Тип сущности используется для описания структуры данных при помощи модели *EDM*. В концептуальной модели типы сущностей конструируются из свойств и описывают структуру основных концептуальных элементов верхнего уровня, таких как сотрудники и роли. Тип сущности является шаблоном для сущностей. Сущность представляет определенный объект (например, определенного сотрудника или его роль в бизнес-процессе). Каждая сущность должна иметь уникальный ключ внутри набора сущностей. Набор сущностей представляет собой коллекцию экземпляров определенного типа сущности. Наборы сущностей (и наборы ассоциаций) логически сгруппированы в контейнеры сущностей.

Тип ассоциации применяется для построения описаний связей в модели *EDM*. В концептуальной модели ассоциация представляет собой связь между двумя типами сущностей. Каждая ассоциация имеет две конечные точки ассоциации, которые определяют типы сущностей, участвующие в ассоциации. Каждая конечная точка ассоциации также определяет кратность конечной точки ассоциации, которая указывает число сущностей, могущих присутствовать на одной конечной точке ассоциации.

Типы сущностей содержат свойства, которые определяют их структуру и характеристики. Например, тип сущности «**Сотрудник**» может иметь свойства, такие как идентифицирующий номер, фамилию, имя, отчество, должность, дату рождения и адрес электронной почты.

Свойства в концептуальной модели аналогичны свойствам, которые определены применительно к классу в приложении или атрибуту реляционной базы данных. Свойство может содержать примитивные данные (такие как строка, целое число, дата, логическое значение) или структурированные данные (такие как сложный тип).

Концептуальная модель является специфическим представлением структуры некоторых данных в виде сущностей и связей. Одним из способов представления концептуальной модели является схема. На Рисунок 23 приведена схема концептуальной модели базы данных ***TitlePersonal*** – "***Персонал***" с двумя типами сущностей (***Employee*** – Сотрудник и ***Title*** – роль/должность) и одной ассоциативной связью 1:* (один ко многим).

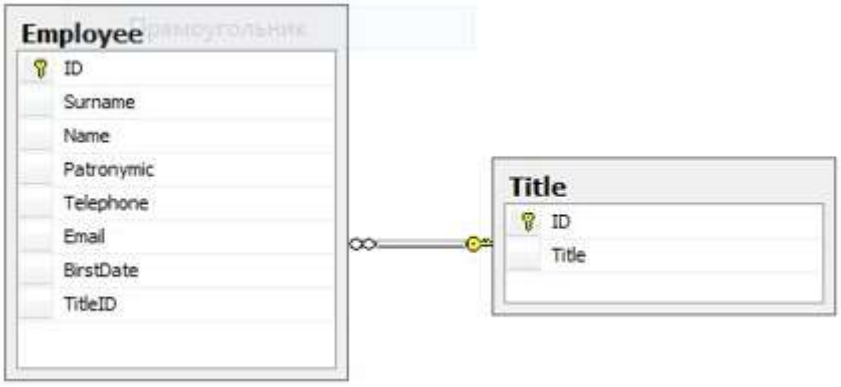

Рисунок 1 – Концептуальная модель базы данных

Таблица ***Employee*** содержит данные по сотруднику. Атрибутами таблицы являются:

- ***ID*** – код сотрудника;
- ***Surname*** – фамилия;
- ***Name*** – имя;
- ***Patronymic*** – отчество;
- ***BirstDate*** – дата рождения;
- ***Telephone*** – телефон;
- ***Email*** – адрес электронной почты;
- ***TitleID*** – внешний ключ для связи с таблицей *Title*.

Таблица ***Title*** является справочником должностей, имеющихся на предприятии, и включает следующие атрибуты:

- ***ID*** – код должности;
- ***Title*** – наименование должности.

2 Создание нового подключения и базы данных

Прежде чем создавать модель на основе базы данных, создадим саму учебную базу данных для проекта и реализуем подключение к этой базе данных.

Откройте панель **Обозреватель серверов (Server Explorer)** (рисунок 1) (если отсутствует на экране слева, можно открыть с помощью меню **View / Server Explorer**)

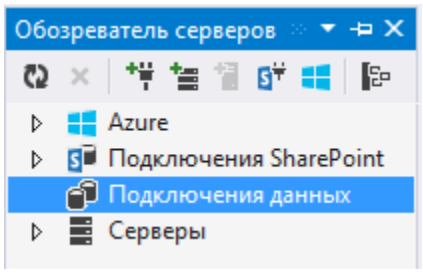

Рисунок 1 – Обозреватель серверов (Server Explorer)

Мы видим возможные варианты подключения, которые зависят от конфигурации, установленной на ПК.

Создадим новое подключение к базе данных (рисунок 2).

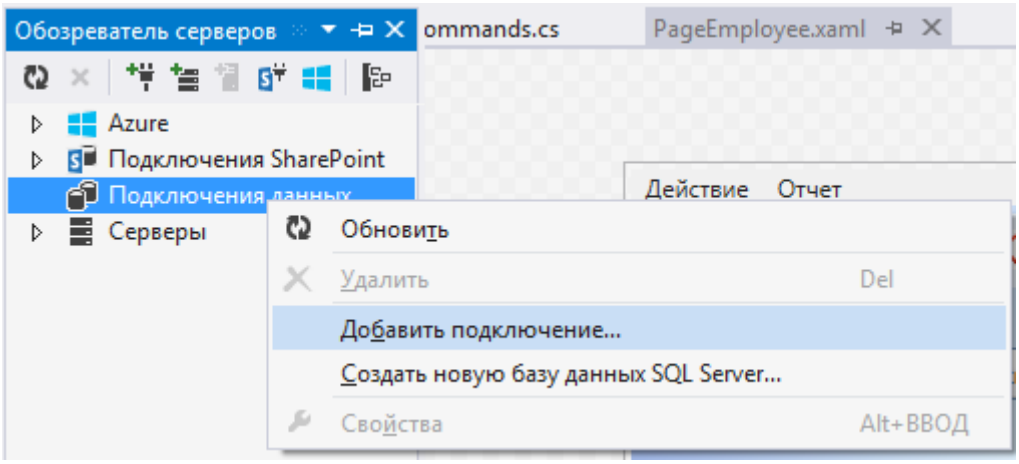

Рисунок 2 – Создание нового подключения к базе данных

В качестве источника данных в этом проекте выберем **Файл базы данных Microsoft SQL Server** (Рисунок 3), укажите имя файла базы данных *TitleEmployee* (рисунок 4), согласитесь с созданием нового файла базы данных типа Microsoft SQL Server (рисунок 5).

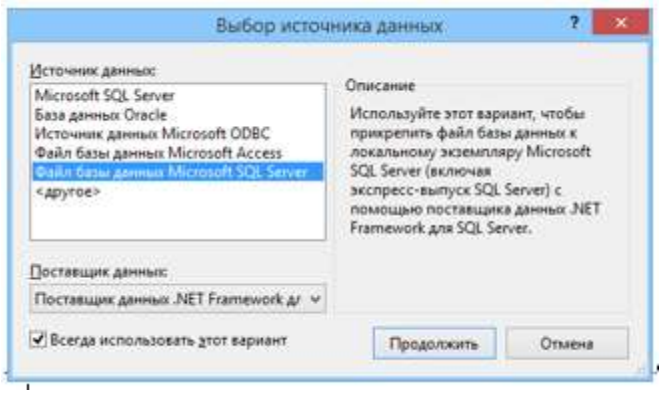

Рисунок 3 – Выбор источника данных

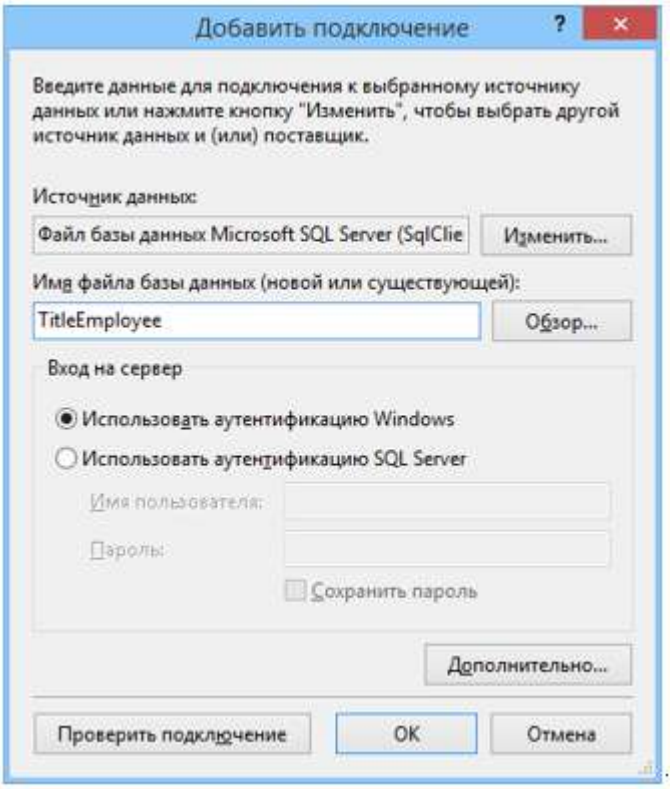

Рисунок 4 – Создание файла базы данных для подключения

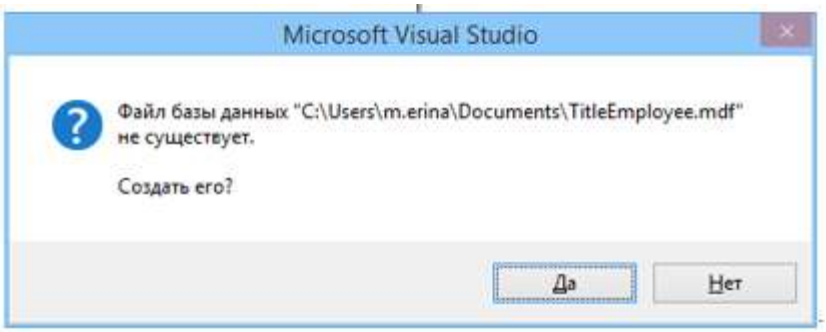

Рисунок 5 – Предупреждение о создании нового файла базы данных

В **Обозревателе серверов (Server Explorer)** появилось подключение данных с базой данных ***TitleEmployeer.mdf*** (рисунок 6). Однако пока созданная база данных пустая, не содержит ни одного объекта (нет таблиц, триггеров, запросов и т.д.).

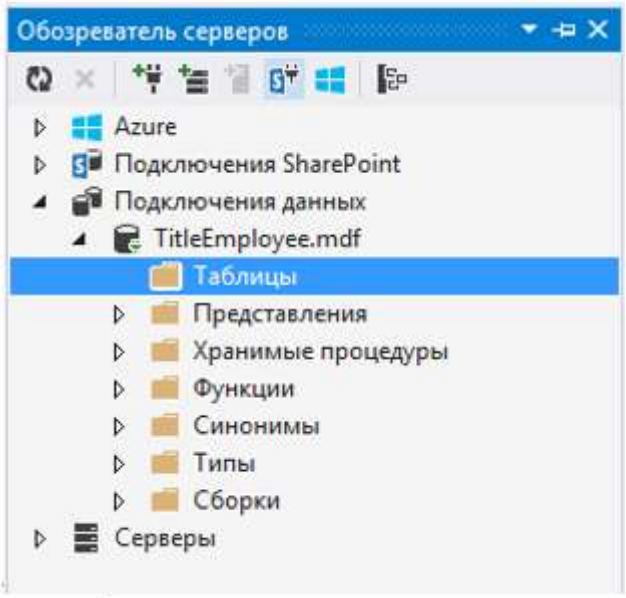

Рисунок 6 – Созданное подключение данных

Создадим структуру базы данных, содержащую две таблицы *Employee* и *Title*. Структура таблиц должна соответствовать концептуальной модели базы данных, показанной на рисунке 23. В **Обозревателе серверов (Server Explorer)** выберите пункт **Добавить новую таблицу (Add New Table)** (рисунок 7).

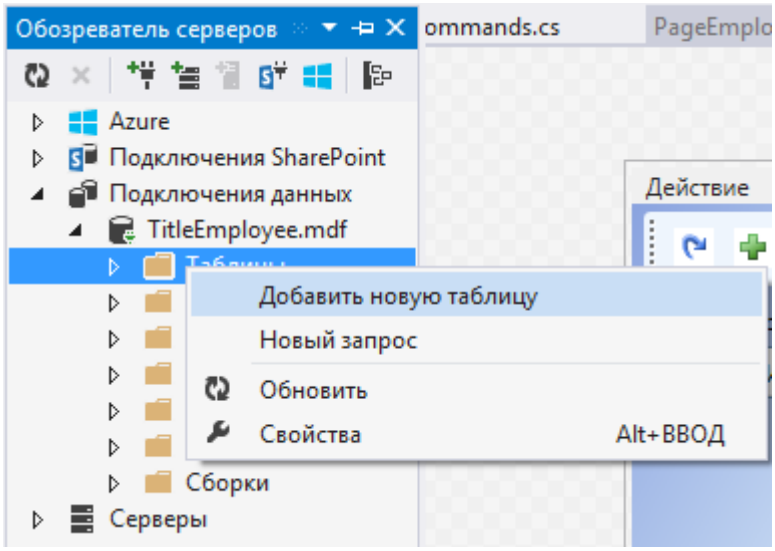

Рисунок 7 –Добавление новой таблицы

В открывшемся конструкторе таблиц базы данных создайте таблицу ***Employee***, которая содержит данные по сотруднику (рисунок 8). Атрибутами таблицы являются:

- ***ID*** – код сотрудника, ***ключевое поле***, тип ***int***, запрет нулевых значений ***Допустимы значения NULL (Allow Nulls) = False (не отмечено)***, автоинкрементное ***Identity Specification = True***;
- ***Surname*** – фамилия, тип ***nchar(100)***, запрет нулевых значений *Allow Nulls = False*;
- ***Name*** – имя, тип ***nchar(100)***;
- ***Patronymic*** – отчество, тип ***nchar(100)***;
- ***BirstDate*** – дата рождения, тип ***date***;
- ***Telephone*** – телефон, тип ***nchar(10)***;
- ***Email*** – адрес электронной почты, тип ***nchar(30)***;
- ***TitleID*** – внешний ключ для связи с таблицей ***Title***, тип ***int***, запрет нулевых значений ***Allow Nulls = False***.

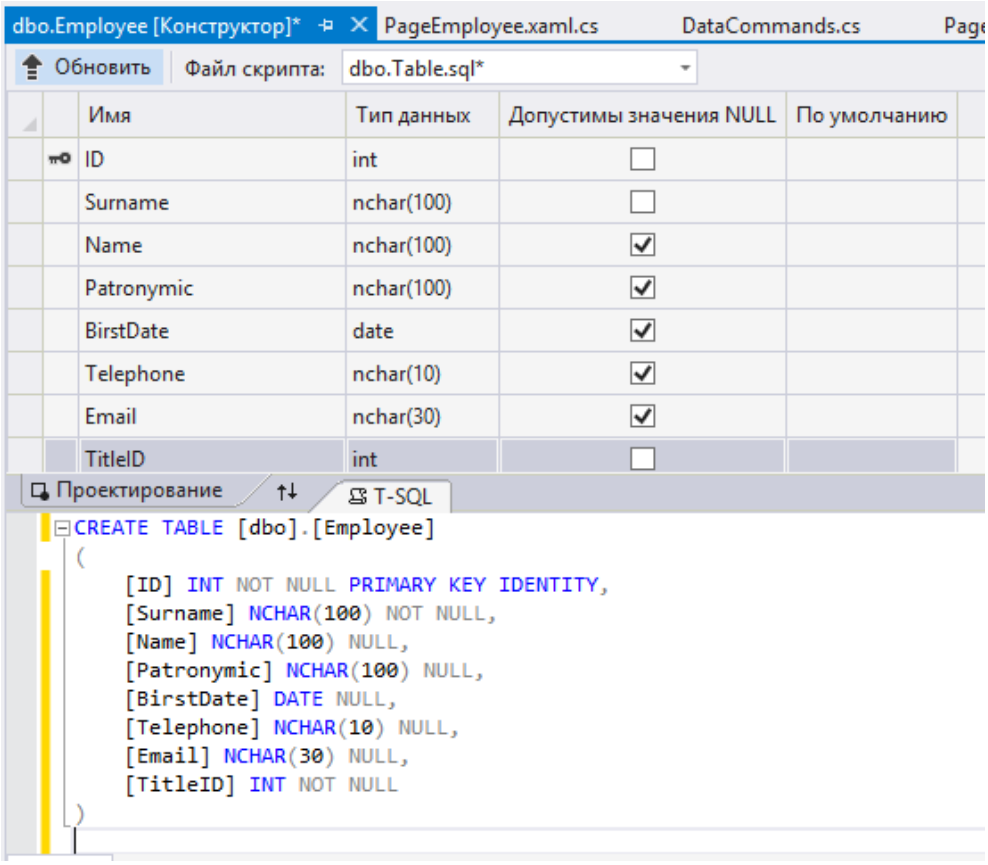

Рисунок 9 – Конструктор таблицы *Employee*

Для того, чтобы внесенные изменения попали в базу данных необходимо воспользоваться кнопкой **Обновить** (**Update**).

Аналогичным образом создайте таблицу ***Title*** (рисунок 10), которая является справочником должностей, имеющихся на предприятии, и включает следующие атрибуты:

– ***ID*** – код должности, ключевое поле, тип *int*, запрет нулевых значений ***Allow Nulls = False***, автоинкрементное ***Identity Specification = True***;

– ***Title*** – наименование должности, тип ***nchar(100)***, запрет нулевых значений ***Allow Nulls = False***.

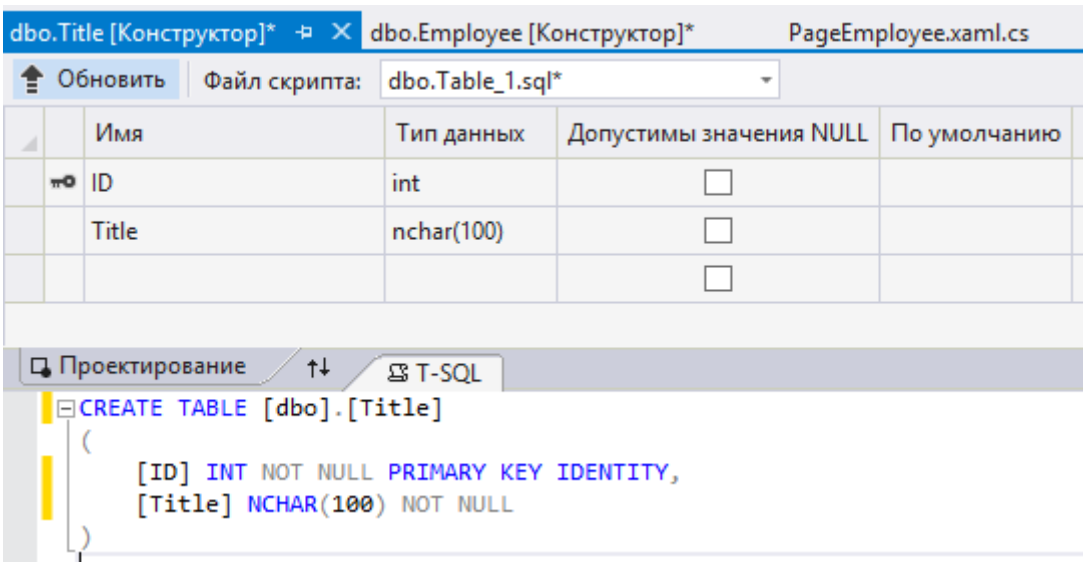

Рисунок 10 – Конструктор таблицы *Title*

Не забудьте обновить таблицу для записи внесенных изменений в физическую структуру базы данных.

В соответствии с концептуальной моделью базы данных (рисунок 1), между таблицами имеется отношение «Один-ко-многим». Создадим это отношение. Снова откройте конструктор таблицы *Employee*. Добавьте новый внешний ключ (рисунок 12).

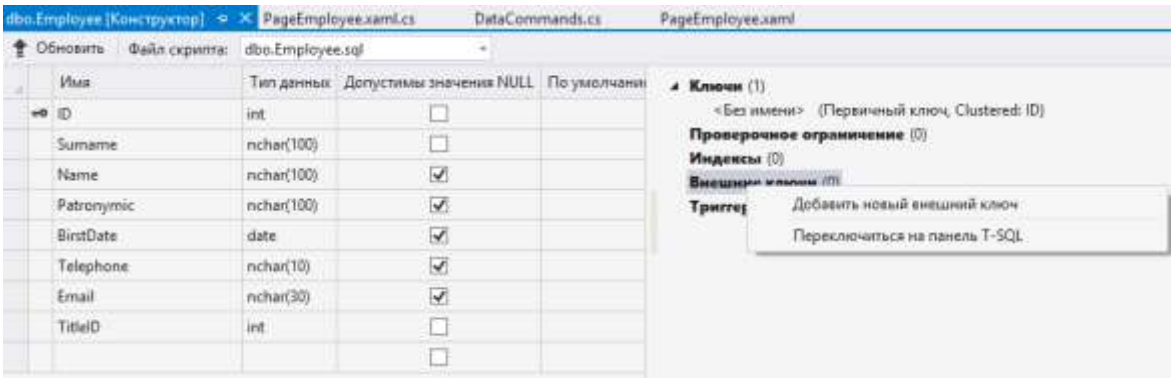

Рисунок 12 – Добавление нового внешнего ключа

Назовите внешний ключ ***FK_Employee_ToTitle***. При этом в описание структуры добавится шаблон для создания отношения. Внесите в шаблон исправления, которые создадут необходимое отношение (рисунок 13). После внесения изменений необходимо обновить таблицу.

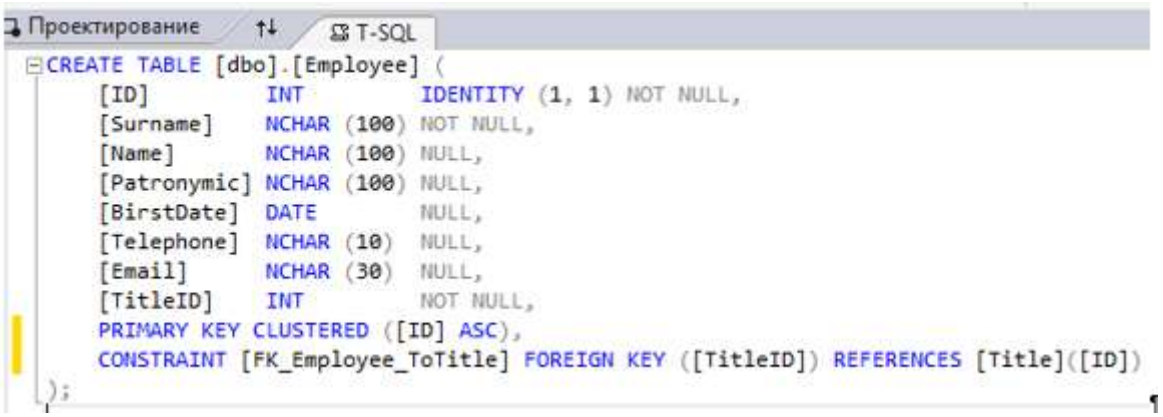

Рисунок 13 – Код для генерации внешнего ключа

Вернувшись в **Обозреватель серверов (Server Explorer)** и обновив созданное подключение, можно увидеть базу данных ***TitleEmployee.mdf*** (рисунок 14).

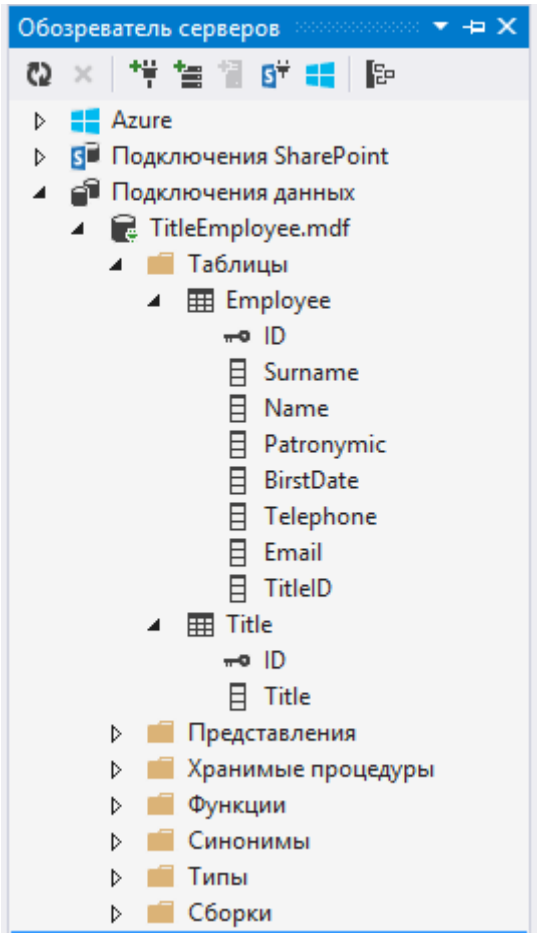

Рисунок 15. Созданное подключение к базе данных

3. Создание EDM-модели

На всякий случай сделайте копию своего проекта!!! Проект на данный момент не должен содержать ошибок!!!

Для создания EDM-модели данных добавим в проект новый элемент – ***Модель ADO.NET EDM*** (Entity Data Model), присвоив файлу модели имя ***TitleEmployee.edmx*** (Рисунок 36).

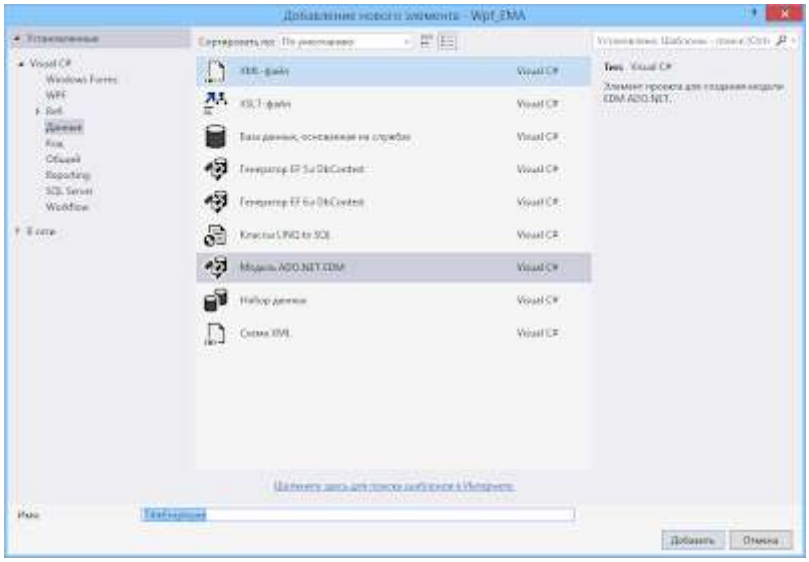
Рисунок 36. Добавление в проект модели EDM

В мастере создания EDM-модели выберем опцию "**Конструктор EF из базы данных**" (**Generate from database**) (Рисунок 37). Для создания соединения с базой данных в окне "Выбор подключения к данным" укажем соединение с базой данных (в рассматриваемом примере – созданная ранее база данных ***TitleEmployee.mdf***) и зададим имя модели данных - ***TitleEmployeeEntities*** (точнее, оставим название по умолчанию) (Рисунок 38).

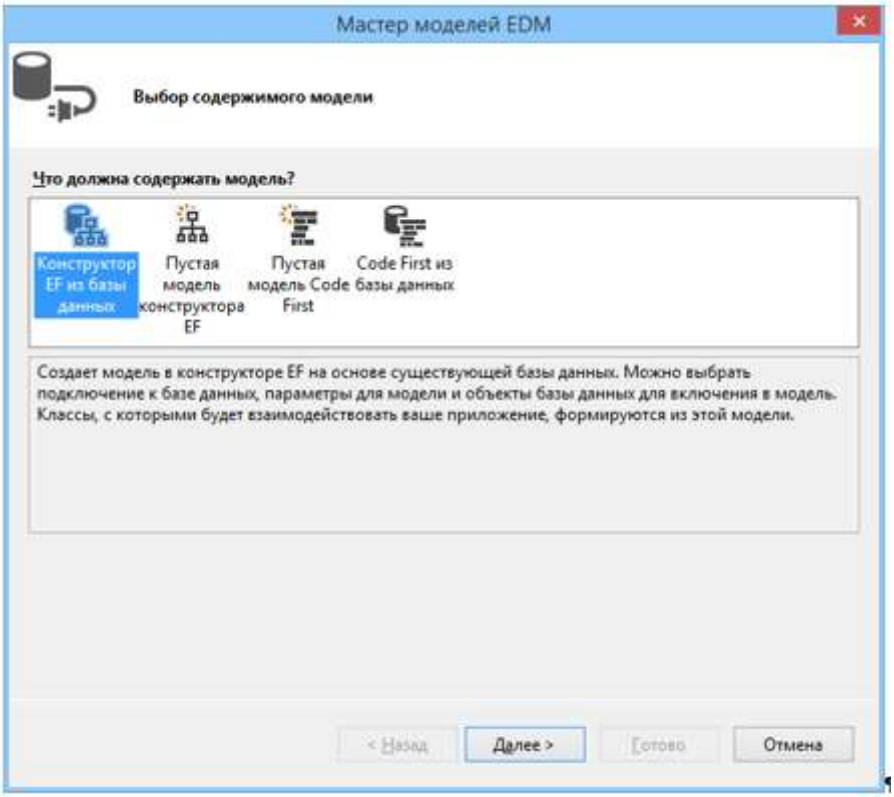

Рисунок 37. Создание модели EDM из базы данных

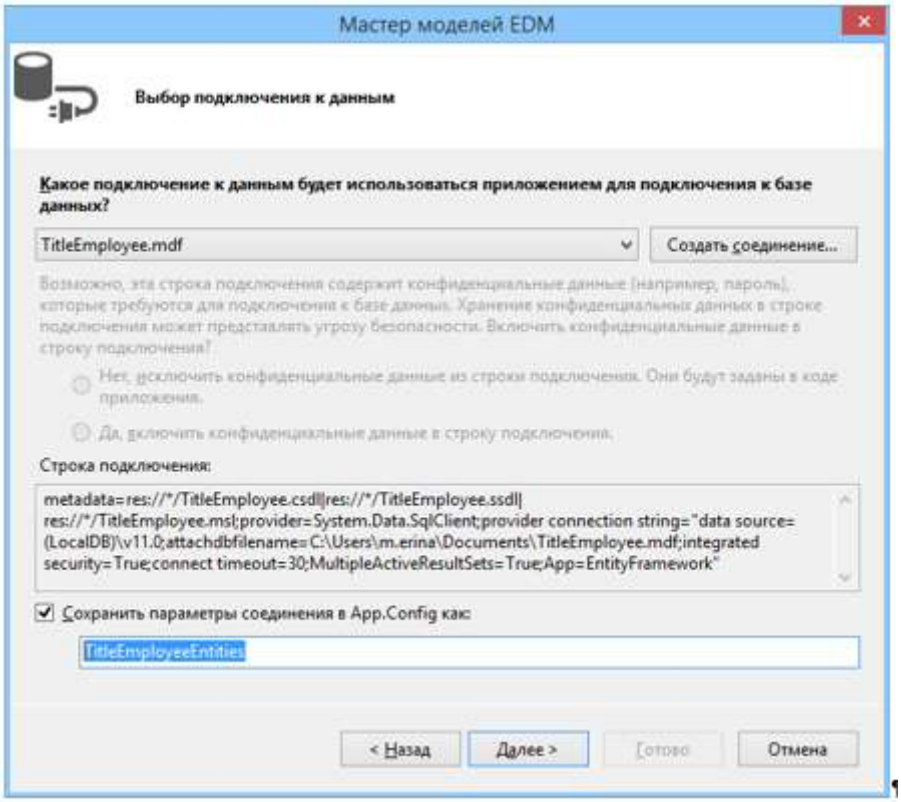

Рисунок 38. Создание соединения с базой данных

Согласитесь с копированием локальной базы данных в выходной каталог приложения. Это означает, что при каждой компиляции проекта в выходной каталог будет копироваться база данных. Все изменения, вносимые в базу данных во время работы с приложением, будут совершаться с этой копией базы данных. Таким образом, ***все данные, которые будут вноситься в базу, при каждой компиляции проекта будут затираться снова пустой копией базы***. Это следует учитывать. При необходимости сохранить данные после работы приложения, можно скопировать базу данных с данными из выходного каталога в каталог проекта.

На следующем шаге работы мастера выберите версию **Entity FrameWork 6.x**

В окне "**Выбор объектов базы данных**" отметим необходимые для приложения таблицы (в нашем случае это таблицы ***Employee*** и ***Title***) и флаг формирования имен объектов во множественном или единственном числе (Рисунок 39).

