



ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания к самостоятельной
и лабораторным работам

Горяинов А.Е., Калентьев А.А.

2018

Краткое содержание

1	Введение	4
2	Лабораторные работы	5
2.1	Развертка внутренней инфраструктуры разработки	7
2.2	Разработка бизнес-логики приложения	44
2.3	Разработка пользовательского интерфейса	72
2.4	Юнит-тестирование	106
2.5	Функциональное расширение и релиз проекта	127
2.6	Составление проектной документации	139
3	Варианты заданий	161
3.1	Записная книжка NoteApp	161
3.2	Контакты ContactsApp	167
4	Техники разработки программного обеспечения	173
4.1	Системы версионного контроля	173
5	Методические указания к самостоятельной работе.....	182
6	Глоссарий	183
7	Список литературы	184

Полное содержание

1	Введение	4
2	Лабораторные работы	5
2.1	Развёртка внутренней инфраструктуры разработки	7
2.1.1	Внутренняя инфраструктура разработки	7
2.1.2	Версионный контроль.....	10
2.1.3	Работа с репозиторием с использованием GitHub и Visual Studio.....	17
2.1.4	Модель ветвления GitFlow.....	38
2.1.5	Задание	39
2.1.6	Вопросы для самоподготовки.....	41
2.1.7	Содержание отчета	41
2.1.8	Список источников.....	42
2.2	Разработка бизнес-логики приложения	44
2.2.1	Особенности разработки логики приложения.....	44
2.2.2	Нотация оформления кода RSDN.....	52
2.2.3	Xml-комментарии	57
2.2.4	Добавление ссылок на проекты внутри решения	60
2.2.5	Добавление ссылок на сторонние библиотеки через NuGet	64
2.2.6	Библиотека сериализации Newtonsoft JSON.NET	66
2.2.7	Задание	67
2.2.8	Вопросы для самоподготовки.....	69
2.2.9	Содержание отчета	69
2.2.10	Список источников	70
2.3	Разработка пользовательского интерфейса	72
2.3.1	Формы и элементы управления Windows Forms	72
2.3.2	Получение данных со стандартных элементов управления	83
2.3.3	Стандартные формы MessageBox, SafeFileDialog, OpenFileDialog	89
2.3.4	Использование проекта бизнес-логики в пользовательском интерфейсе	92
2.3.5	Передача данных между формами.....	95
2.3.6	Верстка	100
2.3.7	Задание	102
2.3.8	Вопросы для самоподготовки.....	103
2.3.9	Содержание отчета	104
2.3.10	Список источников	104
2.4	Юнит-тестирование	106
2.4.1	Организация тестирования в проекте.....	106
2.4.2	Общие принципы написания юнит-тестов.....	108
2.4.3	Цикломатическая сложность и определение требуемых тестов	112
2.4.4	Написание юнит-тестов с использованием NUnit.....	115
2.4.5	Задание	123
2.4.6	Вопросы для самоподготовки.....	124
2.4.7	Содержание отчета	125
2.4.8	Список источников	125
2.5	Функциональное расширение и релиз проекта	127
2.5.1	Сборка установочного пакета.....	127
2.5.2	Автоматизация сборки установочного пакета.....	131
2.5.3	Приёмочное тестирование.....	135
2.5.4	Задание	136
2.5.5	Вопросы для самоподготовки.....	137
2.5.6	Содержание отчета	137
2.5.7	Список источников.....	137
2.6	Составление проектной документации	139
2.6.1	Виды документации на различных этапах разработки	139

2.6.2	Календарный план и смета проекта	144
2.6.3	Пояснительная записка.....	153
2.6.4	Проведение ретроспективы.....	156
2.6.5	Задание	158
2.6.6	Вопросы для самоподготовки.....	159
2.6.7	Содержание отчета	159
2.6.8	Список источников	159
3	Варианты заданий.....	161
3.1	Записная книжка NoteApp	161
3.2	Контакты ContactsApp	167
4	Техники разработки программного обеспечения	173
4.1	Системы версионного контроля	173
4.1.1	Работа с репозиторием через консоль.....	173
4.1.2	Работа с репозиторием сторонними приложениями.....	174
5	Методические указания к самостоятельной работе.....	182
6	Глоссарий	183
7	Список литературы	184

1 Введение

Данное пособие содержит курс лабораторных работ, направленных на освоение процесса разработки программного обеспечения. Последовательность лабораторных работ повторяет основные этапы разработки ПО: создание внутренней инфраструктуры для разработки ПО, написание логики приложения, разработка пользовательского интерфейса, автоматизация тестирования, сборка установочного пакета, проведение приёмочного тестирования, написание технической документации к программе, проведение ретроспективы.

Разработка начинается с получения технического задания с макетами пользовательского интерфейса и «комментариями» от руководителя команды (англ. team leader, или «тимлид»), поясняющими особенности требуемой реализации.

Таким образом, в ходе выполнения лабораторных работ, обучающий будет способен участвовать в командной разработке небольших приложений по всем этапам жизненного цикла приложения.

Данное пособие может быть использовано частично или полностью в курсах дисциплин по программированию (таких как «Объектно-ориентированное программирование», «Системное программное обеспечение», «Новые технологии в программировании», «Технология разработки программного обеспечения») следующих специальностей:

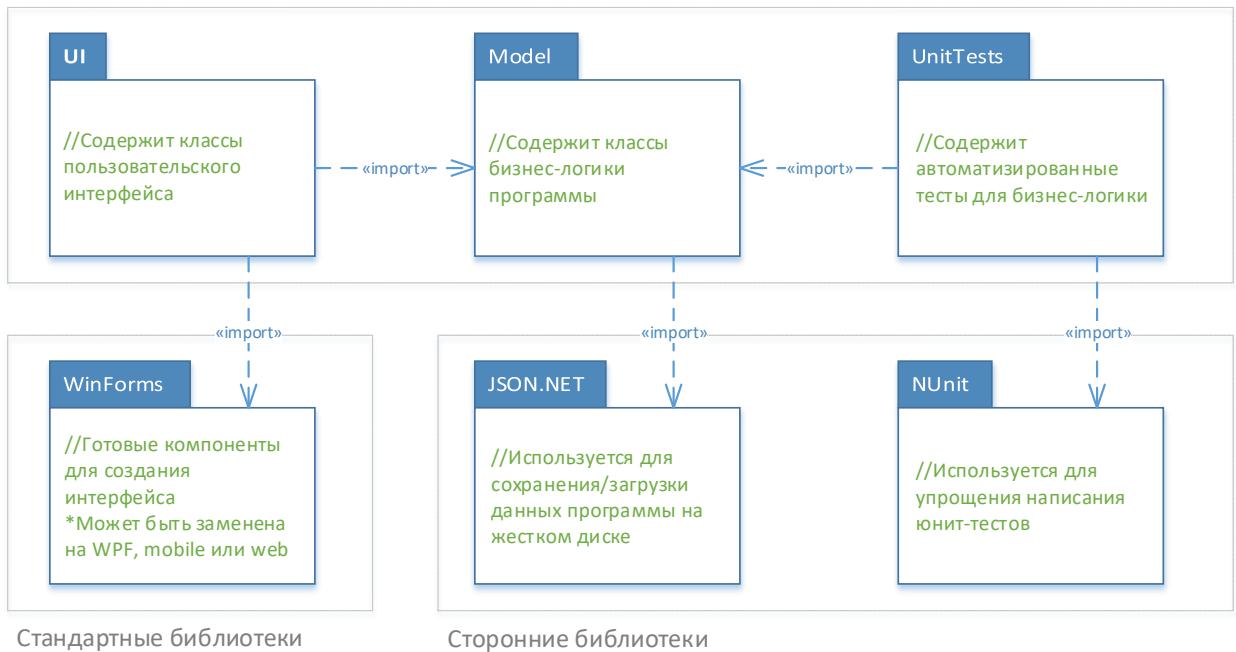
- 09.03.01 «Информатика и вычислительная техника»
- 09.03.02 «Информационные системы и технологии»
- 09.03.03 «Прикладная информатика»
- 09.03.04 «Программная инженерия»
- 15.03.04 «Автоматизация технологических процессов и производств»
- 27.03.02 «Управление качеством»
- 27.03.03 «Системный анализ и управление»
- 27.03.04 «Управление в технических системах»
- 38.03.05 «Бизнес-информатика»

и других специальностей, в учебный план которых включено изучение основ разработки ПО.

2 Лабораторные работы

Несмотря на различия в функциональном назначении приложений, многие архитектурные решения в приложениях схожи: это и разбиение программы на отдельные пакеты логики и пользовательского интерфейса, это и использование одинаковых сторонних библиотек. Обобщенная архитектура всех вариантов заданий представлена следующей диаграммой пакетов:

Разрабатываемые сборки



Стандартные библиотеки

Сторонние библиотеки

Как показано на диаграмме пакетов, изначально предполагается разработка приложений на языке C# с использованием набора компонентов Windows Forms. Несмотря на то, что использование Windows Forms в реальных проектах на 2018 год крайне мало, данный набор компонентов относительно прост в изучении и хорошо подходит для учебных задач. Однако, при желании обучающегося и готовности преподавателя, предлагаемые варианты заданий могут быть выполнены как с использованием другой платформы (WPF, мобильные приложения iOS и Android и др.), так и на другом языке программирования. Разумеется, это повлечет также и набор подключаемых библиотек, и набор используемых приложений, включая среду разработки.

В первую очередь, обучающемуся стоит ознакомиться с заданиями ко всем лабораторным работам – всем техническим заданием –, чтобы детально представлять разрабатываемое приложение и требуемый объём работы. Вариант задания назначается преподавателем.

Задания к лабораторным работам разбиты согласно естественным этапам разработки приложений:

- 1) **Развертка внутренней инфраструктуры разработки** – включает установку всех требуемых приложений, освоение системы версионного контроля Git с использованием сервиса GitHub и создание решения приложения в среде разработки.
- 2) **Разработка логики приложения** – включает написание классов, обеспечивающих хранение и обработку данных пользователя, их сохранение и загрузку на

жестком диске пользователя. Разрабатываемые классы должны отвечать требованиям оформления кода, документированности, а также защиты целостности данных.

- 3) ***Разработка пользовательского интерфейса*** – включает создание компонентов пользовательского интерфейса согласно представленным в техническом задании макетам. Интерфейс должен обеспечивать проверку и защиту от ввода некорректных данных.
- 4) ***Юнит-тестирование*** – включает написание юнит-тестов для автоматического тестирования логики приложения, написанной во второй лабораторной работе.
- 5) ***Функциональное расширение и релиз программы*** – включает разработку новых функциональных возможностей согласно ТЗ и замечаниям руководителя, сборку установщика и проведение приёмочного тестирования.
- 6) ***Техническая документация*** – включает написание трёх документов: пояснительной записки к разработанному проекту с UML-диаграммами, ретроспективы по процессу разработки, и календарного плана на разработку аналогичного приложения.

2.1 Развёртка внутренней инфраструктуры разработки

Цель работы: изучить пакет программ, используемых при разработке десктоп-приложений и получить умения их развертки на рабочей машине.

Задачи:

1. Ознакомиться с перечнем программ, используемых при разработке десктоп-приложений.
2. Установить требуемые приложения.
3. Изучить принцип работы с системами версионного контроля и модели ветвления при командной и индивидуальной разработке.
4. Создать репозиторий проекта.
5. Создать решение в репозитории.

2.1.1 Внутренняя инфраструктура разработки

Инфраструктура разработки ПО – набор программных и аппаратных средств, а также правила взаимодействия в команде, обеспечивающих процесс разработки ПО. Как правило, в инфраструктуру разработки входят: компьютеры разработчиков, сервер/серверы; программное обеспечение для написания кода, проектирования, тестирования, дизайна и макетирования и т.д.; перечень должностей в команде, их должностные обязанности и регламент взаимодействия участников разработки между собой – для поддержания дисциплины и контроля выполнения хода работы.

Чем больше команда разработчиков, тем более важную роль играет именно регламент взаимодействия участников. Это объясняется тем, что в большей команде происходит более явное разделение обязанностей, и также сильнее различия в опыте работы отдельно взятых участников. Например, если в команде из двух человек каждому приходится выполнять самые разные задачи – разработку, тестирование, дизайн – в зависимости от текущих потребностей, а для общения можно использовать обычный чат или даже поговорить напрямую; то в команде из ста человек приходится выделять отделы разработки, тестирования, поддержки и другие – где в каждом отделе есть собственный руководитель, перед которым более высоким руководством ставятся строго определенные задачи. Без строго определенной иерархии и дисциплины добиться эффективной разработки в больших коллективах невозможно. И чем больше команда, тем строже должна быть дисциплина.

Если же говорить о программных и аппаратных средствах, то и их ключевая цель – упростить выполнение этапов разработки и передачи результатов между этапами. Такие средства, как правило, направлены на решение наиболее распространенных, рутинных, но при этом затратных задач. Для того, чтобы понять, какие инструменты могут понадобиться в инфраструктуре, необходимо вспомнить ключевые этапы разработки:

- 1) Определение проблемы и **постановка задачи**.
- 2) **Составление технического задания**, планирование сроков и ресурсов на разработку.
- 3) **Макетирование** (разработка пользовательского интерфейса, прототипирование) – определение внешнего вида будущей системы.
- 4) **Проектирование** - определение всех необходимых внутренних компонентов будущей системы.

- 5) **Разработка** – реализация внешних и внутренних компонентов согласно макетам и проектной документации.
- 6) **Тестирование** – проверка соответствия разработанной системы исходному техническому заданию
- 7) **Внедрение** – передача готовой системы заказчику или конечным пользователям.

Стоит обратить внимание, что на практике линейность данных этапов не соблюдается, а некоторые из них подразделяются на более мелкие этапы. Так, например, тестирование разделяется на: планирование тестирования, проведение тестирования отдельно разработанных компонентов системы, тестирование всей системы. С практической точки зрения, не имеет смысла выполнять всё тестирование строго после разработки – чем позже обнаруживаются ошибки в системе, тем дороже стоит их исправление. Поэтому планирование тестирования выполняется сразу же после утверждения ТЗ или проектирования, тестирование отдельных компонентов выполняется параллельно разработке по мере готовности компонентов. И только тестирование всех системы выполняется после завершения разработки – хотя в случае обнаружения ошибок процесс снова возвращается к разработке.

Также надо помнить, что некоторые этапы могут быть и вовсе исключены из процесса. Например, в случае, если заказчик самостоятельно составил макеты пользовательского интерфейса и включил их в техническое задание, этап макетирования исключается. Исключению может подлежать любой этап в зависимости от требований заказчика или команды разработки.

Для каждого из этапов нужны собственные средства для решения внутренних проблем:

- 1) Составление технического задания – средства общения с заказчиком (чат, мессенджеры, почта и т.д.), программы для написания документов (текстовые редакторы), программы для оценки ресурсов, планирования (календари, специализированные калькуляторы, диаграммы Гантта), системы контроля спецификации (разделяют ТЗ на отдельные короткие спецификации с уникальными номерами, на которые в дальнейшем можно ссылаться в переписке между разработчиками)
- 2) Макетирование – графические редакторы (Photoshop, Visio), программы для создания скетчей и рабочих прототипов (например, InVision, Proto.io, Principle, Sketch). Если графические редакторы позволяют создавать только статичные картинки интерфейса, то такие приложения как InVision и Principle позволяют создавать динамические прототипы, которые пользователь может открыть в браузере или на мобильном телефоне, выполнять переходы между страницами и т.д. Такие прототипы могут отдаваться целевой аудитории, и они, например, попробовав прототип у себя на мобильном телефоне, могут сказать, удобно ли им такое приложение или нет.
- 3) Проектирование – программы для разработки диаграмм пакетов/классов и других видов чертежей (Spark Architect Enterprise [\[18\]](#), Draw.io, Visio). Помимо инструментов ручного создания диаграмм, такие программы как Architect Enterprise могут генерировать исходный код программы на основе диаграммы, а также создавать диаграммы на основе уже написанного исходного кода. Такая функциональность упрощает этапы разработки и составления проектной документации для уже разработанных программных систем.

- 4) Разработка – помимо сред разработки (Visual Studio [1], Xcode, Rider и др.), важную роль для данного этапа играют системы версионного контроля (git [6], mercurial и т.д.), системы управления проектом и системы непрерывной интеграции. Системы версионного контроля позволяют разработчикам сохранять промежуточные версии исходного кода, а также объединять исходный код нескольких разработчиков единую кодовую базу. Системы управления проектом позволяют распределять задачи по разработке среди программистов, контролировать сроки их выполнения. Системы непрерывной интеграции позволяют выполнять автоматическую сборку программы, её тестирование, а также сборку установочных пакетов или размещение веб-сервиса на удаленном рабочем сервере.
- 5) Тестирование – приложения для составления тестовых планов, приложения для записи сценариев тестирования – позволяют автоматизировать тестирование по заданному сценарию. Если сценарий в результате своего выполнения получит неправильный ожидаемый результат, то о невыполнении сценария будет сообщено тестировщику или разработчику.
- 6) Поддержка – системы контроля ошибок. Зачастую они внедрены в системы управления проектом, чтобы в единой системе контролировать все задачи по разработке/исправлению ПО. Однако в систему управления проектом доступ открыт только для разработчиков, то в систему контроля ошибок доступ часто открыт и для обычных пользователей. Так, любой пользователь, который обнаружил ошибку в программе, заносит её в систему контроля ошибок. После исправления ошибки, пользователю приходит уведомление об исправлении ошибки.
- 7) Внедрение – инструменты данного этапа в большей степени зависят от проекта (один заказчик или программа на продажу; десктоп-, веб- или мобильное приложение) и выделить какие-то конкретные инструменты сложно.

Чем сложнее проект и больше команда, тем больше инструментов задействованы в разработке. Например, если вы занимаетесь разработкой небольшого проекта на два-три месяца в команде из трёх человек - скорее всего вы не будете использовать системы контроля спецификаций, программы для разработки чертежей и диаграмм, сложные системы управления проектом и автоматизированное тестирование, но наверняка будете использовать систему версионного контроля и макетирование. Однако, в команде из пятидесяти человек и проектом с продолжительностью в год-два будут использованы все из описанных инструментов.

В ходе данной лабораторной работы предполагается развернуть минимально необходимую инфраструктуру для дальнейшей разработки ПО. В первую очередь, это система версионного контроля – наиболее востребованный инструмент в разработке ПО. По этой причине, большая часть лабораторной работы посвящена изучению данного инструмента.