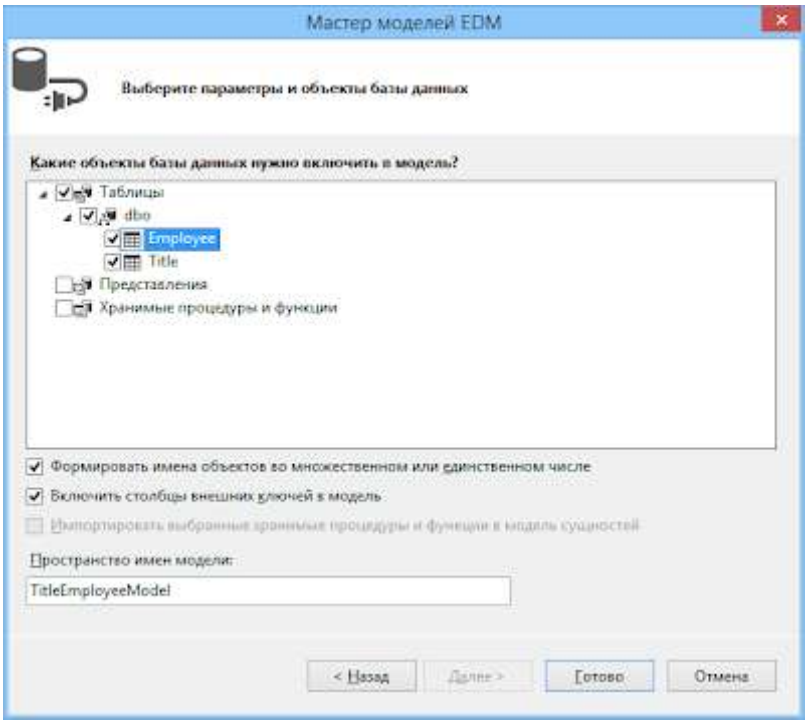

Рисунок 39. Выбор таблиц базы данных

При завершении работы мастера создания EDM-модели в проект будет добавлен файл ***TitleEmployee.edmx*** (Рисунок 40), ссылки на необходимые библиотеки и конфигурационный файл.

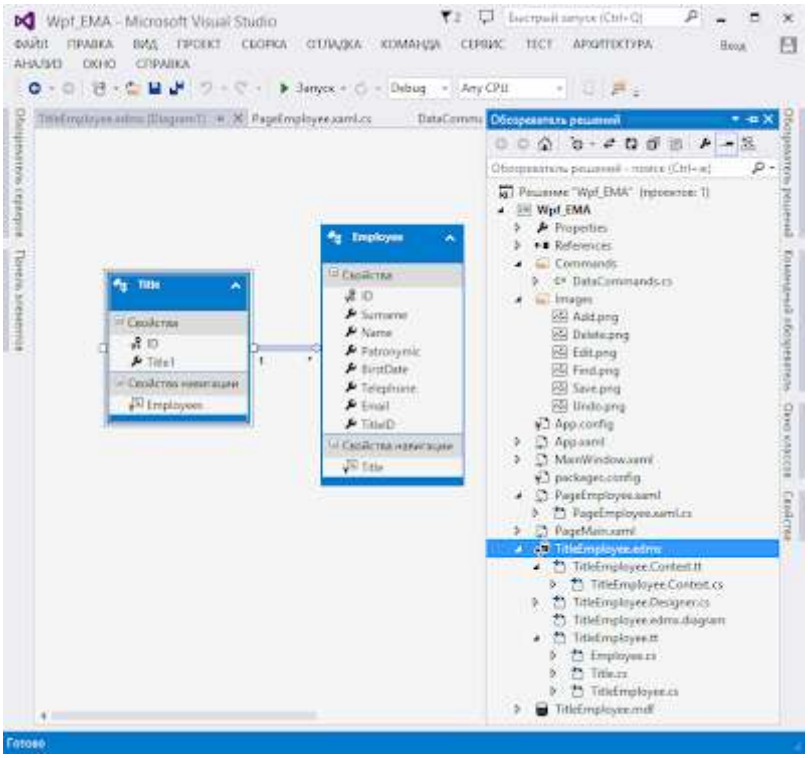

Рисунок 40. Проект с моделью EDM

Автоматически сгенерированный класс ***TitleEmployeeEntities***, который наследуется от класса ***DbContext***, представляет сущности базы данных ***TitlePerson***, содержит свойства, моделирующие таблицы ***Employee*** и ***Title***, связи между таблицами.

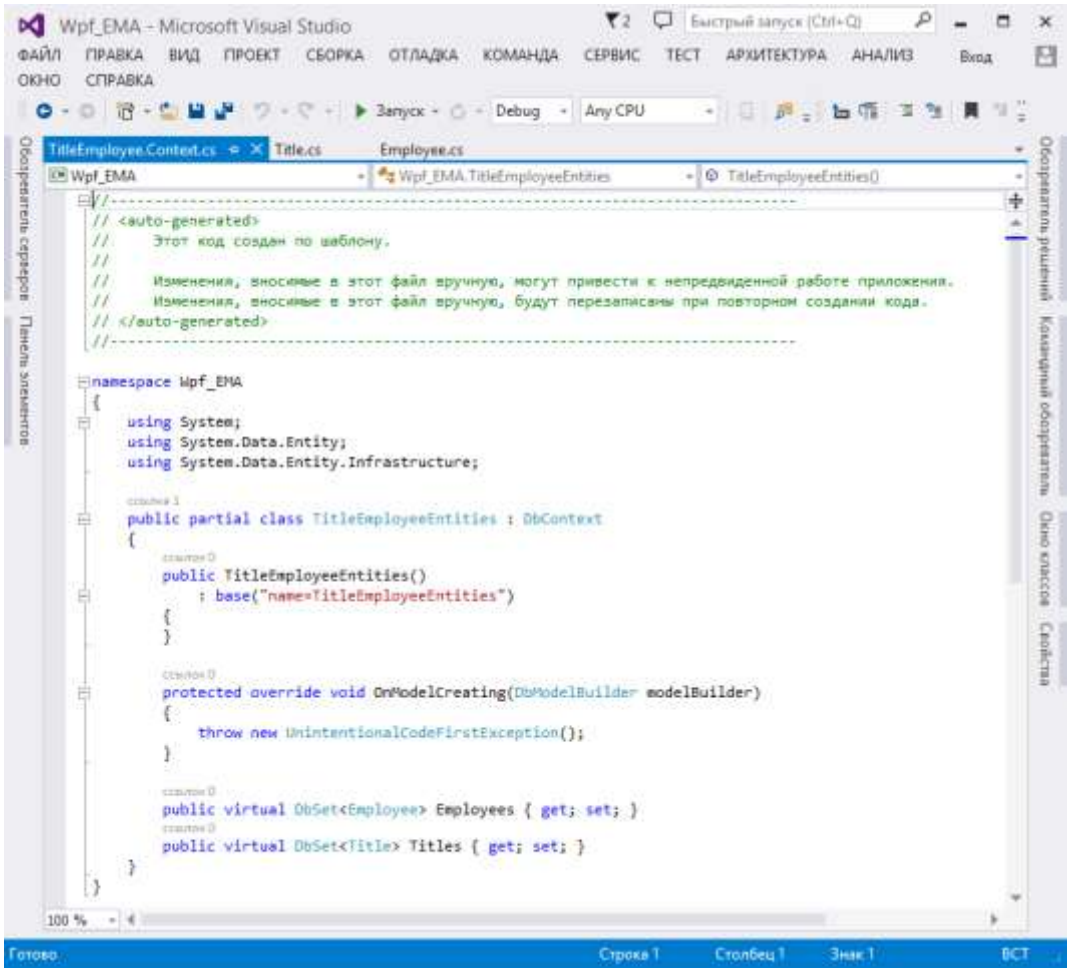

Рисунок 41. Класс ***TitleEmployeeEntities***

Также были автоматически сгенерированы два класса, отражающие сущности ***Employee*** и ***Title***.

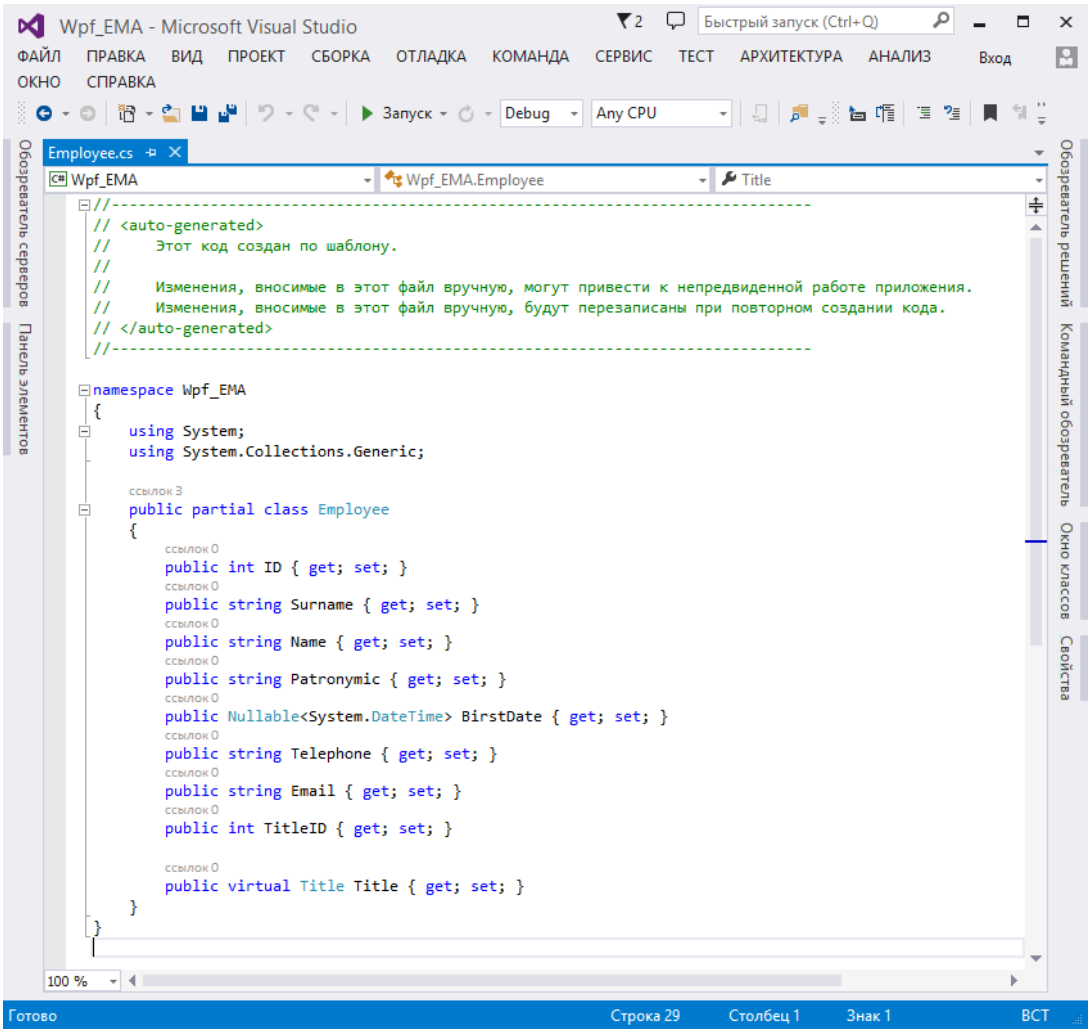

Рисунок 42. Класс, отражающий сущность *Employee*

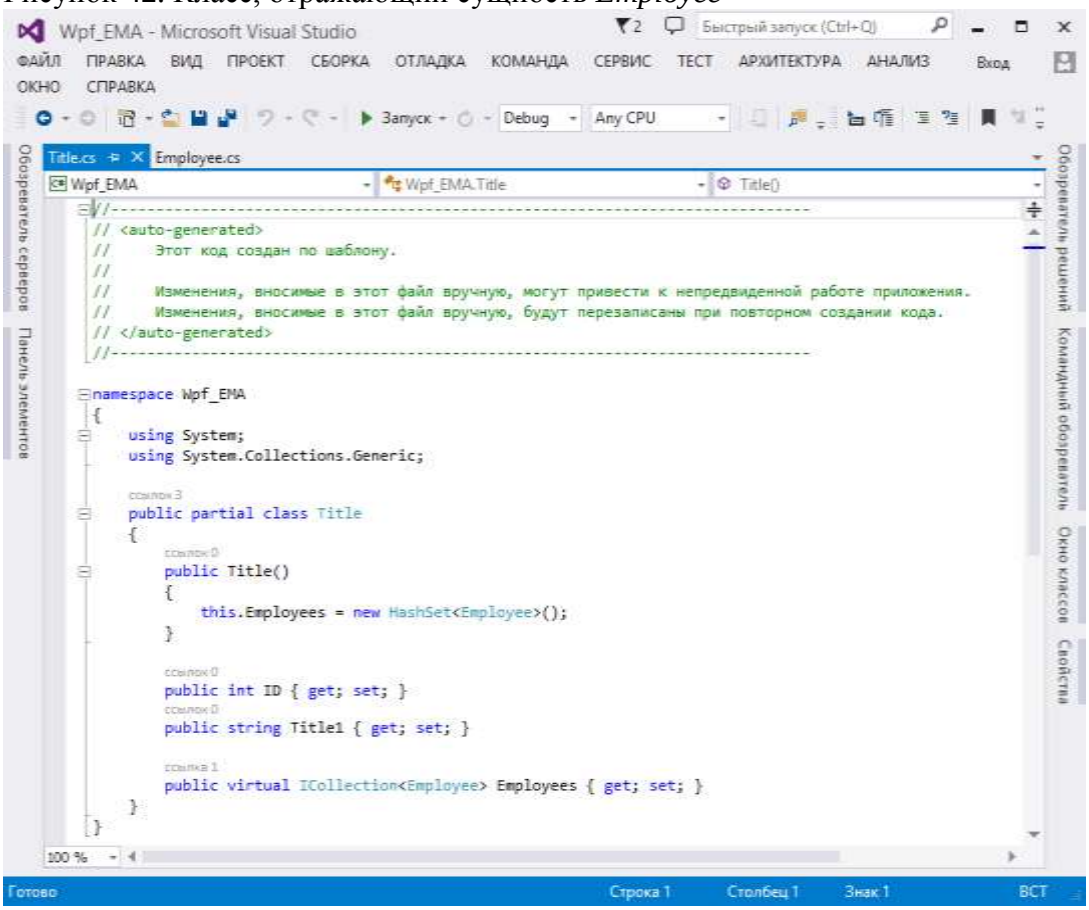

Рисунок 43. Класс, отражающий сущность *Title*

При создании модели данных в проекте автоматически был сгенерирован конфигурационный файл App.Config, который содержит строку соединения с базой данных (выделена желтым цветом)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->

<section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <connectionStrings>

<add name="TitleEmployeeEntities" connectionString="metadata=res://*/TitleEmployee.csdl|res://*/TitleEmployee.ssdl|res://*/TitleEmployee.msl;provider=System.Data.SqlClient;provider connection string=&amp;quot;data source=(LocalDB)\v11.0;attachdbfilename=|DataDirectory|\TitleEmployee.mdf;integrated security=True;connect timeout=30;MultipleActiveResultSets=True;App=EntityFramework&amp;quot;" providerName="System.Data.EntityClient" />
  </connectionStrings>
  <entityFramework>

<defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <providers>

<provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
</configuration>
```

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.1 Инструментальные средства этапа разработки баз данных компьютерных систем и комплексов

Практическая работа № 4
Нормализация реляционной базы данных

Цель: Освоить навыки построения нормализованных реляционных баз данных.

Выполнив работу, Вы будете:

уметь:

– У6. проектировать логическую и физическую схемы базы данных, используя современные case-средства.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, среда программирования Visual Studio 2019, система управления базами данных Microsoft SQL Server (Express).

Задание:

1. Изучите рекомендуемую литературу.
2. Выясните, является ли спроектированная вами база данных реляционной. При необходимости внесите изменения.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.1 Инструментальные средства этапа разработки баз данных компьютерных систем и комплексов

Практическая работа № 5
Создание индексов, триггеров в базе данных

Цель: Приобрести навыки создания индексов, триггеров в базе данных.

Выполнив работу, Вы будете:

уметь:

– У6. проектировать логическую и физическую схемы базы данных, используя современные case-средства.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, среда программирования Visual Studio 2019, система управления базами данных Microsoft SQL Server (Express).

Задание:

1. Изучите рекомендуемую литературу.
2. Определите список необходимых индексов для таблиц базы данных, создайте их.
3. Определите список необходимых триггеров для таблиц базы данных, создайте триггеры и проведите отладку.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.2 Инструментальные средства этапа разработки программного кода компьютерных систем и комплексов

Практическая работа № 6
Проектирование пользовательского интерфейса

Цель: составить общие рекомендации к интерфейсу приложения, спроектировать интерфейс приложения, используя технологию wireframe.

Выполнив работу, Вы будете:

уметь:

- У7 разрабатывать графический интерфейс приложения;

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, среда программирования Visual Studio 2019, система управления базами данных Microsoft SQL Server (Express).

Задание:

1. Изучите рекомендуемую литературу.
2. Составьте общие рекомендации к интерфейсу вашего приложения: стиль форм, логотип, иконка приложения, шрифты, цветовая палитра, внутренний отступ для графических элементов и т.д.
3. Спроектируйте интерфейс вашего приложения, используя технологию wireframe.
4. Протестируйте интерфейс приложения на разных типах пользователей, зафиксируйте замечания.
5. Проанализируйте замечания, внесите изменения в интерфейс приложения.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.2 Инструментальные средства этапа разработки программного кода компьютерных систем и комплексов

Практическая работа № 7
Разработка пользовательского интерфейса

Цель: Освоить основные приемы разработки интерфейса WPF-приложений на основе страничной организации приложения, создания меню, панели команд, табличных элементов управления и системы команд для выполнения задач приложения.

Выполнив работу, Вы будете:
уметь:
- У7 разрабатывать графический интерфейс приложения.

Материальное обеспечение:
Пакет Microsoft Office, доступ к Internet ресурсам, среда программирования Visual Studio 2019, система управления базами данных Microsoft SQL Server (Express).

Задание:
1. Изучите рекомендуемую литературу.
2. Выделите основные характеристики проекта и представить их в виде ментальной карты.
3. Проведите сравнение проектной и операционной видов деятельности.
4. Выделите функции менеджера проектов. Определить компетенции (soft и hard skills), которыми должен обладать менеджер проектов. Представить функции, компетенции и их взаимосвязь графически.

Ход работы:

Разработка корпоративных приложений на базе WPF

1. Создание приложения

Как правило, корпоративное приложение представляет собой программу, реализующую определенную бизнес-задачу (бизнес-функцию). Приложение должно взаимодействовать с данными, которые располагаются в базе данных информационной системы. Архитектура приложения обычно включает слой представления, бизнес-логики и данных. Функциональность каждого слоя приложения во многом определяется предметной областью информационной системы, но имеются и общие, основополагающие функции, которые присущи практически любому корпоративному приложению. Так в приложении необходимо разработать слой представления, который обеспечит интерфейс пользователя с системой. Интерфейс может быть создан с использованием Windows окон и страниц WPF, которые наполняются различными визуальными элементами контроля. Элементы контроля должны поддерживать визуальное представление функциональности системы для пользователя, проводить верификацию вводимых данных и взаимодействовать с бизнес-классами. Слой бизнес-логики приложения должен обеспечивать основную функциональность приложения: формировать бизнес-классы, реализовывать алгоритмы обработки данных, обеспечивать соединение с данными и их кэширование. Реализация данного слоя приложения может быть построена на базе классов, реализующих бизнес-логику, методами классов интерфейсных элементов или методами классов модели данных. Слой данных должен обеспечить взаимодействие приложения с данными системы управления базами данных. В корпоративных приложениях для этого наиболее целесообразно использовать платформу ADO.NET Entity Framework и модель EDM (Entity Data Model). Модель EDM описывает структуру данных независимо от формы хранения.

Для изучения вопросов проектирования корпоративных приложений рассмотрим основные подходы при разработке отдельной функции информационной системы, которая обеспечивает обработку данных по сотрудникам компании. Для учебного примера используется база данных TitlePersonal с небольшим набором таблиц и полей, а функциональность приложения предполагает

обеспечение ввода, корректировки и удаление данных о сотрудниках компании. Разрабатываемое приложение должно обеспечивать хранение и обработку следующих данных по сотрудникам компании: фамилия; имя; отчество; должность; дата рождения; телефон; адрес электронной почты.

Функции приложения:

1. просмотр данных по сотрудникам;
2. ввод данных по новому сотруднику;
3. редактирование данных по сотруднику;
4. удаление данных по сотруднику;
5. поиск данных по сотруднику.

Для разработки приложения необходимо открыть среду Microsoft Visual Studio (рисунок 1) и нажать «Создание проекта».

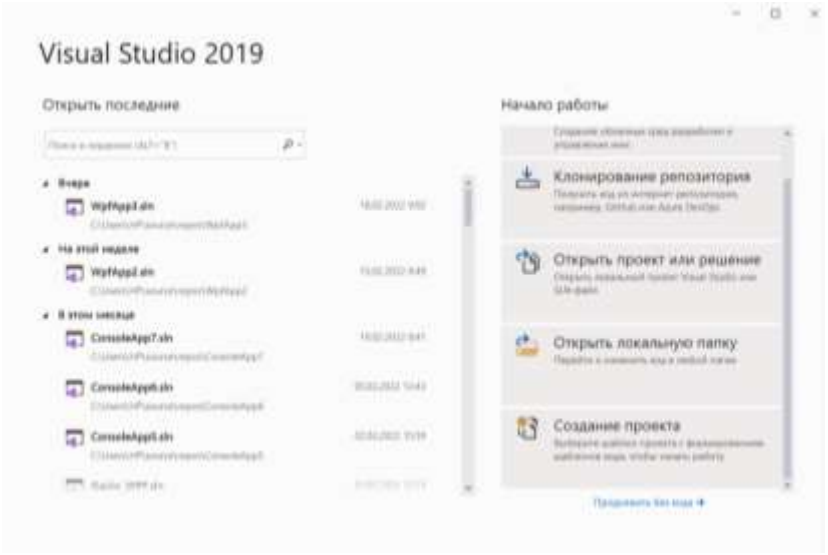

Рисунок 1 – Создание проекта

В окне «Создание проекта» необходимо среди имеющихся шаблонов задать «Приложение WPF» с библиотекой .NET Framework и нажать «Далее» (рисунок 2)

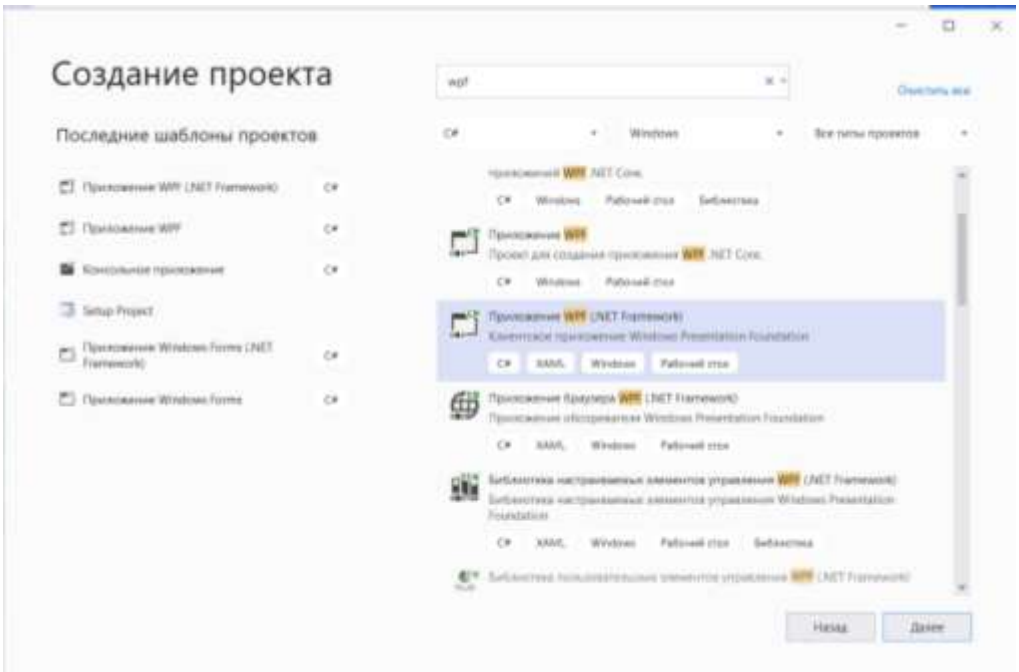

Рисунок 2 – Окно создания проекта

В поле «Имя» ввести имя проекта «WPF_FIO», где FIO – ваша фамилия (рисунок 3).

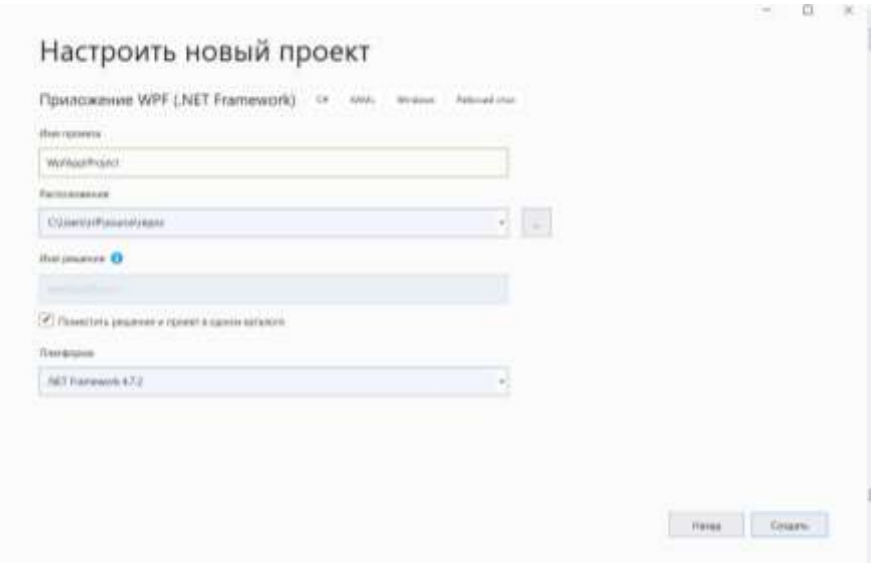

Рисунок 3 – Задание имени проекта

После нажатия кнопки «Создать» будет сформирован шаблон проекта. При этом инструментальная система сгенерирует следующий XAML-документ:

```
<Window x:Class="Wpf_Erina.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Дизайнер проекта приведен на рисунке 4.

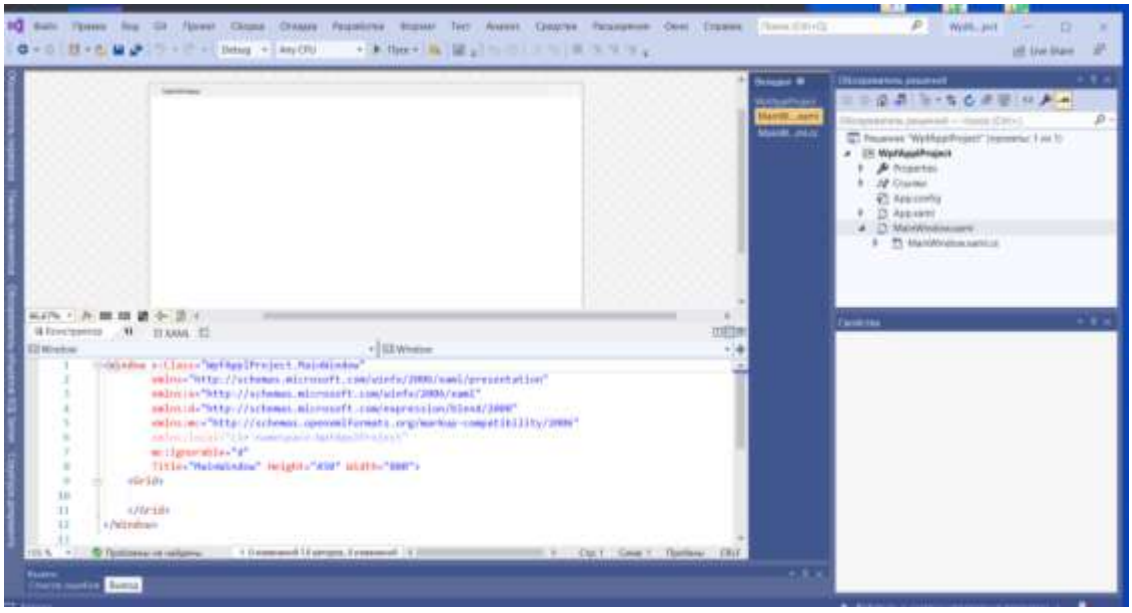

Рисунок 4 – Окно дизайнера проекта

В приведенном XAML-документе имеется один элемент верхнего уровня <*Window*>. Дескриптор </*Window*> завершает весь документ. В XAML-документе приведено имя класса *MainWindow:*

```
x:Class="Wpf_Erina.MainWindow"
```

b два пространства имен

```
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

и три свойства:

```
        Title="MainWindow" Height="350" Width="525"
```

Каждый атрибут соответствует определенному свойству класса *Window*. Приведенные атрибуты предписывают WPF создать окно с надписью *MainWindow* и размером 350x525 единиц. При компиляции и запуске проекта приложения на дисплей выводится окно, приведенное на рисунке 5.

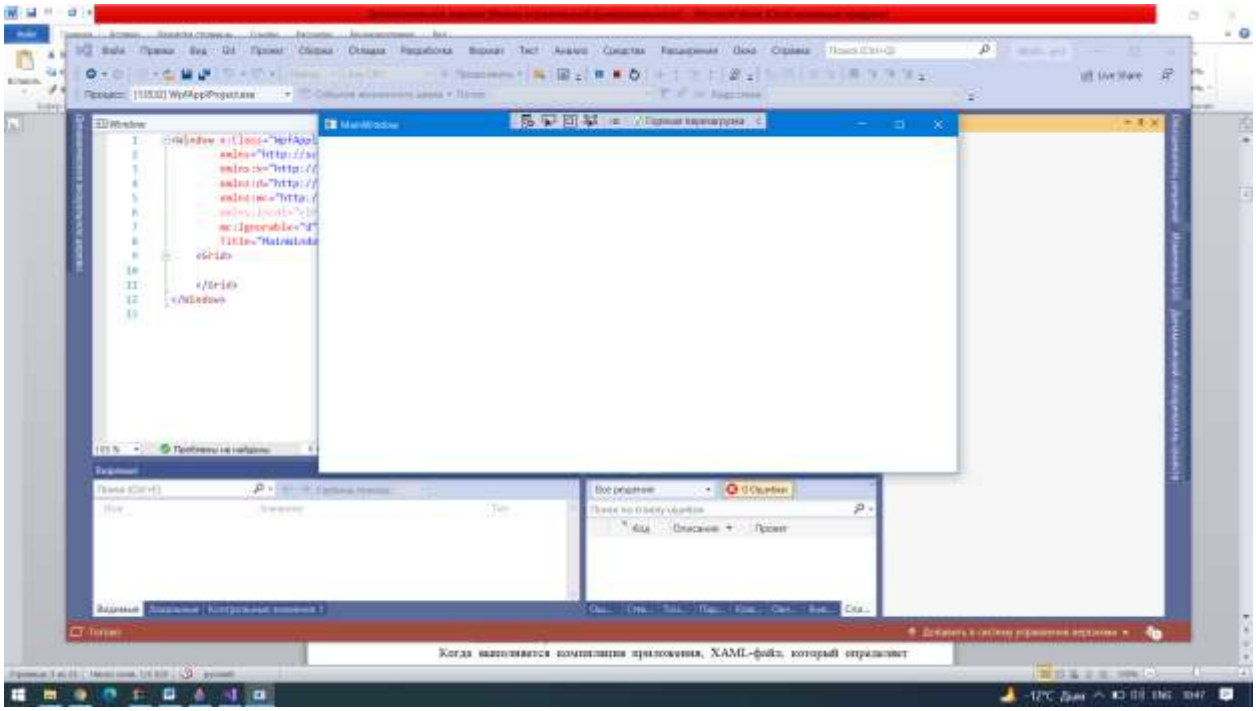

Рисунок 5 – Окно *MainWindow* проекта

Когда выполняется компиляция приложения, XAML-файл, который определяет пользовательский интерфейс (*MainWindow.xaml*), транслируется в объявление типа CLR , которое объединяется с логикой приложения из файла класса отдельного кода (*MainWindow.xaml.cs*).

Метод *InitializeComponent()* генерируется во время компиляции приложения и в исходном коде не присутствует.

Для программного управления элементами управления, описанными в XAML-документе, необходимо задать XAML атрибут *Name*. Так для задания имени элементу *Grid* необходимо записать следующую разметку:

```
<Grid Name="grid">

</Grid>
```

2. Формирование начальной страницы приложения

В последнее время разработчики корпоративных приложений начали осознавать преимущества технологий веб-дизайна, которые базируются на качественном дизайне, четком и понятном интерфейсе. Технология WPF позволяет создавать страничную модель (приложения), с готовыми средствами навигации. Как правило, для каждой страницы приложения создается файл XAML и файл отдельного кода, например на языке C#. При компиляции такого приложения компилятор создает производный класс страницы, который объединяет написанный код с генерируемыми автоматически связующими элементами.

Страницы можно размещать внутри окон и внутри других страниц. В WPF при создании страничных приложений контейнером наивысшего уровня могут быть следующие объекты:

NavigationWindow, который представляет собой несколько видоизмененную версию класса *Window*;

Frame, находящийся внутри другого окна или другой страницы;

Frame, обслуживаемый непосредственно в Internet Explorer.

Для вставки страницы внутрь окна будем использовать класс *Frame* (рисунок 6). Автоматически будет сгенерирован экземпляр класса *Frame* с фиксированными границами (рисунок 7).

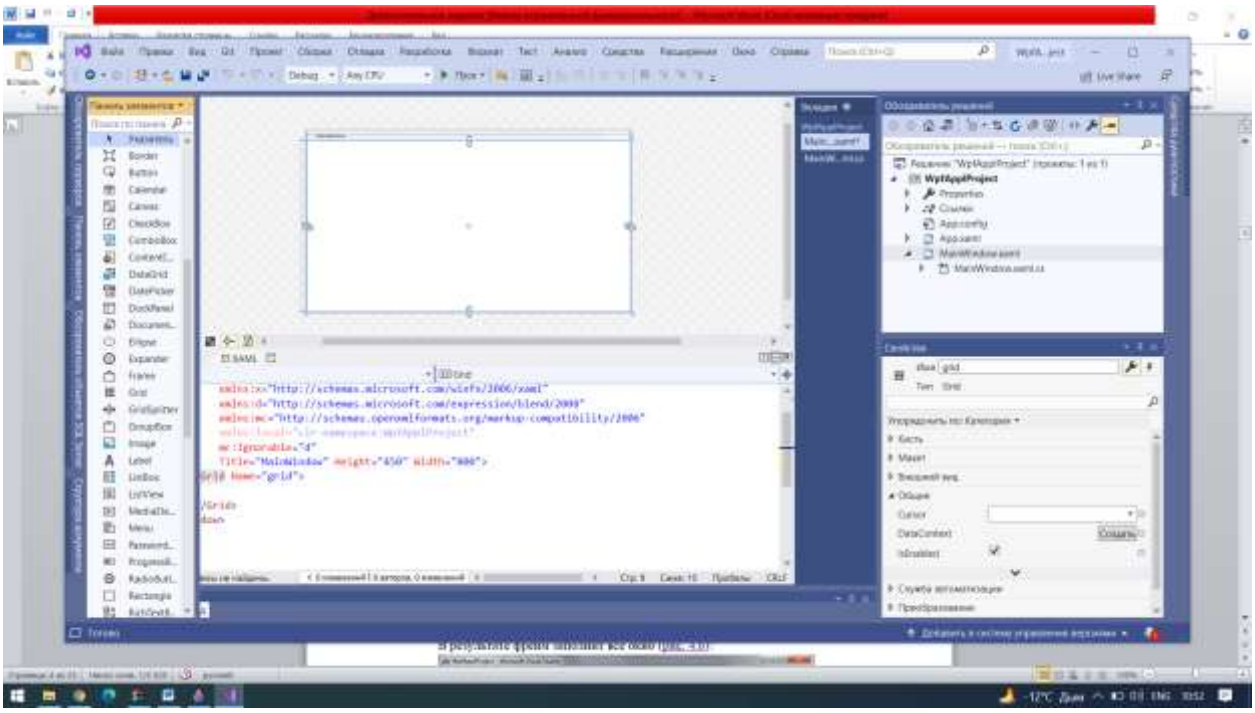

Рисунок 6 – Выбор класса *Frame* в панели инструментов

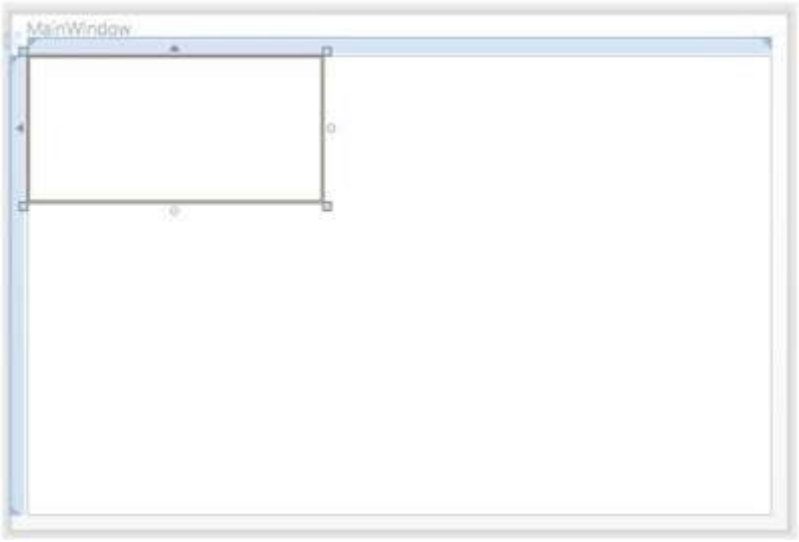

Рисунок 7 – Фрейм с фиксированными границами

В XAML-документ проекта будет добавлена следующая строка:

```
<Frame Content="Frame" HorizontalAlignment="Left" Height="100" Ma
rgin="144,95,0,0" VerticalAlignment="Top" Width="100"/>
```

С учетом того, что создается страничное приложение, размеры фрейма не нужно фиксировать, поэтому изменим описание свойств фрейма:

```
<Frame Name ="frame1" Margin="3" />
```

В результате фрейм заполнит всё окно (рисунок 8):

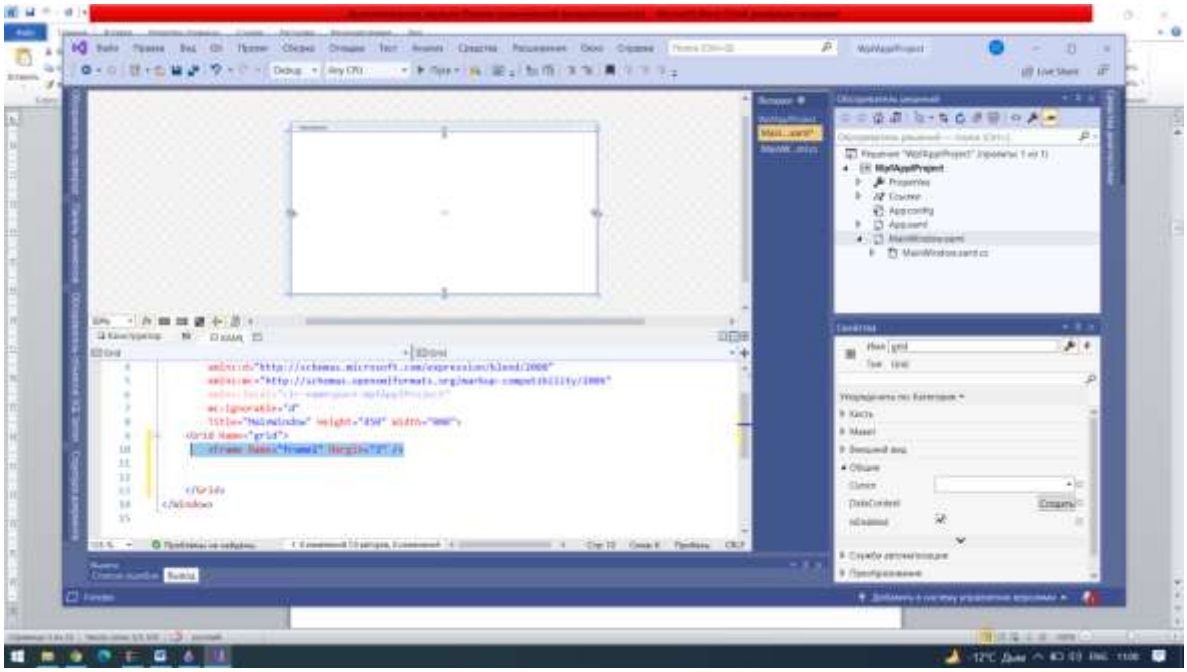

Рисунок 8 – Фрейм, заполняющий всё окно

Созданный фрейм необходим для размещения в нем страницы WPF-приложения. Класс *Page* допускает наличие только одного вложенного элемента. Он не является элементом управления содержимым и наследуется от класса *FrameworkElement*. Класс *Page* имеет небольшой набор дополнительных свойств, которые позволяют настраивать его внешний вид, взаимодействовать с контейнером и использовать навигацию. Для перехода на другую страницу необходимо использовать навигацию.

Класс *Page* имеет следующие свойства:

Background – принимает кисть, которая устанавливает заливку для фона;

Content – принимает один элемент, который отображается на странице. Обычно в роли такого элемента выступает контейнер макета (элемент *Grid* или *StackPanel*);

Foreground, FontFamile, FontSize – определяет используемый по умолчанию внешний вид для текста внутри страницы. Значение этих свойств наследуется элементами внутри страницы.

WindowWidth, WindowHeight – определяет внешний вид окна, в которое упаковывается страница (не действуют, если страница обслуживается во фрейме);

NavigationService – возвращает ссылку на объект *NavigationService*, которую можно использовать для отправки пользователя на другую страницу программным путем;

KeepAlive – определяет, должен ли объект страницы оставаться действующим после перехода пользователя на другую страницу;

ShowsNavigationUI – определяет, должны ли в обслуживающем данную страницу хосте отображаться навигационные элементы управления;

Title – устанавливает имя, которое должно применяться для страницы в хронологии навигации;

Window Title – устанавливает заголовок окна, который отображается в строке заголовка.

Добавим в проект начальную страницу. Для этого в Обозревателе решений щелкнем правой кнопкой мыши на проекте *Wpf_FIO*. В контекстном меню выберем пункт **Добавить** (1 – на рисунке 9), а в раскрывающемся меню пункт **Страница** (2 на рисунке 10).

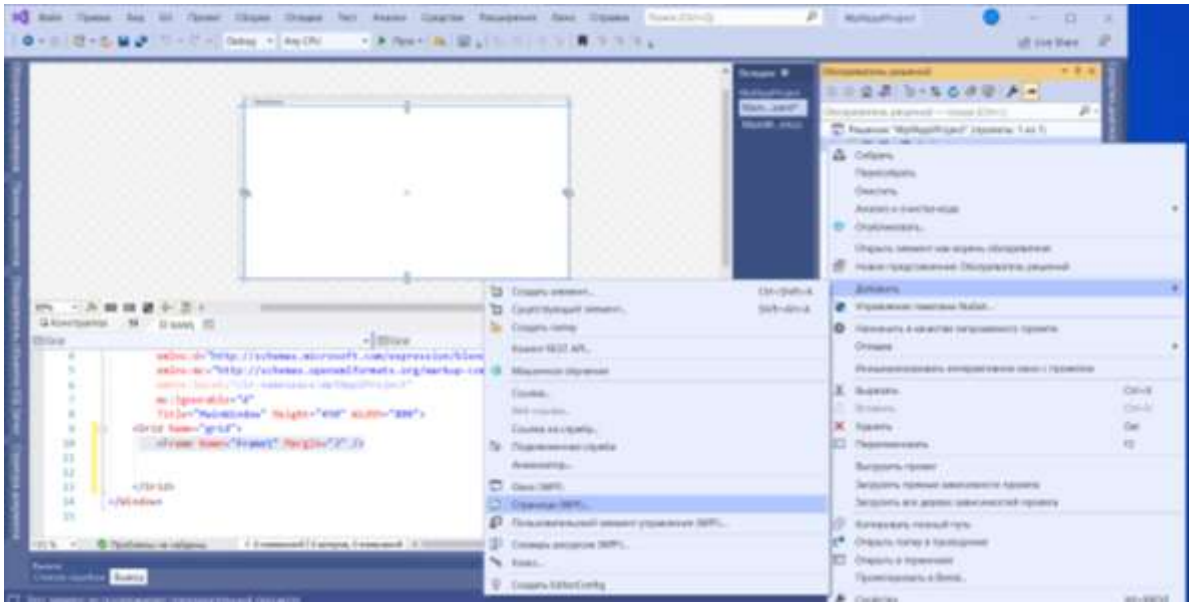

Рисунок 9 – Меню создания страницы

В окне Добавления нового элемента необходимо выбрать шаблон **Страница (WPF)** (1) и задать имя страницы *PageMain* (2 на рисунке 10).

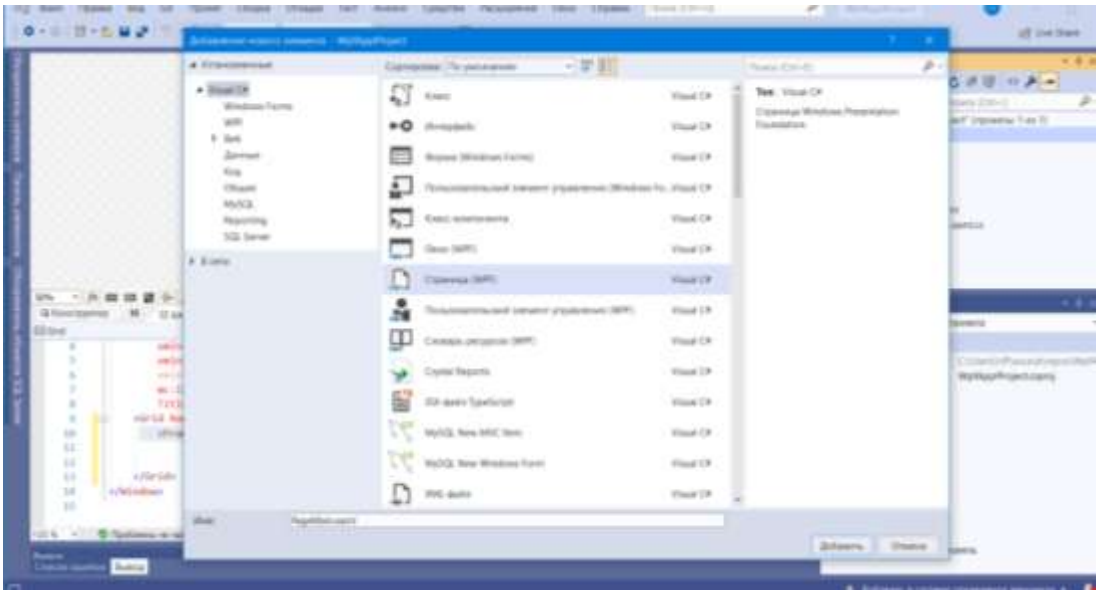

Рисунок 10 – Окно добавления нового элемента

В дизайнере проекта сгенерируется страница *PageMain.xaml* (рисунок 11).

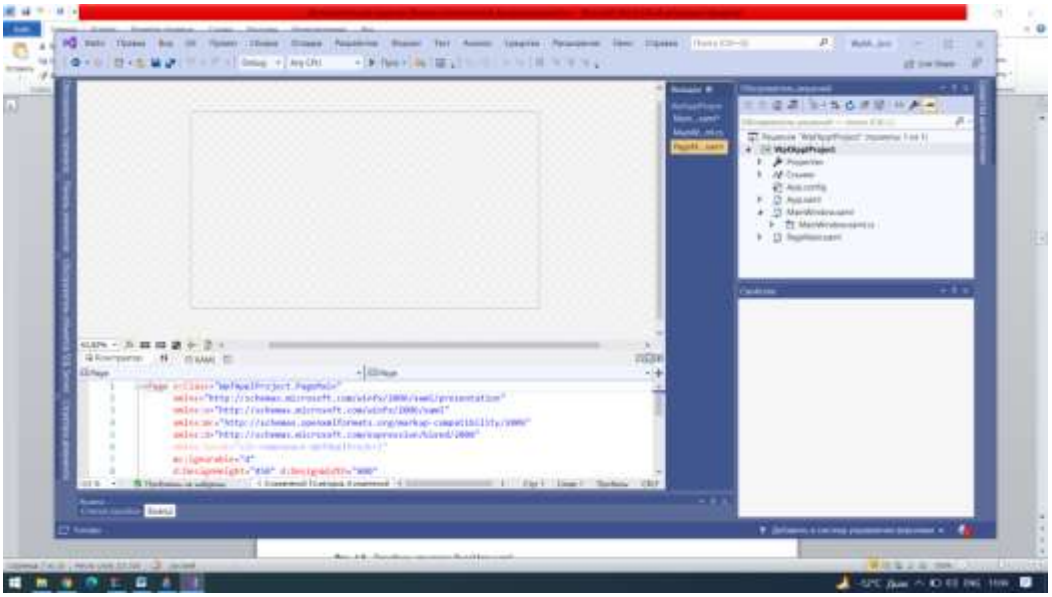

Рисунок 11 – Дизайнер страницы *PageMain.xaml*

В сгенерированной странице в качестве контейнера верхнего уровня используется *Grid*. Заменим *Grid* на контейнер *StackPanel*.

Объект *StackPanel* имеет свойство *Background*, которое предоставляет кисть для рисования фоновой области элемента. Данное свойство является сложным, т.е. должно задаваться в следующем виде:

```
<StackPanel.Background>
   ...
</StackPanel.Background>
```

Классы кистей наследуются от класса *System.Windows.Media.Brush* и обеспечивают различные эффекты при заполнении фона, переднего плана или границ элементов. Некоторые классы кистей приведены ниже:

LineaGradientBrush – рисует область, используя линейное градиентное заполнение, представляющее собой плавный переход от одного цвета к другому;

RadialGradientBrush – рисует область, используя радиальное градиентное заполнение, представляющее собой плавный переход от одного цвета к другому в радиальном направлении от центральной точки;

ImageBrush – рисует область, используя графическое изображение, которое может растягиваться, масштабироваться или многократно повторяться;

DrawingBrush – рисует область, используя объекты *Drawing*, которые могут быть пользовательскими фигурами или битовыми картами;

VisualBrush – рисует область, используя объекты *Visual*.

Создадим градиентную заливку созданной страницы. Для определения градиентной заливки необходимо создать объект *LinearGradientBrush*. Далее необходимо заполнить свойство *LinearGradientBrush.GradientStops* коллекцией объектов *GradientStop*. Каждый объект *GradientStop* имеет свойства *Offset* и *Color*, значение которых задаются в соответствии с синтаксисом " свойство - атрибут ". Свойство *Offset* может принимать значение от 0 до 1, определяя области заполнения градиентом от одного цвета к другому.

Далее приведен фрагмент XAML-документа для определения градиентной заливки страницы.

```
<StackPanel>
    <StackPanel.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Offset="0.00" Color="#FF87A0DE"/>
                <GradientStop Offset="0.50" Color="Azure"/>
                <GradientStop Offset="1.00" Color="#FF009CFF"/>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </StackPanel.Background>
</StackPanel>
```

Добавим в страницу текстовую строку "Система внутреннего учета инвестиционной компании".

```
<TextBlock Margin="25,30,25,5" FontSize="20">
    Система внутреннего учета инвестиционной компании
</TextBlock>
```

Подключим созданную страницу к фрейму. Для этого в разметке фрейма необходимо указать источник его заполнения *Source*, а также определим свойства *Foreground*, *BorderBrush* и *Background*.

```
<Grid Name="grid">
    <Frame Name ="frame1" Margin="3"
           Source="PageMain.xaml"
           Foreground="#FFE4D8D8"
           BorderBrush="#FFF7F3F3"
           Background="#FFB3D9E5"/>
</Grid>
```

Главная страница приложения в дизайнере представлена на Рисунок 10.

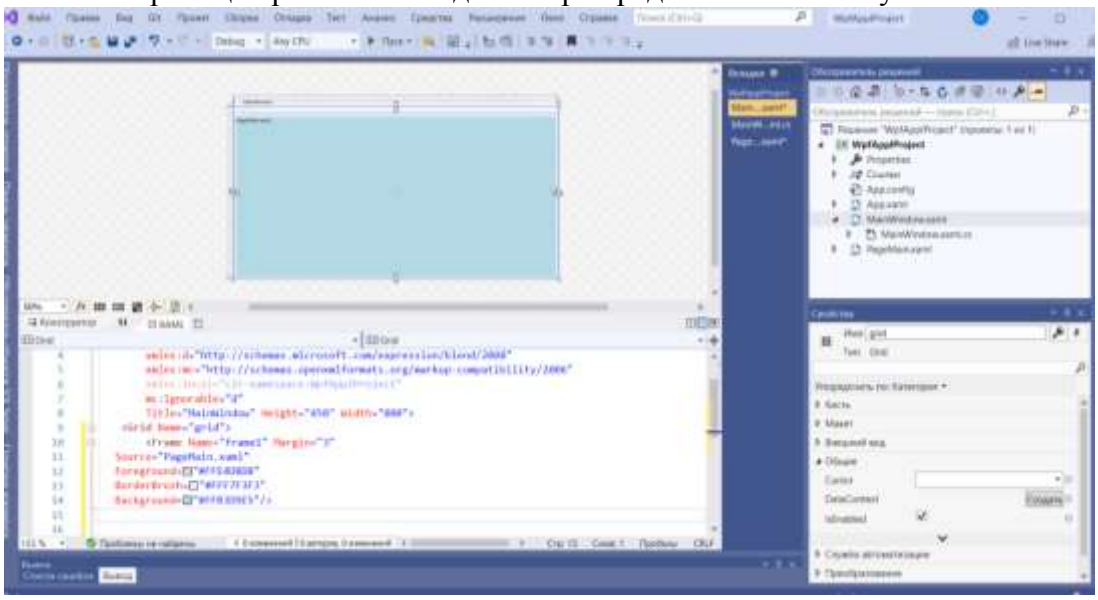

Рисунок 10. Главная страница WPF-приложения в дизайнере

Изменим свойство *Title* окна класса *Window*, присвоив ему значение "*Информационная система ВУ*". Результат компиляции и выполнения WPF-приложения приведен на Рисунок 11.

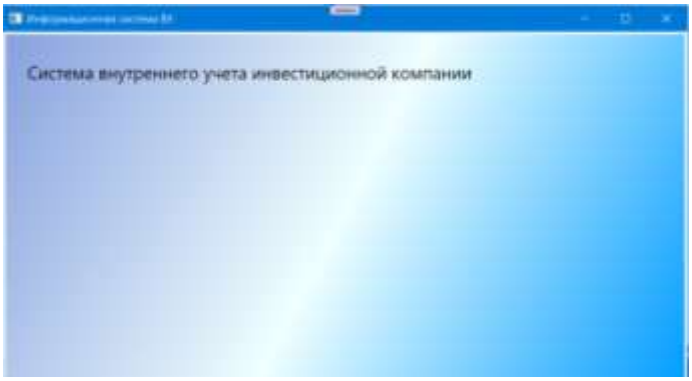

Рисунок 11. Главная страница WPF-приложения

Контейнером верхнего уровня главной страницы приложения является *StackPanel*. Для данной панели свойству *Background*, которое определяет кисть для рисования фоновой области, установлено сложное значение, определяющее градиентную заливку. С учетом того, что в приложении предполагается использовать аналогичную градиентную кисть и для других страниц, то целесообразным является сформировать значение для данной кисти в виде ресурса, разместив его на уровне приложения. Для этого найдите в Обозревателе решения файл приложения App.xaml и добавьте в него ресурс:

```
<Application.Resources>
    <LinearGradientBrush x:Key="BackgroundWindowResource">
        <GradientStop Offset="0.00" Color="#FF87A0DE"/>
        <GradientStop Offset="0.50" Color="Azure"/>
        <GradientStop Offset="1.00" Color="#FF009CFF"/>
    </LinearGradientBrush>
</Application.Resources>
```

Сформированный ресурс для градиентной кисти имеет ключ *BackgroundWindowResource*. Введение в XAML описание приложения ресурса требует провести изменение свойства *Background* панели *StackPanel*, использую расширения разметки и ссылку на статический ресурс *BackgroundWindowResource*.

```
<StackPanel Background="{StaticResource BackgroundWindowResource}">
    <TextBlock Margin="25,30,25,5" FontSize="20">
        Система внутреннего учета инвестиционной компании
    </TextBlock>
</StackPanel>
```

На странице WPF-приложения можно размещать элементы пользовательского интерфейса (элементы управления) для обеспечения взаимодействия пользователя с бизнес-логикой системы.

3. Навигация страничного приложения

Основная страница должна обеспечивать переход на другие страницы, обеспечивающие интерфейс для отдельных функций и выход из системы. Для перехода на другие страницы будем использовать гиперссылки. Гиперссылки позволяют пользователю перемещаться с одной страницы на другую. Элемент гиперссылки, соответствующий объекту класса *Hyperlink*, определяется следующей строкой:

<Hyperlink NavigateUri="PageName.xaml">Текст Гиперссылки</Hyperlink>

Класс *Hyperlink* имеет свойство *NavigateUri*. Данное свойство определяет на какую страницу будет переходить приложение при щелчке на соответствующей гиперссылке. Например, *NavigateUri="Page2.xaml"*.

В WPF гиперссылки являются не отдельными, а внутристроковыми потоковыми элементами, которые должны размещаться внутри другого поддерживающего их элемента. Это можно сделать, например в элементе *TextBlock*, который для гиперссылки является контейнером.

На первой странице создадим следующие гиперссылки: Сотрудники, Клиенты, Договора, Ценные бумаги, Сделки и Справка. XAML-описание гиперссылки для страницы Сотрудники приведено ниже.

```
<TextBlock Margin="25,30,5, 5">
    <Hyperlink NavigateUri="PageEmployee.xaml">Сотрудники</Hyperlink>
</TextBlock>
```

Остальные гиперссылки добавьте самостоятельно.

Для реализации функциональности первого окна WPF-приложения осталось добавить кнопку выхода.

```
<Button Margin="25,15" Width="60" HorizontalAlignment="Left"
        Command="Close">Выход</Button>
```

Результат компиляции и выполнения WPF-приложения приведен на Рисунок 12.

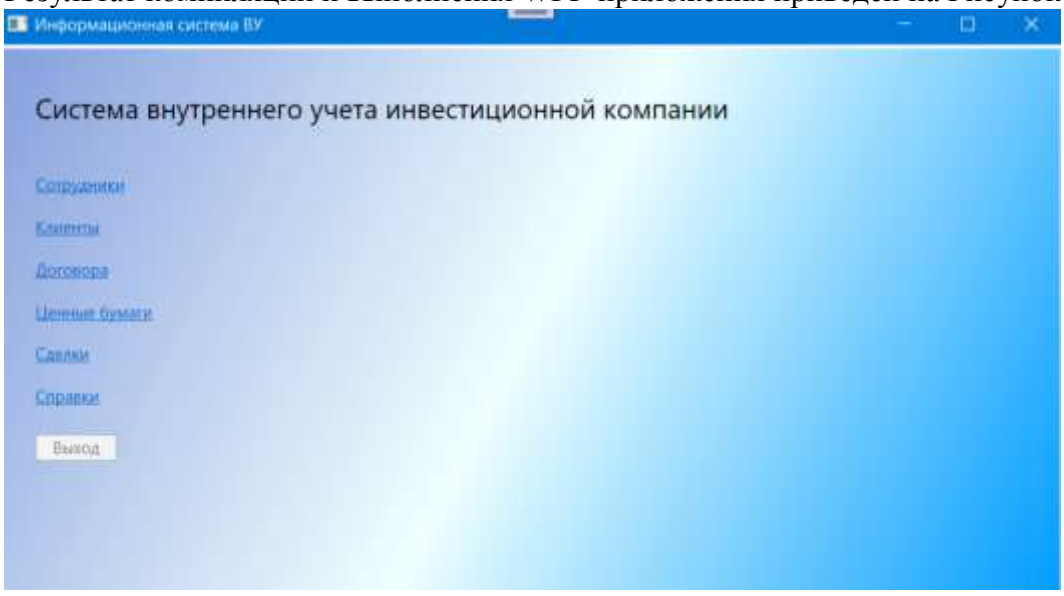

Рисунок 12. Начальная страница WPF-приложения с гиперссылками

По условию учебного примера будем постепенно добавлять функциональность приложения в части обработки данных по сотрудникам компании.

Создадим страницу с именем *PageEmployee*. В качестве основного контейнера будем использовать панель *StackPanel*. Определим для неё градиентную заливку, аналогичную начальной странице приложения.

```
<Page x:Class="Wpf_Erina.PageEmployee"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
    Title="PageEmployee">
    <StackPanel Margin="3" Background="{StaticResource BackgroundWindowResource}">
    </StackPanel>
</Page>
```

Страница *PageEmployee* должна обеспечивать пользователю просмотр данных по сотрудникам, ввод данных по новому сотруднику, редактирование, поиск и удаление данных. Доступ пользователя к функциональности системы будем предоставлять через различные меню.

4. Проектирование интерфейса

В WPF можно создать основное и контекстное меню. Кроме того, можно создать панель инструментов с кнопками, которые будут реализовывать функциональность, аналогичную пунктам меню.

Основное меню создается с помощью класса *Menu*, который представляет Windows элементы управления меню, позволяющие иерархически организовать элементы, связанные с командами и обработчиками событий. Меню формируют из объектов *MenuItem* (имя

пункта меню) и *Separator* (разделитель). Класс *MenuItem* имеет свойство *Header*, которое определяет текст элемента меню. Данный класс может хранить коллекцию объектов *MenuItem*, которая соответствует подпунктам меню. Класс *Separator* отображает горизонтальную линию, разделяющую пункты меню.

Добавим в *StackPanel* страницы *PageEmployee* XAML- описание меню, которое на верхнем уровне будет содержать два пункта *Действие* и *Отчет*. Пункт *Действие* включает подпункты *Отменить*, *Создать*, *Редактировать*, *Сохранить*, *Найти* и *Удалить*. Между пунктами *Отменить*, *Создать* и пунктами *Найти*, *Удалить* добавлены разделительные линии.

```
<Menu>
    <MenuItem Header="Действие" >
        <MenuItem Header="Отменить" ></MenuItem>
        <Separator></Separator>
        <MenuItem Header="Создать" ></MenuItem>
        <MenuItem Header="Редактировать" ></MenuItem>
        <MenuItem Header="Сохранить" ></MenuItem>
        <MenuItem Header="Найти" />
        <Separator></Separator>
        <MenuItem Header="Удалить" ></MenuItem>
    </MenuItem>
    <MenuItem Header="Отчет"></MenuItem>
</Menu>
```

Запустим приложение и выберем ссылку Сотрудники на странице *PageEmployee*. При выборе пункта меню Действие появляется выпадающий список и пунктами подменю (Рисунок 13).

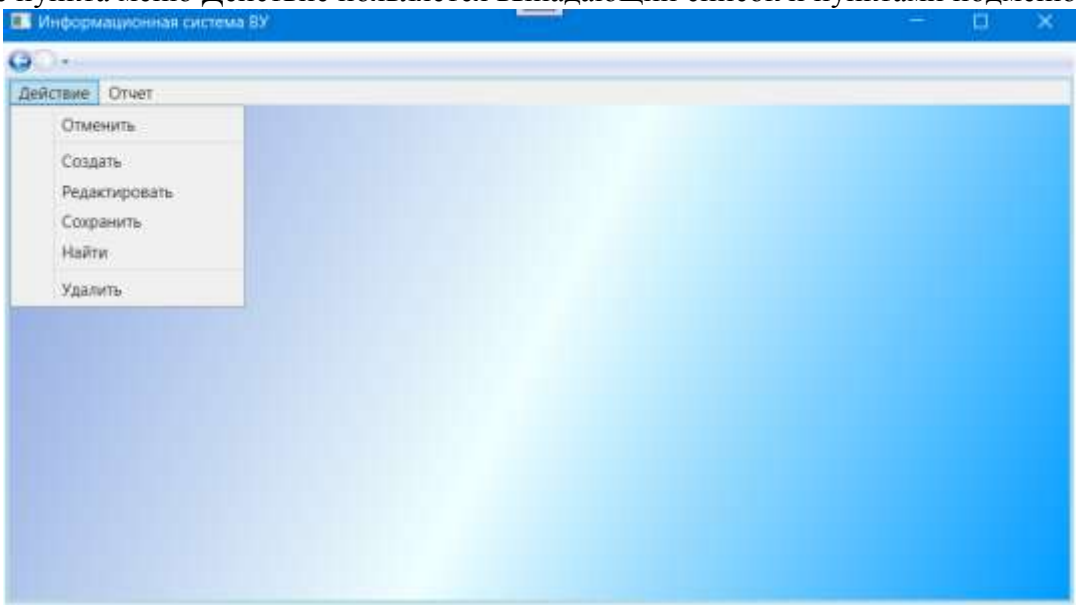

Рисунок 13. Главное меню страницы *PageEmployee*

Панель инструментов представляет специализированный контейнер для хранения коллекции элементов, обычно кнопок. Расположим в панели инструментов кнопки, функциональное назначение которых будет соответствовать подпунктам меню *Действие*, то есть *Отменить*, *Создать*, *Редактировать*, *Сохранить*, *Найти* и *Удалить*.

На лицевой стороне кнопок поместим графическое изображение соответствующего действия. Для этого добавим в файл проекта папку *Images* и в неё включим графические объекты, которые можно найти в стандартной библиотеке графических объектов Visual Studio (в файле **VS2013 Image Library.zip**, его всегда можно бесплатно скачать с официального сайта Microsoft**, для вас архив можно скачать по ссылке:** VS2013 Image Library).

После добавления графических файлов в проект они будут отображены в обозревателе решений (Рисунок 14).

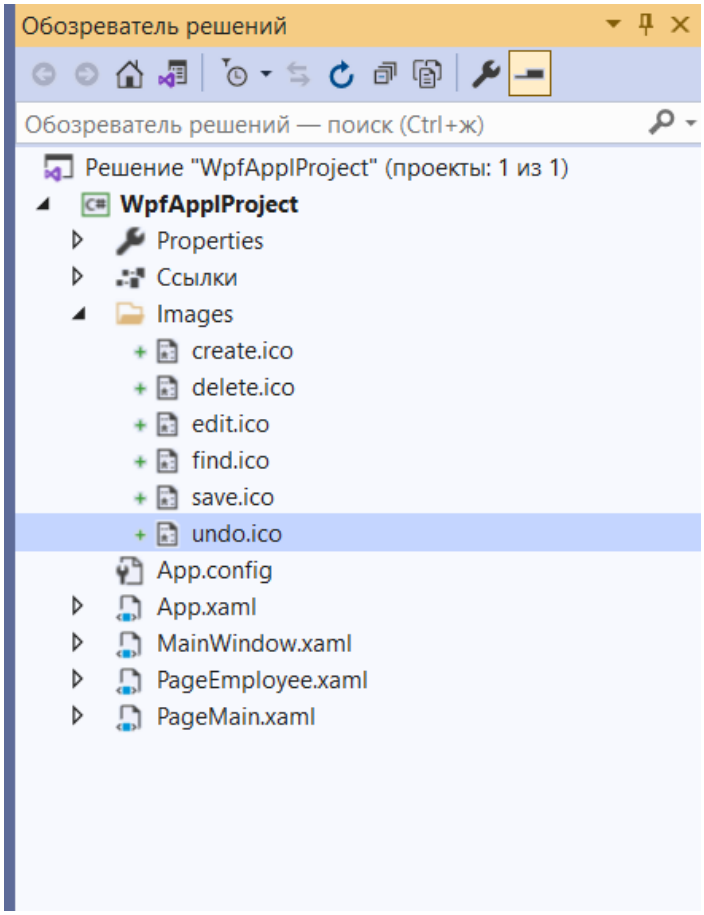

Рисунок 14. Включение в проект папки *Images* с файлами рисунков

Теперь можно разработать XAML-описание панели инструментов для страницы *PageEmployee*.

Для каждой кнопки зададим свойство *Name* – имя объекта в программе и свойство *ToolTip* с текстом всплывающей подсказки при наведении указателя мыши на кнопку.

Свойство *Margin* определяет внешние отступы для кнопки. Задание графического объекта для кнопки осуществляется определением для объекта *Image* источника *Source*, который должен соответствовать полному пути к графическому файлу.