В ходе выполнения лабораторных работ вам потребуются следующие программы и сервисы:

- 1) Среда разработки: *Microsoft Visual Studio 2018 Community* [1, 2, 3, 4].
- 2) Вспомогательные плагины для среды разработки: *JetBrains Resharper*.
- 3) Система версионного контроля: *git* с использованием сервиса *GitHub.com* [6, 7, 8].
- 4) Сборка установочных пакетов: *InnoSetup* [15, 16].
- 5) Microsoft Word.

- 6) Создание диаграмм технической документации: *Visio* или *Spark Enterprise Architect* [18, 19, 20, 21].

Краткое описание приложений:

Microsoft Visual Studio 2018 Community – бесплатная среда разработки. Несмотря на громоздкость по сравнению с аналогами (SharpDevelop, JetBrains Rider, Microsoft Visual Studio Code), данная среда содержит визуальные дизайнеры, которые в значительной степени упрощают верстку пользовательских интерфейсов приложений.

JetBrains Resharper – платная утилита, устанавливаемая поверх Visual Studio, содержащая полезные инструменты для написания кода. Утилита в реальном времени анализирует написанный вами код, находит в нём ошибки и предлагает варианты их исправления. Благодаря этому значительно ускоряется процесс написания кода, а также его качество. Также плагин имеет интеграцию с библиотеками автоматизированного тестирования, позволяя проводить тестирование ПО непосредственно в среде разработки Visual Studio. Для студентов вузов существует бесплатная лицензия, предоставляемая при отправке фотографии студенческого билета или указании персонального университетского e-mail.

GitHub.com – веб-сервис, позволяющий хранить промежуточные версии вашего исходного кода. Таким образом, в случае написания непоправимых ошибок в исходном коде или его потери, сервис позволит восстановить ваш проект. Также является обязательным инструментом для организации командной работы, используемым во многих ИТ-компаниях. Предоставляет бесплатную лицензию для малых проектов.

InnoSetup – бесплатное и относительно простое приложение для создания установочных пакетов для десктоп-приложений.

Microsoft Visio – платное приложение, дополняющее пакет программ Microsoft Office. Являясь векторным графическим редактором, подходит для создания типографической продукции, чертежей, диаграмм и планировок. Содержит готовые графические элементы для UML-диаграмм и блок-схем, что позволяет его использовать при разработке ПО, макетировании и прототипировании. Имеет бесплатный пробный период.

Enterprise Architect – платное приложение, специализированное для создания диаграмм в области разработки ПО. Содержит готовые графические элементы для создания большинства видов программных диаграмм, в том числе и в нотации UML. Важным преимуществом в сравнении с большинством аналогов, – это наличие инструментов для автоматической генерации диаграмм на основе исходного кода и генерации исходного кода на основе диаграмм. Имеет бесплатный пробный период.

2.1.2 Версионный контроль

Описание этой главы основано на книге [6]. Более подробную информацию о работе с Git и особенности его устройства можно взять напрямую из источника или из видео-курса [14].

Что такое управление версиями? Система управления версиями (СУВ) — это система, сохраняющая изменения в одном или нескольких файлах так, чтобы потом можно было восстановить определённые старые версии. Для примеров в этой книге мы будем использовать исходные коды программ, но на самом деле можно управлять версиями практически любых типов файлов.

Если вы графический или веб-дизайнер и хотите хранить каждую версию изображения или макета — вот это вам наверняка нужно — то пользоваться системой управления

версиями будет очень мудрым решением. Она позволяет вернуть файлы к прежнему виду, вернуть к прежнему состоянию весь проект, сравнить изменения с какого-то времени, увидеть, кто последним изменил модуль, который дал сбой, кто создал проблему, и так далее. Вообще, если, пользуясь СУВ, вы всё испортили или потеряли файлы, всё можно легко восстановить. Кроме того, издержки на всё это будут очень маленькими.

Локальные системы управления версиями. Многие люди, чтобы управлять версиями, просто копируют файлы в другой каталог (более продвинутые ещё пишут текущую дату в название каталога). Такой подход очень распространён, потому что прост, но он ещё и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать и перезаписать файлы не туда, куда хотел.

Чтобы решить эту проблему, программисты уже давно разработали локальные СУВ с простой базой данных, в которой хранятся все изменения нужных файлов (см. рис.1). Одной из наиболее популярных СУВ данного типа является rcs, которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита rcs устанавливается вместе с Developer Tools. Эта утилита основана на работе с наборами патчей между парами изменений (патч — файл, описывающий различие между файлами), которые хранятся в специальном формате на диске. Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи.

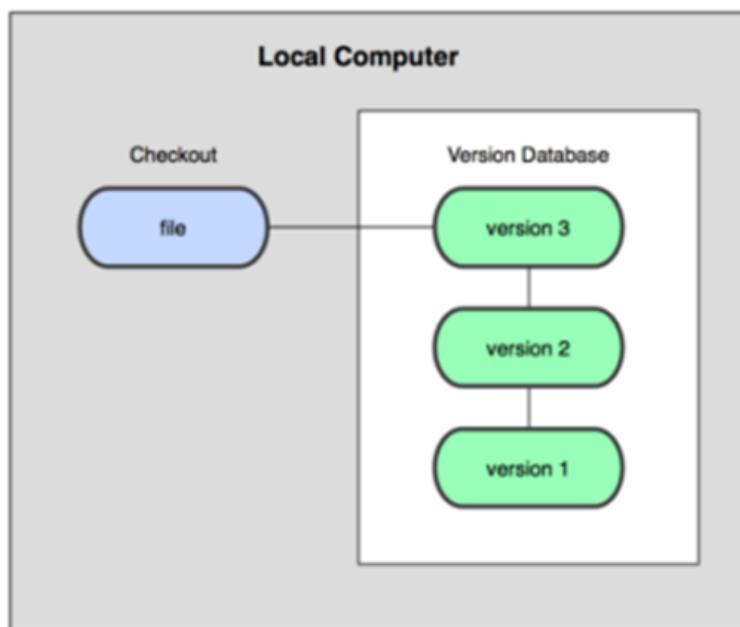


Рисунок 1 – Схема хранения версий на локальном компьютере

Централизованные системы управления версиями. Следующей большой проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы управления версиями (ЦСУВ). В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все отслеживаемые файлы, и ряд клиентов, которые получают копии файлов из него. Много лет это был стандарт управления версиями:

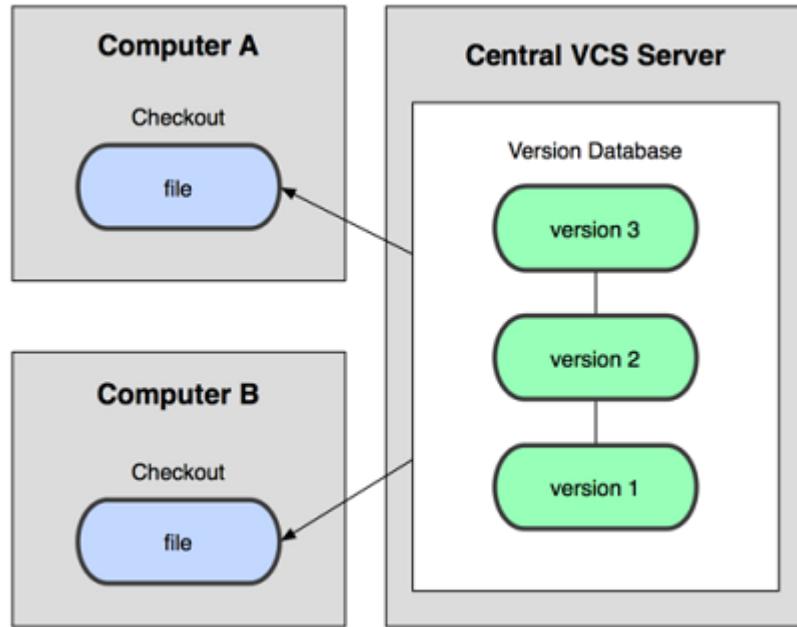


Рисунок 2 – Схема централизованной СУВ

Такой подход имеет множество преимуществ, особенно над локальными СУВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСУВ гораздо легче, чем локальные базы на каждом клиенте.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новые версии. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей. Локальные системы управления версиями подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределенные системы контроля версий. В такой ситуации применяют распределенные системы управления версиями (РСУВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто забирают последние версии файлов, а полностью копируют репозиторий. Поэтому в случае, когда по той или иной причине отключается сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, создаётся полная копия всех данных:

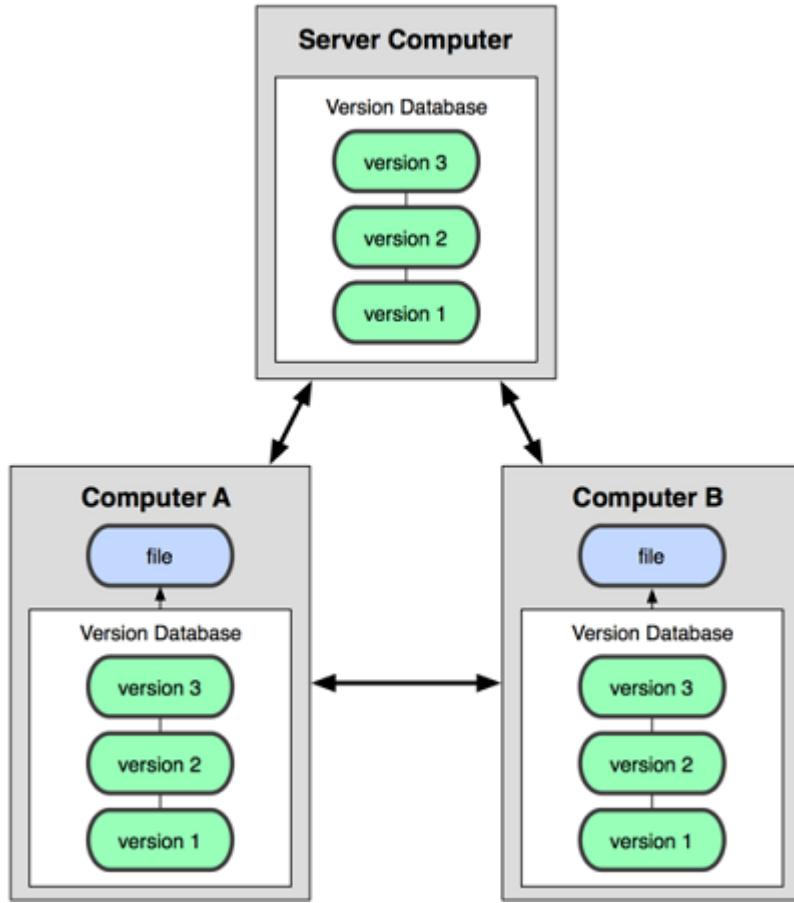


Рисунок 3 – Схема распределенной СУБ

Кроме того, в большей части этих систем можно работать с несколькими удаленными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

История появления Git. Своему появлению Git обязан проблемам, возникшим в ходе разработки ядра Linux. Ядро Linux — действительно очень большой открытый проект. Большую часть существования ядра Linux (1991-2002) изменения вносились в код путем приёма патчей и архивирования версий. В 2002 году проект перешёл на проприетарную РСУВ Bit-Keeper.

В 2005 году отношения между сообществом разработчиков ядра Linux и компанией, разрабатывавшей BitKeeper, испортились, и право бесплатного пользования продуктом было отменено. Это подтолкнуло разработчиков Linux (и в частности Линуса Торвальдса, создателя Linux) разработать собственную систему, основываясь на опыте, полученном за время использования BitKeeper. Основные требования к новой системе были следующими:

- Скорость;
- Простота дизайна;
- Поддержка нелинейной разработки (тысячи параллельных веток);
- Полная распределенность;
- Возможность эффективной работы с такими большими проектами как ядро Linux (как по скорости, так и по размеру данных).

С момента рождения в 2005 г. Git разрабатывали так, чтобы он был простым в использовании, сохранив свои первоначальные свойства. Он невероятно быстр, очень эффективен

для больших проектов, а также обладает превосходной системой ветвления для нелинейной разработки.

Особенности хранения данных в системе Git. Главное отличие Git от любых других СУВ (например, Subversion и ей подобных) — это то, как Git смотрит на данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени:

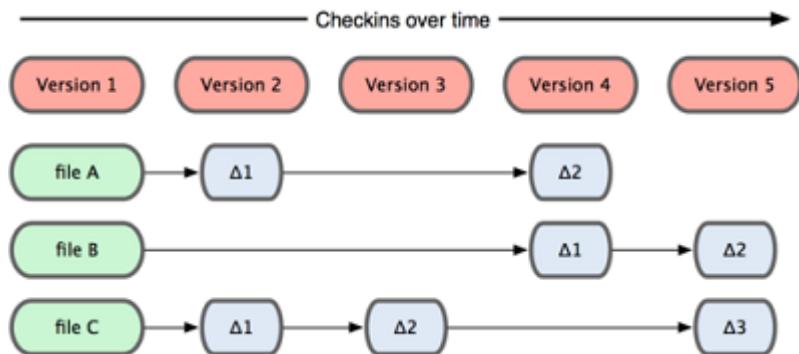


Рисунок 4 – Схема представления данных в других СУВ

Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл:

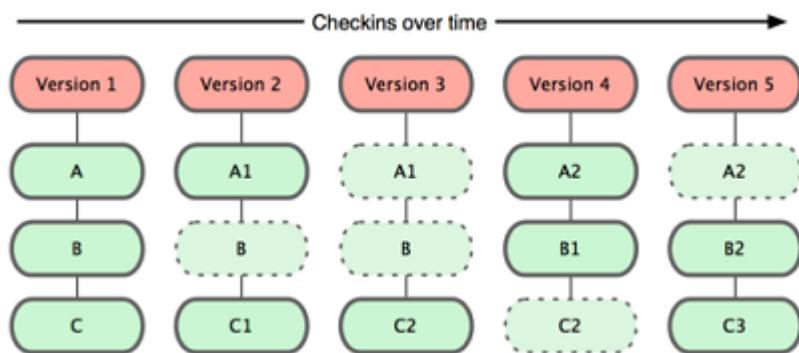


Рисунок 5 – Схема представления данных в Git

Это важное отличие Git от практически всех других систем управления версиями. Из-за него Git вынужден пересмотреть практически все аспекты управления версиями, которые другие системы взяли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто СУВ. В главе 3, коснувшись работы с ветвями в Git, мы узнаем, какие преимущества даёт такое понимание данных.

Почти все операции в Git – локальные. Для совершения большинства операций в Git необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Если вы пользовались централизованными системами, где практически на каждую операцию накладывается сетевая задержка, вы оцените скорость работы

с Git. Поскольку вся история проекта хранится локально у вас на диске, большинство операций выглядят практически мгновенными.

К примеру, чтобы показать историю проекта, Git-у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у сервера СУБ или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или VPN. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а VPN клиент не работает, всё равно можно продолжать работать. Во многих других системах это невозможно или же крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория). Вроде ничего серьёзного, но потом вы удивитесь, насколько это меняет дело.

Git следит за целостностью данных. Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм, используемый Git для вычисления контрольных сумм, называется SHA-1 хеш. Это строка из 40 шестнадцатеричных знаков (0-9 и a-f), которая вычисляется на основе содержимого файла или структуры каталога, хранимого Git. SHA-1 хеш выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Работая с Git, вы будете постоянно встречать эти хеши, поскольку они широко используются. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого.

Чаще всего данные в Git только добавляются. Практически все действия, которые вы совершаете в Git, только добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой СУБ, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их очень сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.

Поэтому пользоваться Git — удовольствие, потому что можно экспериментировать, не боясь серьёзно что-то поломать.

В Git файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. «Зафиксированный» значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проекте с использованием Git есть три части: каталог Git (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).

Каталог Git — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Стандартный рабочий процесс с использованием Git выглядит примерно так:

- 1) Вы изменяете файлы в вашем рабочем каталоге.
- 2) Вы подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
- 3) Вы делаете коммит. При этом слепки из области подготовленных файлов сохраняются в каталог Git.

Если рабочая версия файла совпадает с версией в каталоге Git, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

Сервис GitHub - – это один из крупнейших веб-сервисов для хостинга ИТ-проектов и их совместной разработки. Сервис основан на системе контроля версий Git.

Сервис бесплатен для проектов с открытым исходным кодом и предоставляет им все возможности (включая SSL), а для частных проектов предлагаются различные платные тарифные планы.

Для работы с сервисом необходимо зарегистрироваться, зайдя на [7]. После регистрации – заведите репозиторий (зайдите во вкладку Repositories и нажмите на New). При создании нового репозитория нужно определиться для чего он нужен, для публичных проектов или для личного закрытого использования, в зависимости от этого нужно выбрать Public или Private. При создании Private репозитория необходимо будет заплатить по тарифу 7\$ в месяц (стоимость на 5.09.2018). Введите имя репозитория и его описание и нажмите на зелёную кнопку Create repository.

После создания репозитория будет доступна возможность его клонирования (физического переноса файлов репозитория на локальную машину для дальнейшей работы или переноса репозитория на другой сервис контроля версий, поддерживающий Git). Клонирование возможно по двум протоколам HTTPS и SSH. Работа по HTTPS достаточно проста и будет рассмотрена ниже. Работа по SSH протоколу подразумевает наличия SSH ключей – публичного и приватного. Генерация и работа с последними не представляет никакой особой сложности, выполняется она, например, с помощью утилиты PUTTYgen. В данном пособии эти вопросы рассматриваться не будут, поэтому читатель может ознакомиться с генерацией и использованием ключей самостоятельно.

GitHub предоставляет большое количество сопутствующих сервисов для разработки, например отслеживание коммитов, просмотр различий между версиями файлов напрямую в браузере (при этом сервис поддерживает синтаксическую подсветку для большинства существующих языков программирования), использование сервиса в качестве системы управления проектами с возможностью создания задач, их отслеживания и т.п., создание документации на основе Wiki, отслеживание активности по проекту с помощью графиков и пр.

Создатели GitHub называют свой проект «социальной сетью для разработчиков». Кроме размещения кода, участники могут общаться, комментировать правки друг друга, а также следить за новостями знакомых. Работая с системами, подобными GitHub, начинающий программист развивает навыки изучения чужого кода и его критической оценки. Очень часто на ресурсе можно найти код именитых разработчиков, изучение которого будет приучать к правильным стандартам и приёмам кодирования. Помимо этого, достаточно опытный разработчик может поучаствовать в интересующем Open Source проекте, опять же развивая определённые профессиональные компетенции.

Сегодня очень часто при приёме на работу продвинутый работодатель, вместо выдачи тестового задания просит ссылку на публичный репозиторий на GitHub и делает вывод по находящимся там проектам о качествах кандидата-программиста. Поэтому освоение Git, как одной из СУБ и GitHub, как публичного хранилища репозиториев – позволит студенту иметь определённые навыки для дальнейшей работы разработчиком.