```
<ToolBar Name="ToolBar1" Margin="3">
    <Button Name="Undo" ToolTip="Отменить редактирование/создание" Margin="5,2,5,2">
        <Image Source="Images/Undo.png" />
    </Button>
    ...
</ToolBar>
```

В дизайнере Visual Studio страница *PageEmployee* примет вид, приведенный на Рисунок 15.

Добавьте все необходимые кнопки на панель инструментов приложения.

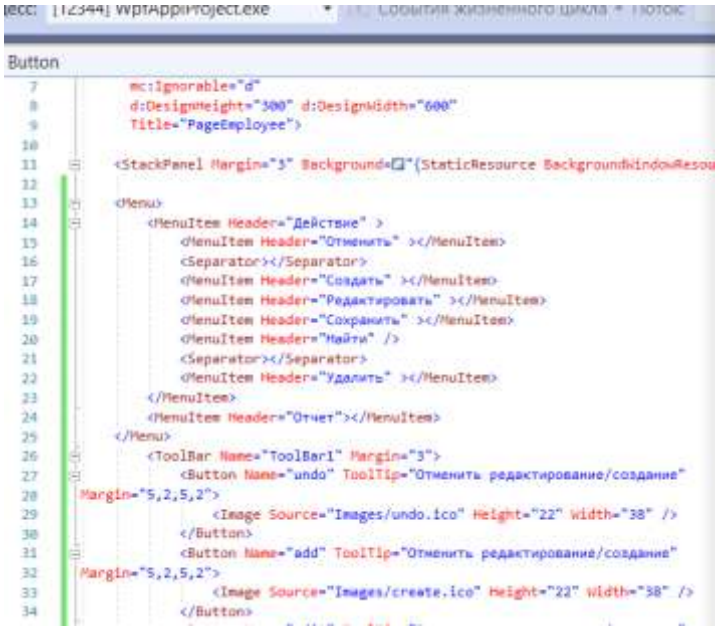

Рисунок 15. Страниц *PageEmployee* с панелью инструментов в дизайнере

После запуска приложения и выборе кнопки панели инструментов Добавить (Add) страница *PageEmployee* будет иметь вид, приведенный на Рисунок 16.

Рисунок 16. Страница *PageEmployee* с панелью инструментов

Следующим шагом проектирования интерфейсных элементов для страницы *PageEmployee* является решение задачи формирования отображения данных по сотрудникам предприятия. Данная задача может быть решена различными способами: с помощью элементов контроля *ListBox*, *ListView*, *TextBox*, *TextBlock*, *ComboBox*, *DataGrid* и других.

В учебном примере будем использовать элемент контроля *DataGrid* для табличного отображения данных о сотрудниках. В качестве заголовка таблицы применим текстовый блок "Список сотрудников".

Класс *DataGrid* представляет элемент управления, отображающий данные в настраиваемой сетке строк и столбцов. По умолчанию *DataGrid* автоматически создает столбцы, на основе источника данных. При этом генерируются следующие типы столбцов:

DataGridTextColumn – для отображения в ячейках столбцов текстового содержимого;

DataGridCheckBoxColomn – для отображения в ячейках столбцов логических данных;

DataGridComboBoxColomn – для отображения в ячейках столбцов данных, когда имеется набор элементов для выбора;

DataGridHyperlinkColomn – для отображения в ячейках столбцов элементов *Uri*.

Если разработчика не устраивает автоматическая генерация столбцов *DataGrid* можно её отключить. Для этого свойству *AutoGenerateColumns* необходимо присвоить значение *false*. Далее можно создать собственный набор столбцов (*Columns*), используя существующие типы столбцов или создать новый тип столбца с помощью шаблона *DataGridTemplateColumn*.

DataGrid поддерживает множество способов настройки отображения данных. В Таблица 1 приведен список стандартных сценариев.

Таблица 1. Сценарии настройки отображения данных

Сценарий	Подход
Переменные цвета фона	Задайте для свойства *AlternationIndex* значение 2 или больше, а затем назначьте объект *Brush* свойствам *RowBackground* и *AlternatingRowBackground*.
Определение поведения при выборе ячейки и строки	Установите свойства *SelectionMode* и *SelectionUnit*.
Настройка внешнего вида заголовков, ячеек и строк	Примените новый *Style* к свойствам *ColumnHeaderStyle*, *RowHeaderStyle*, *CellStyle* или *RowStyle*.
Доступ к выбранным элементам	Проверьте свойство *SelectedCells*, чтобы получить выделенные ячейки, и свойство *SelectedItems*, чтобы получить выбранные строки.
Настройка взаимодействия с пользователем	Установите свойства *CanUserAddRows*, *CanUserDeleteRows*, *CanUserReorderColumns*, *CanUserResizeColumns*, *CanUserResizeRows* и *CanUserSortColumns*.
Отмена или изменение автоматически созданных столбцов	Обработать событие *AutoGeneratingColumn*.
Заморозка столбца	Задайте для свойства *FrozenColumnCount* значение 1 и переместите столбец в крайнюю левую позицию, задав для свойства *DisplayIndex* значение 0.
В качестве источника данных используются данные XML.	Привяжите *ItemsSource* в *DataGrid* к запросу *XPath*, представляющему коллекцию элементов. Создайте каждый столбец в *DataGrid*. Свяжите каждый столбец, указав *XPath* для привязки к запросу, который получает свойство из источника элемента.

XAML-документ описания интерфейсных элементов страницы *PageEmployee* имеет следующий вид.

```
<TextBlock Margin="5" >Список сотрудников</TextBlock>
<DataGrid Name="DataGridEmployee" >
    <DataGrid.Columns>
        <DataGridTextColumn Header="Фамилия"/>
        ...
        <DataGridComboBoxColumn Header="Должность" />
        ...
        <DataGridTemplateColumn Header="Дата рождения" />
    </DataGrid.Columns>
</DataGrid>
```

Для полей Фамилия, Имя, Отчество, Телефон и Электронная почта используется тип столбца *DataGridTextColumn*. Так как должность сотрудника задается в соответствии с данными из справочника базы данных, то есть предоставляется список, из которого можно произвести выбор, для данного столбца используется тип *DataGridComboBoxColumn*. Для столбца Дата рождения используется специфичный шаблон, то есть тип *DataGridTemplateColumn*, который будет описан позднее.

С учетом добавленных интерфейсных элементов страница *PageEmployee* в дизайнере примет вид, приведенный на Рисунок 17.

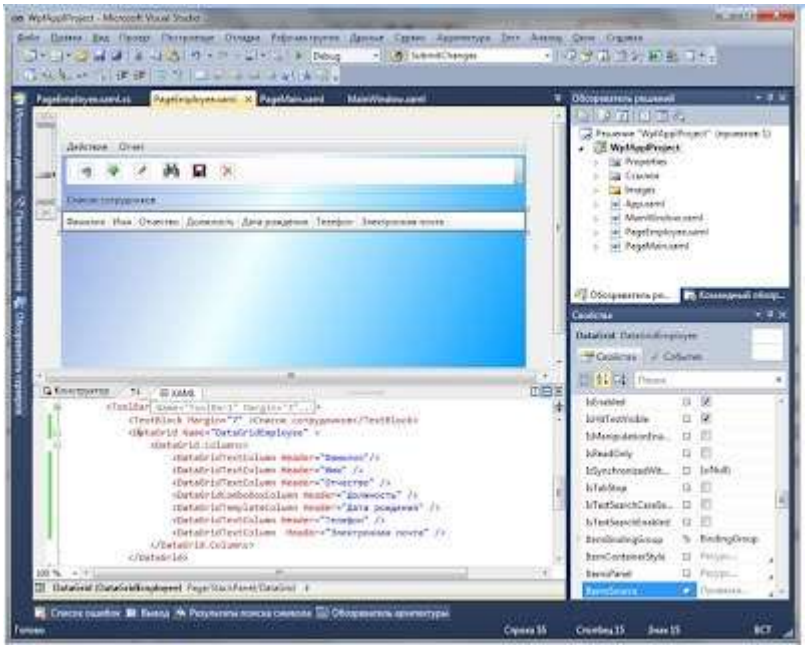

Рисунок 17. Страница *PageEmployee* с элементом *DataGrid* в дизайнере

После запуска приложения страница *PageEmployee* будет иметь вид, приведенный на Рисунок 18.

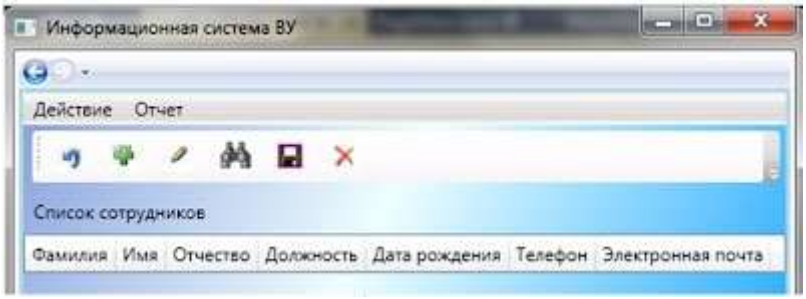

Рисунок 18. Страница *PageEmployee* с элементом *DataGrid*

На данном этапе проектирования приложения на странице *PageEmployee* размещены все необходимые элементы контроля.

Форма представления результата:

7. Цель работы
8. Введение
9. Программно-аппаратные средства, используемые при выполнении работы
10. Описание работы
11. Заключение (выводы)
12. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.2 Инструментальные средства этапа разработки программного кода компьютерных систем и комплексов

Практическая работа № 8
Разработка программных модулей

Цель: Сформировать понятие и основные характеристики проекта. Выяснить причины необходимости ведения проектной деятельности и управления проектом. Определить отличия проекта и операционной деятельности.

Выполнив работу, Вы будете:

уметь:

- У8 осуществлять разработку кода информационных систем и программных средств на языках высокого уровня;

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам.

Задание:

5. Изучите рекомендуемую литературу.
6. Выделите основные характеристики проекта и представить их в виде ментальной карты.
7. Проведите сравнение проектной и операционной видов деятельности.
8. Выделите функции менеджера проектов. Определить компетенции (soft и hard skills), которыми должен обладать менеджер проектов. Представить функции, компетенции и их взаимосвязь графически.

Ход работы:

1. Разработка бизнес-логики

Ваш проект на данный момент не должен содержать ошибок!

Для реализации функциональности приложения в части обработки информации о сотрудниках предприятия необходимо для страницы *PageEmployee* установить механизм запуска задач (*Отменить*, *Создать*, *Редактировать*, *Сохранить*, *Найти* и *Удалить*) при выборе соответствующих пунктов меню и нажатии на кнопки панели инструментов. Технология WPF предлагает ***модель команд*** для выполнения такой привязки.

Модель команд обеспечивает делегирование событий определенным командам и управление доступностью элементов управления в зависимости от состояния соответствующей команды. В WPF ***команда*** представляет собой ***задачу приложения*** и ***механизм слежения*** за тем, когда она может быть выполнена. В то же время сама команда не содержит конкретного кода выполнения задачи. Одна и та же команда может быть привязана к одному или нескольким интерфейсным элементам приложения. ***Инициируют команду источники***, которые могут быть различными элементами управления, например пункты меню *MenuItem* или кнопки – *Button*. Целевым объектом команды является элемент, для которого предназначена эта команда.

Классы, реализующие команды должны поддерживать интерфейс *ICommand*. В этом интерфейсе определены два метода *Execute, CanExecute* и событие *CanExecuteChanged*.

В WPF имеется ***библиотека базовых команд***. Команды доступны через статические свойства следующих статических классов:

- *ApplicationCommands*;
- *NavigationCommands*;
- *EditingCommands*;
- *MediaCommands*.

Для создания пользовательских команд целесообразно использовать классы *RoutedCommand*, который имеет реализацию интерфейса *ICommand*.

В разрабатываемом приложении для функций *Отменить*, *Создать*, *Сохранить* и *Найти* будем использовать *стандартные команды из*

библиотеки WPF – статический класс *ApplicationCommands*, а для функций *Редактировать* и *Удалить* спроектируем *пользовательские команды*.

Для разработки пользовательских команд добавим в проект папку *Commands* и в ней создадим класс *DataCommands*. (рисунок 1).

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

namespace Wpf_Erina.Commands
{
    public class DataCommands
    {
        public static RoutedCommand Delete { get; set; }
        public static RoutedCommand Edit { get; set; }
        static DataCommands()
        {
            InputGestureCollection inputs = new InputGestureCollection();
            inputs.Add(new KeyGesture(Key.E, ModifierKeys.Control, "Ctrl+E"));
            Edit = new RoutedCommand("Edit", typeof(DataCommands), inputs);
            inputs = new InputGestureCollection();
            inputs.Add(new KeyGesture(Key.D, ModifierKeys.Control, "Ctrl+D"));
            Delete = new RoutedCommand("Delete", typeof(DataCommands), inputs);
        }
    }
}
```

Рисунок 1 – Код класса *DataCommands*

В классе *DataCommands* объявлены два свойства: *Delete* и *Edit* типа *RoutedCommand*. Класс *RoutedCommand* определяет команду, реализующую *ICommand*. В конструкторе данного класса определяется объект *input* типа *InputGestureCollection*. Класс *InputGestureCollection* представляет упорядоченную коллекцию объектов *InputGesture*, которые позволяют с помощью класса *KeyGesture* задать комбинацию клавиш для вызова команды.

Для использования страницей *PageEmployee* пользовательских команд в XAML-документе необходимо добавить пространство имен, где расположен класс *DataCommands*. Данному пространству имен присвоим ссылку *command*.

xmlns:command="clr-namespace:Wpf_Erina.Commands"

Теперь для страницы *PageEmployee* сформируем коллекцию объектов *CommandBinding*, которая осуществляет привязку команд для данного элемента и объявляет связь между командой, ее событиями и обработчиками.Для функций *Отменить*, *Создать*, *Сохранить* и *Найти*, основанных на ***стандартных командах из библиотеки WPF***, привязка выглядит так, как показано на рисунке 2.

```
<Page.CommandBindings>
    <CommandBinding Command="Undo" Executed="UndoCommandBinding_Executed"
        CanExecute="UndoCommandBinding_CanExecute" />
</Page.CommandBindings>
```

Рисунок 2 – Привязка – связь между командой, ее событиями и обработчиками.

Для класса *CommandBinding* свойство *Command* определяет ссылку на соответствующую команду, а свойства *Executed* и *CanExecute* задают обработчики событий при выполнении команды.

Добавьте привязки для следующих команд: Отменить – Undo, Создать – New, Сохранить – Save, Найти – Find.

Для функций *Редактировать* и *Удалить*, использующих пользовательские команды, необходимо указать, что команда берется из пространства имен Wpf_Erina.Commands (рисунок 3).

```
<CommandBinding Command="{x:Static command:DataCommands.Edit}"
                Executed="EditCommandBinding_Executed" CanExecute="EditCommandBinding_CanExecute"/>
```

Рисунок 3 – Привязки для функций *Редактировать* и *Удалить*,

Добавьте привязки для команд Редактировать – Edit и Удалить – Delete.

Итак, на странице приложения используются следующие команды: *Отменить*, *Создать*, *Редактировать*, *Поиск*, *Сохранить* и *Удалить*. Команды могут быть доступны или недоступны пользователю при работе приложения. Это проверяет метод *CanExecute* при генерации события *CanExecuteChanged*, которое вызывается при изменении состояния команды. Доступность команд определяется состоянием, в котором находится приложение. В тоже время выполнение какой-либо команды переводит, как правило, приложение в какое-либо другое состояние. Для проектируемого приложения можно определить следующие состояния:

- первоначальная загрузка страницы (1);
- просмотр данных по всем сотрудникам (2);
- редактирование данных по отдельному сотруднику (3);
- создание новой записи по сотруднику в базе данных (4).

На рисунке 4 приведена диаграмма состояний приложения, на которой кружками обозначены состояния, а дуги соответствуют переходам при выполнении определенной команды.

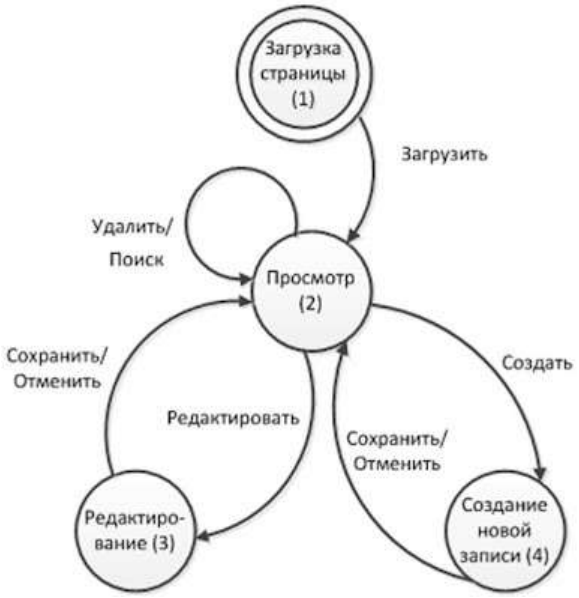

Рисунок 4 – Диаграмма состояний приложения

На основе диаграммы состояний построим таблицу доступности команд в различных состояниях (таблица 1).

Таблица 1 – Доступность команд в различных состояниях приложения

Состояние	Доступность команд					
	Сохранить Save	Отменить Undo	Создать New	Поиск Find	Редактировать Edit	Удалить Delete
1	false	false	true	true	true	true
2	false	false	true	true	true	true
3	true	true	false	false	false	false
4	true	true	false	false	false	false

Из таблицы 2 видно, что в приложении режим доступности команд в состояниях 1 и 2 противоположно режиму доступности команд в состояниях 3 и 4. Фактически для приложения имеются два режима (один объединяет состояния 1 «Первоначальная загрузка» и 2 «Просмотр данных по всем сотрудникам», а второй – состояния 3 «Редактирование данных по одному сотруднику» и 4 «Создание новой записи по сотруднику»), управлять которыми можно с помощью логической переменной.

В код программы класса *PageEmployee* введем логическое поле *isDirty* для управления доступностью команд.

```
private bool isDirty = true;
```

В код класса добавим обработчики команд, определяющие бизнес-логику приложения (реализация методов *Executed*). На данном этапе проектирования системы обработчики будут содержать только вывод сообщений о вызове команды и изменение поля *isDirty*. Код обработчика команды *Отменить* приведен ниже.

```
private void UndoCommandBinding_Executed(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Отмена");
    isDirty = true;
}
```

Добавьте ВСЕ обработчики команд Executed

В дальнейшем в обработчики добавим код для обеспечения требуемой функциональности.

В коде класса остается добавить обработчики, которые управляют доступностью команд (реализация метода *CanExecute*). Так как при анализе Таблица 2 было выявлено, что для приложения различимо только два состояния доступности команд, то и обработчиков тоже достаточно иметь в программе два.

```
private void EditCommandBinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = isDirty;
}

private void SaveCommandBinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = !isDirty;
}
```

Исправьте в XAML-документе в привязке команд к обработчикам событий, у всех остальных команд привязку к одному из этих обработчиков. При этом учтите, что команды первой группы устанавливают значение поля *isDirty* в Истину, а команды второй группы устанавливают ее в Ложь.

Теперь необходимо модифицировать XAML-документ в части задания свойства *Command* при описании пунктов меню и панели инструментов для привязки команд.

При описании меню (*Menu*) XAML-документ модифицирован следующим образом.

```
<Menu>
    <MenuItem Header="Действие">
        <MenuItem Header="Отменить" Command="Undo"></MenuItem>
        <Separator></Separator>
        <MenuItem Header="Создать" Command="New"></MenuItem>
        <MenuItem Header="Редактировать" Command="command:DataCommands.Edit"></MenuItem>
```

Соответствующие изменения XAML-документа необходимо провести и для панели инструментов *ToolBar*.

```
<ToolBar Name="ToolBar1" Margin="3">
    <Button Name="Undo" Command="Undo" ToolTip="Отменить редактирование/создание" Margin="5,2,2,2">
        <Image Source="Images/Undo.png"/>
    </Button>
    <Button Name="Add" Command="New" ToolTip="Добавить нового сотрудника" Margin="2,2,2,2">
        <Image Source="Images/Add.png"/>
    </Button>
    <Button Name="Edit" Command="command:DataCommands.Edit" ToolTip="Редактировать сотрудника" Margin="2,2,2,2">
        <Image Source="Images/Edit.png"/>
    </Button>
```

Внесите изменения для всех пунктов меню и всех кнопок панели инструментов.

При выполнении приложения различные состояния доступности пунктов меню приведены на Рисунок 20 и Рисунок 21.

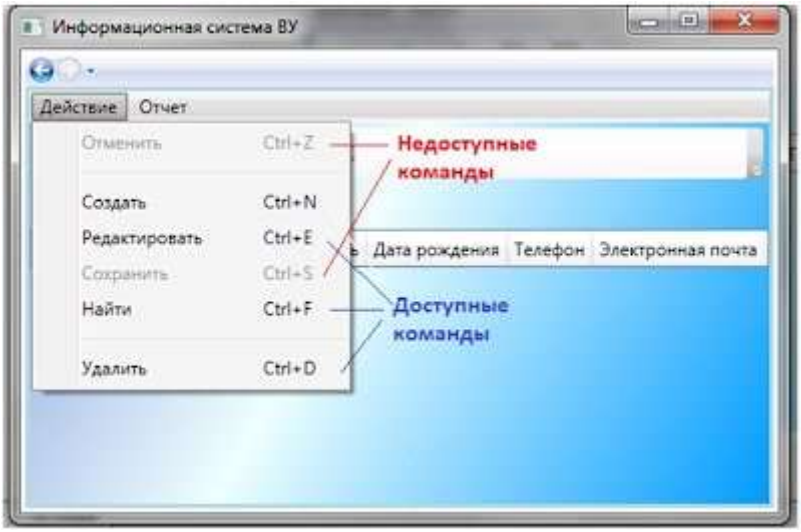

Рисунок 20. Состояние доступности пунктов меню после загрузки приложения

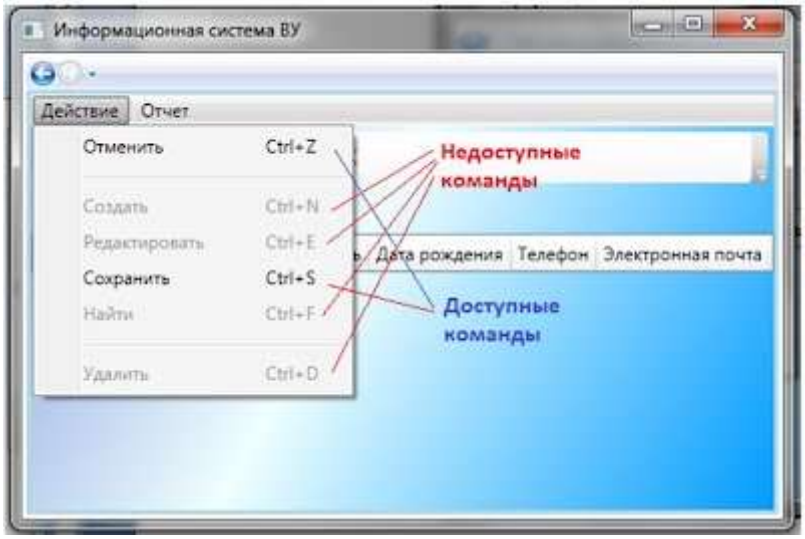

Рисунок 21. Состояние доступности пунктов меню после редактирования данных

При выборе какого-либо пункта меню, например пункта Создать, система выдает сообщение (Рисунок 22).

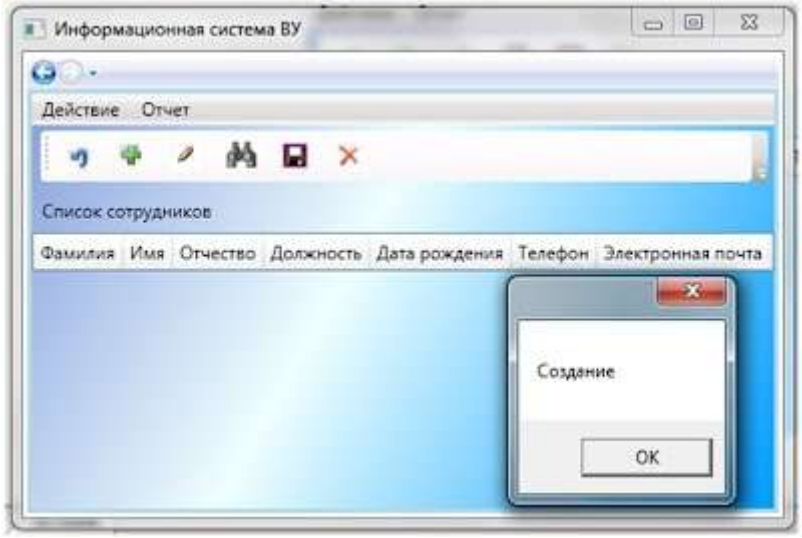

Рисунок 22. Выбор пункта меню *Создать*

Следующим этапом разработки приложения является создание источника данных, который обеспечит взаимодействие приложения с базой данных.

Ключевые термины

Приложение WPF, пространство имен, атрибут, частичный класс, простое свойство, сложное свойство, фрейм, гиперссылка, меню, панель инструментов, команда, привязка команд.

MainWindow, xmlns, partial, InitializeComponent, NavigationWindow, Frame, FrameworkElement, Page, Hyperlink, NavigateUri, Menu, MenuItem, DataGrid, DataGridTextColomn, DataGridCheckBoxColomn, DataGridComboBoxColomn, DataGridHyperlinkColomn, DataGridTemplateColumn, ICommand, Execute, CanExecute, CanExecuteChanged, CommandBinding.

Взаимодействие приложения WPF с базой данных. Создание EDM-модели

Для создания EDM-модели данных добавим в проект новый элемент – ***Модель ADO.NET EDM*** (Entity Data Model), присвоив файлу модели имя ***TitleEmployee.edmx*** (Рисунок 36).

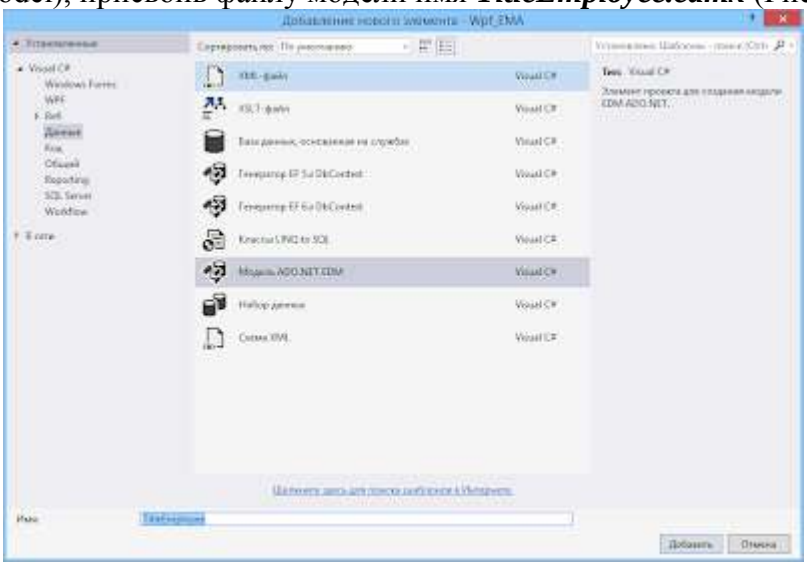

Рисунок 36. Добавление в проект модели EDM

В мастере создания EDM-модели выберем опцию "**Конструктор EF из базы данных**" (**Generate from database**) (Рисунок 37). Для создания соединения с базой данных в окне "Выбор подключения к данным" укажем соединение с базой данных (в рассматриваемом примере – созданная ранее база данных ***TitleEmployee.mdf***) и зададим имя модели данных - ***TitleEmployeeEntities*** (точнее, оставим название по умолчанию) (Рисунок 38).

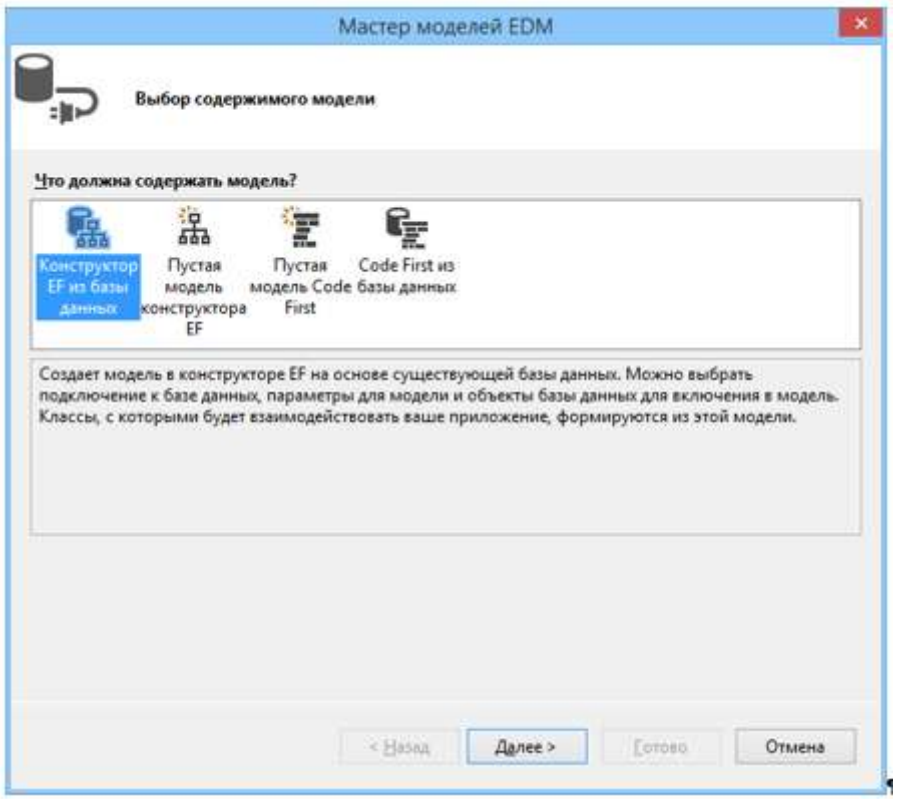

Рисунок 37. Создание модели EDM из базы данных

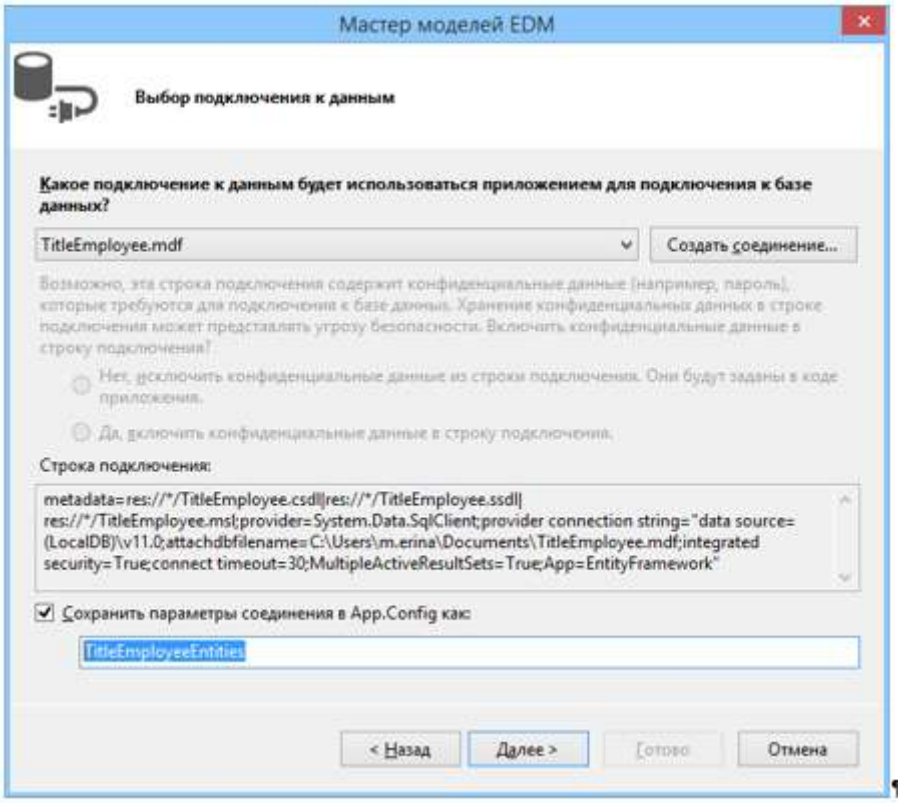

Рисунок 38. Создание соединения с базой данных

Согласитесь с копированием локальной базы данных в выходной каталог приложения. Это означает, что при каждой компиляции проекта в выходной каталог будет копироваться база данных. Все изменения, вносимые в базу данных во время работы с приложением, будут совершаться с этой копией базы данных. Таким образом, ***все данные, которые будут вноситься в базу, при каждой компиляции проекта будут затираться снова пустой копией базы***. Это следует учитывать. При необходимости сохранить данные после работы приложения, можно скопировать базу данных с данными из выходного каталога в каталог проекта.

На следующем шаге работы мастера выберите версию **Entity FrameWork 6.x**

В окне "**Выбор объектов базы данных**" отметим необходимые для приложения таблицы (в нашем случае это таблицы ***Employee*** и ***Title***) и флаг формирования имен объектов во множественном или единственном числе (Рисунок 39).

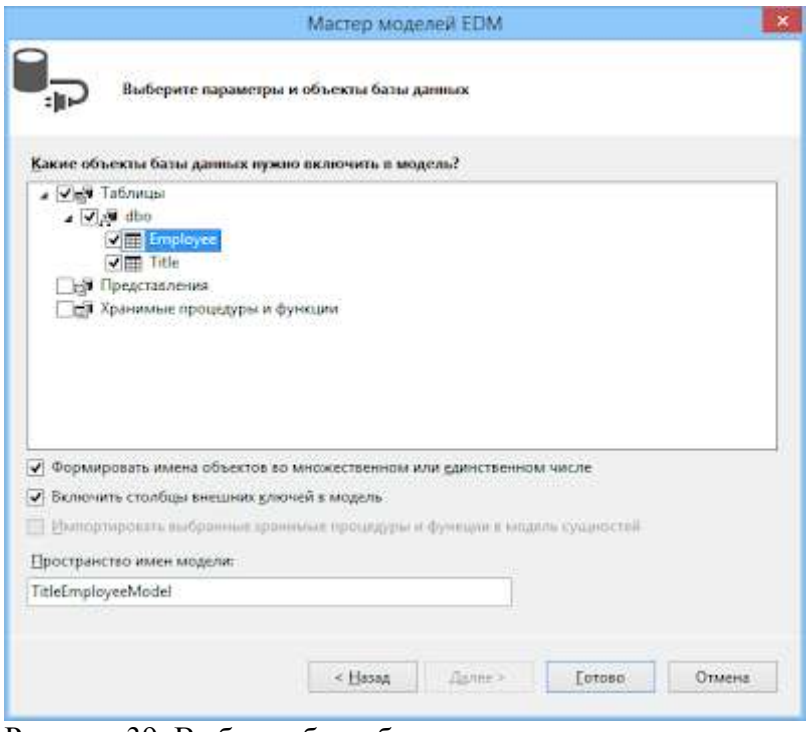

Рисунок 39. Выбор таблиц базы данных

При завершении работы мастера создания EDM-модели в проект будет добавлен файл ***TitleEmployee.edmx*** (Рисунок 40), ссылки на необходимые библиотеки и конфигурационный файл.

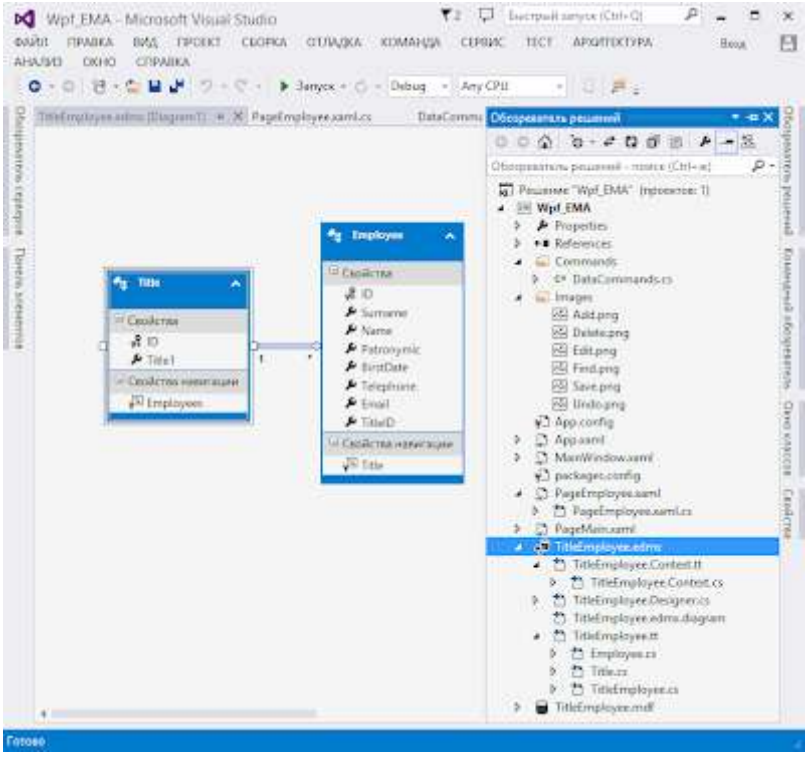

Рисунок 40. Проект с моделью EDM

Автоматически сгенерированный класс ***TitleEmployeeEntities***, который наследуется от класса ***DbContext***, представляет сущности базы данных ***TitlePerson***, содержит свойства, моделирующие таблицы ***Employee*** и ***Title***, связи между таблицами.

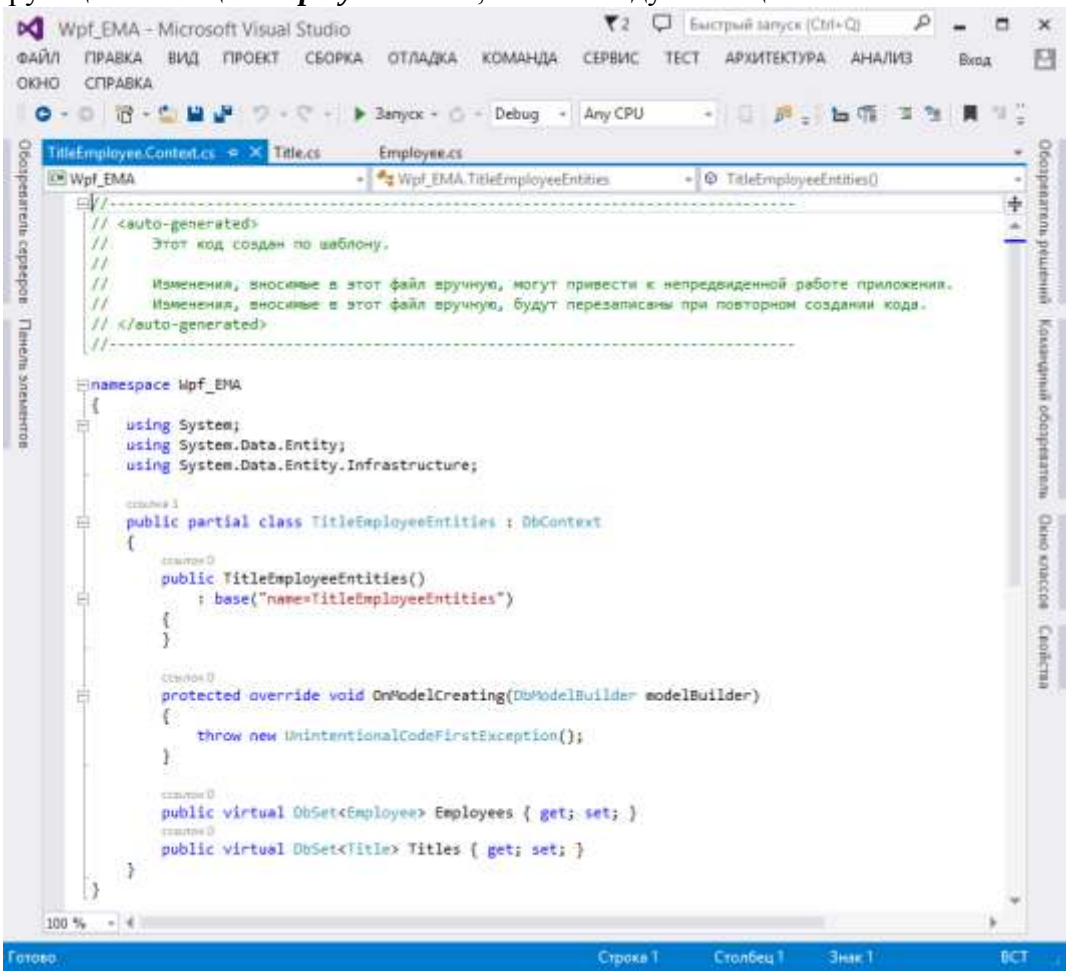

Рисунок 41. Класс ***TitleEmployeeEntities***

Также были автоматически сгенерированы два класса, отражающие сущности ***Employee*** и ***Title***.

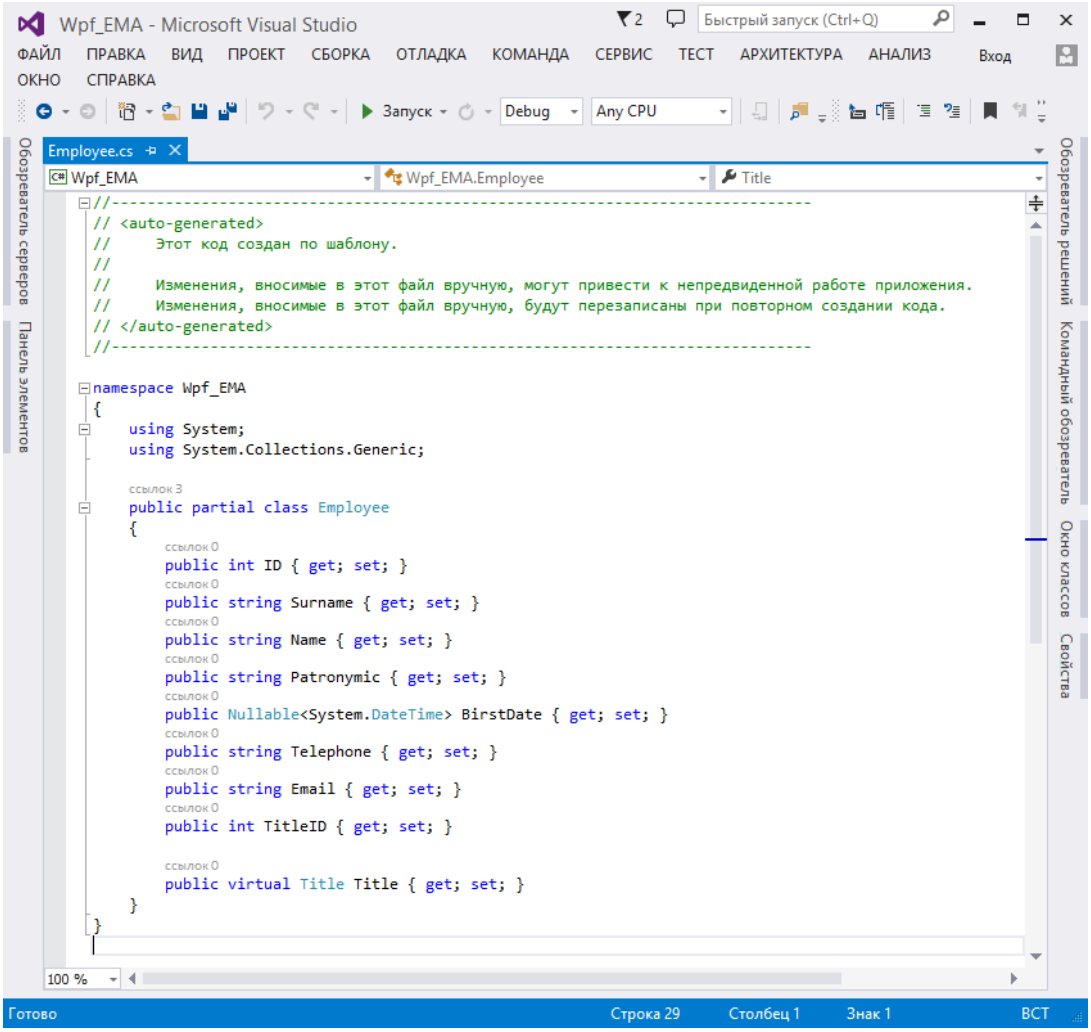

Рисунок 42. Класс, отражающий сущность *Employee*

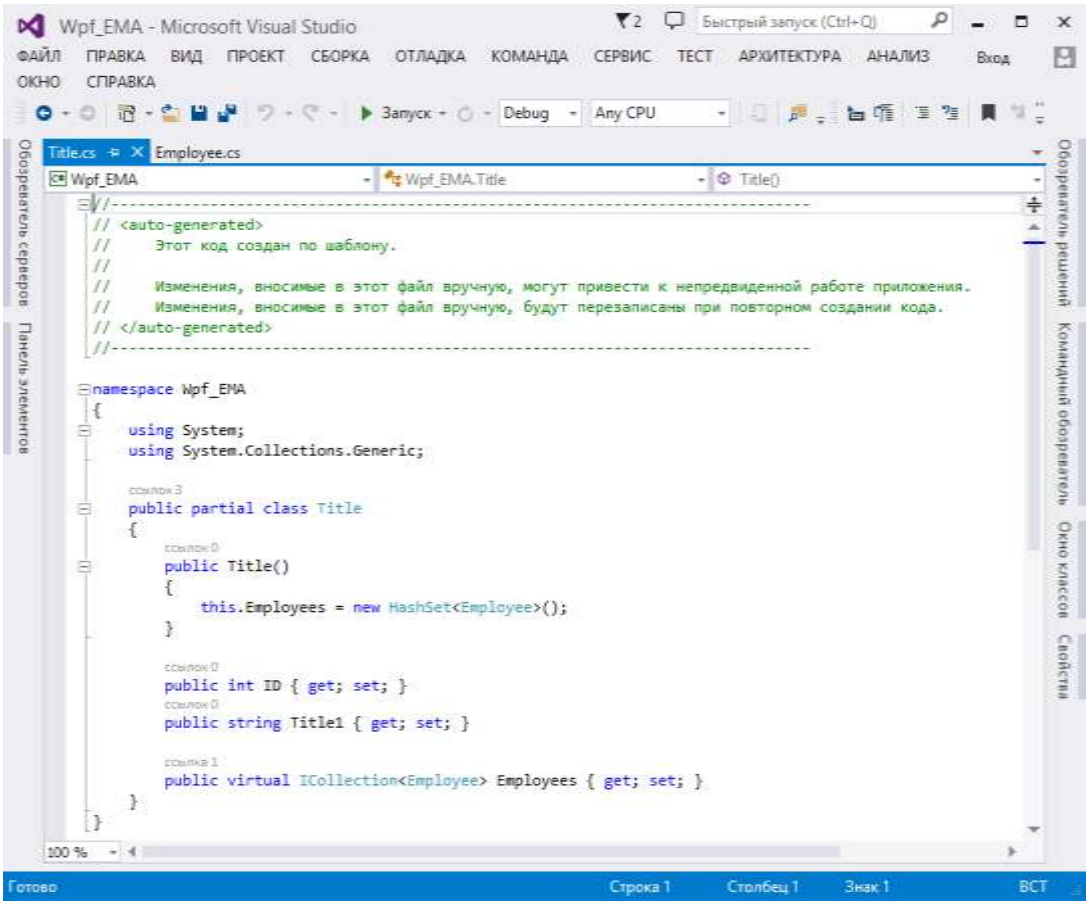

Рисунок 43. Класс, отражающий сущность *Title*

При создании модели данных в проекте автоматически был сгенерирован конфигурационный файл App.Config, который содержит строку соединения с базой данных (выделена жирным)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
      <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->

<section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <connectionStrings>

<add name="TitleEmployeeEntities" connectionString="metadata=res://*/TitleEmployee.csdl|res://*/TitleEmployee.ssdl|res://*/TitleEmployee.msl;provider=System.Data.SqlClient;provider connection string=&amp;quot;data source=(LocalDB)\v11.0;attachdbfilename=|DataDirectory|\TitleEmployee.mdf;integrated security=True;connect timeout=30;MultipleActiveResultSets=True;App=EntityFramework&amp;quot;" providerName="System.Data.EntityClient" />
  </connectionStrings>
  <entityFramework>

<defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">
```

```
        <parameters>
          <parameter value="v11.0" />
        </parameters>
      </defaultConnectionFactory>
      <providers>

<provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
      </providers>
    </entityFramework>
  </configuration>
```

Привязка элементов управления к источнику данных

Для взаимодействия приложения с базой данных в коде класса *PageEmployee* необходимо объявить статическое свойство контекста данных *DataEntitiesEmployee* сформированной EDM-модели. На данном этапе проектирования приложения это свойство можно объявлять только общедоступным, то есть *public*, но в дальнейшем нам потребуется использовать это свойство в других классах без создания экземпляра класса *PageEmployee*, что возможно только в случае, если это свойство будет статическим.

public static TitleEmployeeEntities DataEntitiesEmployee { get; set; }

Также необходимо объявить коллекцию *ListEmployee* для работы приложения с коллекцией объектов *Employee*.

ObservableCollection<Employee> ListEmployee;

И инициализировать их в конструкторе

```
public PageEmployee()
{
    DataEntitiesEmployee = new TitleEmployeeEntities();
    InitializeComponent();
    ListEmployee = new ObservableCollection<Employee>();
}
```

Формирование данных для приложения, которые должны предоставляться из базы данных, будем создавать при загрузке страницы *PageEmployee*. Для этого в XAML-документ *Page* добавим свойство *Loaded*.

Loaded="Page_Loaded"

В код класса *PageEmployee* включаем обработчик *Page_Loaded*.

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
    var employees = DataEntitiesEmployee.Employees;
    var queryEmployee = from employee in employees
                        orderby employee.Surname
                        select employee;
    foreach (Employee emp in queryEmployee)
    {
        ListEmployee.Add(emp);
    }
    DataGridEmployee.ItemsSource = ListEmployee;
}
```

В поле *employees* хранится результат выполнения запроса, возвращающего коллекцию типизированных сущностей с любым количеством элементов. Запрос сформируем с помощью технологии LINQ.

```
var queryEmployee = from employee in employees
                    orderby employee.Surname
                    select employee;
```

Результаты запроса отсортируем по фамилии сотрудника (*orderby employee.Surname*). Далее формируем коллекцию *ListEmployee* сотрудников и источник данных для сетки *DataGridEmployee*.

```
foreach (Employee emp in queryEmployee)
{
    ListEmployee.Add(emp);
}
DataGridEmployee.ItemsSource = ListEmployee;
```

В результате проектирования в приложении сформирована коллекция с данными из таблицы *Employee* базы данных *TitleEmployee*. Следующей задачей является настройка сетки *DataGridEmployee* для корректного отображения данных.

Вначале модифицируем общее описание *DataGrid*.

```
<DataGrid Name="DataGridEmployee"
          ItemsSource="{Binding}"
          AutoGenerateColumns="False"
          HorizontalAlignment="Left"
          MaxWidth="1000" MaxHeight="295"
          RowBackground="#FFE6D3EF"
          AlternatingRowBackground="#FC96CFD4"
          BorderBrush="#FF1F33EB"
          BorderThickness="3"
          RowHeight="25"
          Cursor="Hand"
          CanUserAddRows="False"
          CanUserDeleteRows="False" Width="985">
```

Определим привязку для источника данных.

ItemsSource="{Binding}"

Отменим автоматическую генерацию столбцов.

AutoGenerateColumns="False"

Установим привязку к левому краю страницы.

HorizontalAlignment="Left"

Определим максимальные размеры сетки.

MaxWidth="1000" MaxHeight="295"

Установим основной и альтернативный цвета заливки сетки.

RowBackground="#FFE6D3EF"

AlternatingRowBackground="#FC96CFD4"

Определим цвет заливки и толщину линии для рамки сетки.

BorderBrush="#FF1F33EB"

BorderThickness="3"

Определим высоту строк сетки.

RowHeight="25"

Проверьте, что у вас НЕ установлен параметр (в приложении будет реализовано редактирование данных в таблице):

IsReadOnly="True"

Переопределим форму курсора при наведении указателя мыши на таблицу *DataGridEmployee*.

Cursor="Hand"

1) Привязка текстового поля

Для каждого текстового столбца зададим свойство привязки *Binding*. В расширении разметки привязки данного свойства определим свойство класса *Employee*, к которому привязывается колонка. Для колонки *Фамилия* это свойство *Surname*. Далее указываем режим двусторонней привязки (*Mode = TwoWay*), который определяет синхронное обновление как источника, так и приемника привязки. Способ обновления источника привязки задается свойством *UpdateSourceTrigger*, которое принимает значение *PropertyChanged*, что соответствует немедленному обновлению источника привязки каждый раз при изменении свойства цели привязки.

```
<DataGridTextColumn Header="Фамилия"
    Binding="{Binding Surname, Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged}"/>
```

Внесите изменения для всех ТЕКСТОВЫХ столбцов DataGrid.

После запуска приложения страница *PageEmployee* имеет вид, приведенный на рисунке 44. (Для этого, конечно, таблица базы данных должна быть заполнена!)

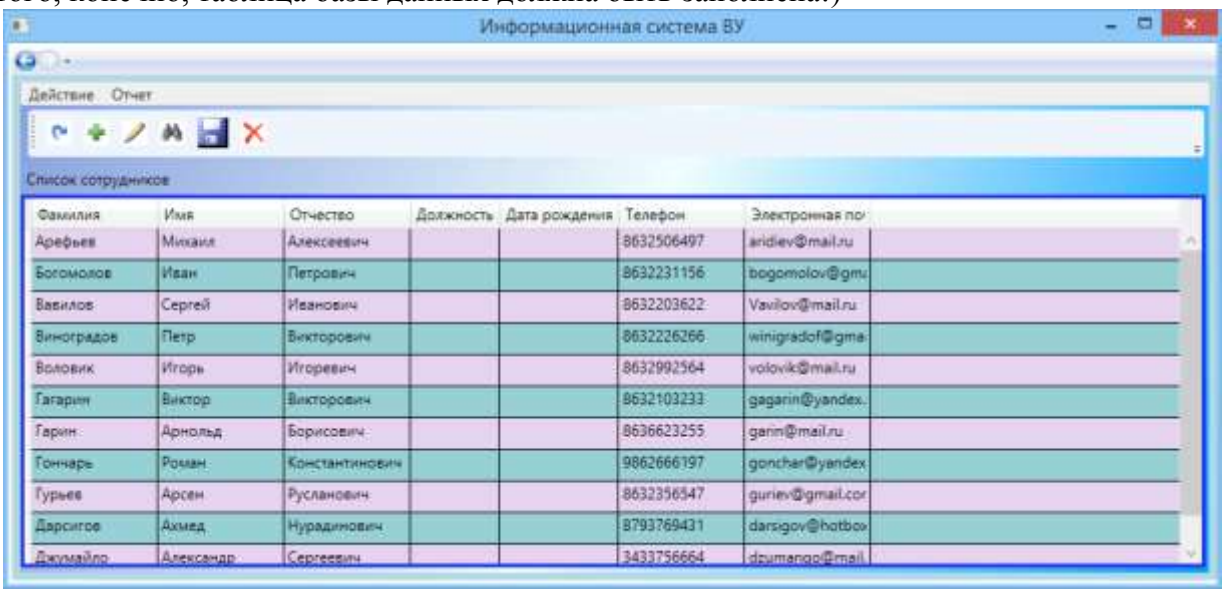

Рисунок 44. Страница *PageEmployee* с привязанными в таблице текстовыми полями

Заполните таблицы БД корректными значениями. Сделать это можно, выбрав пункт меню «Показать таблицу данных» в окне Обозреватель серверов. Заполнять начните с таблицы Title!

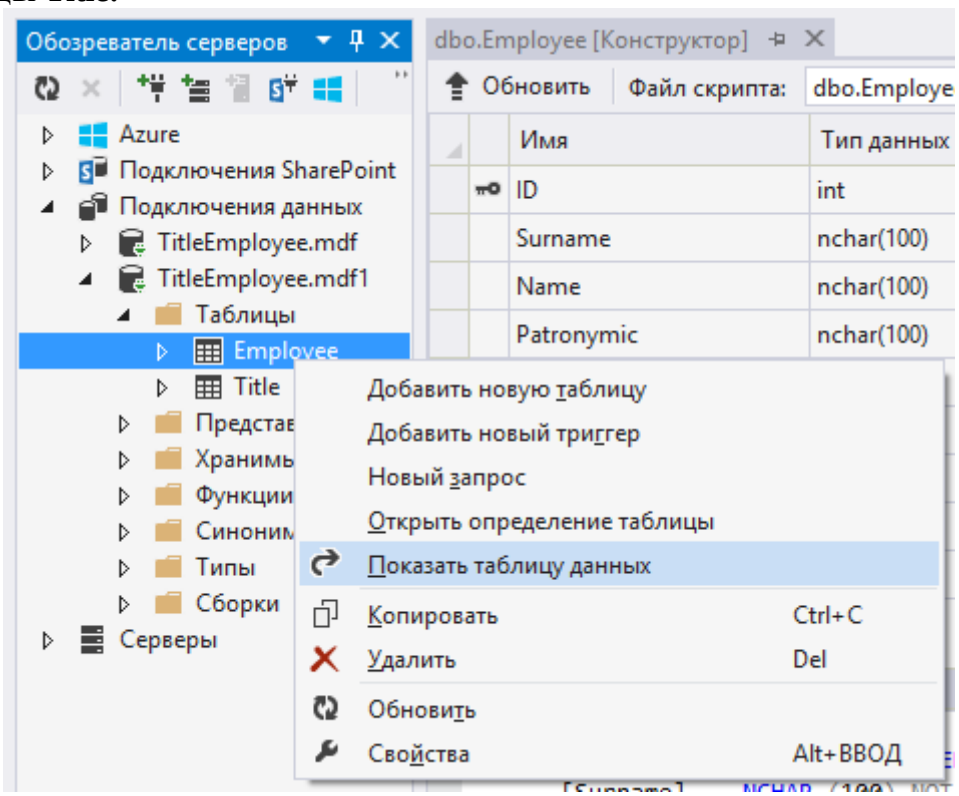

Рисунок 45. Добавление данных в таблицу БД

dbo.Title [Данные] | dbo.Employee [Данные]

Максимальное количество строк: 1000

	ID	Surname	Name	Patronymic	BirstDate	Telephone	Email	TitleID
	16	Арефьев	Михаил	Алексеевич	26.01.1985	8632506497	aridiev@mail.ru	1
	17	Богомолов	Иван	Петрович	24.02.1994	8632231156	bogomolov@gmail.com	2
	18	Вавилов	Сергей	Иванович	23.03.1996	8632203622	Vavilov@mail.ru	6
	19	Виноградов	Петр	Викторович	03.03.1990	8632226266	winigradof@gmail.com	4
	20	Воловик	Игорь	Игоревич	25.11.1983	8632992564	volovik@mail.ru	5
	21	Гагарин	Виктор	Викторович	30.01.1986	8632103233	gagarin@yandex.ru	2
	22	Гарин	Арнольд	Борисович	03.04.1988	8636623255	garin@mail.ru	5
	23	Гончарь	Роман	Константинович	13.06.1991	9862666197	gonchar@yandex.ru	2
	24	Гурьев	Арсен	Русланович	18.05.1990	8632356547	guriev@gmail.com	3
	25	Дарсигов	Ахмед	Нурадинович	22.09.2000	8793769431	darsigov@hotbox.ru	1
▶	26	Джумайло	Александр	Сергеевич	12.08.1998	3433756664	dzumango@mail.ru	2
*	N...	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 46. Таблица *Employee* в процессе заполнения

2) Привязка даты

При привязке даты к колонке "Дата рождения" ставится задача для невыделенной ячейки представлять дату в виде цифр дня, месяца и года, разделенных точками, а для выделенной ячейки – использовать класс для выбора даты.

Решение данной задачи можно осуществить с помощью разработки двух шаблонов данных *DataTemplate*.

В первом шаблоне, для которого задан ключ *DateTemplate*, используется элемент управления *TextBlock*, свойство *Text* которого привязывается к атрибуту *BirstDate* таблицы *Employee*. Данные в привязке форматируются свойством *StringFormat* и строковые данные выравниваются по центру. Этот шаблон используется для визуализации данных в режиме их просмотра.

```
<DataTemplate x:Key="DateTemplate" >
    <TextBlock Text="{Binding BirstDate, StringFormat={}{0:dd\.}{0:MM\.}{0:yyyy}}"
               VerticalAlignment="Center" HorizontalAlignment="Center" />
</DataTemplate>
```

Второй шаблон с ключом *EditingDateTemplate* использует класс *DatePicker*. Этот шаблон используется в режиме редактирования данных.

```
<DataTemplate x:Key="EditingDateTemplate">
    <DatePicker SelectedDate="{Binding BirstDate, Mode=TwoWay, UpdateSourceTrigger=PropertyChan
</DataTemplate>
```

Разработанные шаблоны необходимо добавить к ресурсам в XAML-документ страницы *PageEmployee*:

<Page.Resources>
ЗДЕСЬ ВСТАВЛЕНЫ ОПРЕДЕЛЕННЫЕ ВЫШЕ ШАБЛОНЫ
</Page.Resources>

Тогда в самой таблице столбец с датой будет представлен следующим кодом:

```
<DataGridTemplateColumn Header="Дата рождения"
                        CellTemplate="{StaticResource DateTemplate}"
                        CellEditingTemplate="{StaticResource EditingDateTemplate}"/>
```

Невыделенную ячейку колонки (*CellTemplate*) свяжем с ресурсом *DateTemplate*, а редактируемую ячейку (*CellEditingTemplate*) – с ресурсом *EditingDateTemplate*.

При запуске приложения страница *PageEmployee* принимает вид, показанный на рисунке 47.

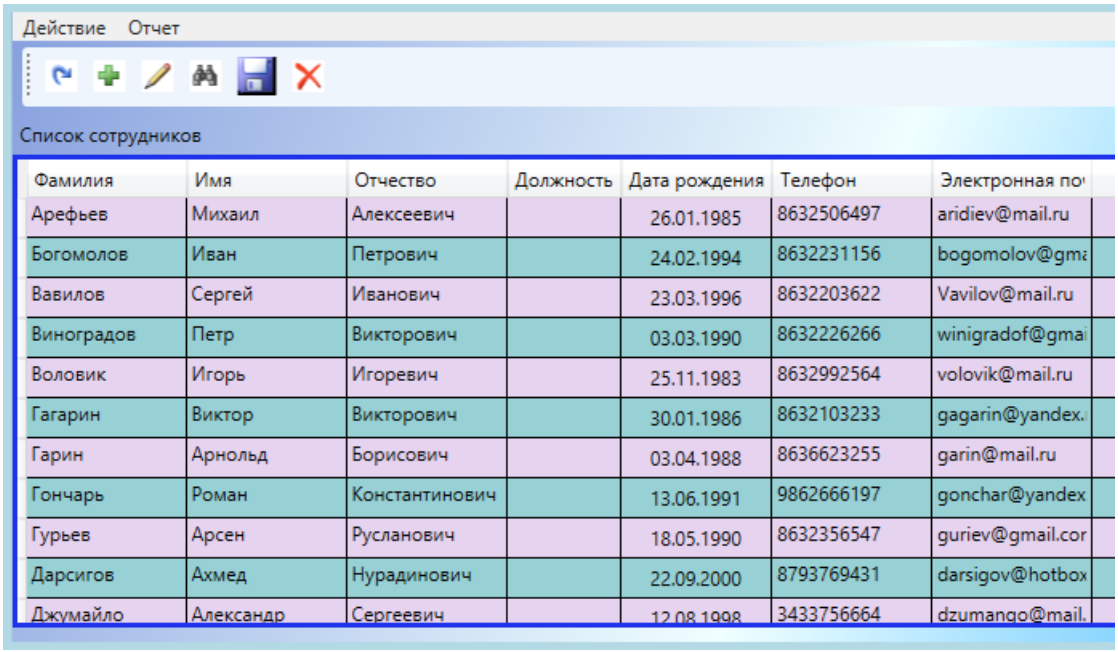

Рисунок 47. Страница *PageEmployee* с привязанным столбцом "Дата рождения"

Ячейка столбца для отображения даты типа *DataGridTemplateColumn* имеет три представления (рисунок 48). Если ячейка не выделена, то представление обычное текстовое (рисунок 48, а). При выделении ячейки прорисовывается свернутое содержание класса *DatePicker* (рисунок 48, б). При щелчке на ячейке появляется календарь (рисунок 48, в).