GitHub позволяет работать по системе ветвлений напрямую на ресурсе, но в методическом пособии будет рассмотрена локальная система ветвлений.

Применительно к лабораторным работам: использование GitHub или любого другого публичного веб-сервиса для хранения Git репозитория позволит преподавателю оценить работу студента с репозиторием.

Инструменты работы с Git. Для работы с Git репозиториями существует множество инструментов, выбор которых зависит от нескольких факторов: используемой операционной системы и привычного варианта использования программ (с помощью пользовательского интерфейса: Mac и Windows стиль, - или с помощью консоли - *nix системы). Стоит отметить, что опытные разработчики чаще используют управление Git-репозиторием с помощью командной строки. Также, в большинстве современных сред разработки есть встроенные инструменты для работы с системами версионного контроля.

Далее рассмотрим работу с git с использованием сервиса GitHub и встроенных средств Visual Studio.

2.1.3 Работа с репозиторием с использованием GitHub и Visual Studio

Для работы с сервисом GitHub и работы с его репозиториями необходимо зарегистрироваться. Если вы всё сделали верно и зашли в аккаунт, перед вами появится следующее окно:

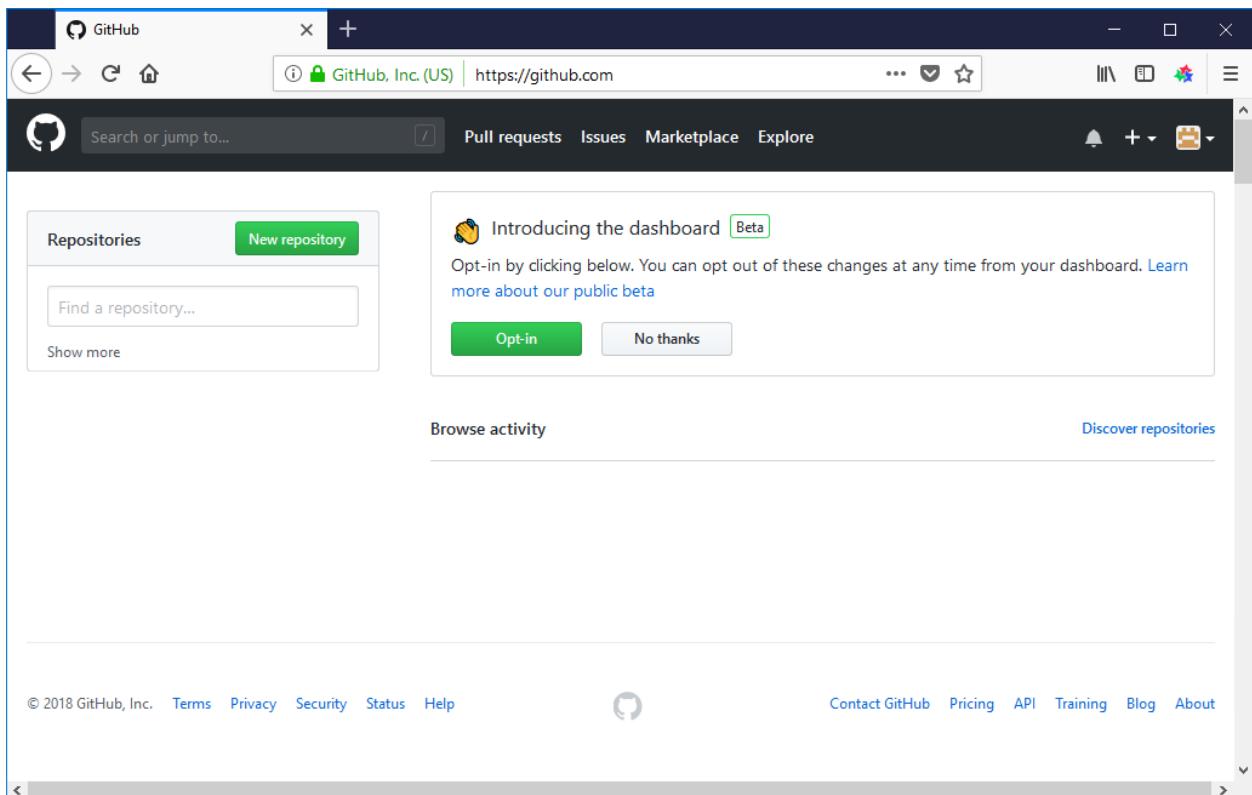


Рисунок 6 – Страница пользователя GitHub

В данном окне показывается информация о всех ваших текущих репозиториях и проектах, в которых вы участвуете на GitHub. Сразу же после регистрации данное окно будет пустым, как на представленном рисунке. Однако по мере вашей работы окно будет заполняться полезной статистикой.

После регистрации можно создавать собственные репозитории. Для создания репозитория необходимо нажать кнопку **New Repository** в левом верхнем углу или нажать на знак плюса в верхнем правом углу на панели пользователя. Создание репозитория происходит на отдельной странице (см. рис. далее). В данном окне необходимо указать:

- 1) название репозитория;
- 2) описание проекта;
- 3) указать доступность репозитория (открытый или закрытый репозиторий);
- 4) указать шаблон файла `gitignore`;
- 5) указать тип лицензии проекта.

Название должно осмысленным и отражать суть проекта. В противном случае: а) в будущем вы запутаетесь в собственных репозиториях; б) в ваших репозиториях запутаются люди, которых вы будете подключать к разработке своих будущих проектов.

Для бесплатных аккаунтов можно создавать только открытые репозитории – их исходный код сможет скачать любой пользователь GitHub, но не сможет делать коммиты без вашего разрешения. Закрытые репозитории позволяют указывать конкретных пользователей, которые смогут видеть исходный код, для остальных репозиторий будет невиден. Однако использование закрытого репозитория требует ежемесячной оплаты.

Стоит обратить внимание на файл `gitignore`. `Gitignore` это файл, который лежит в корневой папке репозитория и содержит список расширений файлов, которые не следует автоматически добавлять в репозиторий при коммитах. Например, в репозиторий не имеет

смысла добавлять результаты компиляции, находящиеся в папках bin проектов Visual Studio – так как эти файлы можно всегда получить с помощью компиляции, а хранение их в репозитории – лишняя нагрузка. Также в gitignore можно указать расширения для файлов настроек среды разработки – среда разработки создает файлы ваших настроек и помещает их в папку с проектом. Так как это ваши индивидуальные настройки среды, помещать их в репозиторий нет смысла. В противном случае, у другого разработчика, скачавшего ваш репозиторий, сбываются его собственные настройки среды. Более того, он сможет записать свои настройки в ваш репозиторий, и тогда настройки сбываются уже у вас.

Чтобы не писать файлы gitignore вручную, GitHub предлагает готовые шаблоны файлов для различных языков программирования и сред разработки. Так, в случае данного курса, вам необходимо выбрать шаблон Visual Studio.

The screenshot shows the GitHub interface for creating a new repository. At the top, there's a dark header bar with the GitHub logo, a search bar, and navigation links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the right side of the header are icons for notifications, a plus sign, and a gear for settings. Below the header, the main title 'Create a new repository' is displayed in bold. A sub-instruction below it says 'A repository contains all the files for your project, including the revision history.' The form fields include 'Owner' (set to 'goryainovalex'), 'Repository name' ('NoteApp'), and a 'Description (optional)' field containing the text 'NoteApp - приложение, разрабатываемое в рамках учебного курса "Технологии разработки ПО"'. There are two radio button options for visibility: 'Public' (selected) and 'Private'. Underneath, a checked checkbox labeled 'Initialize this repository with a README' has a descriptive note: 'This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.' At the bottom of the form are two buttons: 'Add .gitignore: VisualStudio' and 'Add a license: MIT License', followed by a small help icon. A large green 'Create repository' button is at the very bottom of the form.

Рисунок 7 – Страница создания репозитория в GitHub

Последним шагом указывается лицензия. Это может показаться ненужным шагом, но лицензию стоит указывать всегда. Даже если вы не хотите ограничивать использование своего кода, то в лицензии необходимо отказаться от ответственности в случае некорректной работы вашего кода. Иначе, если кто-то использует ваш код в своих коммерческих проектах, а затем из-за ошибки в вашем коде он потеряет пользователей и прибыль – он сможет подать на вас в суд. Законодательство некоторых стран это позволяет. Поэтому указывайте лицензию своих репозиториев. Если вы не хотите как-либо ограничивать использование

своего кода, выбирайте лицензию MIT License. С её переводом можно ознакомиться в Интернете. После нажатия кнопки Create repository вы перейдете на страницу вашего репозитория:

The screenshot shows a GitHub repository page for 'NoteApp' by 'goryainovalex'. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the header, the repository name 'goryainovalex / NoteApp' is displayed, along with buttons for 'Unwatch', 'Star', 'Fork', and a search bar. A banner below the header reads 'NoteApp - приложение, разрабатываемое в рамках учебного курса "Технологии разработки ПО"' with an 'Edit' button. The main content area shows repository statistics: 1 commit, 1 branch, 0 releases, and 1 contributor. Below these stats are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The file list includes '.gitignore', 'LICENSE', and 'README.md', all with 'Initial commit' status and 'just now' timestamp. The 'README.md' file is expanded, showing its content: 'NoteApp - приложение, разрабатываемое в рамках учебного курса "Технологии разработки ПО"'. At the bottom of the page, there are links for 'Contact GitHub', 'Pricing', 'API', 'Training', 'Blog', and 'About', along with a GitHub logo.

Рисунок 8 – Страница репозитория в GitHub

Изначально в вашем репозитории будет три файла: gitignore, файл лицензии и файл readme. Если ознакомитесь с данным окном, то можете заметить: вкладки проекта Code (просмотр исходного кода), Issues (список задач), Wiki (база знаний проекта), Settings (настройки репозитория). В ходе данного курса вам потребуется только вкладка Code, однако помните, что GitHub не только хранит код, и предоставляет инструменты для полноценного управления проектами и назначением задач команде разработчиков. Ознакомьтесь с ними самостоятельно.

Также пользователь может перейти на другую ветку, просмотреть историю коммитов текущей ветки, создать или загрузить файлы, найти файл, и, главное, клонировать репозиторий на свой компьютер.

Сервис GitHub позволяет просматривать текстовые файлы. Для этого необходимо нажать на имя файла, и его содержимое появится в новом окне. Для примера ознакомьтесь с содержимым файла gitignore:

The screenshot shows a GitHub repository page for 'goryainovalex / NoteApp'. The top navigation bar includes links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the right, there are icons for notifications, a plus sign, and a dropdown menu. Below the header, the repository name 'goryainovalex / NoteApp' is displayed, along with a star count of 1, a fork count of 0, and a 'Unwatch' button. A navigation bar below the repository name includes 'Code' (selected), 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Insights', and 'Settings'. The main content area shows a commit history for the '.gitignore' file. The first commit, made by 'goryainovalex' at '65c40a6 a minute ago', is titled 'Initial commit'. It has 1 contributor. The file contains 331 lines (265 sloc) and is 5.45 KB. The code content is as follows:

```
1 ## Ignore Visual Studio temporary files, build results, and
2 ## files generated by popular Visual Studio add-ons.
3 ##
4 ## Get latest from https://github.com/github/gitignore/blob/master/VisualStudio.gitignore
5
6 # User-specific files
7 *.suo
8 *.user
9 *.userosscache
10 *.sln.docstates
11
12 # User-specific files (MonoDevelop/Xamarin Studio)
13 *.userprefs
14
15 # Build results
16 [Dd]ebug/
17 [Dd]ebugPublic/
18 [Rr]elease/
19 [Rr]eleases/
20 x64/
21 x86/
22 bld/
23 [Bb]in/
24 [Oo]bj/
25 [Ll]og/
```

Рисунок 9 – Окно просмотра файла на примере gitignore в GitHub

Как видно, файл `gitignore` содержит перечень расширений файлов и имен папок, которые должны игнорироваться при добавлении файлов. Например, строки под номерами 16, 18 и 23 указывают, что версионный контроль должен игнорировать папки `debug`, `release` и `bin`. Как говорилось ранее, эти папки содержат результаты компиляции, а потому их хранение под версионным контролем не целесообразно.

Для клонирования репозитория на локальную машину пользователя необходимо нажать кнопку `Clone or download` и нажать кнопку `Open in Visual Studio` (среда разработки должна быть предварительно установлена):

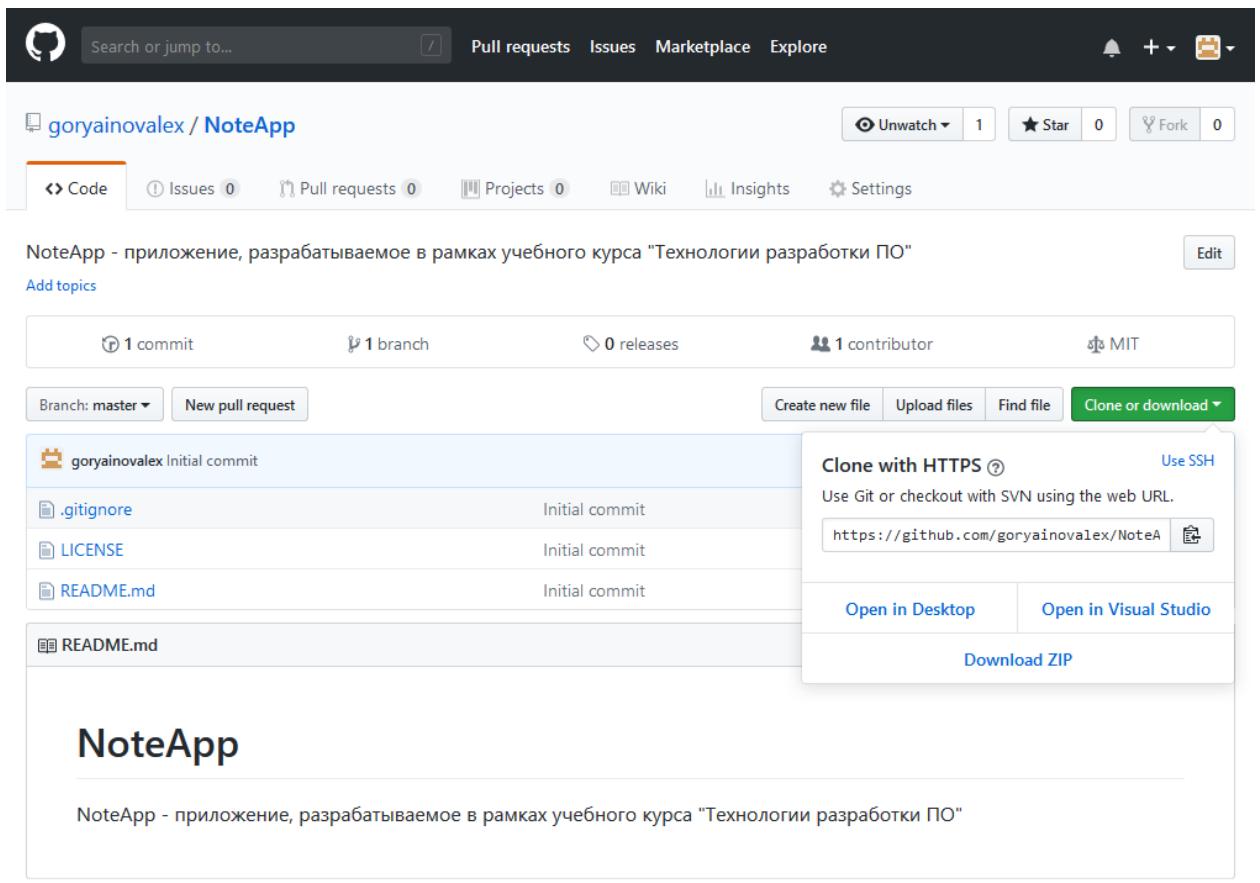


Рисунок 10 – Расположение кнопки клонирования репозитория с GitHub

При нажатии на кнопку может появится окно с выбором приложения, которое должно выполнить клонирование. В нашем случае, это Visual Studio:

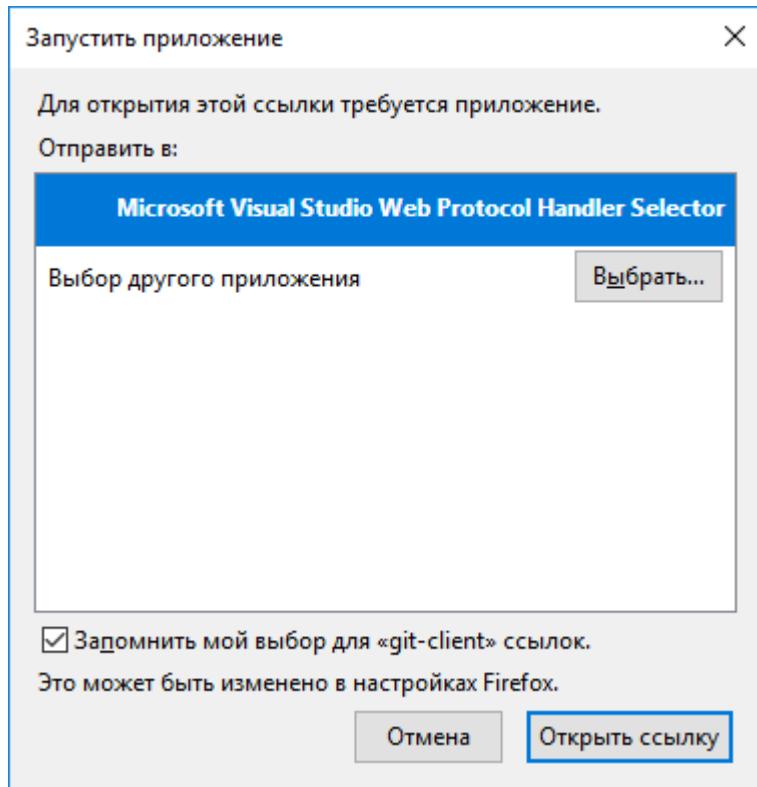


Рисунок 11 – Окно выбора программы для обработки ссылки с репозиторием

После нажатия «Открыть ссылку» запустится среда разработки Visual Studio. После запуска в правой части экрана появится панель Team Explorer. Панель Team Explorer в Visual Studio предназначена для работы с версионным контролем. При клонировании репозитория на панели появится два поля: ссылка на удаленный репозиторий (будет заполнена автоматически ссылкой из GitHub) и ссылка на локальную папку, куда необходимо клонировать удаленный репозиторий. Укажите целевую папку (можно оставить по умолчанию) и нажмите Clone.