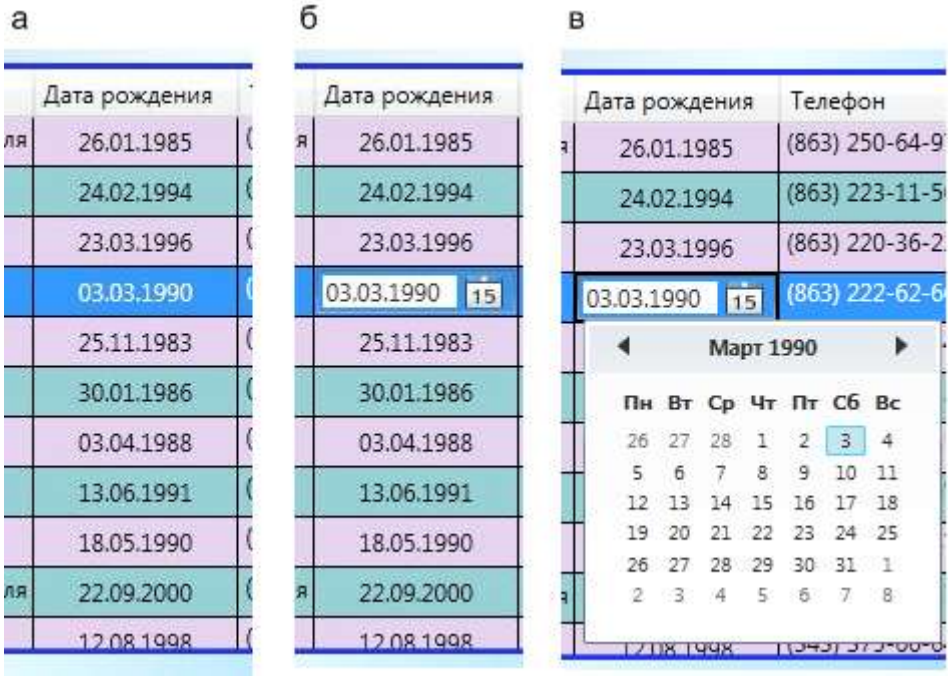

Рисунок 48. Отображение даты в сетке DataGrid

3) Привязка выпадающего списка с полем из другой таблицы

Привязка данных к колонке типа *DataGridComboBoxColumn* требует определенной подготовительной работы. В таблице *Employee* модели данных хранится не текстовое значение должности сотрудника, а внешний ключ для таблицы *Title*, где находится наименование должности. В EDM-модели можно получить значение атрибута из связанных таблиц. В нашем случае для таблицы *Employee* имеется атрибут *Title.Title1*, который посредством имеющихся связей в модели обеспечивает доступ к таблице *Title* из таблицы *Employee*. Это можно использовать для просмотра данных по сотруднику, но при редактировании данных необходимо обеспечить выбор должности для сотрудника из списка должностей, которые определены в таблице *Title*. И это невозможно сделать при привязке колонки *Должность* к полю *Title.Title1*.

Для обеспечения возможности работы с коллекцией в таблицы *Title* в приложении добавим в проект папку *Model* и в ней создадим класс *ListTitle*.

```
class ListTitle : ObservableCollection<Title>
{
    ссылок 0
    public ListTitle()
    {
        DbSet<Title> titles = PageEmployee.DataEntitiesEmployee.Titles;
        var queryTitle = from title in titles select title;
        foreach (Title titl in queryTitle)
        {
            this.Add(titl);
        }
    }
}
```

Класс *ListTitle* наследуется от класса обобщенной коллекции *ObservableCollection<T>* и его назначение создавать коллекцию объектов *Title*. Поле *titles* является запросом типа *DbSet<Title>*. Данному полю присваивается свойство *Titles* контекста данных *DataEntitiesEmployee*, который определен в классе *PageEmployee*. В классе *ListTitle* не рекомендуется создавать собственный контекст данных, а лучше использовать контекст, созданный в классе *PageEmployee*. Именно поэтому свойства *DataEntitiesEmployee* объявлено в классе *PageEmployee* статическим.

Запрос LINQ получает данные из базы *TitlePersonal* в поле *queryTitle* и затем в цикле *foreach* формируется коллекция класса *ListTitle*.

Для использования класса *ListTitle* в XAML-документе класса *PageEmployee* необходимо объявить данный класс как ресурс. При этом нужно подключить пространство имен *WpfApplProject.Model*.

```
xmlns:core ="clr-namespace:Wpf_EMA.Model"
```

Затем определим ресурс страницы с ключом *listTitle*.

```
<Page.Resources>
  <core:ListTitle x:Key="listTitle" />
</Page.Resources>
```

Далее модифицируем XAML описание для типа *DataGridComboBoxColumn*.

```
<DataGridComboBoxColumn Header="Должность"
                        ItemsSource="{Binding Source={StaticResource listTitle}}"
                        DisplayMemberPath="Title1"
                        SelectedValueBinding="{Binding Path=TitleID, Mode=TwoWay, UpdateSourceT
                        SelectedValuePath="ID"/>
```

Источник выпадающего списка задается как статический ресурс *{StaticResource listTitle}*. Выводимое в ячейки колонки поле должно соответствовать полю *Title1* таблицы *Title* EDM-модели (*DisplayMemberPath="Title1"*). Выбираемый в списке параметр (*SelectedValue*) должен быть привязан к полю *TitleID* таблицы *Employee*.

Выбор в свойства *SelectedValue* производится по пути определенному свойством *SelectedValuePath* (*SelectedValuePath="ID"*).

При запуске приложения страница PageEmployee принимает вид, показанный на рисунке 49.

Рисунок 49. Страница *PageEmployee* с привязанным столбцом "Должность"

Ячейка столбца типа *DataGridComboBoxColumn* имеет три представления (Рисунок 50). Если ячейка не выделена, то представление обычное текстовое (Рисунок 50, а). При выделении ячейки прорисовывается представление *ComboBox* (Рисунок 50, б). При раскрытии списка появляется список элемента *ComboBox* (Рисунок 50, в).

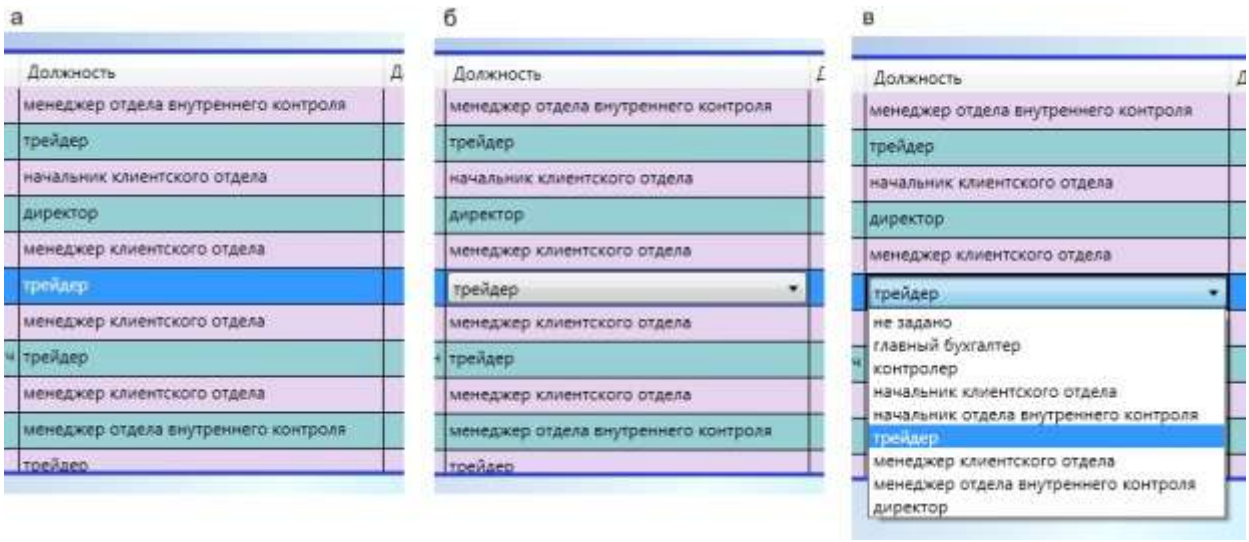

Рисунок 50. Отображение *ComboBox* в сетке *DataGrid*

Форма представления результата:

13. Цель работы
14. Введение
15. Программно-аппаратные средства, используемые при выполнении работы
16. Описание работы
17. Заключение (выводы)
18. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.3 Инструментальные средства этапа отладки и тестирования программ

Практическая работа № 9
Модульное тестирование

Цель: Сформировать понятие и основные характеристики проекта. Выяснить причины необходимости ведения проектной деятельности и управления проектом. Определить отличия проекта и операционной деятельности.

Выполнив работу, Вы будете:

уметь:

– У9 выполнять отладку и тестирование информационных систем и программных средств.

Материальное обеспечение:

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio.

Задание:

Изучите основные принципы разработки тестовой модели программного обеспечения и постройте тестовую модель для разработанного программного продукта (лабораторное задание № 10); сформируйте метрики для оценки качества программного продукта; составьте тест-план; сформируйте тестовую документацию.

Порядок выполнения работы:

1. Изучить предлагаемый теоретический материал по теме.
2. Построить тестовую модель для разработанного программного продукта (лабораторное задание № 10).
3. Сформировать метрику для оценки качества разработанного программного продукта.
4. Составить тест-план для разработанного программного продукта.
5. Сформировать тестовую документацию для разработанного программного продукта.
6. Разработать диаграмму вариантов использования и диаграмму состояний и последовательности с использованием языка UML для заданной предметной области в соответствии с требованиями.
7. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения

Для разработки тестовых сценариев и выполнения тестов используются системы управления тестированием, существенно повышающие производительность тест-дизайнеров и тестировщиков, а также обеспечивающие видимость уровня качества приложений среди всех участников проекта.

Тестовые сценарии неразрывно связаны с требованиями, изменения в которых должны своевременно отражаться в тестовой документации, что позволяет сделать система управления жизненным циклом разработки приложений, при помощи механизма трассировок.

При выполнении теста тестировщик отмечает результат прохождения одного шага или всего тестового сценария, прикрепляет обнаруженные ошибки и другую вспомогательную информацию: скриншоты, дампы, логи и т.п.

Тестовые сценарии удобно объединять в тест-планы по назначению:

– тестирование релиза, то есть очередной версии продукта;
– тестирование развертывания;
– тестирование удобства использования;
– конфигурационное тестирование;
– тестирование безопасности и т.п.

Зачастую ручное тестирование превращается в рутину и занимает значительное время, что отрицательно сказывается на скорости выпуска релизов. Автоматизация тестирования позволяет:

– высвободить ресурсы для проведения более сложных видов тестирования;

- снизить количество дефектов, доходящих до стадии контроля качества;
- ускорить выпуск релизов.

Сведение результатов автоматических и ручных тестов в системе управления качеством, позволяет всем участникам проекта видеть уровень качества очередного релиза, контролировать его изменение и опираться на эту информацию при планировании своей работы.

Получение результатов тестирования и их анализ

Получение результатов тестирования напрямую зависит от средств тестирования. В моем случае это был встроенный в 1С механизм отладки и система контроля ошибок.

Перед получением результата программа проходит несколько уровней:

1. Тестирование компонентов — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто тестирование компонентов осуществляется разработчиками программного обеспечения.
2. Интеграционное тестирование — тестируются интерфейсы между компонентами, подсистемами или системами. При наличии резерва времени на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.
3. Системное тестирование — тестируется интегрированная система на её соответствие требованиям.
4. Альфа-тестирование — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приёмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программа.
5. Бета-тестирование — в некоторых случаях выполняется распространение предварительной версии (в случае проприетарного программного обеспечения иногда с ограничениями по функциональности или времени работы) для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Часто для свободного и открытого программного обеспечения стадия альфа-тестирования характеризует функциональное наполнение кода, а бета-тестирования — стадию исправления ошибок. При этом как правило на каждом этапе разработки промежуточные результаты работы доступны конечным пользователям.

Метрики для оценки качества программного продукта

В настоящее время в программной инженерии еще не сформировалась окончательно система метрик. Действуют разные подходы к определению их набора и методов измерения [10.11-10.13].

Система измерения включает метрики и модели измерений, которые используются для количественной оценки качества ПО.

При определении требований к ПО задаются соответствующие им внешние характеристики и их атрибуты (подхарактеристики), определяющие разные стороны управления продуктом в заданной среде. Для набора характеристик качества ПО, приведенных в требованиях, определяются соответствующие метрики, модели их оценки и диапазон значений мер для измерения отдельных атрибутов качества.

Согласно стандарту [1.16] метрики определяются по модели измерения атрибутов ПО на всех этапах ЖЦ (промежуточная, внутренняя метрика) и особенно на этапе тестирования или функционирования (внешние метрики) продукта.

Остановимся на классификации метрик ПО, правилах для проведения метрического анализа и процесса их измерения.

Типы метрик.

Существует три типа метрик:

- метрики программного продукта, которые используются при измерении его характеристик - свойств;

- метрики процесса, которые используются при измерении свойства процесса ЖЦ создания продукта.

метрики использования.

Метрики программного продукта включают:

- внешние метрики, обозначающие свойства продукта, видимые пользователю;
- внутренние метрики, обозначающие свойства, видимые только команде разработчиков.

Внешние метрики продукта - это метрики:

- надежности продукта, которые служат для определения числа дефектов;
- функциональности, с помощью которых устанавливаются наличие и правильность реализации функций в продукте;
- сопровождения, с помощью которых измеряются ресурсы продукта (скорость, память, среда);применимости продукта, которые способствуют определению степени доступности для изучения и использования;
- стоимости, которыми определяется стоимость созданного продукта.

Внутренние метрики продукта включают:

- метрики размера, необходимые для измерения продукта с помощью его внутренних характеристик;
- метрики сложности, необходимые для определения сложности продукта;
- метрики стиля, которые служат для определения подходов и технологий создания отдельных компонентов продукта и его документов.

Внутренние метрики позволяют определить производительность продукта и являются релевантными по отношению к внешним метрикам.

Внешние и внутренние метрики задаются на этапе формирования требований к ПО и являются предметом планирования и управления достижением качества конечного программного продукта.

Метрики продукта часто описываются комплексом моделей для установки различных свойств, значений модели качества или прогнозирования. Измерения проводятся, как правило, после калибровки метрик на ранних этапах проекта. Общая мера - степень трассируемости, которая определяется числом трасс, прослеживаемых с помощью моделей сценариев типа UML и оценкой количества:

- требований;
- сценариев и действующих лиц;
- объектов, включенных в сценарий, и локализация требований к каждому сценарию;
- параметров и операций объекта и др.

Стандарт ISO/IEC 9126-2 определяет следующие типы мер:

мера размера ПО в разных единицах измерения (число функций, строк в программе, размер дисковой памяти и др.);

- мера времени (функционирования системы, выполнения компонента и др.);
- мера усилий (производительность труда, трудоемкость и др.);
- мера учета (количество ошибок, число отказов, ответов системы и др.).

Специальной мерой может служить уровень использования повторных компонентов и измеряется как отношение размера продукта, изготовленного из готовых компонентов, к размеру системы в целом. Данная мера используется также при определении стоимости и качества ПО. Примеры метрик:

- общее число объектов и число повторно используемых;
- общее число операций, повторно используемых и новых операций;
- число классов, наследующих специфические операции;
- число классов, от которых зависит данный класс;
- число пользователей класса или операций и др.

При оценке общего количества некоторых величин часто используются среднестатистические метрики (среднее число операций в классе, наследников класса или операций класса и др.).

Как правило, меры в значительной степени являются субъективными и зависят от знаний экспертов, производящих количественные оценки атрибутов компонентов программного продукта.

Примером широко используемых внешних метрик программ являются метрики Холстеда - это характеристики программ, выявляемые на основе статической структуры программы на конкретном языке программирования: число вхождений наиболее часто встречающихся операндов и операторов; длина описания программы как сумма числа вхождений всех операндов и операторов и др.

На основе этих атрибутов можно вычислить время программирования, уровень программы (структурированность и качество) и языка программирования (абстракции средств языка и ориентация на проблему) и др.

В качестве метрик процесса могут быть время разработки, число ошибок, найденных на этапе тестирования и др. Практически используются следующие метрики процесса:

- общее время разработки и отдельно время для каждой стадии;
- время модификации моделей;
- время выполнения работ на процессе;
- число найденных ошибок при инспектировании;
- стоимость проверки качества;
- стоимость процесса разработки.

Метрики использования служат для измерения степени удовлетворения потребностей пользователя при решении его задач. Они помогают оценить не свойства самой программы, а результаты ее эксплуатации - эксплуатационное качество. Примером может служить - точность и полнота реализации задач пользователя, а такжезатраченные ресурсы (трудозатраты, производительность и др.) на эффективное решение задач пользователя. Оценка требований пользователя проводится с помощью внешних метрик.

Стандартная оценка значений показателей качества

Оценка качества ПО согласно четырехуровневой модели качества начинается с нижнего уровня иерархии, т.е. с самого элементарного свойства оцениваемого атрибута показателя качества согласно установленных мер. На этапе проектирования устанавливают значения оценочных элементов для каждого атрибута показателя анализируемого ПО, включенного в требования.

По определению стандарта ISO/IES 9126-2 метрика качества ПО представляет собой "модель измерения атрибута, связываемого с показателем его качества". При измерении показателей качества данный стандарт позволяет определять следующие типы мер:

- меры размера в разных единицах измерения (количество функций, размер программы, объем ресурсов и др.);
- меры времени - периоды реального, процессорного или календарного времени (время функционирования системы, время выполнения компонента, время использования и др.);
- меры усилий - продуктивное время, затраченное на реализацию проекта (производительность труда отдельных участников проекта, коллективная трудоемкость и др.);
- меры интервалов между событиями, например, время между последовательными отказами;
- счетные меры - счетчики для определения количества обнаруженных ошибок, структурной сложности программы, числа несовместимых элементов, числа изменений (например, число обнаруженных отказов и др.).

Метрики качества используются при оценке степени тестируемости с помощью данных (безотказная работа, выполнимость функций, удобство применения интерфейсов пользователей, БД и т.п.) после проведения испытаний ПО на множестве тестов.

Наработка на отказ как атрибут надежности определяет среднее время между появлением угроз, нарушающих безопасность, и обеспечивает трудноизмеримую оценку ущерба, которая наносится соответствующими угрозами.Очень часто оценка программы проводится по числу строк. При сопоставлении двух программ, реализующих одну прикладную задачу, предпочтение

отдается короткой программе, так как её создает более квалифицированный персонал и в ней меньше скрытых ошибок и легче модифицировать. По стоимости она дороже, хотя времени на отладку и модификацию уходит больше. Т.е. длину программы можно использовать в качестве вспомогательного свойства для сравнения программ с учетом одинаковой квалификации разработчиков, единого стиля разработки и общей среды.

Если в требованиях к ПО было указано получить несколько показателей, то просчитанный после сбора данных показатель умножается на соответствующий весовой коэффициент, а затем суммируются все показатели для получения комплексной оценки уровня качества ПО.

На основе измерения количественных характеристик и проведения экспертизы качественных показателей с применением весовых коэффициентов, нивелирующих разные показатели, вычисляется итоговая оценка качества продукта путем суммирования результатов по отдельным показателям и сравнения их с эталонными показателями ПО (стоимость, время, ресурсы и др.).

Т.е. при проведении оценки отдельного показателя с помощью оценочных элементов просчитывается весомый коэффициент k -метрика, j -показатель, i -атрибут. Например, в качестве j -показателя возьмем переносимость. Этот показатель будет вычисляться по пяти атрибутам ($i = 1, \ldots, 5$), причем каждый из них будет умножаться на соответствующий коэффициент k_i .

Все метрики j -атрибута суммируются и образуют i -показатель качества. Когда все атрибуты оценены по каждому из показателей качества, производится суммарная оценка отдельного показателя, а потом и интегральная оценка качества с учетом весовых коэффициентов всех показателей ПО.

В конечном итоге результат оценки качества является критерием эффективности и целесообразности применения методов проектирования, инструментальных средств и методик оценивания результатов создания программного продукта на стадиях ЖЦ.

Для изложения оценки значений показателей качества используется стандарт [10.4], в котором представлены следующие методы: измерительный, регистрационный, расчетный и экспертный (а также комбинации этих методов). Измерительный метод базируется на использовании измерительных и специальных программных средств для получения информации о характеристиках ПО, например, определение объема, числа строк кода, операторов, количества ветвей в программе, число точек входа (выхода), реактивность и др.

Регистрационный метод используется при подсчете времени, числа сбоев или отказов, начала и конца работы ПО в процессе его выполнения.

Расчетный метод базируется на статистических данных, собранных при проведении испытаний, эксплуатации и сопровождении ПО. Расчетными методами оцениваются показатели надежности, точности, устойчивости, реактивности и др.

Экспертный метод осуществляется группой экспертов - специалистов, компетентных в решении данной задачи или типа ПО. Их оценка базируется на опыте и интуиции, а не на непосредственных результатах расчетов или экспериментов. Этот метод проводится путем просмотра программ, кодов, сопроводительных документов и способствует качественной оценки созданного продукта. Для этого устанавливаются контролируемые признаки, которые коррелированны с одним или несколькими показателями качества и включены в опросные карты экспертов. Метод применяется при оценке таких показателей, как анализируемость, документируемость, структурированность ПО и др.

Для оценки значений показателей качества в зависимости от особенностей используемых ими свойств, назначения, способов их определения используются:

- шкала метрическая (1.1 - абсолютная, 1.2 - относительная, 1.3 - интегральная);
- шкала порядковая (ранговая), позволяющая ранжировать характеристики путем сравнения с опорными;
- классификационная шкала, характеризующая наличие или отсутствие рассматриваемого свойства у оцениваемого программного обеспечения.

Показатели, которые вычисляются с помощью метрических шкал, называются количественные, а определяемые с помощью порядковых и классификационных шкал - качественные.

Атрибуты программной системы, характеризующие ее качество, измеряются с использованием метрик качества. Метрика определяет меру атрибута, т.е. переменную, которой присваивается значение в результате измерения.Для правильного использования результатов измерений каждая мера идентифицируется шкалой измерений.

Стандарт ISO/IES 9126-2 рекомендует применять 5 видов шкал измерения значений, которые упорядочены от менее строгой к более строгой:

- номинальная шкала отражает категории свойств оцениваемого объекта без их упорядочения;
- порядковая шкала служит для упорядочивания характеристики по возрастанию или убыванию путем сравнения их с базовыми значениями;
- интервальная шкала задает существенные свойства объекта (например, календарная дата);
- относительная шкала задает некоторое значение относительно выбранной единицы;
- абсолютная шкала указывает на фактическое значение величины (например, число ошибок в программе равно 10).

Из всех областей программной инженерии надежность ПС является самой исследованной областью. Ей предшествовала разработка теории надежности технических средств, оказавшая влияние на развитие надежности ПС. Вопросами надежности ПС занимались разработчики ПС, пытаясь разными системными средствами обеспечить надежность, удовлетворяющую заказчика, а также теоретики, которые, изучая природу функционирования ПС, создали математические модели надежности, учитывающие разные аспекты работы ПС (возникновение ошибок, сбоев, отказов и др.) и позволяющие оценить реальную надежность. В результате надежность ПС сформировалась как самостоятельная теоретическая и прикладная наука [10.5-10.10, 10.16-10.24].

Надежность сложных ПС существенным образом отличается от надежности аппаратуры. Носители данных (файлы, сервер и т.п.) обладают высокой надежностью, записи на них могут храниться длительное время без разрушения, поскольку физическому разрушению они не подвергаются.

С точки зрения прикладной науки надежность - это способность ПС сохранять свои свойства (безотказность, устойчивость и др.), преобразовывать исходные данные в результаты в течение определенного промежутка времени при определенных условиях эксплуатации. Снижение надежности ПС происходит из-за ошибок в требованиях, проектировании и выполнении. Отказы и ошибки зависят от способа производства продукта и появляются в программах при их исполнении на некотором промежутке времени.

Для многих систем (программ и данных) надежность - главная целевая функция реализации. К некоторым типам систем (реального времени, радарные системы, системы безопасности, медицинскоеоборудование со встроенными программами и др.) предъявляются высокие требования к надежности, такие, как отсутствие ошибок, достоверность, безопасность и др.

Таким образом, оценка надежности ПС зависит от числа оставшихся и не устраненных ошибок в программах. В ходе эксплуатации ПС ошибки обнаруживаются и устраняются. Если при исправлении ошибок не вносятся новые или, по крайней мере, новых ошибок вносится меньше, чем устраняется, то в ходе эксплуатации надежность ПС непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность системы и соответственно ее качество.

Надежность является функцией от ошибок, оставшихся в ПС после ввода его в эксплуатацию. ПС без ошибок является абсолютно надежным. Но для больших программ абсолютная надежность практически недостижима. Оставшиеся необнаруженные ошибки проявляют себя время от времени при определенных условиях (например, при некоторой совокупности исходных данных) сопровождения и эксплуатации системы.

Для оценки надежности ПС используются такие статистические показатели, как вероятность и время безотказной работы, возможность отказа и частота (интенсивность) отказов. Поскольку в качестве причин отказов рассматриваются только ошибки в программе, которые не могут самоустраниться, то ПС следует относить к классу невосстанавливаемых систем.

При каждом проявлении новой ошибки, как правило, проводится ее локализация и исправление. Строго говоря, набранная до этого статистика об отказах теряет свое значение, так

как после внесения изменений программа, по существу, является новой программой в отличие от той, которая до этого испытывалась.

В связи с исправлением ошибок в ПС надежность, т.е. ее отдельные атрибуты, будут все время изменяться, как правило, в сторону улучшения. Следовательно, их оценка будет носить временный и приближенный характер. Поэтому возникает необходимость в использовании новых свойств, адекватных реальному процессу измерения надежности, таких, как зависимость интенсивности обнаруженных ошибок от числа прогонов программы и зависимость отказов от времени функционирования ПС и т.п.

К факторам гарантии надежности относятся:

- риск как совокупность угроз, приводящих к неблагоприятным последствиям и ущербу системы или среды;
- угроза как проявление неустойчивости, нарушающей безопасность системы;
- анализ риска - изучение угрозы или риска, их частота и последствия;
- целостность - способность системы сохранять устойчивость работы и не иметь риска;

Риск преобразует и уменьшает свойства надежности, так как обнаруженные ошибки могут привести к угрозе, если отказы носят частотный характер.

Основные понятия в проблематике надежности ПС

Формально модели оценки надежности ПС базируются на теории надежности и математическом аппарате с допущением некоторых ограничений, влияющих на эту оценку. Главным источником информации, используемой в моделях надежности, является процесс тестирования, эксплуатации ПС и разного вида ситуации, возникающие в них. Ситуации порождаются возникновением ошибок в ПС, требуют их устранения для продолжения тестирования.

Базовыми понятиями, которые используются в моделях надежности ПС, являются [10.5-10.10].

Отказ ПС (failure) - это переход ПС из работающего состояния в нерабочее или когда получаются результаты, которые не соответствуют заданным допустимым значениям. Отказ может быть вызван внешними факторами (изменениями элементов среды эксплуатации) и внутренними - дефектами в самой ПС.

Дефект (fault) в ПС - это последствие использования элемента программы, который может привести к некоторому событию, например, в результате неверной интерпретации этого элемента компьютером (как ошибка (fault) в программе) или человеком (ошибка (error) исполнителя). Дефект является следствием ошибок разработчика на любом из процессов разработки - в описании спецификаций требований, начальных или проектных спецификациях, эксплуатационной документации и т.п. Дефекты в программе, не выявленные в результате проверок, являются источником потенциальных ошибок и отказов ПС. Проявление дефекта в виде отказа зависит от того, какой путь будет выполнять специалист, чтобы найти ошибку в коде или во входных данных. Однако не каждый дефект ПС может вызвать отказ или может быть связан с дефектом в ПС или среды. Любой отказ может вызвать аномалию от проявления внешних ошибок и дефектов.

Ошибка (error) может быть следствием недостатка в одном из процессов разработки ПС, который приводит к неправильной интерпретации промежуточной информации, заданной разработчиком или при принятии им неверных решений.

Интенсивность отказов - это частота появления отказов или дефектов в ПС при ее тестировании или эксплуатации.

При выявлении отклонения результатов выполнения от ожидаемых во время тестирования или сопровождения осуществляется поиск, выяснение причин отклонений и исправление связанных с этим ошибок.

Модели оценки надежности ПС в качестве входных параметров используют сведения об ошибках, отказах, их интенсивности, собранных в процессе тестирования и эксплуатации.

Классификация моделей надежности

Как известно, на данный момент времени разработано большое количество моделей надежности ПС и их модификаций. Каждая из этих моделей определяет функцию надежности, которую можно вычислить при задании ей соответствующих данных, собранных во время функционирования ПС. Основными данными являются отказы и время. Другие дополнительные параметры связаны с типом ПС, условиями среды и данных.

Ввиду большого разнообразия моделей надежности разработано несколько подходов к классификации этих моделей. Такие подходы в целом основываются на истории ошибок в проверяемой и тестируемой ПС на этапах ЖЦ. Одной из классификаций моделей надежности ПО является классификация Хетча [10.10]. В ней предлагается разделение моделей на прогнозирующие, измерительные и оценочные (рисунок 1).

Прогнозирующие модели надежности основаны на измерении технических характеристик создаваемой программы: длина, сложность, число циклов и степень их вложенности, количество ошибок на страницу операторов программы и др.

Например, модель Мотли-Брукса основывается на длине и сложности структуры программы (количество ветвей, циклов, вложенность циклов), количестве и типах переменных, а также интерфейсов. В этих моделях длина программы служит для прогнозирования количества ошибок, например, для 100 операторов программы можно смоделировать интенсивность отказов.

Рисунок 1 – Классификация моделей надежности

Модель Холстеда прогнозирует количество ошибок в программе в зависимости от ее объема и таких данных, как число операций (n_1) и операндов (n_2), а также их общее число (N_1, N_2).

Время программирования программы предлагается вычислять по следующей формуле:

$$T = \frac{n_1 N_2 (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n_1}{2 n_2 S},$$

где S - число Страуда (Холстед принял равным 18 - число умственных операций в единицу времени).

Объем вычисляется по формуле:

$$V = (2 + n_2^*) \log_2 (2 + n_2^*),$$

где n_2^* - максимальное число различных операций.

Измерительные модели предназначены для измерения надежности программного обеспечения, работающего с заданной внешней средой. Они имеют следующие ограничения:

программное обеспечение не модифицируется во время периода измерений свойств надежности;

обнаруженные ошибки не исправляются;

измерение надежности проводится для зафиксированной конфигурации программного обеспечения.

Типичным примером таких моделей являются модели Нельсона и РамамуртиБастани и др.Модель оценки надежности Нельсона основывается на выполнении k-прогонов программы при тестировании и позволяет определить надежность

$$R(k) = exp[-\sum \nabla t_j \lambda(t)],$$

где t_j - время выполнения j -прогона, $\lambda(t) = -[\ln(1 - q_i)]$ и при $q_i \leq 1$ она интерпретируется как интенсивность отказов.

В процессе испытаний программы на тестовых n_l прогонах оценка надежности вычисляется по формуле

$$R(l) = \frac{1 - n_l}{k},$$

где k - число прогонов программы.

Таким образом, данная модель рассматривает полученные количественные данные о проведенных прогонах.

Оценочные модели основываются на серии тестовых прогонов и проводятся на этапах тестирования ПС. В тестовой среде определяется вероятность отказа программы при ее выполнении или тестировании.

Эти типы моделей могут применяться на этапах ЖЦ. Кроме того, результаты прогнозирующих моделей могут использоваться как входные данные для оценочной модели. Имеются модели (например, модель Муссы), которые можно рассматривать как оценочную и в то же время как измерительную модель [10.16, 10.17].

Другой вид классификации моделей предложил Гоэл [10.18, 10.19], согласно которой модели надежности базируются на отказах и разбиваются на четыре класса моделей:

без подсчета ошибок;

с подсчетом отказов;

с подсевом ошибок;

модели с выбором областей входных значений.

Модели без подсчета ошибок основаны на измерении интервала времени между отказами и позволяют спрогнозировать количество ошибок, оставшихся в программе. После каждого отказа оценивается надежность и определяется среднее время до следующего отказа. К таким моделям относятся модели Джелински и Моранды, Шика Вулвертона и Литвуда-Вералла [10.20, 10.21].

Модели с подсчетом отказов базируются на количестве ошибок, обнаруженных на заданных интервалах времени. Возникновение отказов в зависимости от времени является стохастическим процессом с непрерывной интенсивностью, а количество отказов является случайной величиной. Обнаруженные ошибки, как правило, устраняются и поэтому количество ошибок в единицу времени уменьшается. К этому классу моделей относятся модели Шумана, Шика- Вулвертона, Пуассоновская модель и др. [10.21-10.24].

Модели с подсевом ошибок основаны на количестве устраненных ошибок и подсеве, внесенном в программу искусственных ошибок, тип и количество которых заранее известны. Затем определяется соотношение числа оставшихся прогнозируемых ошибок к числу искусственных ошибок, которое сравнивается с соотношением числа обнаруженных действительных ошибок к числу обнаруженных искусственных ошибок. Результат сравнения используется для оценки надежности и качества программы. При внесении изменений в программу проводится повторное тестирование и оценка надежности. Этот подход к организации тестирования отличается громоздкостью и редко используется из-за дополнительного объема работ, связанных с подбором, выполнением и устранением искусственных ошибок.

Модели с выбором области входных значений основываются на генерации множества тестовых выборок из входного распределения, и оценка надежности проводится по полученным отказам на основе тестовых выборок из входной области. К этому типу моделей относится модель Нельсона и др.