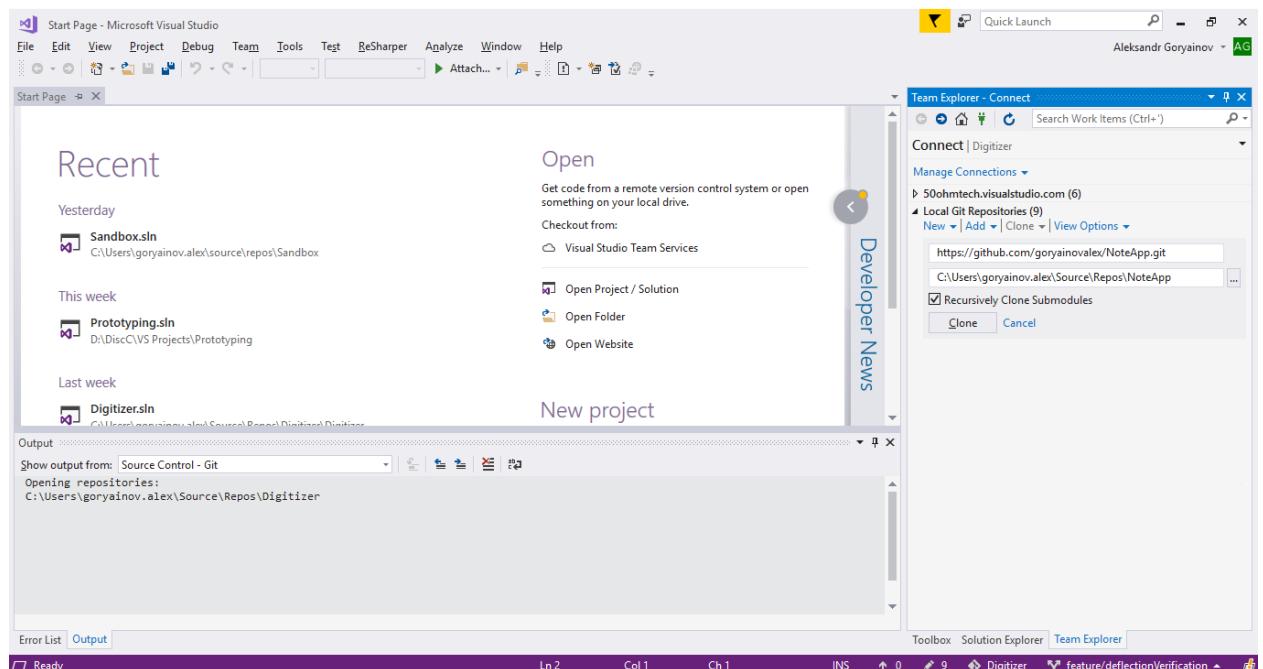


Рисунок 12 – Окно клонирования репозитория в Visual Studio

После завершения клонирования вы можете открыть целевую папку и убедитесь, что все файлы из GitHub были скопированы на ваш компьютер:

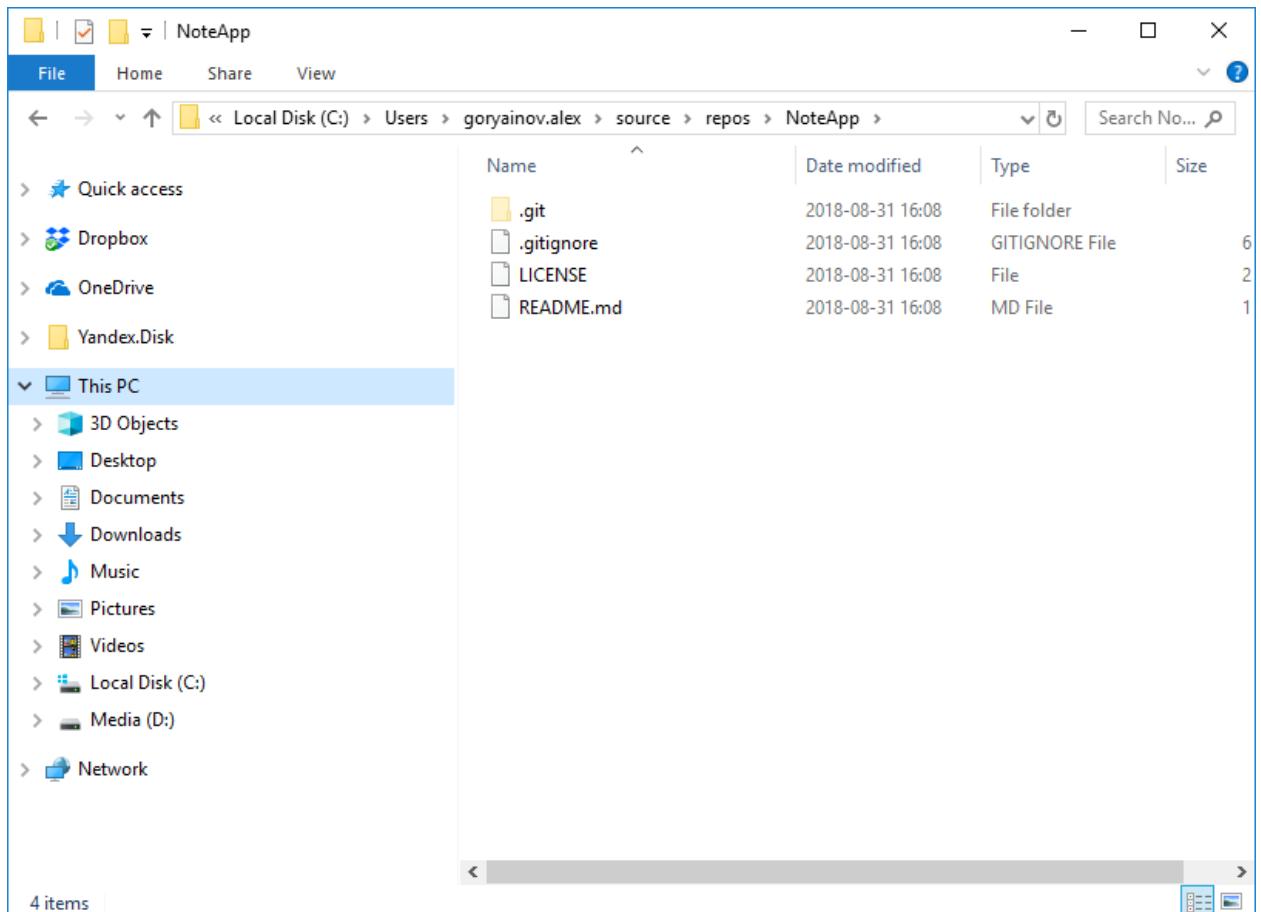


Рисунок 13 – Папка склонированного репозитория на компьютере пользователя

Папка репозитория обладает рядом особенностей. В частности, система версионного контроля будет отслеживать все изменения в этой папке: добавление, редактирование и удаление файлов. Каждое изменение запоминается версионным контролем, и в дальнейшем система предложит вам зафиксировать эти изменения – сделать коммит.

После клонирования репозитория панель Team Explorer изменит свой вид – главное меню панели Team Explorer (Home). Из данного меню можно перейти на подстраницы: несохраненных изменений в папке репозитория Changes; синхронизации локального репозитория с удаленным репозиторием origin; управления ветками репозитория Branches; настроек репозитория Settings. Также данная панель позволяет открывать и создавать новые решения в папке репозитория – кнопки New, Open под главным меню панели.

Сейчас в нашем репозитории есть только три файла, но нет ни одного решения Visual Studio, где мы могли бы разрабатывать программу. Об этом свидетельствует надпись There were no solutions found на панели Team Explorer:

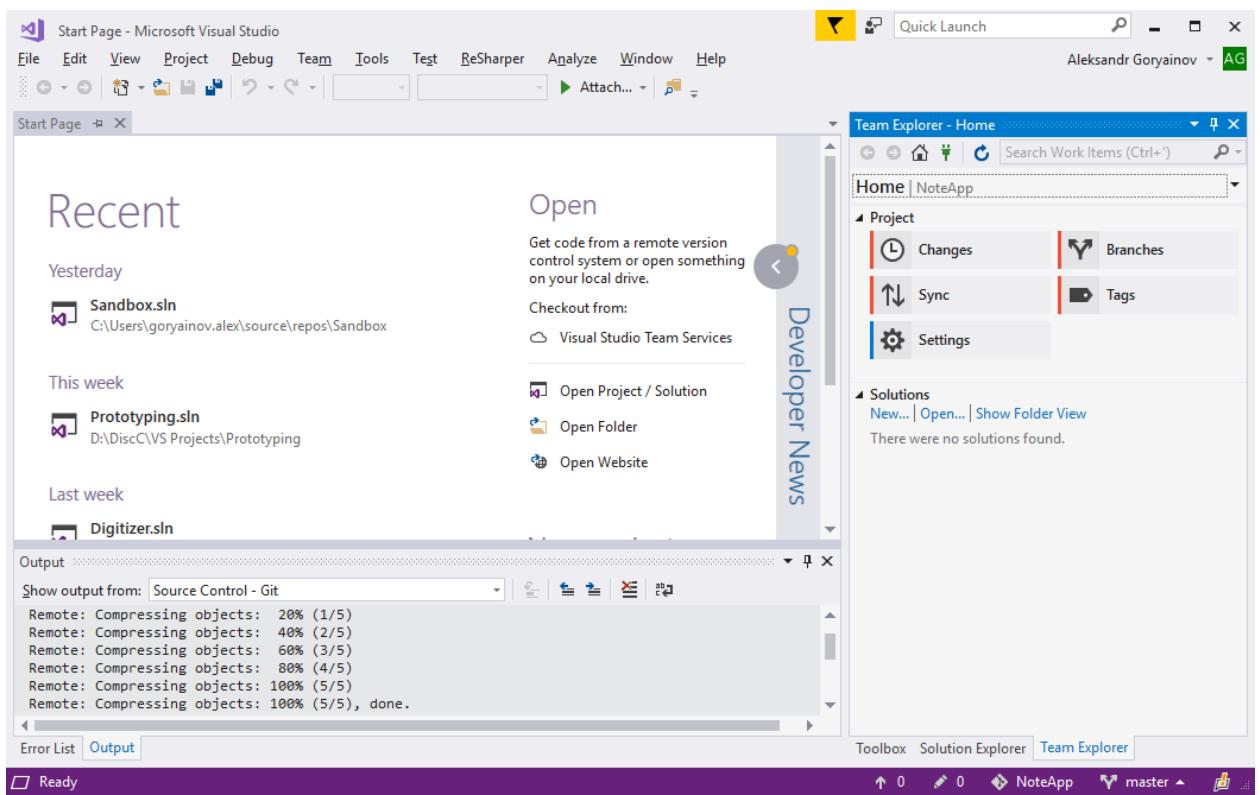


Рисунок 14 – Панель Team Explorer открытого репозитория

Нажмем кнопку New и создадим решение для нашей разрабатываемой программы:

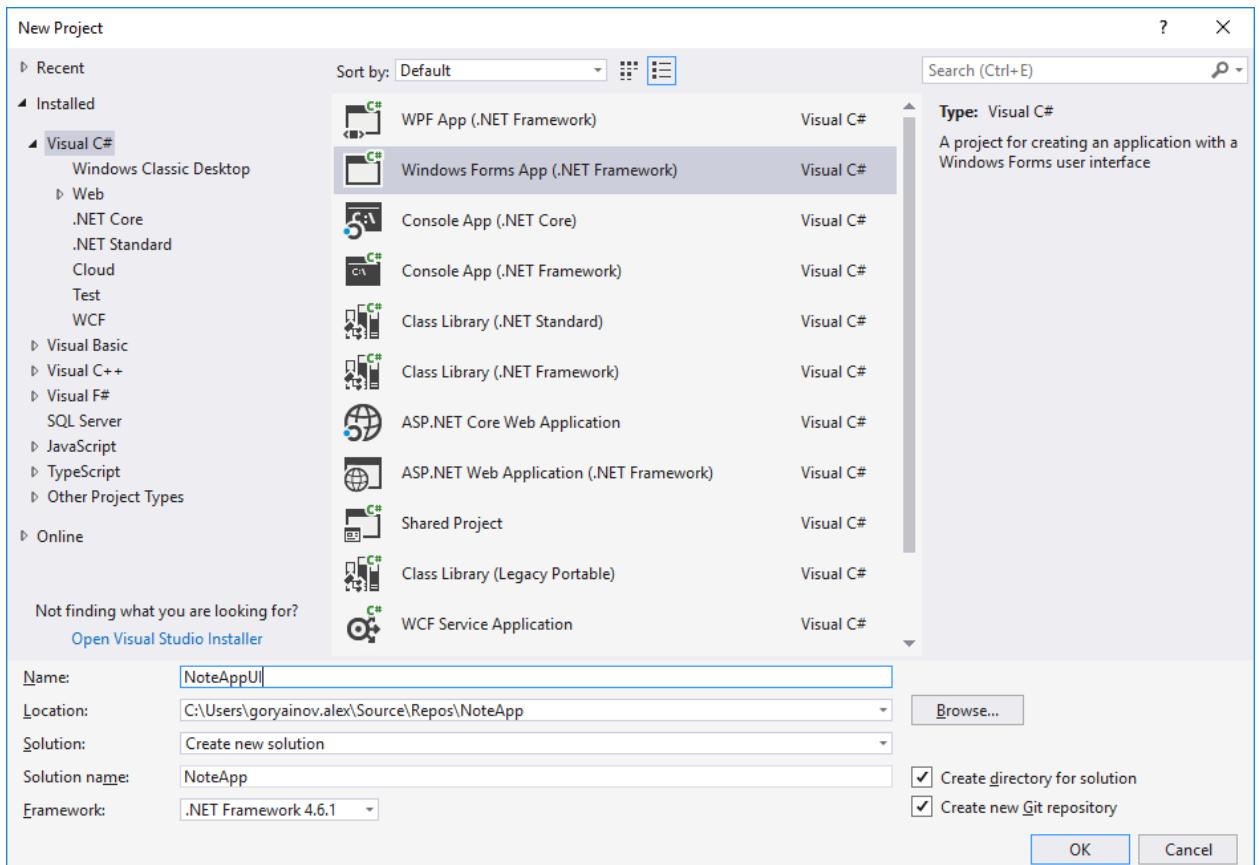


Рисунок 15 – Создание нового решения в репозитории

Стартовым проектом программы выберем проект Windows Forms App, в качестве названия решения укажем название программы, а к названию проекта добавим приставку UI (например, NoteAppUI). Такие приставки в названиях проектов позволяют лучше ориентироваться в решении и сразу находить проекты, относящиеся к пользовательскому интерфейсу.

После создания решения будет открыта пустая форма для создания программы. Мы можем переходить непосредственно к разработке программы:

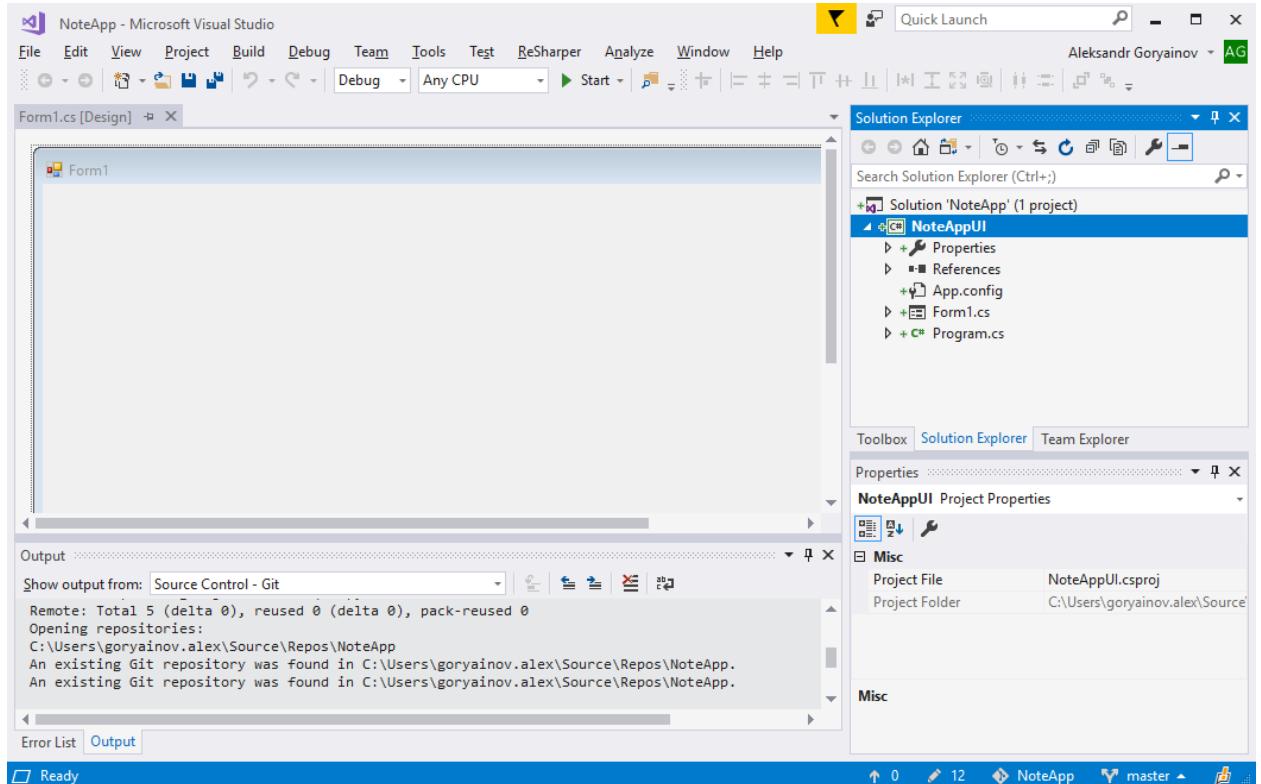


Рисунок 16 – Созданное решение в Visual Studio

В ходе работы вам придется часто переключаться между панелями Solution Explorer и Team Explorer. Поэтому разместите их как вам удобно для работы. Зайдите в папку репозитория на своём компьютере и убедитесь, что решение находится в папке:

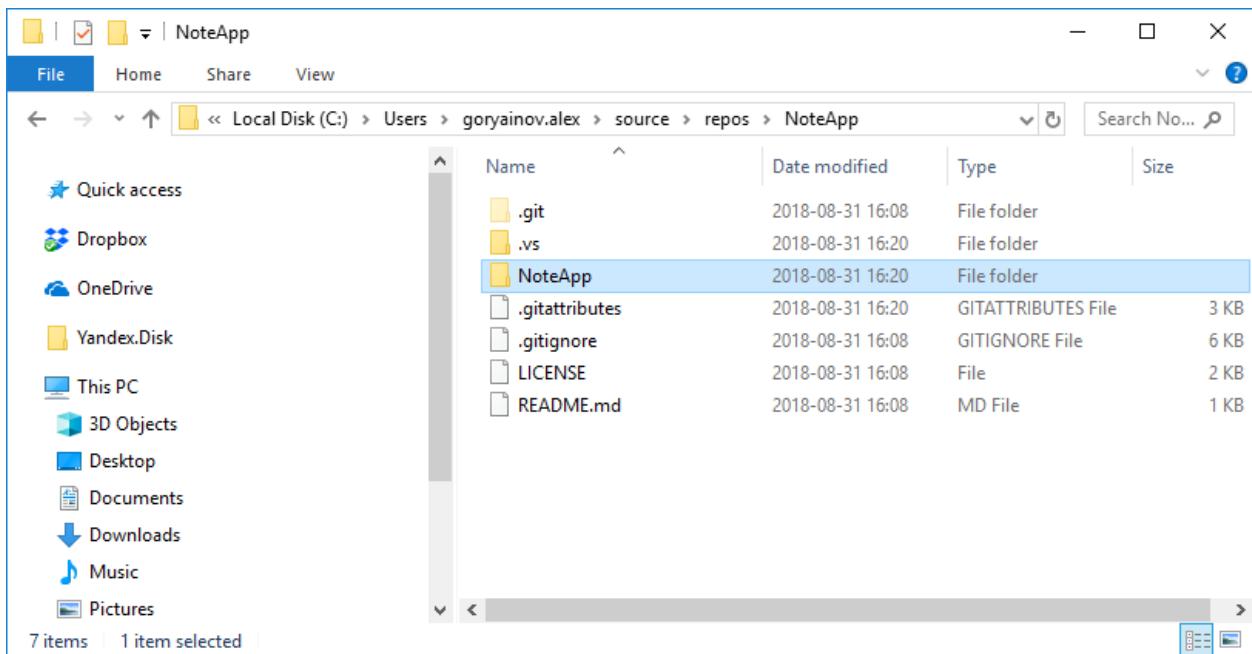


Рисунок 17 – Папка с решением на компьютере пользователя

После того, как мы создали решение, в папке репозитория появились новые файлы. Сейчас эти файлы не сохранены в репозитории. Это можно отследить через панель TeamExplorer на подменю Changes. На панели будет представлен список всех несохраненных изменений в репозитории, а также предложение сохранить их (сделать коммит). Как видно на рисунке, сейчас в репозитории есть 12 несохраненных изменений, все они связаны с созданием нового решения:

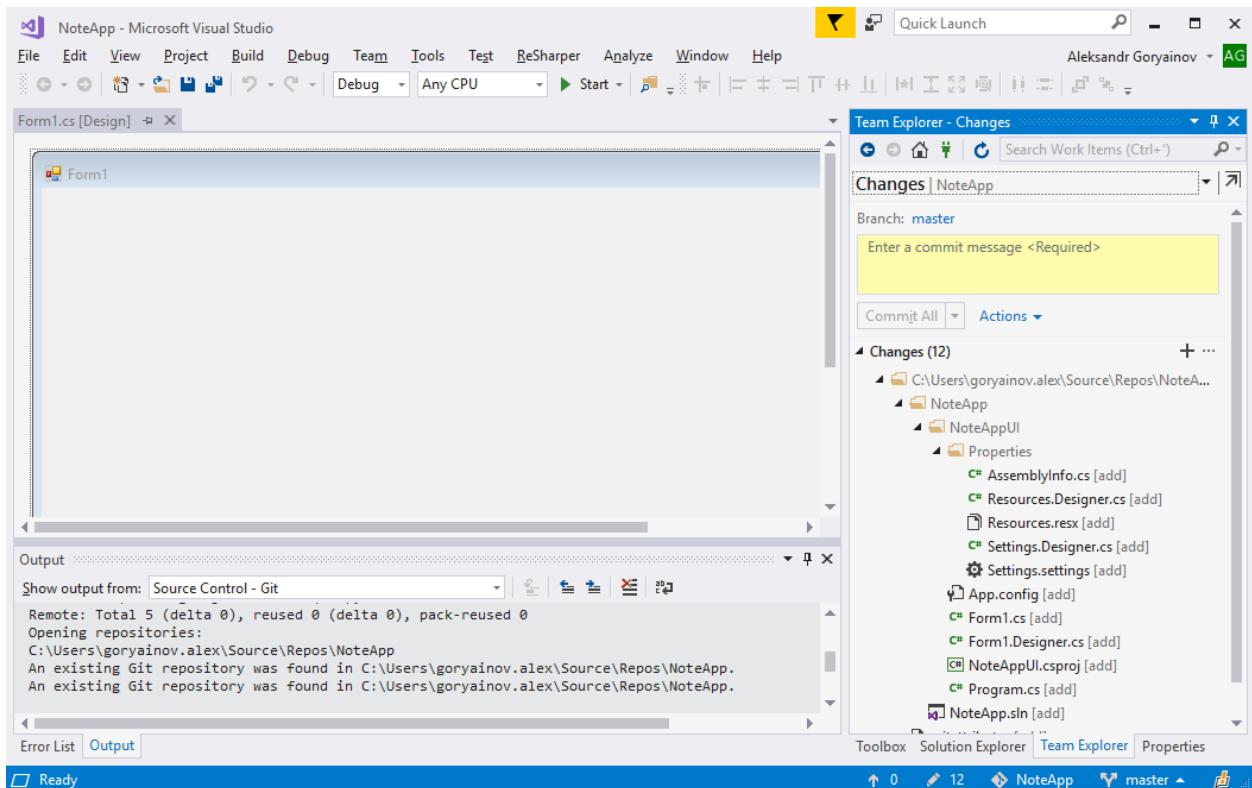


Рисунок 18 – Список несохраненных изменений на панели Team Explorer

Также обратите внимание на синюю строку состояния Visual Studio. В правом нижнем углу дается краткая информация о вашем текущем репозитории: его название (NoteApp), текущая ветка (master), количество незафиксированных изменений (12) и количество коммитов в локальном репозитории, несинхронизированных с удаленным origin (0). Нажатие по любому из компонентов откроет соответствующее подменю панели TeamExplorer.

Зафиксируем текущие изменения в репозитории. Для этого вверху панели TeamExplorer введем комментарий и нажмем Commit All. Комментарий должен быть осмысленным, и кратко объяснить, что было изменено в репозитории. Лучше всего, если вы будете описывать в комментарии какую функциональность вы добавили в программу и перечисляли названия ключевых файлов/классов, которые были изменены. Правильный комментарий значительно упрощает в будущем работу с репозиторием, если в какой-то момент вы случайно удалите репозиторий или обнаружите критические ошибки в проекте. Поверьте, такие ситуации у вас будут чаще, чем хотелось бы.

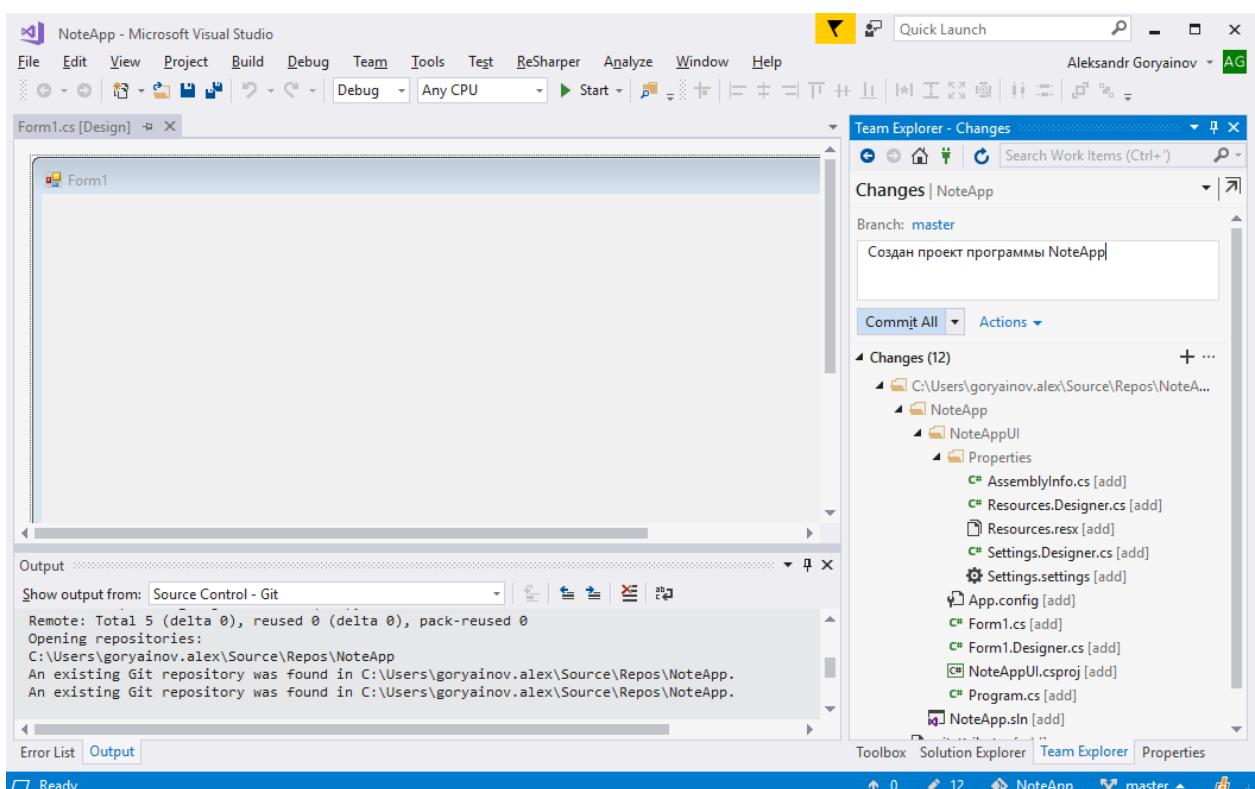


Рисунок 19 – Выполнение коммита через панель Team Explorer

После коммита на панели надпись об успешном проведении коммита, а список нессохраненных изменений станет пуст – это значит, что теперь все изменения были сохранены в репозитории и в случае необходимости к ним можно будет вернуться.

Обратите внимание, что теперь в строке состояния Visual Studio справа внизу количество несинхронизированных коммитов изменилось с нуля на единицу:

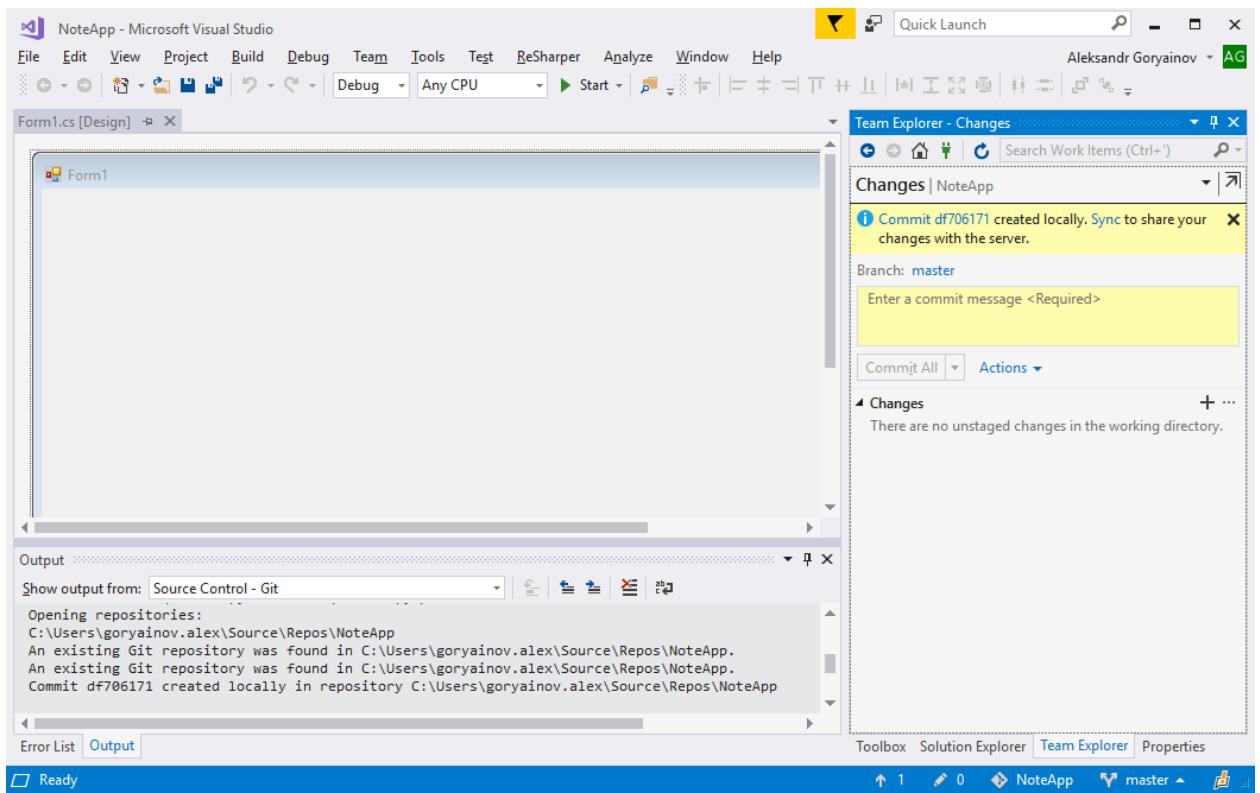


Рисунок 20 – Уведомление о выполнленном коммите на панели Team Explorer

Это цифра означает, что локальный репозиторий отличается от репозитория origin на GitHub ровно на один коммит. То есть, коммит с созданным проектом программы теперь сохранен у вас в репозитории, но не в репозитории GitHub. Для того, чтобы изменения с вашего репозитория попали в удаленный репозиторий GitHub, необходимо выполнить синхронизацию репозиториев. Для этого перейдите на подменю Synchronization панели Team Explorer:

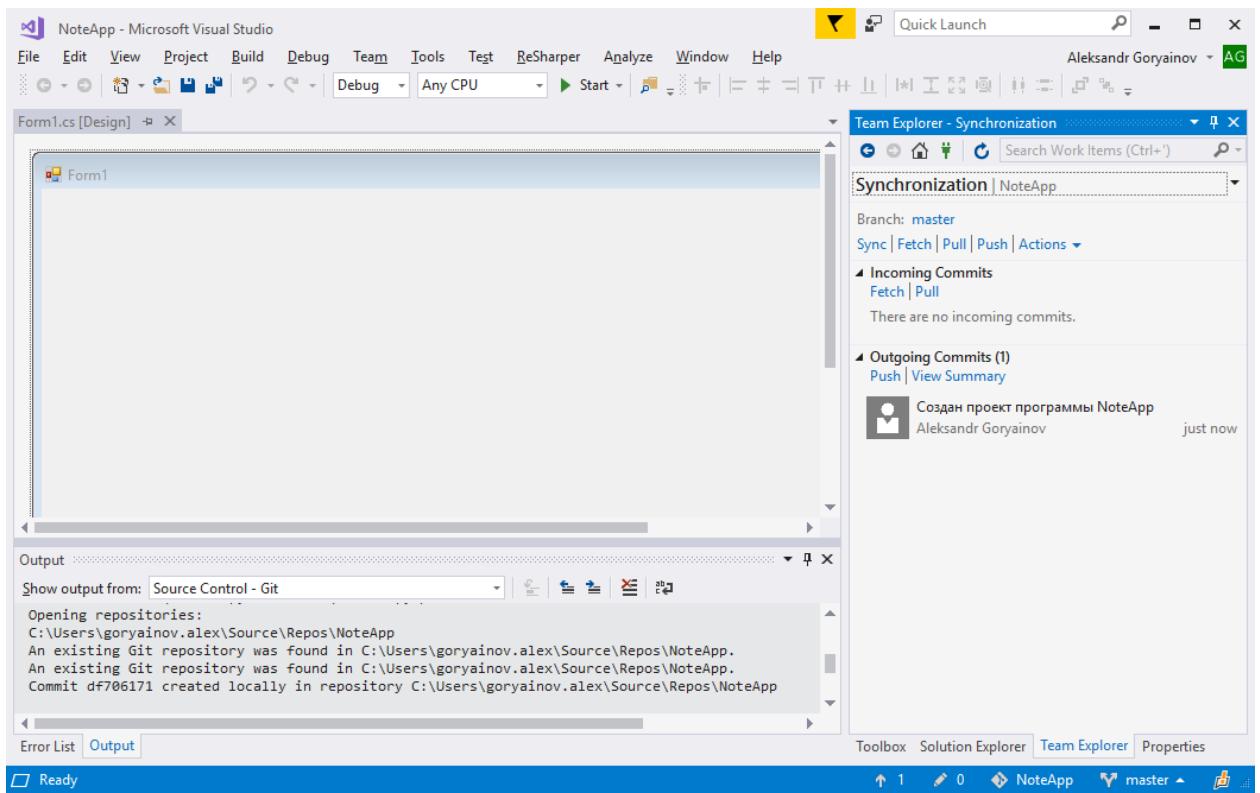


Рисунок 21 – Синхронизация локального репозитория с origin через панель Team Explorer

На этой панели описываются входящие и исходящие коммиты. Исходящие коммиты – это коммиты, которые выполнены в вашем репозитории, но которых еще нет в удаленном репозитории. Входящие коммиты – это коммиты, которые появились в удаленном репозитории, но которых пока нет в вашем репозитории (входящие коммиты появятся, если в вашей команде будут другие разработчики, создающие коммиты в своих репозиториях).