Таким образом, классификация моделей роста надежности относительно процесса выявления отказов, фактически разделена на две группы:

модели, которые рассматривают количество отказов как марковский процесс;

модели, которые рассматривают интенсивность отказов как пуассоновский процесс.

Фактор распределения интенсивности отказов разделяет модели на экспоненциальные, логарифмические, геометрические, байесовские и др.

Марковские и пуассоновские модели надежности

Марковский процесс характеризуется дискретным временем и конечным множеством состояний. Временной параметр пробегает неотрицательные числовые значения, а процесс (цепочка) определяется набором вероятностей перехода $p_{ij}(n)$, т.е. вероятностью перейти на n -шаге из состояния i в состояние j. Процесс называется однородным, если он не зависит от n. В моделях, базирующихся на процессе Маркова, предполагается, что количество дефектов, обнаруженных в ПС, в любой момент времени зависит от поведения системы и представляется в виде стационарной цепи Маркова [10.5, 10.7, 10.10]. При этом количество дефектов конечное, но является неизвестной величиной, которая задается для модели в виде константы. Интенсивность отказов в ПС или скорость прохода по цепи зависит лишь от количества дефектов, которые остались в ПС. К этой группе моделей относятся: Джелински- Моранды [10.20], Шика-Вулвертона, Шантикумера [10.21] и др.

Ниже рассматриваются некоторые модели надежности, которые обеспечивают рост надежности ПО (модели роста надежности [10.7, 10.10]), находят широкое применение на этапе тестирования и описывают процесс обнаружения отказов при следующих предположениях:

все ошибки в ПС не зависят друг от друга с точки зрения локализации отказов;

интенсивность отказов пропорциональна текущему числу ошибок в ПС (убывает при тестировании программного обеспечения);

вероятность локализации отказов остается постоянной;

локализованные ошибки устраняются до того, как тестирование будет продолжено;

при устранении ошибок новые ошибки не вносятся.

Приведем основные обозначения величин при описании моделей роста надежности:

m - число обнаруженных отказов ПО за время тестирования;

X_i - интервалы времени между отказами $i-1$ и i, при $i=1,\ldots,m$;

S_i - моменты времени отказов (длительность тестирования до i -отказа), $S_i = X_k$ при $i=1,\ldots,m$;

T - продолжительность тестирования ПО (время, для которого определяется надежность);

N - оценка числа ошибок в ПО в начале тестирования;

M - оценка числа прогнозированных ошибок;

$M\mathrm{T}$ - оценка среднего времени до следующего отказа;

$E(T_p)$ - оценка среднего времени до завершения тестирования;

$Var(T_p)$ - оценка дисперсии;

$R(t)$ - функция надежности ПО;

$Z_i(t)$ - функция риска в момент времени t между $i-1$ и i -отказами;

c - коэффициент пропорциональности;

b - частота обнаружения ошибок.

Далее рассматриваются несколько моделей роста надежности, основанные на этих предположениях и использовании результатов тестирования программ в части отказов, времени между ними и др.

Модель Джелинского-Моранды. В этой модели используются исходные данные, приведенные выше, а также:

m - число обнаруженных отказов за время тестирования;

X_i - интервалы времени между отказами;

T - продолжительность тестирования.

Функция риска $Z_i(t)$ в момент времени t расположена между $i-1$ и i имеет вид:

$$Z_i(t) = c(N - n_{i-1}),$$

где $i=1,\ldots,m$; $T_{i-1} < t < T_i$.

Эта функция считается ступенчатой кусочнопостоянной функцией с постоянным коэффициентом пропорциональности и величиной ступени - c. Оценка параметров c и N производится с помощью системы уравнений:

$$\sum_{i=1}^{m} \frac{1}{N - N_{i-1}} - \sum_{i=1}^{m} cX_i = 0,$$
$$\frac{n}{c} - NT - \sum_{i=1}^{m} X_i n_{i-1} = 0$$

При этом суммарное время тестирования вычисляется так: $T = \sum_{i=1}^{m} X_i$

Выходные показатели для оценки надежности относительно указанного времени T включают:

число оставшихся ошибок $M_m = N - m$;

среднее время до текущего отказа $M\mathrm{T}_m = 1/(N - m)c$;

среднее время до завершения тестирования и его дисперсию

$$E(T_p) = \sum_{i=1}^{N-n} \frac{1}{ic},$$

$$Var(T_p) = \sum_{i=1}^{N-n\}} \frac{1}{(ic)^2}.$$

При этом функция надежности вычисляется по формуле:

$$Rm(t) = exp(-(N - m)ct),$$

при $t > 0$ и числе ошибок, найденных и исправленных на каждом интервале тестирования, равным единице.

Модель Шика-Вулвертона. Модель используется тогда, когда интенсивность отказов пропорциональна не только текущему числу ошибок, но и времени, прошедшему с момента последнего отказа. Исходные данные для этой модели аналогичны выше рассмотренной модели Джелински-Моранды:

m - число обнаруженных отказов за время тестирования,

X_i - интервалы времени между отказами,

T - продолжительность тестирования.

Функции риска $Z_i(t)$ в момент времени между $i - 1$ и $i - m$ отказами определяются следующим образом:

$$Z_i(t) = c(N - n_{i-1}), \text{где } i = 1, \ldots, m;\ T_{i-1} < t < T_i$$
$$T = \sum_{i=1}^{m} X_i$$

Эта функция является линейной внутри каждого интервала времени между отказами, возрастает с меньшим углом наклона. Оценка с и N вычисляется из системы уравнений:

К выходным показателям надежности относительно продолжительности T относятся:

число оставшихся ошибок $Mm = N - m$;

среднее время до следующего отказа MTт = (p / (2 (N - m) c))1/2;

среднее время до завершения тестирования и его дисперсия

$$E(T_p) = \sum_{i=1}^{N-m} \sqrt{\frac{\pi}{2ic}},$$

$$Var(T_p) = \sum_{i=1}^{N-m} \frac{2 - \pi/2}{ic}.$$

Функция надежности вычисляется по формуле:

$$R_T(t) = exp(-\frac{(N-m)ct^2}{2}), t \geq 0.$$

Модели пуассоновского типа базируются на выявлении отказов и моделируются неоднородным процессом, который задает $\{M(t), t \geq 0\}$ - неоднородный пуассоновский процесс с функцией интенсивности $\lambda(t)$, что соответствует общему количеству отказов ПС за время его использования t .

Модель Гоело-Окумото. В основе этой модели лежит описание процесса обнаружения ошибок с помощью неоднородного пуассоновского процесса, ее можно рассматривать как модель экспоненциального роста. В этой модели интенсивность отказов также зависит от времени. Кроме того, в ней количество выявленных ошибок трактуется как случайная величина, значение которой зависит от теста и других условных факторов.

Исходные данные этой модели:

m - число обнаруженных отказов за время тестирования;

X_i - интервалы времени между отказами;

T - продолжительность тестирования.

Функция среднего числа отказов, обнаруженных к моменту t , имеет вид

$$m(t) = N(1 - e^{-bt}),$$

где b - интенсивность обнаружения отказов и показатель роста надежности $q(t) = b$.

Функция интенсивности $\lambda(t)$ в зависимости от времени работы до отказа равна

$$\lambda(t) = Nb^{-b}, t \geq 0.$$

Оценка b и N получаются из решения уравнений:

$$m/N - 1 + \exp\{(-bT)\} = 0$$
$$m/b - \sum_{i=1}^{m}\{t_i\} - N_m \exp\{(-bT)\} = 0.$$

Выходные показатели надежности относительно времени T определяют:

среднее число ошибок, которые были обнаружены в интервале $[0, T]$, по формуле $E(N_T) = Nexp(-bT)$,

функцию надежности

$$R_T(t) = exp(N(e^{-bt} - e^{-bt(t+\mathrm{T})})), t \geq 0.$$

В этой модели обнаружение ошибки трактуется как случайная величина, значение которой зависит от теста и операционной среды.

В других моделях количество обнаруженных ошибок рассматривается как константа.В моделях роста надежности исходной информацией для расчета надежности являются интервалы времени между отказами тестируемой программы, число отказов и время, для которого определяется надежность программы при отказе. На основании этой информации по моделям определяются показатели надежности вида:

вероятность безотказной работы;

среднее время до следующего отказа;

число необнаруженных отказов (ошибок);

среднее время дополнительного тестирования программы.

Модель анализа результатов прогона тестов использует в своих расчетах общее число экспериментов тестирования и число отказов. Эта модель определяет только вероятность безотказной работы программы и выбрана для случаев, когда предыдущие модели нельзя использовать (мало данных, некорректность вычислений). Формула определения вероятности безотказной работы по числу проведенных экспериментов имеет вид

$$P = 1 - Nex/N,$$

где Nex - число ошибочных экспериментов, N - число проведенных экспериментов для проверки работы ПС.

Таким образом, можно сделать вывод о том, что модели надежности ПС основаны на времени функционирования и/или количестве отказов (ошибок), полученных в программах в процессе их тестирования или эксплуатации. Модели надежности учитывают случайный марковский и пуассоновский характер соответственно процессов обнаружения ошибок в программах, а также характер и интенсивность отказов.

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 3.3 Инструментальные средства этапа отладки и тестирования программ

Практическая работа № 10
Интеграционное тестирование

Цель: Сформировать понятие и основные характеристики проекта. Выяснить причины необходимости ведения проектной деятельности и управления проектом. Определить отличия проекта и операционной деятельности.

Выполнив работу, Вы будете:
уметь:
– У9 выполнять отладку и тестирование информационных систем и программных средств;

Материальное обеспечение:
Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio.

Задание:
1. Выполните оценку необходимого количества тестов для определения качества программного приложения для тест кейса:

Действие	Ожидаемый результат
1. Открываем форму отправки сообщения	Форма открыта Все поля по умолчанию пусты Обязательные поля помечены - * Кнопка "Отправить" не активна
2. Заполняем поля формы: Тип обращения Контактное лицо Контактный телефон Сообщение	Поля заполнены Кнопка "Отправить" - активна (Enabled)
3. Нажимаем кнопку "Отправить"	Если введенные данные корректны - Сообщение "Заявка отправлена"выведено на экран. Новая заявка появилась в списке на странице "Заявки". Если введенные данные НЕ корректны -; Валидационное сообщение со всеми ошибками выведено на экран. Заявка НЕ появилась в списке на странице "Заявки".

2. Оцените необходимое количество тестов для разработанного программного продукта (лабораторное задание № 10), разработайте тестовые сценарии (тест-кейсы), проведите тестирование. Определите задачи доработки программного продукта. Сформируйте тестовую документацию.

Порядок выполнения работы:
1. Выполнить оценку необходимого количества тестов для определения качества программного приложения для тест кейса
2. Изучить предлагаемый теоретический материал по теме.
3. Определить необходимые виды тестирования для разработанного программного продукта (лабораторное занятие № 10).
4. Оценить необходимое количество тестов.
5. Разработать тестовые сценарии (тест-кейсы).
6. Провести тестирование:
7. Определить задачи доработки программного продукта.

8. Сформировать тестовую документацию.
9. Сформировать отчет о работе.

Ход работы:

Краткие теоретические сведения

Тестовое Покрытие (Test Coverage)

Тестовое Покрытие- это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

Если рассматривать тестирование как "проверку соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов", то именно этот конечный набор тестов и будет определять тестовое покрытие:

Чем выше требуемый уровень тестового покрытия, тем больше тестов будет выбрано, для проверки тестируемых требований или исполняемого кода.

Сложность современного программного обеспечения и инфраструктуры сделало невыполнимой задачу проведения тестирования со 100% тестовым покрытием. Поэтому для разработки набора тестов, обеспечивающего более менее высокий уровень покрытия можно использовать специальные инструменты либо техники тест дизайна.

Существует 2 широко применяемых подхода к оценке и измерению тестового покрытия:

Покрытие требований (Requirements Coverage)- оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).

Покрытие кода (Code Coverage)- оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей программного обеспечения.

Различия: Метод покрытия требований сосредоточен на проверке соответствия набора проводимых тестов требованиям к продукту, в то время как анализ покрытия кода - на полноте проверки тестами, разработанной части продукта (исходного кода).

Ограничения: Метод оценки покрытия кода не выявит нереализованные требования, так как работает не с конечным продуктом, а с существующим исходным кодом. Метод покрытия требований может оставить непроверенными некоторые участки кода, потому что не учитывает конечную реализацию.

Покрытие требований (Requirements Coverage)

Расчет тестового покрытия относительно требований проводится по формуле:

Tcov = (Lcov/Ltotal) * 100%,

где: Tcov- тестовое покрытие, Lcov - количество требований, проверяемых тест кейсами, Ltotal- общее количество требований

Для измерения покрытия требований, необходимо проанализировать требования к продукту и разбить их на пункты. Опционально каждый пункт связывается с тест кейсами, проверяющими его. Совокупность этих связей - и является матрицей трассировки. Проследив связи, можно понять какие именно требования проверяет тестовый случай.

Тесты, не связанные с требованиями не имеют смысла. Требования, не связанные с тестами - это "белые пятна", т.е. выполнив все созданные тест кейсы, нельзя дать ответ реализовано данное требование в продукте или нет.

Для оптимизации тестового покрытия при тестировании на основании требований, наилучшим способом будет использование стандартных техник тест дизайна. Пример разработки тестовых случаев по имеющимся требованиям рассмотрен в разделе: "Практическое применение техник тест дизайна при разработке тест кейсов"

Покрытие кода (Code Coverage)

Расчет тестового покрытия относительно исполняемого кода программного обеспечения проводится по формуле:

Tcov = (Ltc/Lcode) * 100%,

где: Tcov- тестовое покрытие, Ltc- кол-ва строк кода, покрытых тестами, Lcode- общее кол-во строк кода.

В настоящее время существует инструментарий (например, Clover), позволяющий проанализировать в какие строки были вхождения во время проведения тестирования, благодаря чему можно значительно увеличить покрытие, добавив новые тесты для конкретных случаев, а

также избавиться от дублирующих тестов. Проведение такого анализа кода и последующая оптимизация покрытия достаточно легко реализуется в рамках тестирования белого ящика (white-box testing) при модульном, интеграционном и системном тестировании; при тестировании же черного ящика (black-box testing) задача становится довольно дорогостоящей, так как требует много времени и ресурсов на установку, конфигурацию и анализ результатов работы, как со стороны тестировщиков, так и разработчиков.

Техники тест дизайна (Test Design Technics)

Многие люди тестируют и пишут тестовые случаи (test cases), но не многие пользуются специальными техниками тест дизайна. Постепенно, набираясь опыта они осознают, что постоянно делают одну и ту же работу, поддающуюся конкретным правилам. И тогда они находят, что все эти правила уже описаны.

Предлагаю вам ознакомиться с кратким описанием наиболее распространенных техник тест дизайна:

Эквивалентное Разделение (**Equivalence Partitioning - EP**). Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала - 0.

Анализ Граничных Значений (**Boundary Value Analysis - BVA**). Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям, имеющим ограничения.

Причина / Следствие (**Cause/Effect - CE**). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как "Имя", "Адрес", "Номер Телефона" а затем, нажать кнопку "Добавить" - эта "Причина". После нажатия кнопки "Добавить", система добавляет клиента в базу данных и показывает его номер на экране - это "Следствие".

Предугадывание ошибки (**Error Guessing - EG**). Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. Например, спецификация говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки.

Исчерпывающее тестирование (**Exhaustive Testing - ET**) - это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

Практическое применение техник тест дизайна при разработке тест кейсов

Многие знают, что такое тест дизайн, но не все умеют его применять. Чтобы немного прояснить ситуацию, мы решили предложить Вашему вниманию последовательный подход к разработке тестовых случаев (тест кейсов), используя самые простейшиетехники тест дизайна:

Эквивалентное Разделение (Equivalence Partitioning), далее в тексте -EP

Анализ Граничных Значений (Boundary Value Analysis), далее в тексте -BVA

Предугадывание ошибки (Error Guessing), далее в тексте -EG

Причина / Следствие (Cause/Effect), далее в тексте -CE

План разработки тест кейсовпредлагается следующий:

Анализ требований.

Определение набора тестовых данных на основании EP,BVA,EG.

Разработка шаблона теста на основании CE.

Написание тест кейсов на основании первоначальных требований, тестовых данных и шагов теста.

Далее на примере, рассмотрим предложенный подход.

Пример:

Протестировать функциональность формы приема заявок, требования к которой предоставлены в следующей таблице:

Элемент	Тип элемента	Требования
Тип обращения	combobox	Набор данных: Консультация Проведение тестирования Размещение рекламы Ошибка на сайте *- на процесс выполнения операции приема заявок не влияет.
Контактное лицо	editbox	1. Обязательное для заполнения 2. Максимально 25 символов 3. Использование цифр и спец символов не допускается
Контактный телефон	editbox	Обязательное для заполнения Допустимые символы "+" и цифры "+" можно использовать только в начале номера Допустимые форматы: начинается с плюса - 11-15 цифр +31612361264 +375291438884 без плюса - 5-10 цифр, например: 0613261264 2925167
Сообщение	text area	1. Обязательное для заполнения 2. Максимальная длина 1024 символа
Отправить	button	Состояние: 1. По умолчанию - не активна (Disabled) 2. После заполнения обязательных полей становится активна (Enabled) Действия после нажатия 1. Если введенные данные корректны - отправка сообщения 2. Если введенные данные НЕ корректны - валидационное сообщение

Вариант использования (иногда его может и не быть) представлен на рисунке 1.

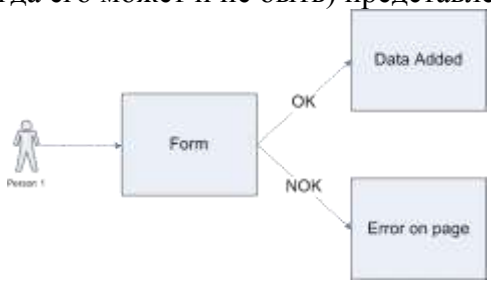

Рисунок 1 – Вариант использования

1. Анализ требований

Читаем, анализируем требования и выделяем для себя следующие нюансы:

какие из полей обязательные для заполнения?

имеют ли поля ограничения по длине или по размерности (границы)?

какие из полей имеют специальные форматы?

2.Определение набора тестовых данных

Отталкиваясь от требований к полям, используя техники тест дизайна начинаем определение набора тестовых данных:

в зависимости от того обязательное поле или нет, определим какие поля необходимо проверить на пустое значение, так как оно может вызывать ошибку (В результирующей таблице оранжевыйцвет)

т.к. исчерпывающее тестирование не представляется возможным из-за огромного числа всевозможных комбинаций значений, в первую очередь необходимо определитьминимальный набор данных. Это можно сделать используя такиетехники, какEPиBVA. (В результирующей таблицеголубойцвет)

На форме присутствует поле, имеющее составной тип (цифры используются совместно с символами), обладает специальным форматом данных и поэтому выделение тестовых данных для него - это достаточно трудоемкая задача. В пределах данной статьи ограничимся только простой проверкой форматов и основных требований описанных в форме приема заявок.

По завершению генерации данных используя стандартные техники, можно добавить некоторое количество значений на основании личного опыта (техника EG) - это будет использование спец. символов, очень длинных строк, разных форматов данных, регистров в строках (Upper, Lowwer, Mixed cases), отрицательные и нулевые значения, кейворды Null - NaN - Infinity и т.д. Сюда можно включить все, что вы полагаете может вывести приложение из строя (В результирующей таблицефиолетовыйцвет)

Примечание:

Отметим, что количество тестовых данных после окончательной генерации будет достаточно большим, даже при использовании специальных техник тест дизайна. Поэтому ограничимся лишь несколькими значениями для каждого поля, так как цель данной статьи показать именно процесс создания тест кейсов, а не процесс получения конкретных тестовых данных.

2.1 Выбор тестовых данных для каждого отдельно взятого поля

Поле Тип обращения. Так как все данные входят в 1 класс эквивалентности, то есть не изменяют сам процесс выполнения приема заявки, берем любою (1-ю) позицию в листе с ожидаемым результатом ОК. Но т.к. реализовано поле как лист, имеет также смысл рассмотреть и граничные условия (техника BVA), т.е. берем первый и последний элементы. Итого: 1-я и последняя позиции в листе. Ожидаемый результат при использовании - ОК.

Поле Контактное лицо. Это обязательное поле размером от 1 до 25 символов (включая границы). Проверка на обязательность добавляет к тестовым данным пустое значение. Проведем анализ граничных условий (BVA), получим набор: 0, 1, 2, 24, 25 и 26 символов. Пустое значение (0 символов) уже было добавлено при анализе обязательности поля для ввода, поэтому при BVA мы не будем добавлять его еще раз. (если его добавить второй раз, произойдет дублирование тестовых данных, которое не приведет к нахождению новых дефектов, а значит повторное добавление в домен не имеет смысла). В связи с тем, что значения 2 и 24 символа являются, с нашей точки зрения, некритичными, их можно не добавлять. В итоге получаем, что минимальный набор данных для тестирования поля - это строки 1 и 25 - ОК, и 0 (пустое значение), 26 символов - NOK.

поле Контактный телефонсостоит из нескольких частей: код страны, код оператора, номер телефон (который может быть составной и разделенный дефисами). Для определения правильного набора тестовых данных необходимо рассматривать каждую составную часть по-отдельности. Применяя BVA и EP, получим:

для номеров с плюсомПо BVA получим номера с 10, 11, 12 и 14, 15, 16 цифрами, где 10 и 16 - NOK, а 11, 12, 14, 15 - OK Рассматривая полученные данные с позиции EP выделим, что 11, 12, 14, 15 входят в один класс эквивалентности. Поэтому при тестировании мы можем использовать любое из них, но так как 11 и 15 - это границы интервала, то на наш взгляд их пропускать нельзя. Следовательно мы можем уменьшить набор значений до двух, исключив 12 и 14, а оставив 11 и 15 для проверки граничных условий. Итого имеем: 11 и 15 цифр - OK, (+12345678901, +123456789012345) 10 и 16 цифр - NOK; (+1234567890, +12345678901234567890)

для номеров без плюса:По BVA получим номера с 4, 5, 6 и 9, 10, 11 цифрами. Действуя аналогично примеру для номеров телефонов с плюсом, исключим значения 6 и 9, оставив 5 и 10. Итого имеем: 5 и 10 цифр - OK, (12345, 1234567890) 4 и 11 цифр - NOK; (1234, 12345678901)

поле Сообщение. подбор данных проводим по аналогии с полем Контактное лицо. На выходе получаем значения: строки 1 и 1024 - ОК, и 1025 символов - NOK.

Результирующая таблица данных, для использования при последующем составлении тест кейсов

Поле	OK/NOK	Значение	Комментарий

Тип обращения	OK	Консультация	первый в списке
		Ошибка на сайте	последний в списке
	NOK		
Контактное лицо	OK	йцукенгшщзйцукенгшщзйцуке	25 символов нижний регистр
		а	1 символ
		ЙЦУКЕНГШЩЗФЫВАПРОЛДЖЯЧСМИ	25 символов ВЕРХНИЙ регистр
		ЙЦУКЕНГШЩЗфывапролджЯЧСМИ	25 символов СМеШаННыЙ регистр
	NOK		пустое значение
		йцукенгшщзйцукенгшщзйцукей	длина больше максимальной(26 символов
		@#$%^&;.?,>|\/№"!()_{}[<~	спец. символы (ASCII)
		12345678901234567890123 45	только цифры
		adsadasdasdas dasdasd asasdsads(...)sas	очень длинная строка (~1Mb)
Контактный телефон	OK	+12345678901	с плюсом - минимальная длина
		+123456789012345	с плюсом - максимальная длина
		12345	без плюса - минимальная длина
		1234567890	без плюса - максимальная длина
	NOK		пустое значение
		+1234567890	с плюсом - < минимальной длины
		+1234567890123456	с плюсом - > максимальной длины
		1234	без плюса - < минимальной длины
		12345678901	без плюса - > максимальной длины
		+YYYXXXyyyxxzz	с плюсом - буквы вместо цифр
		yyyxxxxzz	без плюса - буквы вместо цифр

		+###-$$$-%^-&^-&!	спец. символы (ASCII)
		1232312323123213231232(...)99	очень длинная строка (~1Mb)
Сообщение	OK	йццуйцуйц(...)йцу	максимальная длина (1024 символа)
	NOK		пустое значение
		йццуйцуйц(...)йцуц	длина больше максимальной (1025 символов)
		adsadasdasdas dasdasd asasdsads(...)sas	очень длинная строка (~1Mb)
		@##$$$%^&^&	только спец. символы (ASCII)

Пакеты тестов

Как правило, модульные тесты и тестовые конфигурации разрабатываются самими разработчиками, основная задача тестера в этом случае – организовать все тесты в упорядоченную структуру пакетов и указать, какие пакеты должны исполняться в каких условиях. Вся иерархия тестов хранится в так называемом файле метаданных, имеющем расширение vsmdi. Этот файл находится в системе контроля версий и может быть включен в решение как отдельный элемент (рисунок 2).

Для создания тестового пакета можно воспользоваться меню "Create New Test List", как показано на рисунке 3, и после этого откроется окно для задания имени нового пакета и определения его места в иерархии тестовых пакетов (рисунок 4). В том случае, если для решения уже создан файл метаданных, пакет тестов будет добавлен к нему, если же файла метаданных еще создано не было, он будет создан автоматически.

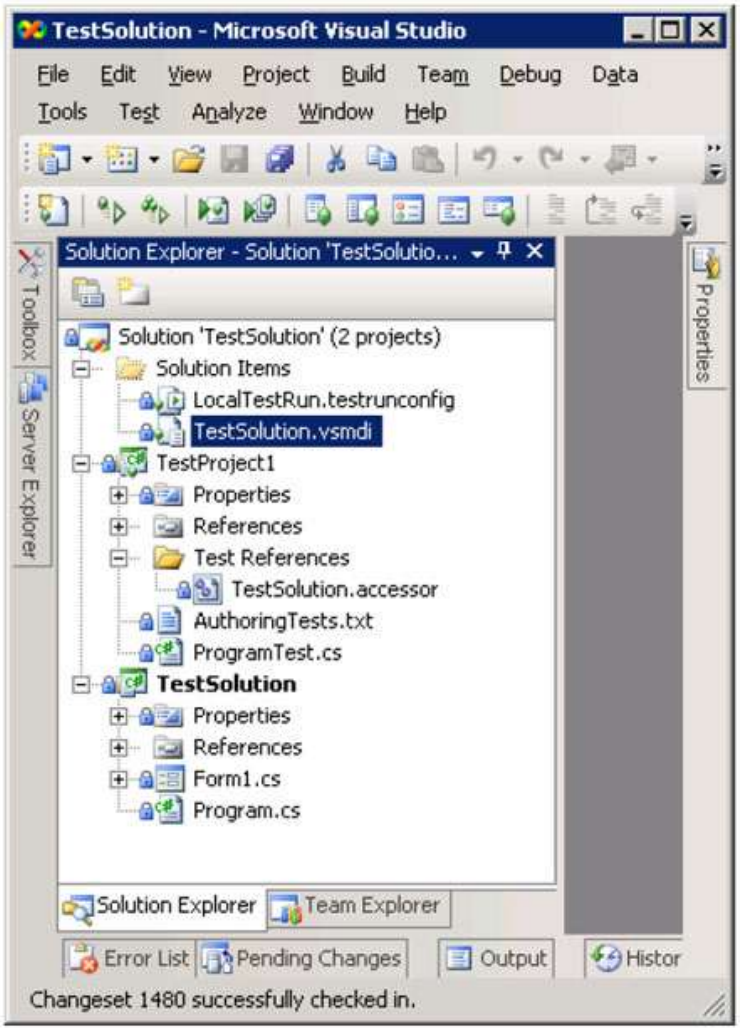

Рисунок 2 – Пакет с иерархией тестов

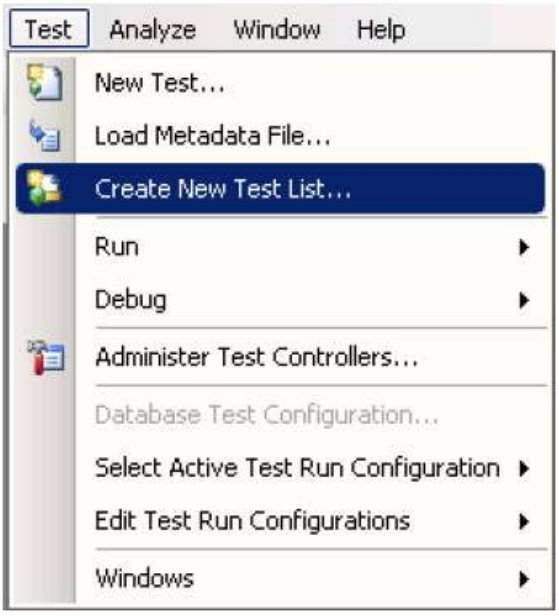

Рисунок 3 – Создание списка тестов

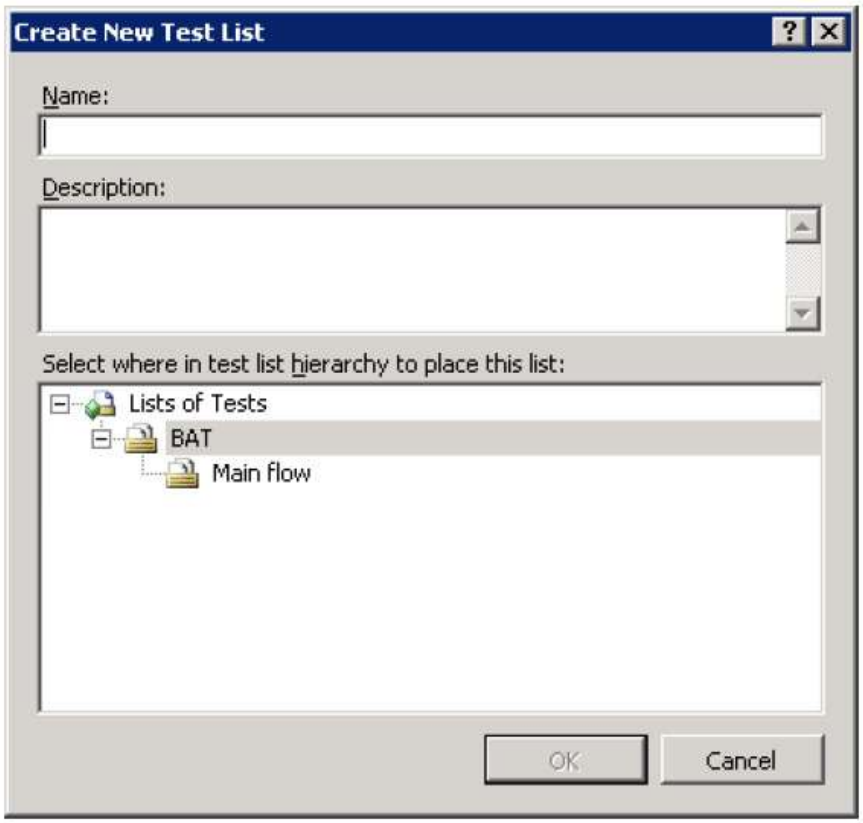

Рисунок 4 – Свойства нового списка тестов

Содержимое пакетов тестов редактируется с помощью специального редактора, показанного на рисунке 5. В пакеты могут быть включены тесты, находящиеся в одном из проектов текущего решения.

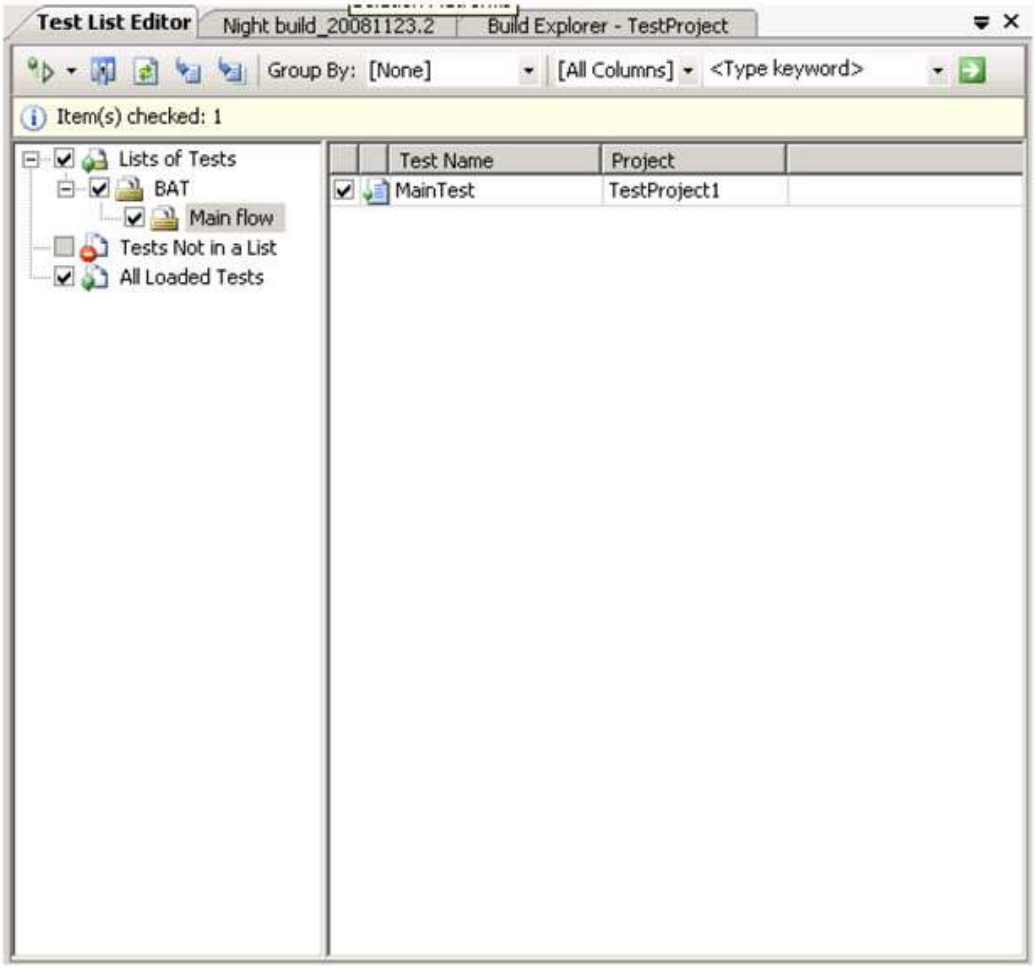

Рисунок 5 – Редактор список тестов

Тестовые пакеты могут использоваться как для ручного прогона тестов определенной тематики (команда "Run checked", рисунок 6), так и для автоматического прогона в рамках автоматической сборки.