Вверху на панели есть кнопки Sync, Fetch, Pull, Push:

Push – отправить свои коммиты в удаленный репозиторий origin;

Pull – получить коммиты из удаленного репозитория в свой локальный репозиторий (перед нажатием Push всегда сначала стоит делать Pull – во избежание возможных конфликтов);

Sync – выполнить полную синхронизацию с удаленным репозиторием (последовательно выполняет Pull и Push). Нажмите кнопку Push или Sync для синхронизации. После успешной синхронизации на панели TeamExplorer появится уведомление:

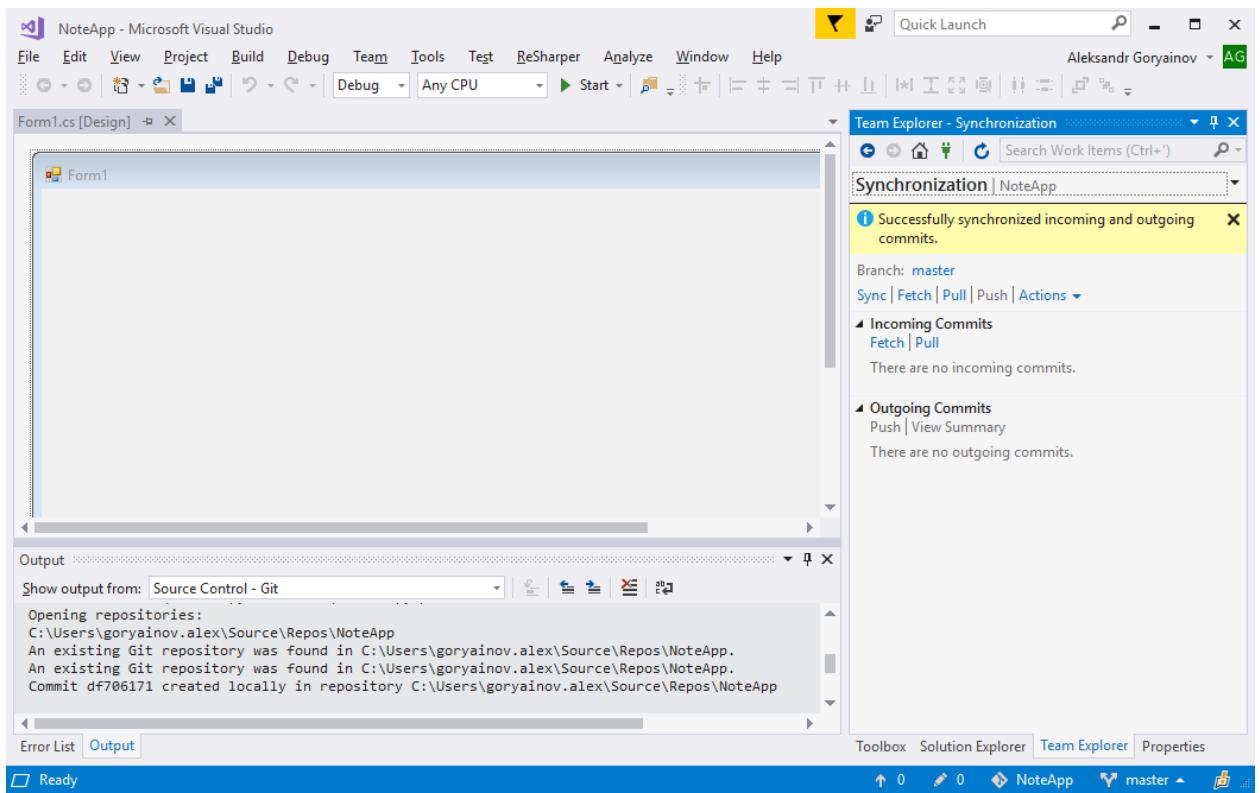


Рисунок 22 – Уведомление о выполненной синхронизации на панели Team Explorer

Средства Visual Studio позволяют просматривать историю коммитов всего репозитория. Для этого нажмите правую кнопку по названию ветки и нажмите View History:

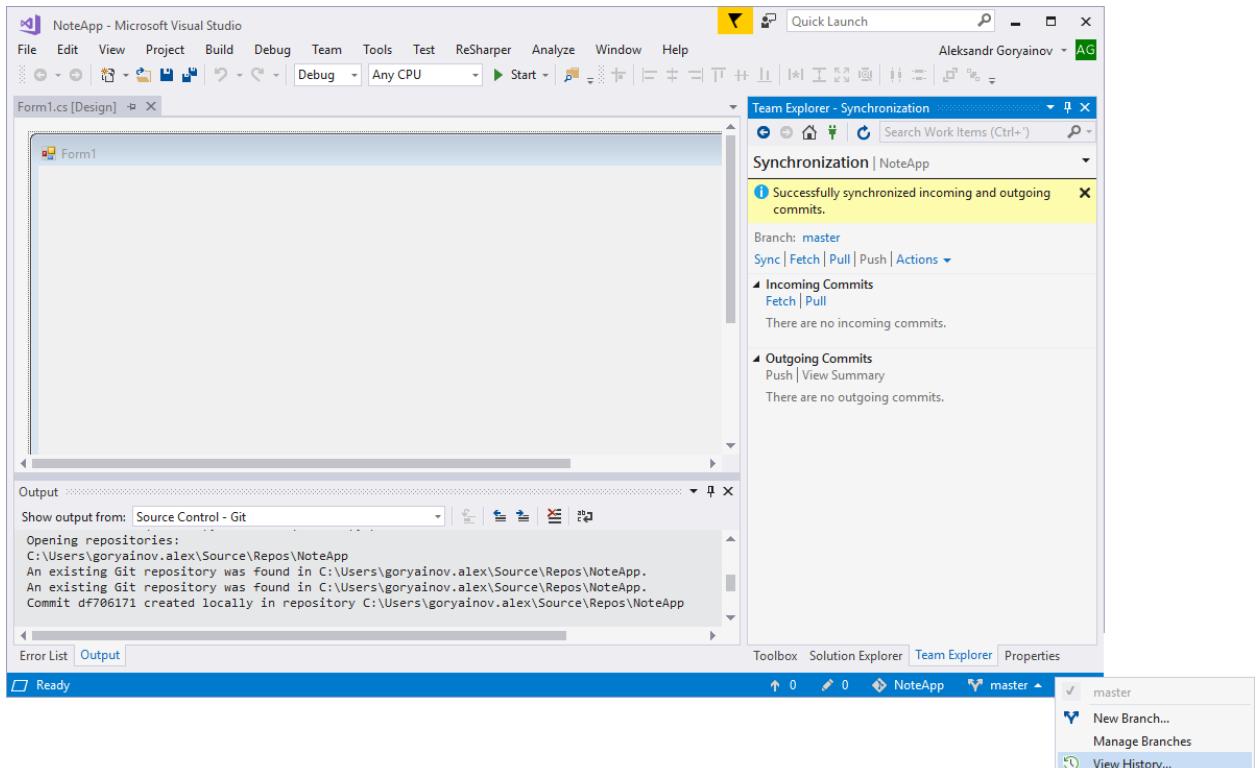


Рисунок 23 – Просмотр истории репозитория через строку состояния Visual Studio

После нажатия в центральной части экрана появится вкладка History, отображающая историю репозитория текущей ветки:

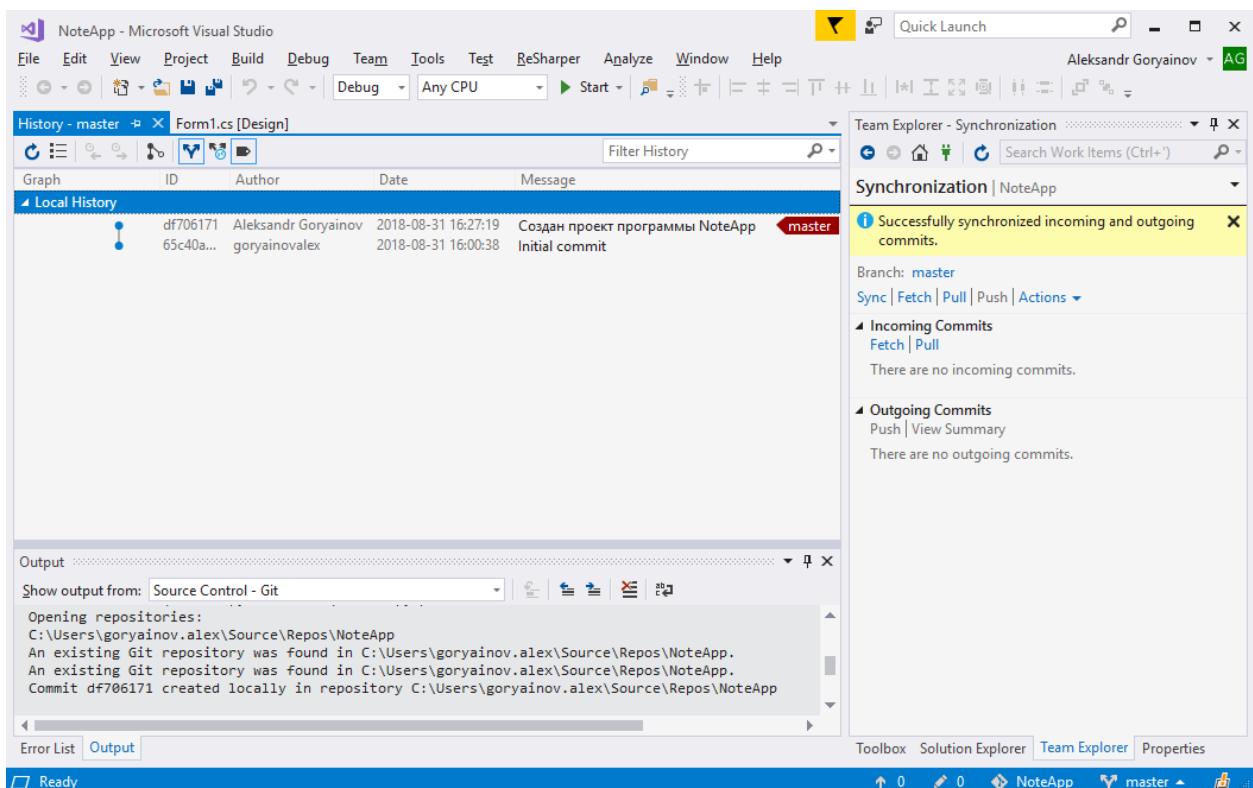


Рисунок 24 – Вкладка истории репозитории (ветка master)

В случае работы со множеством веток на данной вкладке также показывается граф, отображающий создание и слияние веток в репозитории. Сейчас в нашем репозитории только два коммита, однако в реальных компаниях количество коммитов в одном репозитории может легко составлять несколько тысяч уже через год работы.

После синхронизации можно зайти на GitHub и убедиться, что файлы решения действительно были залиты на GitHub:

The screenshot shows the GitHub repository page for 'goryainovalex / NoteApp'. At the top, there are buttons for 'Unwatch' (1), 'Star' (0), and 'Fork' (0). Below the header, a navigation bar includes 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Insights', and 'Settings'. A main title states 'NoteApp - приложение, разрабатываемое в рамках учебного курса "Технологии разработки ПО"' with an 'Edit' button. Below this, a summary bar shows '2 commits', '1 branch', '0 releases', '1 contributor', and 'MIT'. A dropdown menu shows 'Branch: master' and a 'New pull request' button. To the right are links for 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The commit history lists five entries:

Commit	Message	Time
Aleksandr Goryainov	Создан проект программы NoteApp	Latest commit df70617 3 minutes ago
NoteApp	Создан проект программы NoteApp	3 minutes ago
.gitattributes	Создан проект программы NoteApp	3 minutes ago
.gitignore	Initial commit	30 minutes ago
LICENSE	Initial commit	30 minutes ago
README.md	Initial commit	30 minutes ago

Below the commit history is a section titled 'NoteApp' containing the project's description: 'NoteApp - приложение, разрабатываемое в рамках учебного курса "Технологии разработки ПО"'.

Рисунок 25 – Страница репозитория на GitHub после синхронизации

Нажав на кнопку commits (панель над перечнем файлов), можно также просмотреть историю репозитория, но только средствами GitHub.

The screenshot shows the GitHub commit history for the 'master' branch of 'goryainovalex / NoteApp'. At the top, there are buttons for 'Unwatch' (1), 'Star' (0), and 'Fork' (0). Below the header, a navigation bar includes 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Insights', and 'Settings'. A dropdown menu shows 'Branch: master'. The commit history displays two entries:

Commit	Author	Date	Actions
df70617	Aleksandr Goryainov	committed 4 minutes ago	
65c40a6	goryainovalex	committed 31 minutes ago	

At the bottom, there are 'Newer' and 'Older' buttons.

Рисунок 26 – Страница истории репозитория на GitHub после синхронизации

Как правило, ветка master предназначена для хранения только тех коммитов, код которых проверен, протестирован и готов для сборки установщика приложения. Для непосредственной разработки программы создается специальная ветка develop. Для создания ветки перейдите на подменю Branches на панели Team Explorer:

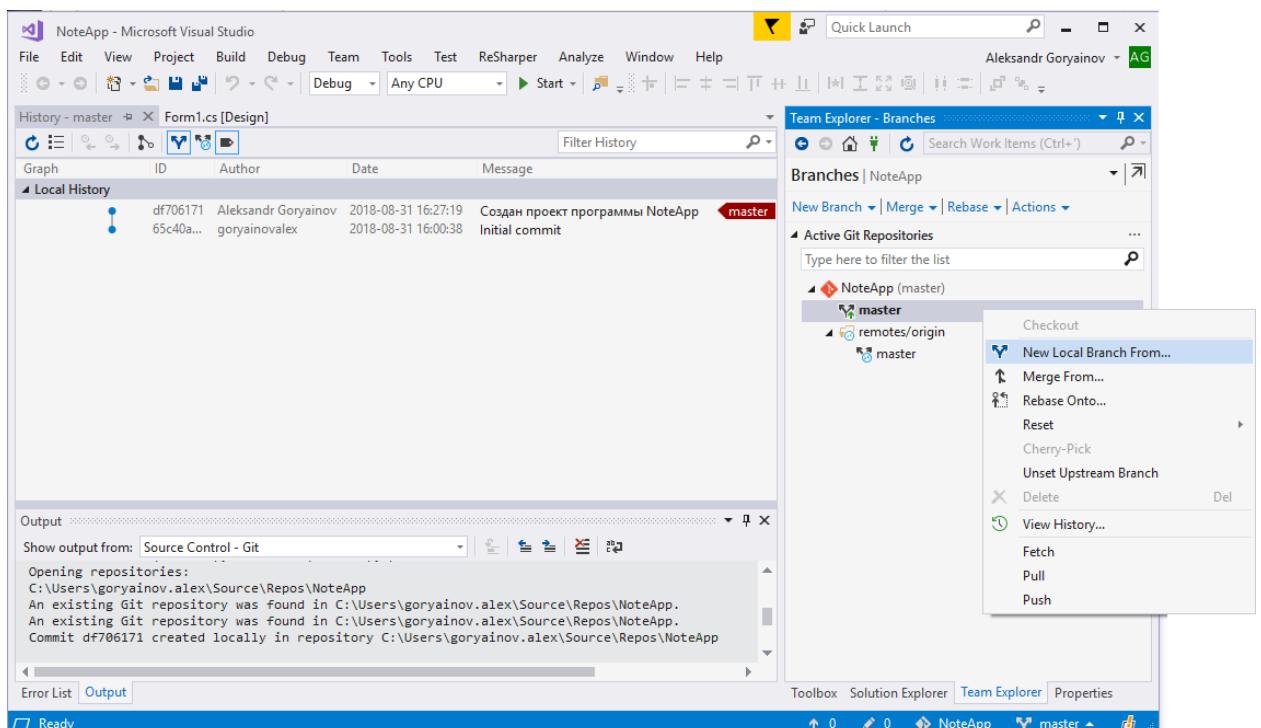


Рисунок 27 – Контекстное меню для создания новой ветки в репозитории

Перед вами появится список всех веток репозитория. Вверху перечисляются ветки репозитория, которые хранятся в вашем локальном репозитории (сейчас это только одна ветка master), а в подпапке remotes/origin перечислены все ветки, которые есть в удаленном оригинальном репозитории (опять же, сейчас это только одна ветка master). Кликните правой кнопкой по локальной ветке master и выберите пункт New Local Brach From (Создать новую локальную ветку из...). В появившейся панели введите название новой ветки (develop) и нажмите Create Branch (Создать ветку):

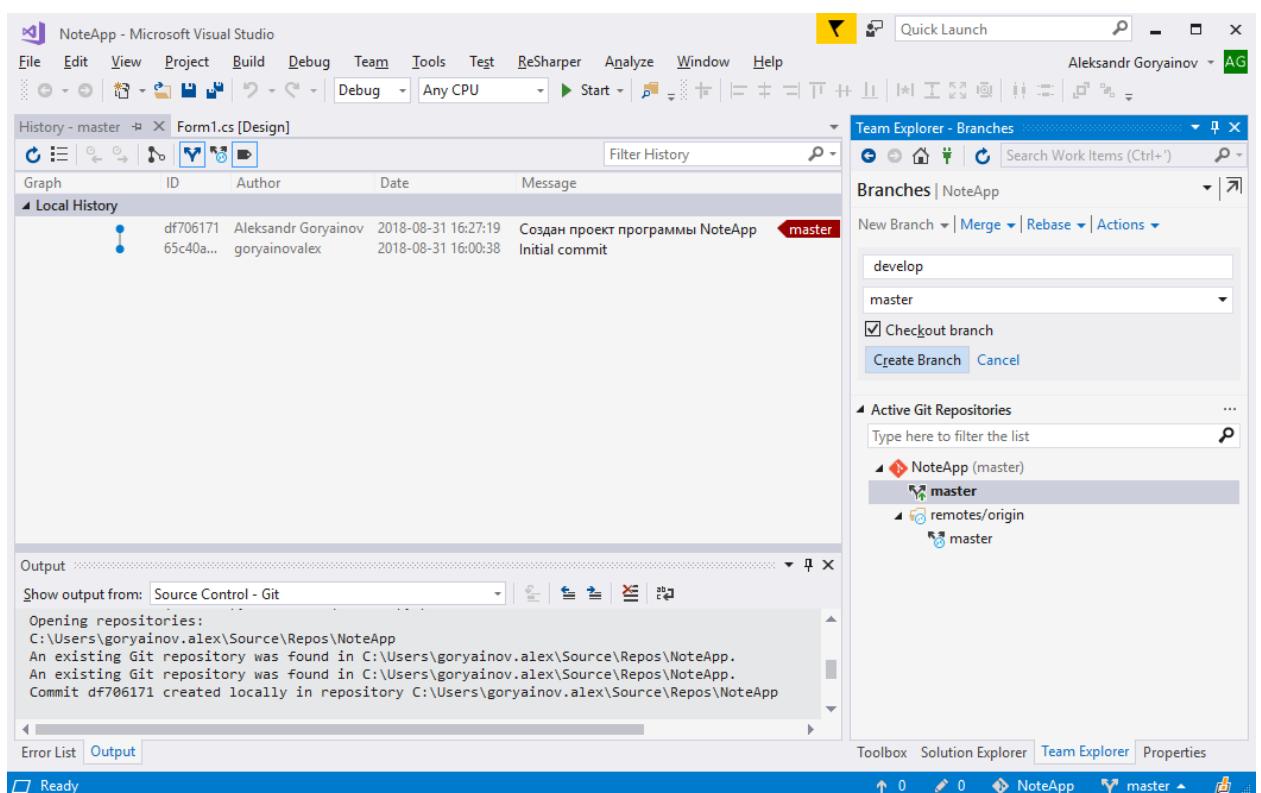


Рисунок 28 – Панель создания ветки в репозитории

Созданная ветка отобразится в списке веток. Двойным кликом перейдите в новую ветку **develop** (она должна быть выделена жирным шрифтом):