Рисунок 6 – Ручной" запуск пакета тестов

Указать тесты, которые будут запускаться при определенной сборке можно при создании файла с описанием сборки MsBuild, или в последствии через модификацию проекта MsBuild. В первом случае достаточно на соответствующем шаге мастера выбрать файл метаданных и отметить галочками интересующие пакеты тестов (рисунок 4.6). Во втором случае необходимо открыть проект MsBuild в редакторе XML, найти элемент MetaDataFile, или вписать необходимые пакеты вручную:

<MetaDataFile
Include="$(BuildProjectFolderPath)/../../TestSolution/TestSolution.vsmdi">
<TestList>BAT/Main flow</TestList>
</MetaDataFile>

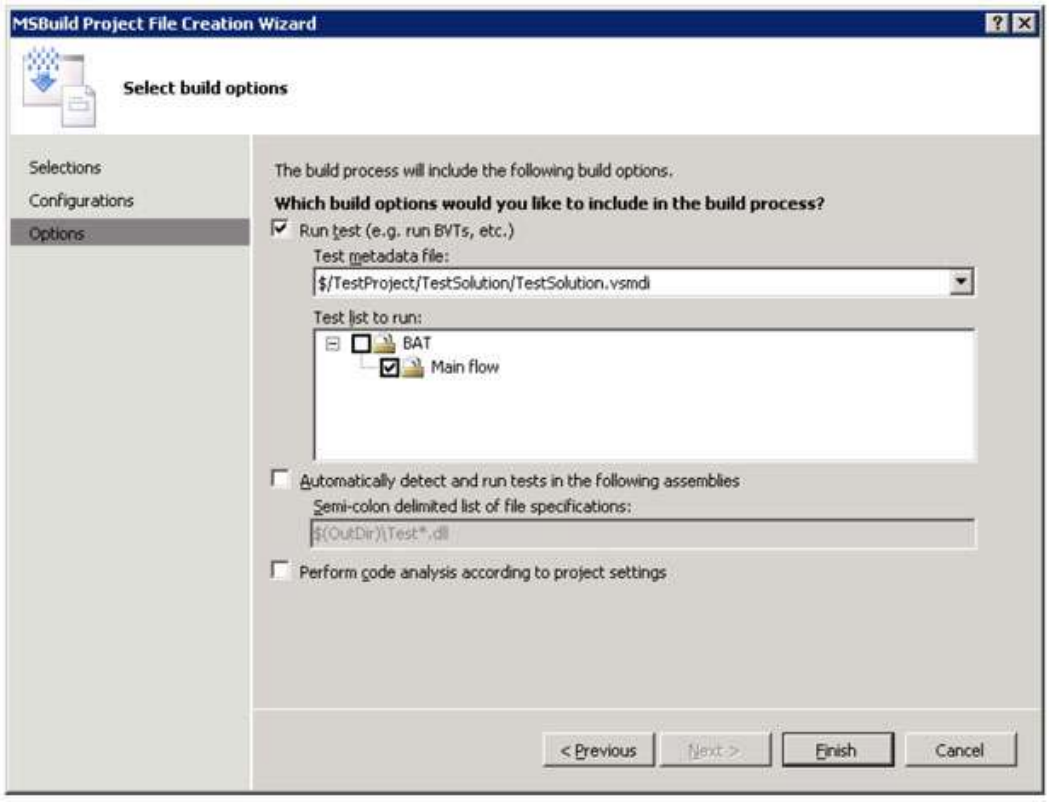

Рисунок 7 – Выбор пакета тестов при автоматической сборке

Автоматическое тестирование Web-приложений

Capture & Playback подход. Этот подход к тестированию пользовательских интерфейсов выглядит очень эффектно и основан на следующей идее. Тестировщик проходится мышкой по окнам, пунктам меню и другим элементам интерфейса. Специальная программа записывает его шаги и потом их воспроизводит в пакетном режиме. То есть очень просто получаются повторяемые тесты. Технически это устраивается так.

Специальное тестовое окружение с той или иной точностью распознает, куда именно было нажато мышкой на экране и создает соответствующий код в специальном скрипте. Потом этот скрипт "прогоняется" в пакетном режиме, воспроизводя действия тестировщиков. Весь вопрос в том, каким образом распознается клик тестировщика мышкой. Идеально, когда этот клик связывается с соответствующим элементом управления интерфейса. То есть если тестировщик нажал кнопку в каком-то диалоге, то в скрипте эта информация сохраняется в полном объеме. Другая, более грубая ситуация имеет место тогда, когда тестовое окружение не может распознать, какой элемент пользовательского интерфейса активировал тестировщик. Тогда в скрипт заносится информация о тех координатах на экране, куда был клик мышкой.

Чем плоха последняя ситуация? Дело в том, что при малейшем изменении пользовательского интерфейса (а это типичная ситуация, ведь ПО развивается, дорабатывается и тестируется одновременно) автоматический тест-скрипт, созданный таким образом, выходит из строя. Там, куда раньше клик мышкой попадал, например, на нужную кнопку, теперь находится совсем другой элемент управления.

Если же тестовое окружение распознало элемент интерфейса, то такой тест-скрипт оказывается более "живучим". Это происходит на уровне перехвата сообщений на уровне операционной системы (в частности, Windows). Но для того, чтобы это было возможно, код приложения должен быть написан "правильным" образом. Далеко не все интерфейсные приложения написаны "правильно".

То, насколько успешно для конкретного приложения можно применить данный подход, определяется несколькими факторами. Основным является то, какая используется платформа (например, Java Swing, AWT, Windows Forms, WPF, etc.) и дополнительные библиотеки с элементами пользовательского интерфейса. Наиболее зрелой платформой с этой точки зрения на данный момент является MS Windows Forms.

Capture & Playback при тестировании Web-интерфейсов. В случае с тестированием Web-интерфейса ситуация с точностью распознавания элементов управления (interface controls) значительно проще, чем при тестировании произвольного пользовательского интерфейса. Взаимодействие с сервером происходит по строго описанному протоколу HTTP, что позволяет при записи перехватить отправляемые и получаемые сообщения. Кроме того, визуальное представление страницы задано в структурированном формате HTML, что позволяет легко опознать отдельные элементы на странице.

В издание Visual Studio Team Edition for Software Testers включен дополнительный пакет, облегчающий автоматизацию тестирования Web-приложений методом Capture & Playback. Он позволяет, как автоматически генерировать простые тестовые сценарии на основе записи действия пользователя, так и писать более точные тесты на любом языке платформы .NET.

Добавить новый Web-тест к решению можно с помощью команды "Test/New Test", выбрав в возникшем диалоге (рисунок 8) тип теста Web Test.

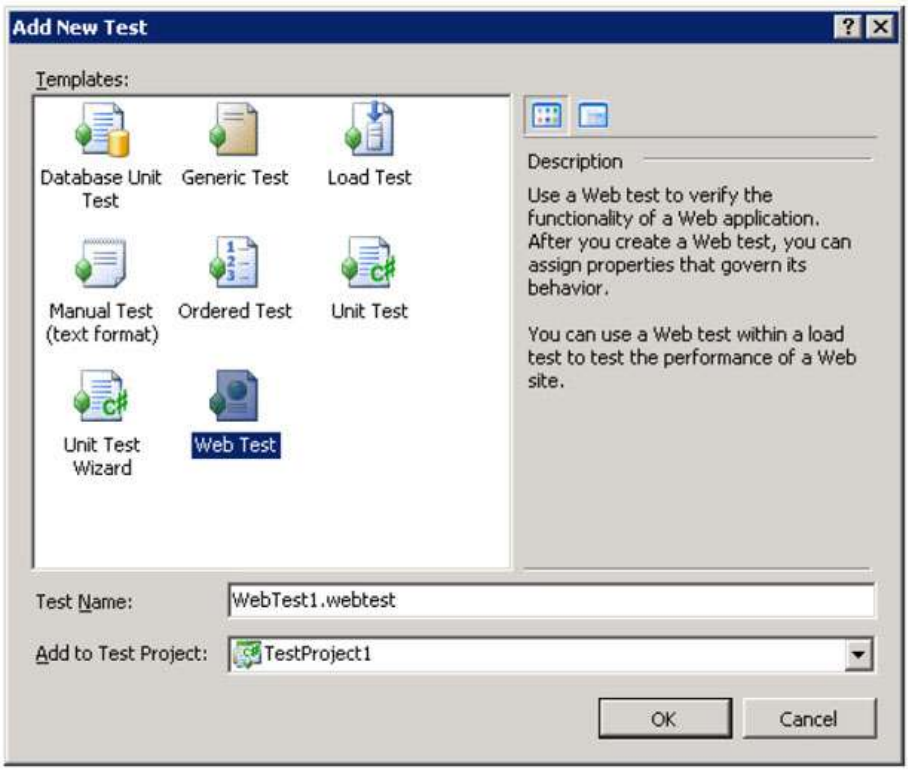

Рисунок 8 – Создание Web-теста

После создания нового теста автоматически будет запущена процедура записи сценария теста в браузере (рисунок 9). На этом этапе достаточно ввести www-адрес приложения, которое следует протестировать, после чего выполнить тестовый "проход" по Web-интерфейсу непосредственно в браузере.

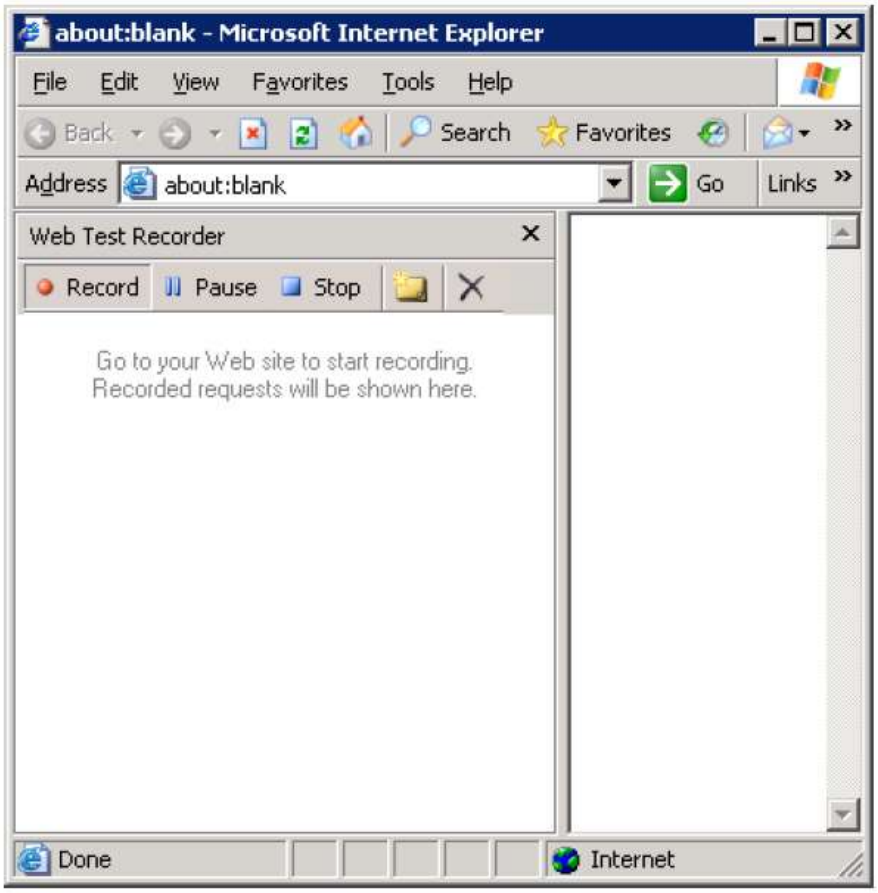

Рисунок 9 – Запись шагов тестировщика Web-приложения в Internet Explorer

После окончания записи будет автоматически сгенерирован тест, включающий все отправленные на сервер http-запросы и все полученные ответы (рисунок 10). При этом генератор автоматически добавит некоторые правила, по которым будет проверяться корректность работы теста.

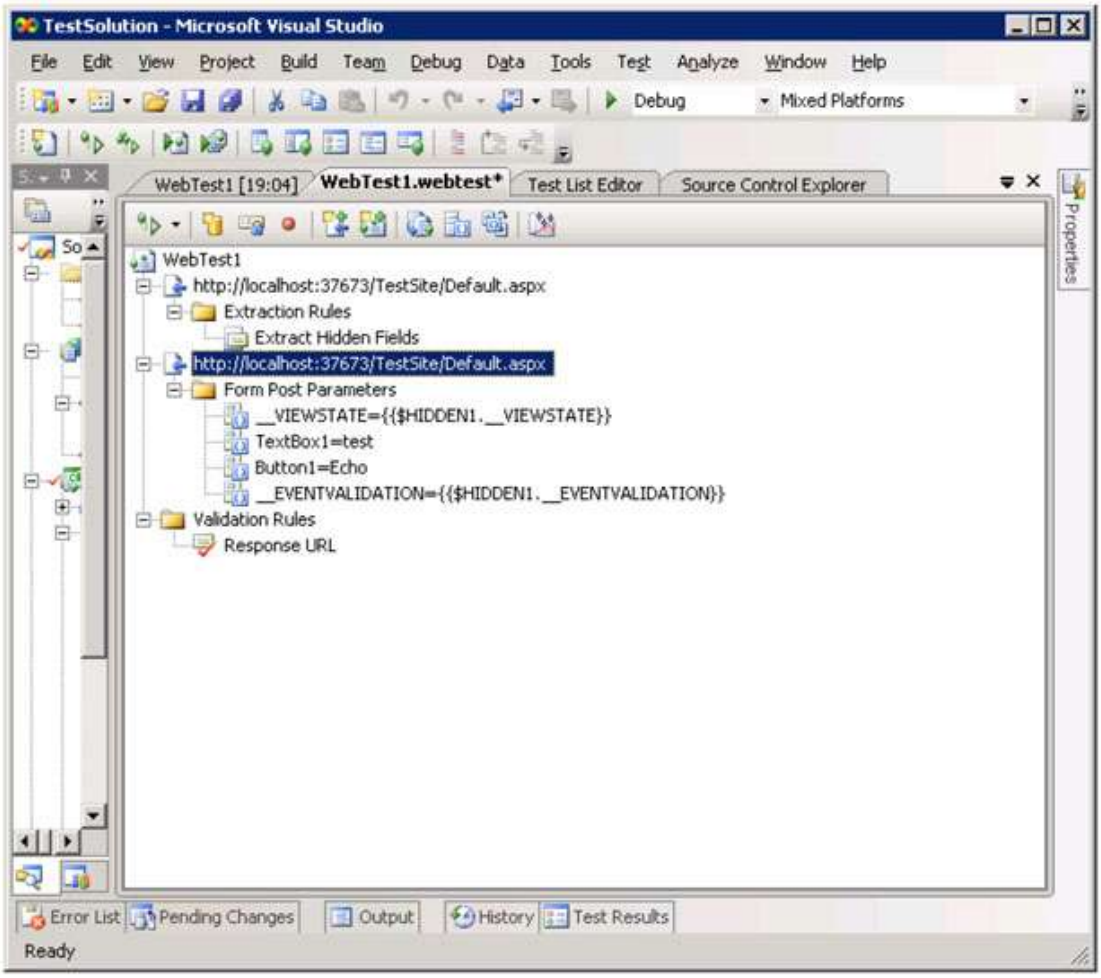

Рисунок 10 – Редактор Web-теста

Для каждого из шагов теста можно, с помощью визуального редактора, добавить дополнительные проверки или опции, управляющие ходом выполнения: поиск подстроки в ответе сервера (тексте полученного HTML), валидацию HTML через задание регулярного выражения, проверка наличия или отсутствия определенных тегов или атрибутов на странице и т.д. Допускается также возможность разработки собственных правил на любом .NET языке. В тех же случаях, когда гибкости редактора недостаточно, можно сгенерировать C# код для данного теста и реализовать необходимую логику "вручную".

При написании правил валидации очень важно правильно выбрать необходимый уровень детализации. Чем более детально сформулировано правило и чем более специфично оно для данного HTML, тем больше вероятность того, что тест придется изменять при изменении кода тестируемого приложения, даже если эти изменения не касались напрямую этой части. Хорошее правило валидации должно проверять только то, что является важной частью бизнес логики приложения или то, что является ключевым свойством данной HTML-страницы. При этом правило не должно проверять детали верстки и дополнительные визуальные эффекты. К сожалению, автоматически сгенерированные правила далеко не всегда оказываются наиболее эффективными.

Созданный в редакторе или в ручную тест является полноправным тестом и может быть включен в тестовый пакет, а следовательно, и в процедуры автоматической сборки (рисунок 11). Однако, для того, чтобы автоматическое тестирование было возможным, необходимо соответствующим образом настроить сервер автоматических сборок – на нем должен быть развернут сервер IIS с тестовым сайтом, а одним из этапов сборки должно быть обновление кода этого сайта.

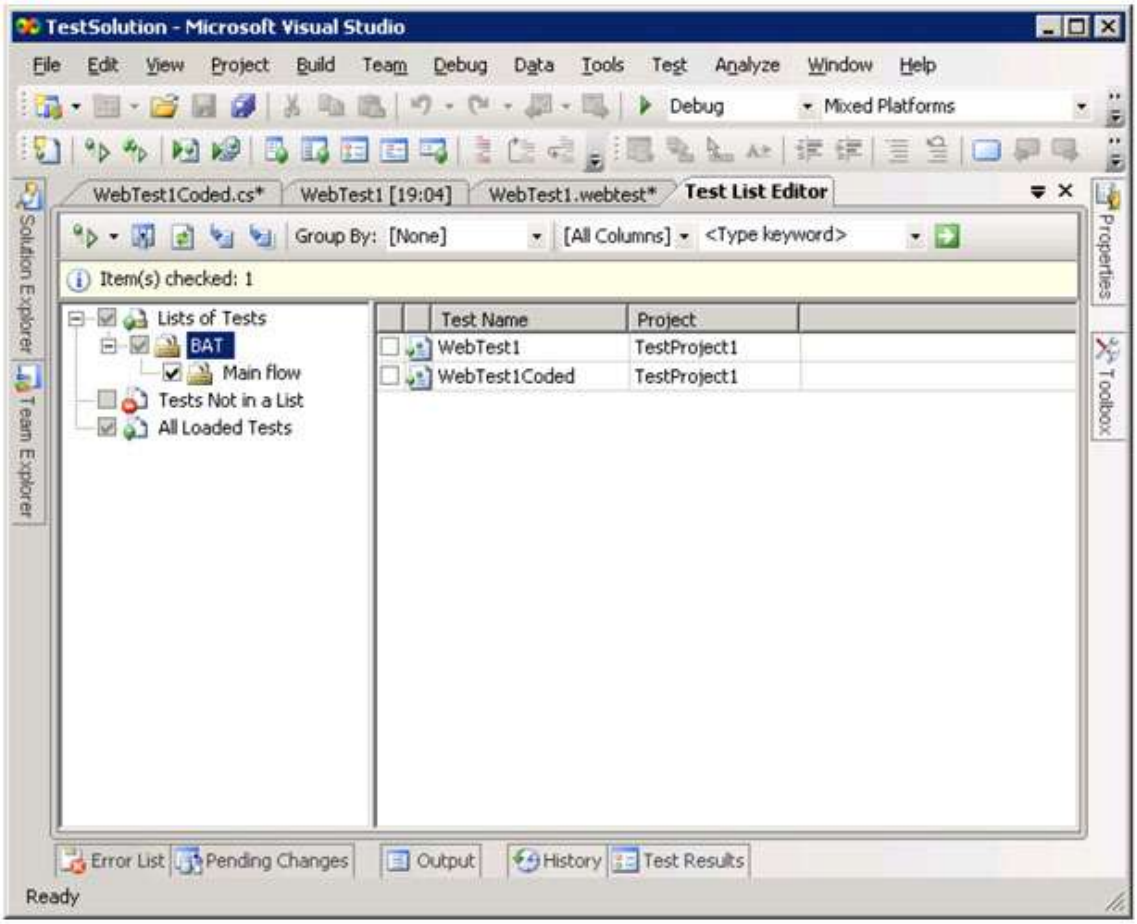

Рисунок 11 – Web-тесты в пакетах тестов

Форма представления результата:

1. Цель работы
2. Введение
3. Программно-аппаратные средства, используемые при выполнении работы.
4. Описание работы
5. Заключение (выводы)
6. Список используемой литературы

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Тема 4.1 Управление версиями программных продуктов

Практическая работа № 11
Основы работы в системе контроля версий git

Цель: приобрести базовые навыки работы в системе контроля версий git.

Выполнив работу, Вы будете:

уметь:

- У04.4. выбирать оптимальный формат, способ и место хранения информации и данных с помощью цифровых инструментов;
- У05.1. использовать средства информационно-коммуникационных технологий для решения профессиональных задач;
- У05.2. использовать специализированное программное обеспечение;

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio, система контроля версий git, GUI-клиент git.

Задание:

Изучите и настройте систему контроля версий при выполнении проекта в среде Visual Studio для разрабатываемого программного продукта.

Порядок выполнения работы:

1. Изучите предлагаемый теоретический материал по теме.
2. Настройте систему контроля версий для разрабатываемого программного продукта, используя средства системы контроля верси git,.
3. Настройте систему контроля версий для разрабатываемого программного продукта, используя GUI-клиент git.
4. Настройте систему контроля версий для разрабатываемого программного продукта при выполнении проекта в среде Visual Studio.
5. Выбрать модули для рефакторинга кода, провести рефакторинг кода.

Ход работы:

Краткие теоретические сведения

Начиная с Visual Studio 2013 Update 1, пользователям Visual Studio доступен Git-клиент, встроенный непосредственно в IDE. Visual Studio уже в течение достаточно долгого времени имеет встроенные функции управления исходным кодом, но они были ориентированы на централизованные системы с блокировкой файлов, и Git не очень хорошо вписывался в такой рабочей процесс. Поддержка Git в Visual Studio 2013 была существенно переработана по сравнению со старой версией, и в результате удалось добиться лучшей интеграции Visual Studio и Git.

Чтобы воспользоваться этой функциональностью, откройте проект, который управляется Git (или выполните git init для существующего проекта) и выберите пункты View (Вид) > Team Explorer (Командный обозреватель) в главном меню. В результате откроется окно "Connect" ("Подключить"), которое выглядит примерно вот так:

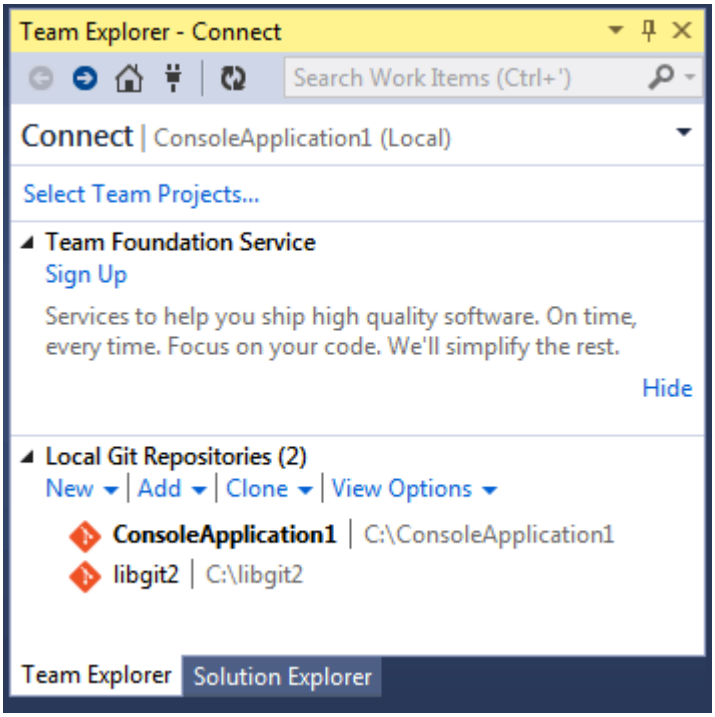

Рисунок 1 – Подключение к Git-репозиторию из окна Team Explorer (Командный обозреватель)

Visual Studio запоминает все проекты, управляемые с помощью Git, которые Вы открыли, и они доступны в списке в нижней части окна. Если в списке нет проекта, который вам нужен, нажмите кнопку "Add" ("Добавить") и укажите путь к рабочей директории. Двойной клик по одному из локальных Git-репозиториев откроет главную страницу репозитория, которая выглядит примерно так "Home" ("Главная") страница Git-репозитория в Visual Studio..

Это центр управления Git; когда вы пишете код, вы, вероятно, проводите большую часть своего времени на странице "Changes" ("Изменения"), но когда приходит время получать изменения, сделанные вашими коллегами по работе, вам необходимо использовать страницы "Unsynced Commits" ("Несинхронизированные коммиты") и "Branches" ("Ветви").

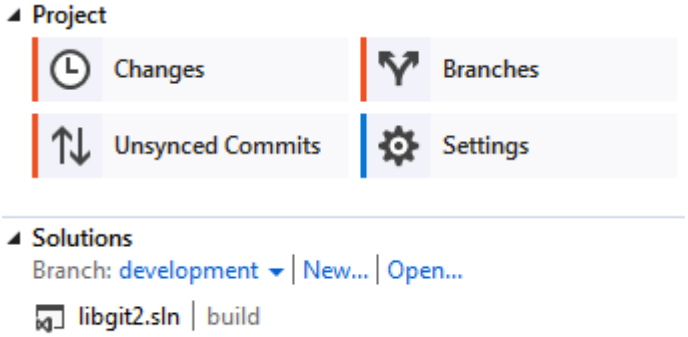

Рисунок 2 – «Home» («Главная») страница Git-репозитория в Visual Studio

В настоящее время Visual Studio имеет мощный задача-ориентированый графический интерфейс для Git. Он включает в себя возможность линейного представления истории, различные средства просмотра, средства выполнения удалённых команд и множество других возможностей. Для просмотра полной документации по данной функциональности (которая здесь не представлена), перейдите на http://msdn.microsoft.com/en-us/library/hh850437.aspx.

Форма представления результата:

Подготовить отчет с линейным представлением истории,

Перечислить различные средства просмотра,

Перечислить средства выполнения удалённых команд, с которыми Вы работали при выполнении проекта.

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».

Тема 4.1 Управление версиями программных продуктов

Практическая работа № 12
Изучение работы в системе контроля версий git. Ветвление. Отмена изменений

Цель: углубить навыки работы в системе контроля версий git в части ветвления и отмены изменений.

Выполнив работу, Вы будете:

уметь:

- У04.4. выбирать оптимальный формат, способ и место хранения информации и данных с помощью цифровых инструментов;
- У05.1. использовать средства информационно-коммуникационных технологий для решения профессиональных задач;
- У05.2. использовать специализированное программное обеспечение;

Пакет Microsoft Office, доступ к Internet ресурсам, Visual Studio, система контроля версий git, GUI-клиент git.

Задание:

В процессе разработки программного продукта в среде Visual Studio проведите ветвление, слияние и отмену изменений. Ведите логирование ваших действий с git-репозиторием.

Порядок выполнения работы:

1. Изучите предлагаемый теоретический материал по теме.
2. В процессе разработки программного продукта в среде Visual Studio проведите ветвление, слияние и отмену изменений. Ведите логирование ваших действий с git-репозиторием.

Ход работы:

Краткие теоретические сведения

Начиная с Visual Studio 2013 Update 1, пользователям Visual Studio доступен Git-клиент, встроенный непосредственно в IDE. Visual Studio уже в течение достаточно долгого времени имеет встроенные функции управления исходным кодом, но они были ориентированы на централизованные системы с блокировкой файлов, и Git не очень хорошо вписывался в такой рабочей процесс. Поддержка Git в Visual Studio 2013 была существенно переработана по сравнению со старой версией, и в результате удалось добиться лучшей интеграции Visual Studio и Git.

Чтобы воспользоваться этой функциональностью, откройте проект, который управляется Git (или выполните git init для существующего проекта) и выберите пункты View (Вид) > Team Explorer (Командный обозреватель) в главном меню. В результате откроется окно "Connect" ("Подключить"), которое выглядит примерно вот так:

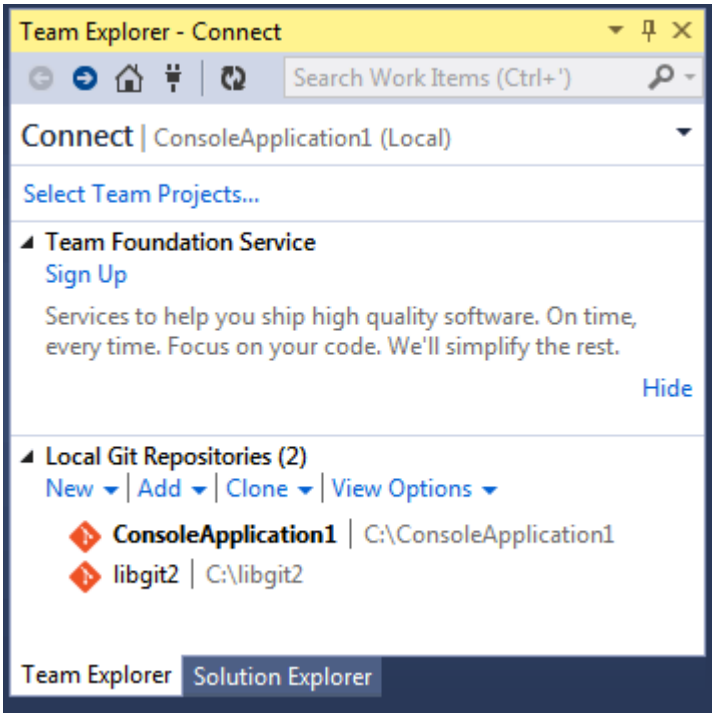

Рисунок 1 – Подключение к Git-репозиторию из окна Team Explorer (Командный обозреватель)

Visual Studio запоминает все проекты, управляемые с помощью Git, которые Вы открыли, и они доступны в списке в нижней части окна. Если в списке нет проекта, который вам нужен, нажмите кнопку "Add" ("Добавить") и укажите путь к рабочей директории. Двойной клик по одному из локальных Git-репозиториев откроет главную страницу репозитория, которая выглядит примерно так "Home" ("Главная") страница Git-репозитория в Visual Studio..

Это центр управления Git; когда вы пишете код, вы, вероятно, проводите большую часть своего времени на странице "Changes" ("Изменения"), но когда приходит время получать изменения, сделанные вашими коллегами по работе, вам необходимо использовать страницы "Unsynced Commits" ("Несинхронизированные коммиты") и "Branches" ("Ветви").

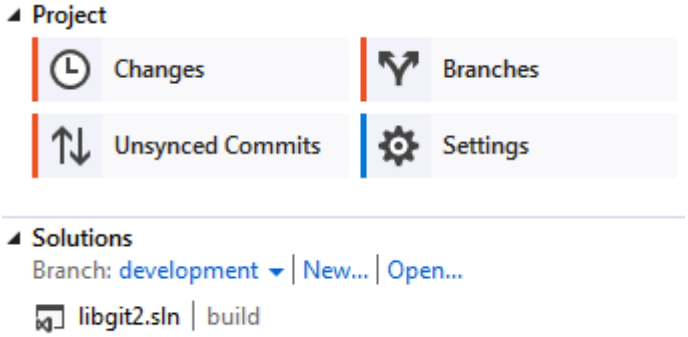

Рисунок 2 – «Home» («Главная») страница Git-репозитория в Visual Studio

В настоящее время Visual Studio имеет мощный задача-ориентированый графический интерфейс для Git. Он включает в себя возможность линейного представления истории, различные средства просмотра, средства выполнения удалённых команд и множество других возможностей. Для просмотра полной документации по данной функциональности (которая здесь не представлена), перейдите на http://msdn.microsoft.com/en-us/library/hh850437.aspx.

Форма представления результата:

Подготовить отчет с линейным представлением истории,

Перечислить различные средства просмотра,

Перечислить средства выполнения удалённых команд, с которыми Вы работали при выполнении проекта.

Критерии оценки:

Работа выполнена полностью и не содержит ошибок, студент грамотно представил отчет – оценка «отлично».

Работа выполнена полностью, но содержит не более двух ошибок, студент грамотно представил отчет – оценка «хорошо».

Работа выполнена с ошибками, студент представил краткий отчет – оценка «удовлетворительно».

Работа выполнена с грубыми ошибками, отчет составлен неграмотно – оценка «неудовлетворительно».