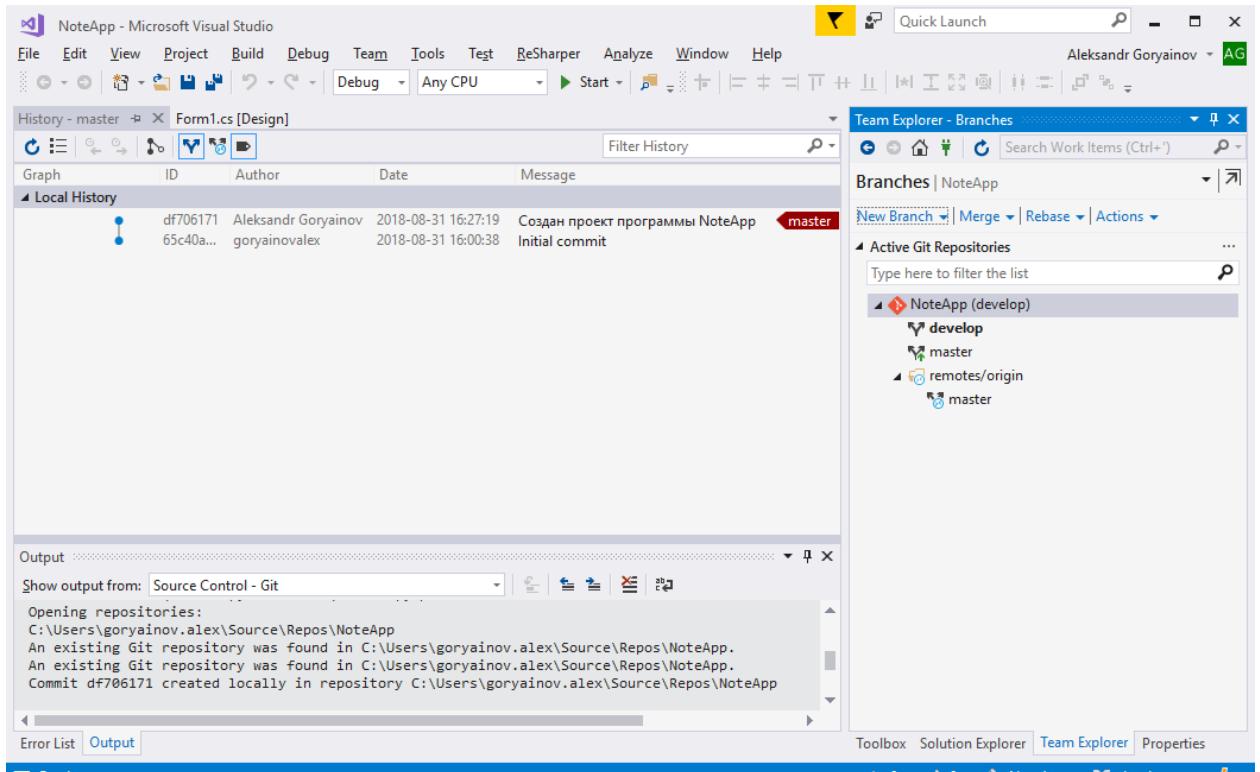


Рисунок 29 – Новая ветка **develop** на панели Team Explorer

Обратите внимание, что название текущей ветки изменилось на панели состояния в правом нижнем углу. Сделаем коммит в новую ветку. Для этого выполним какую-нибудь изменения в проекте. Например, переименуем класс **Form1** в **MainForm** через панель Solution Explorer. На панели Team Explorer – Changes сразу же отобразятся незафиксированные изменения:

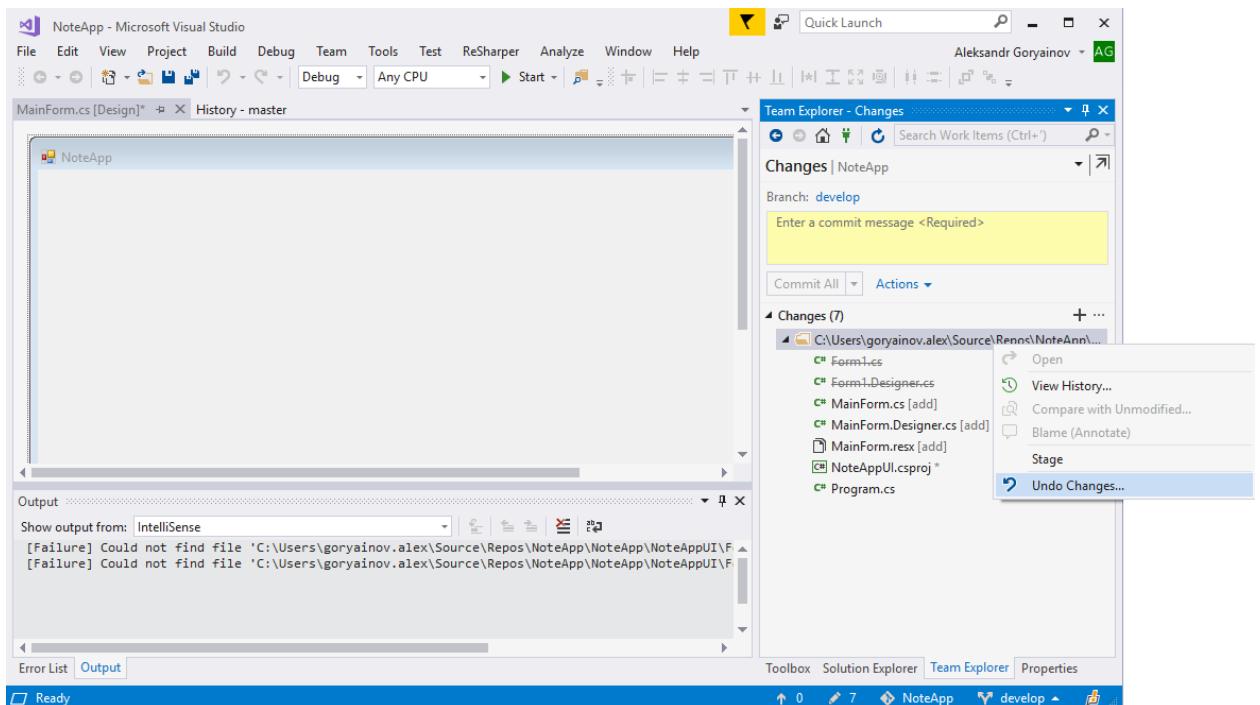


Рисунок 30 – Панель изменений в проекте и контекстное меню для отмены несохраненных изменений в репозитории

Сделайте коммит и синхронизируйтесь с удаленным репозиторием. Заметьте, что до синхронизации ветка develop не существует в удаленном репозитории origin на GitHub, и появится там только после синхронизации. Если вы всё сделали верно, то история ветки develop должна выглядеть следующим образом:

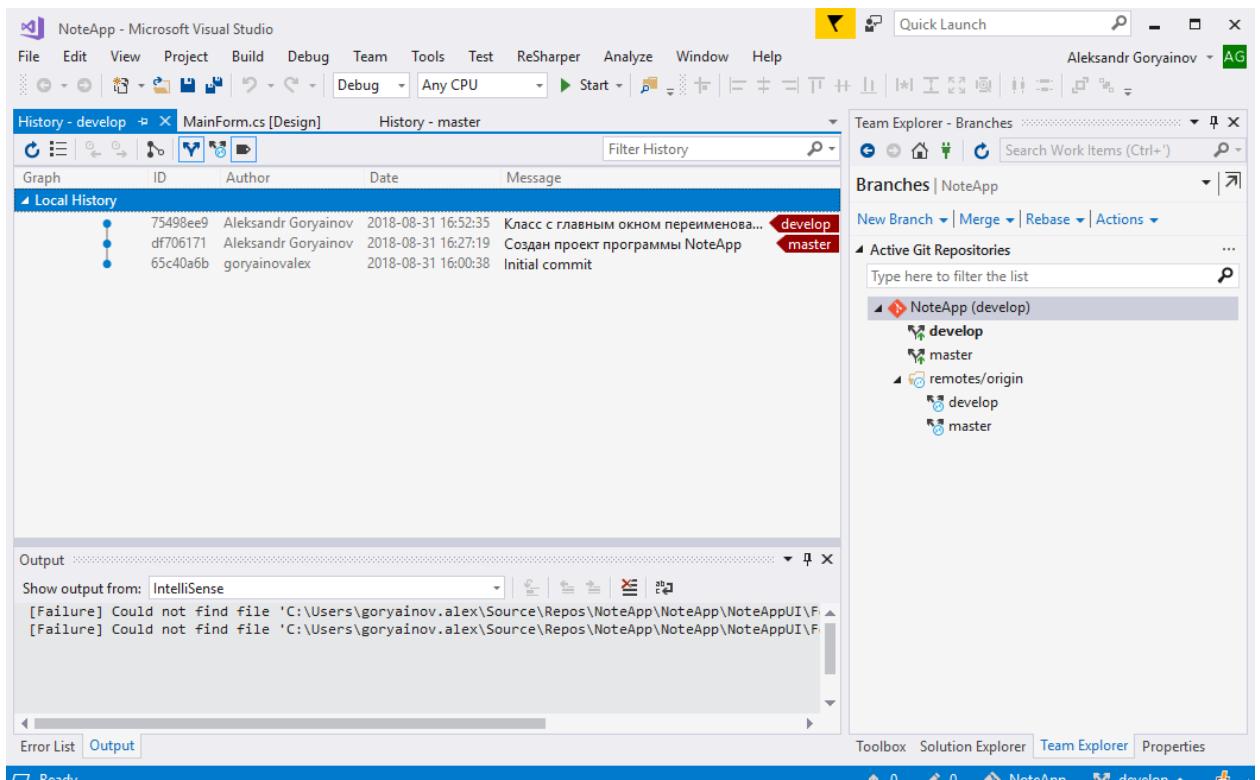


Рисунок 31 – История локального репозитория (ветка develop)

Также вы можете зайти на GitHub и убедиться, что в репозитории действительно появилась ветка develop, а класс Form1 переименован в MainForm.

Описанная функциональность достаточна для одиночной разработки программ. Если вы хотите добавить разработчиков к своему репозиторию, вам необходимо зайти в настройки репозитория на GitHub. Во вкладке Collaborators вы можете добавить аккаунты тех пользователей, которым будет разрешено делать коммиты в ваш репозиторий.

The screenshot shows the GitHub repository settings page for 'goryainovalex / NoteApp'. The 'Settings' tab is selected. On the left, there's a sidebar with options like Options, Collaborators (which is highlighted), Branches, Webhooks, Integrations & services, Deploy keys, Moderation, and Interaction limits. The main content area is titled 'Collaborators' and contains a sub-header 'Push access to the repository'. It says 'This repository doesn't have any collaborators yet. Use the form below to add a collaborator.' Below this is a search bar with placeholder text 'Search by username, full name or email address' and a note: 'You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.' At the bottom right of the search bar is a button labeled 'Add collaborator'.

Рисунок 32 – Страница настроек репозитория на GitHub – вкладка добавления участников в проект

На практике, другим пользователям не разрешают напрямую делать коммиты в любую ветку репозитория – иначе недостаточно ответственный разработчик может испортить своими коммитами ваш репозиторий и добавить критических ошибок в проект. По этой причине, на панели Branches также можно добавить специальные правила, ограничивающие коммиты в ветки master и develop.

The screenshot shows the GitHub repository settings page for 'goryainovalex / NoteApp'. The 'Settings' tab is selected. The sidebar on the left has the 'Branches' option highlighted. The main content area is titled 'Default branch' and contains a note: 'The default branch is considered the "base" branch in your repository, against which all pull requests and code commits are automatically made, unless you specify a different branch.' Below this is a dropdown menu set to 'master'. The next section is titled 'Branch protection rules' with a 'Add rule' button. It contains a note: 'Define branch protection rules to disable force pushing, prevent branches from being deleted, and optionally require status checks before merging. New to branch protection rules? Learn more.' At the bottom right of this section is a note: 'No branch protection rules defined yet.'

Рисунок 33 – Страница настроек репозитория на GitHub – вкладка создания ограничений доступа к веткам репозитория

При ограничении доступа, любой коммит другого пользователя будет делаться с предварительным уведомлением вас и разрешением на включение – это называется pull request («запрос на включение»). Использование запросов на включение описывается в следующем подразделе.

2.1.4 Модель ветвления GitFlow

На практике в компаниях используется так называемая модель ветвления gitflow [12, 13]. Строго говоря, любая модель ветвления git называется gitflow, однако чаще всего под данным термином подразумевают модель, описанную далее.

Модель применяется в случае коллективной разработки, где в команде можно выделить руководителя команды (team lead – «лидер команды», тимлид), ведущих разработчиков (senior developer – «ведущий разработчик»), разработчиков (middle developer) и младших разработчиков (junior developer). Преимущества данной модели – минимизация конфликтов в случае слияния исходного кода разработчиков, исключение внедрения неработающего или плохо оформленного кода в репозиторий, а также частые проверки кода. Описанные преимущества достигаются благодаря разграничению доступа к веткам репозитория для разработчиков с различным опытом.

Согласно в gitflow, в репозитории должны быть следующие ветки:

master – главная и начальная ветка репозитория. Любой код в данной ветке должен быть проверен, протестирован и готов к сборке установщика или внедрению. Ветка master существует всё время разработки проекта.

develop – основная ветка разработки, предназначенная для синхронизации работы всей команды. Когда кто-либо из разработчиков создает стабильную рабочую версию новой функциональности, он синхронизирует свой код в develop. Любой код в данной ветке должен быть проверен и протестирован. Нерабочий код не допускается, так как в противном случае это остановит работу всей команды. Ветка develop создается из ветки master сразу же при создании репозитория и существует всё время разработки проекта. Master и develop – главные ветки репозитория.

feature branches – ветки функциональностей. Это ветки, создаваемые любым разработчиком для разработки новой функциональности. Создаются из ветки develop. В один момент времени может существовать множество веток функциональностей. Например, в ветке feature1 ведет разработку первый разработчик, а в ветке feature2 ведет работу второй разработчик. Когда первый разработчик закончит разработку feature1, он зальёт её код в ветку develop, и создаст ветку feature3 для разработки следующей функциональности. Таким образом, ветки функциональности существуют не всё время разработки проекта, а только во время разработки одного конкретного функционального блока.

hotfixes – ветки с исправлением критических ошибок в релизах, обнаруженных во время эксплуатации/внедрения программы. Ветки создаются из ветки master, а после исправления ошибки, проверки кода и полного тестирования, сливаются в master и develop.

Однако главной особенностью gitflow это разделение прав на синхронизацию с origin/develop. У разработчиков нет прямого доступа делать слияния собственных веток feature с веткой origin/develop, слияние можно произвести только через так называемый **pull request** («запрос на включение»). Запрос на включение подразумевает, что весь код, который разработчик хочет синхронизировать с origin/develop, должен быть проверен руководо-

дителем команды или одним из старших разработчиков. Проверка кода в различных источниках называется термином code review («проверка кода»). Если у старшего разработчика нет замечаний по новому синхронизируемому коду, то он принимает запрос на включение, и синхронизация с origin/develop выполняется успешно. Если же у старшего разработчика находятся замечания, он оставляет комментарии в коде и отклоняет запрос на включение до их исправления.

Благодаря специализированным веткам и работе через запросы на включение решаются несколько распространенных проблем в разработке приложений:

- 1) младший разработчик или новый член команды не могут по случайности синхронизировать в репозиторий некачественный код;
- 2) любая часть кода известна как минимум двум участникам разработки – нет такого кода, работа которого известна только одному разработчику;
- 3) любому участнику команды всегда доступна последняя стабильно работающая версия исходного кода – она всегда находится в ветке develop.

Также обязательным правилом является синхронизация последнего кода из develop в собственную ветку feature перед отправкой запроса на включение. Это объясняется просто: так как пока вы разрабатывали свою часть функциональности, в ветке develop могли попасть блоки кода от других участников команды – во избежание конфликтов при запросе на включение вам необходимо залить последние изменения из ветки develop в свою ветку feature и только после этого отправлять запрос на включение.

Данная модель используется в подавляющем большинстве ИТ-компаний и потому обязательна к изучению. Исключая из данной модели ветвления использование запросов на включение вы получаете эффективную модель для одиночной разработки программ.

2.1.5 Задание

1. Установить требуемые приложения:

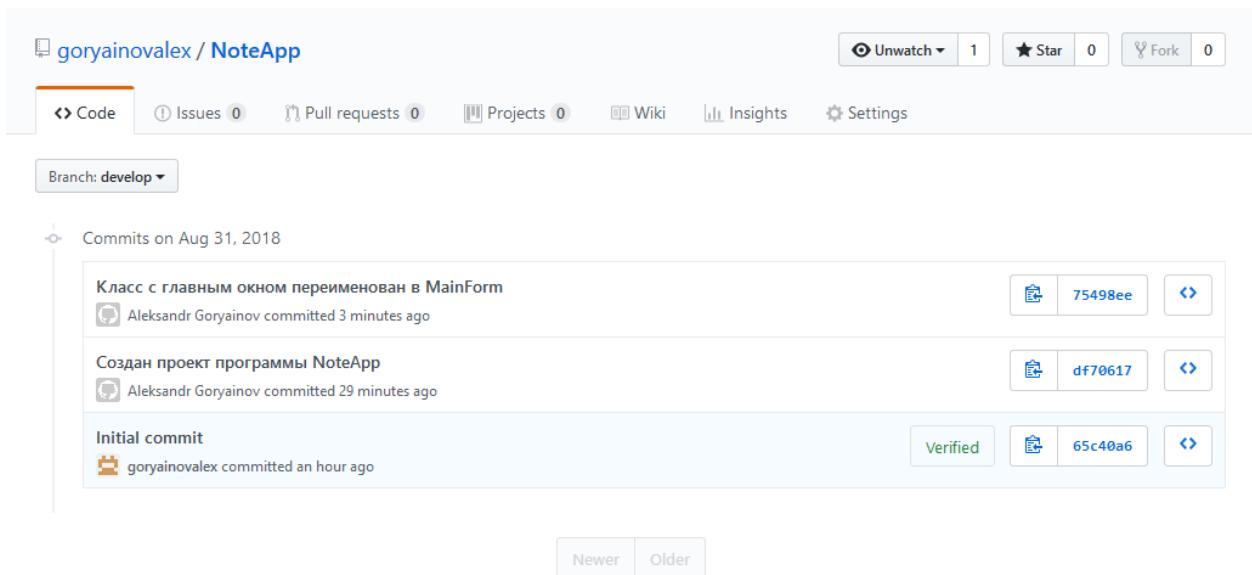
- 1.1. Установите Visual Studio 2018 Community. Предпочтительнее устанавливать английскую версию. При первом запуске приложения выбрать преднастройки языка C#. При первом запуске потребуется создать аккаунт пользователя для сервисов Microsoft.
- 1.2. Установите JetBrains ReSharper. После установки запросите получение бесплатной учебной лицензии на сайте компании JetBrains.
- 1.3. Установите InnoSetup.
- 1.4. Установите Enterprise Architect. При запуске выбирать пробный период.

2. Создать репозиторий на сервисе GitHub:

- 2.1. Создайте аккаунт на сервисе GitHub [\[7\]](#).
- 2.2. Создайте новый репозиторий с именем, соответствующим варианту задания. Например, NoteApp. В описании репозитория (поле Description) указать «Проект выполняется в рамках университетского курса».
- 2.3. При создании репозитория выберите шаблон Visual Studio для файла gitignore и лицензию MIT License.
- 2.4. Клонируйте созданный репозиторий на свой компьютер. При клонировании репозитория через GitHub используйте кнопку **Clone or download > Open in Visual Studio**. При необходимости, ввести логин и пароль от аккаунта GitHub в Visual Studio.

3. Создать решение Visual Studio в репозитории:

- 3.1. Создайте ветку `develop` на основе ветки `master` через боковую панель Visual Studio.
Перейдите в созданную ветку.
 - 3.2. В ветке `develop` добавьте новое решение с исполняемым проектом Windows Forms.
ВНИМАТЕЛЬНО: в окне создания нового решения укажите имя решения согласно варианту задания, а имя проекта укажите как `<name>UI`. Например, `NoteAppUI`. `UI` в имени проекта подчеркивает, что в проекте будут содержаться элементы пользовательского интерфейса программы. Далее, данный проект будем называть проект пользовательского интерфейса.
 - 3.3. Скомпилируйте решение и убедитесь, что программа запускается. После успешного запуска сделайте коммит «Создано решение проекта». Синхронизируйте локальный репозиторий с удаленным для отправки созданной ветки `develop` и последнего коммита на GitHub.
 - 3.4. Добавьте в решение проект библиотеки классов. Имя проекта должно совпадать с именем решения/варианта. Например, `NoteApp`. Имя проекта, совпадающего с именем варианта подчеркивает, что в данном проекте будет содержаться ключевая логика будущего приложения. Далее данный проект будем называть проект логики.
 - 3.5. Добавьте ссылку на проект логики в проект пользовательского интерфейса. Убедитесь, что проект логики «виден» в проекте пользовательского интерфейса. Для этого в исходном коде любой формы пользовательского интерфейса в области подключения пространств имён напишите строчку `using <имя проекта логики>;`. Например, `using NoteApp;`
Если ссылка добавлена не верно, среда разработки при компиляции сообщит об ошибке в этой строке.
 - 3.6. После успешной компиляции и запуска решения, сделайте коммит «Добавлен проект логики» и синхронизируйте проект с удаленным репозиторием.
4. **Проверить, что задание лабораторной работы выполнено верно:**
 - 4.1. Зайдите в свой аккаунт GitHub и откройте репозиторий проекта. Переключитесь на ветку `develop` для просмотра её текущего состояния. Структура файлов в удаленном репозитории должна соответствовать структуре файлов на локальной машине.
Если структура файлов отличается, или содержатся другие папки, например, `bin`, `Release`, `Debug`, `obj` – это значит, вы допустили ошибку при создании репозитория или решения. Например, не указали файл `gitignore` или указали не верные пути при создании проектов. Определите шаг, в котором вы допустили ошибку, и исправьте работу.
 - 4.2. Зайдите в историю коммитов ветки `develop`. История коммитов должна выглядеть следующим образом:



- 4.3. Если история коммитов отличается – это значит, вы не выполняли требуемые коммиты или не синхронизировались с удаленным репозиторием. Также обратите внимание на осмысленность комментариев к коммитам. Правильные комментарии играют важную роль при работе с долгосрочными проектами.
- 4.4. Если все задания выполнены верно, ответьте на вопросы для самоподготовки и переходите к защите лабораторной работы. На защите продемонстрируйте структуру файлов в удаленном репозитории и историю коммитов.

2.1.6 Вопросы для самоподготовки

1. Какие приложения были установлены вами в ходе лабораторной работы и для чего?
2. Чем отличается понятие «решение» от понятия «проект» в Visual Studio?
3. Как добавить ссылку на проект в другом проекте внутри одного решения?
4. Что такое система версионного контроля и для чего она нужна при разработке ПО?
5. Объясните принцип работы систем версионного контроля. Что такое коммит? Что такое синхронизация? Что такое локальный и удаленный репозиторий?
6. Что такое git и что такое GitHub?
7. Что такое gitignore? Почему нежелательно хранение бинарных файлов под версионным контролем? Почему из версионного контроля необходимо исключать результаты компиляции, находящиеся в папках bin, obj, Release, Debug?
8. Что такое ветка? Как происходит слияние веток? Что такое конфликт при слиянии веток?

2.1.7 Содержание отчета

Отчет по лабораторной работе должен содержать:

- Титульный лист, оформленный согласно требованиям ОС ТУСУР [22].
- Структура файлов репозитория в ветке develop (допустимо в виде снимка экрана).
- Текущая история коммитов ветки develop (допустимо в виде снимка экрана соответствующей страницы GitHub или VisualStudio).

Отчёт сдается с указанием даты защиты лабораторной и подписью студента.

2.1.8 Список источников

Среда разработки Visual Studio

1. Visual Studio: официальный сайт. [Электронный ресурс]. / Microsoft. — URL: <https://visualstudio.microsoft.com/ru/> (дата обращения 6.08.2018).
2. [Видео] Первая программа на C# в Visual Studio 2017: видеокурс по языку программирования C#/ Eugene Popov: Youtube-канал. — URL: https://www.youtube.com/watch?v=y1OXLBDZT7k&t=0s&list=PLL-k0Ff5RfqXGhAooRkUpzMLd6_Fpr13I&index=2 (дата обращения 6.08.2018).
3. Введение в C#: язык C# и платформа .NET. [Электронный ресурс]. / Metanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/tutorial/1.1.php> (дата обращения 6.08.2018).
4. Начало работы:Visual Studio. [Электронный ресурс]. / Metanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/tutorial/1.2.php> (дата обращения 6.08.2018).
5. Введение в C#: язык C# и платформа .NET. [Электронный ресурс]. / Metanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/tutorial/1.1.php> (дата обращения 6.08.2018).

Версионный контроль

6. Git — Book. [Электронный ресурс]. — URL:<http://git-scm.com/book/ru/v1> (дата обращения 18.12.2014).
7. GitHub. [Электронный ресурс]. — URL:<https://github.com> (дата обращения 18.12.2014).
8. Git — Downloading Package. [Электронный ресурс]. — URL:<http://git-scm.com/download/win> (дата обращения 18.12.2014).
9. gitignore/VisualStudio.gitignore. [Электронный ресурс]. — URL: <https://github.com/github/gitignore/blob/master/VisualStudio.gitignore> (дата обращения 18.12.2014).
10. Free Mercurial and Git Client for Windows and Mac | Atlassian SourceTree[Электронный ресурс]. — URL: <http://www.sourcetreeapp.com/> (дата обращения 18.12.2014).
11. Git — GUI Clients [Электронный ресурс]. — URL: <http://git-scm.com/downloads/guis> (дата обращения 18.12.2014).
12. A successful Git branching model// nvie.com. [Электронный ресурс]. — URL: <http://nvie.com/posts/a-successful-git-branching-model/> (дата обращения 18.12.2014).
13. Удачная модель ветвления для Git // Хабрахабр [Электронный ресурс]. — URL: <http://habrahabr.ru/post/106912/> (дата обращения 18.12.2014).
14. [Видео] Git – для новичков: видео-курс / Loftblog: Youtube-канал. — URL: <https://www.youtube.com/watch?v=PEKN8NtBDQ0&list=PLY4rE9dstrJyTdVJpv7FibSaXB4BHPInb> (дата обращения 6.08.2018).

Инструменты сборки установочного пакета

15. Inno Setup: официальный сайт. [Электронный ресурс]. / Jordan Russell's software. — URL: <http://www.jrsoftware.org/isinfo.php> (дата обращения 6.08.2018).

16. Inno Setup: полная официальная документация к программе. [Электронный ресурс]. / Jordan Russell's software. — URL: <http://www.jrsoftware.org/ishelp/> (дата обращения 6.08.2018).
17. Inno Setup: создание инсталлятора на примере развертывания C# приложения. [Электронный ресурс]. / блог пользователя maisvendoo, сайт habr.com. — URL: <https://habr.com/post/255807/> (дата обращения 6.08.2018).

Инструменты разработки проектной документации

18. Sparx Enterprise Architect: официальный сайт приложения. [Электронный ресурс]. / Sparx Systems. — URL: <http://www.sparxsystems.com/products/ea/index.html> (дата обращения 6.08.2018).
19. Sparx Enterprise Architect: официальное руководство пользователя. [Электронный ресурс]. / Sparx Systems. — URL: <http://www.sparxsystems.com/resources/user-guides/> (дата обращения 6.08.2018).
20. [Видео] Видео-руководство Sparx Enterprise Architect: установка программы. [Электронный ресурс]. / Sparx Systems. — URL: <http://www.sparxsystems.com/resources/demos/gettingstarted/installingea.html> (дата обращения 6.08.2018).
21. [Видео] Видео-руководство Sparx Enterprise Architect: первый запуск приложения. [Электронный ресурс]. / Sparx Systems. — URL: <http://www.sparxsystems.com/resources/demos/gettingstarted/firstrun.html> (дата обращения 6.08.2018).

Требования к оформлению отчета

22. ОС ТУСУР 01-2013. Работы студенческие по направления подготовки и специальностям технического профиля. Общие требования и правила оформления. – Томск: ТУСУР, 2013. – 57с. – URL: https://storage.tusur.ru/files/40668/rules_tech_01-2013.pdf (дата обращения 27.08.2018)

2.2 Разработка бизнес-логики приложения

Цель работы: изучить типовые требования, предъявляемые к бизнес-логике приложения, получить умения разработки логики приложения с обеспечением данных требований.

Задачи:

1. Изучить требования и процесс разработки логики приложения.
2. Повторить синтаксис языка C# для разработки объектно-ориентированных программ.
3. Разработать классы, необходимые для работы логики приложения.
4. Обеспечить целостность данных классов с помощью свойств и механизма генерации исключений.

2.2.1 Особенности разработки логики приложения

Базовые понятия объектно-ориентированного программирования^[1, 2, 3]. Исполняемое объектно-ориентированное приложение представляет множество *объектов*, взаимодействующих между собой с помощью *сообщений*. Каждый объект является экземпляром некоторого *класса*. Одному классу может принадлежать множество объектов. Класс определяет атрибуты класса – его *поля* – и спецификацию сообщений, которые ему можно посыпать – *методы* класса.

Все статические поля класса, а также их динамические значения в единый момент времени называются *состоянием* объекта. Все функции, описанные внутри класса (методы), вызываемые относительно объекта, и предназначенные для обработки поведения объекта называются *поведением*. Таким образом, *класс* – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью. Свойство системы, позволяющее объединять в едином описании состояние и поведение объектов, а также скрывать детали своей реализации, называется *инкапсуляция*.

Объекты описывают реальные сущности предметной области и содержат только значимые (с точки зрения работы программы) атрибуты. Фактически, все объекты являются *абстракциями* – описанием реальных сущностей с выделением значимых характеристик объекта и отсеиванием незначимых. Процесс создания абстракций называется *абстрагированием*. Абстрагирование является первым шагом при проектировании объектных систем.

Каждый атрибут класса – поле – описывается уникальным именем и типом данных. Каждый метод класса описывается уникальным именем, входными аргументами и выходным значением. Отличие метода от обычной функции заключается в том, что, находясь в единой области видимости с полями класса, метод может обращаться к значениям полей объекта напрямую, т.е. без передачи их в качестве входных аргументов. Отличие от глобальных переменных в данном случае заключается в том, что метод может обратиться только к полям своего класса, но не может напрямую обратиться к полям других классов, в то время как глобальные переменные доступны для всех функций.

Классы позволяют определить доступность своих полей и методов с помощью *модификаторов доступа*. Модификаторы доступа: *private* – видимость только внутри класса; *protected* – видимость только внутри класса и всех классов-наследников; *public* – видимость в любой точке программы, *internal* – видимость только внутри компилируемой сборки. Модификаторы доступа позволяют обеспечить *скрытие реализации*. Чем меньше полей и

методов класса доступно извне, тем легче его использовать, тестировать и модифицировать. Таким образом, разработчик должен обеспечивать максимально возможное (в разумных пределах) скрытие реализации. **Интерфейс класса** – это набор методов класса, доступных для использования другими классами. Поля класса предпочтительно помещать под модификатор доступа `private` для защиты целостности данных, а доступ к их значениям предоставлять с помощью специальных методов: геттеров и сеттеров.

Методы класса можно разделить на следующие группы: 1) конструкторы; 2) деструкторы; 3) геттеры; 4) сеттеры; 5) операционные. **Конструкторы** – методы класса, вызываемые для создания экземпляра класса и содержание инструкции по инициализации объекта. **Деструкторы** – методы, вызываемые при уничтожении объектов класса; как правило, содержат инструкции для освобождения динамической памяти, используемой внутри объекта. **Геттеры** – методы, позволяющие получить текущие значения полей класса. **Сеттеры** – методы, позволяющие установить новые значения в поля класса. **Операционные** – методы, выполняющие обработку текущего состояния объекта или входных данных метода.

В качестве полей и входных аргументов методов могут выступать объекты других классов. Когда объект является локальной переменной метода другого класса, такое взаимодействие объектов называется **использование**. Когда объект является полем другого класса, такая связь называется **агрегированием**. Агрегирование делится на агрегацию и композицию. Для **композиции** время жизни объекта-контейнера и объекта-поля должны совпадать. Для **агрегации** время жизни объекта-контейнера и объекта-поля различны. Агрегация может связывать экземпляры класса как один ко одному, так и один ко многим, так и в других численных соотношениях. Соотношение объектов между собой при связи агрегировании называется **кардинальностью связи**.

Для создания новых классов может использоваться механизм наследования. **Наследование** – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется **базовым** или **родительским**. Новый класс – **потомком**, **наследником** или **производным классом**. С помощью наследования можно создавать новые классы, а также выделять общую реализацию нескольких классов в один базовый класс для уменьшения дублирования кода.

Указатели на базовый класс – уникальные переменные, способные хранить объекты базового класса и всех производных классов. Особенностью этой переменной является то, что через указатель на базовый класс можно вызвать любые методы, описанные в базовом классе – даже если в указателе храниться экземпляр дочернего класса.

Виртуальные функции – функции класса, реализация которых может быть переопределена в дочерних классах. **Чисто виртуальные функции** – это виртуальные функции, не имеющие реализации в базовом классе. Класс, который имеет хотя бы одну чисто виртуальную функцию, называется **абстрактным**. Создать экземпляры абстрактного класса нельзя, так как в противном случае процессор не сможет определить, что делать при вызове чисто виртуальной функции. Абстрактные классы используются как базовые классы некоторой иерархии наследования, где в дочерних классах будет определяться точная реализация чисто виртуальных методов.

Комбинация виртуальных функций и указателя на базовый класс позволяет получить полезное свойство системы. Так, поместив в указатель на базовый класс с виртуальной

функцией объект дочернего класса с новой реализацией виртуальной функции, мы можем вызвать через указатель виртуальную функцию. Однако при вызове функции будет вызвана реализация дочернего класса, а не базового. При этом, выбор реализации для вызова будет определяться не на этапе компиляции, а на этапе исполнения программы – динамически. Это свойство указателя на базовый класс и виртуальных функций используется для реализации полиморфизма.

Полиморфизм (в ООП) – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Для полиморфизма необходимо создать базовый класс с чисто виртуальными функциями – *интерфейс*. В дочерних классах определяется собственная реализация интерфейса. Таким образом, можно сказать, что дочерние классы обладают единым интерфейсом. В дальнейшем, вне иерархии наследования создается указатель на базовый класс, который может хранить экземпляры дочерних полиморфных классов. Обращаясь через указатель к методам базового класса (через общий интерфейс) мы можем вызвать разную реализацию, в зависимости от объекта, который хранится в указателе в данный момент времени.

Помимо описанных выше классов, в языках программирования есть различные вариации классов и иных сущностей. Например, *статические поля* – поля класса, существующие в едином экземпляре и являющиеся общими для всех объектов данного класса. *Статические методы* – методы, вызов которых может быть осуществлен без создания объектов класса; как правило, через обращение к имени класса. *Статические классы* – классы, для которых создается всего лишь один экземпляр в момент запуска приложения и разрушается при завершении; все поля и методы статического класса являются статическими. Вызывать конструкторы или деструкторы статического класса нет необходимости.

Вложенные классы – классы, описание которых находится внутри других классов. Вложенные классы могут быть скрыты под модификатором доступа внутри класса. Таким образом, вложенный класс можно будет использовать только внутри одного класса, но не в других классах. Помимо классов вложенными могут быть перечисления и обычные структуры.

Для графического представления классов используются диаграммы классов в нотации UML (Unified Modeling Language)^[16-18]. Основные элементы диаграмм классов представлены на рисунках (рис.34-35).

При размещении сущностей на диаграмме классов следует руководствоваться следующими правилами:

- не размещать на одной диаграмме более 12-15 сущностей, так как большее количество сущностей сложно воспринимать.

- сущности классов старайтесь располагать сверху вниз, разделяя их на уровни. На верхних уровнях располагайте базовые классы, интерфейсы и классы-контейнеры, на нижних – дочерние классы, реализации интерфейсов, агрегируемые классы. Классы одинаковой значимости располагайте на одном уровне.

- группируйте классы на диаграмме согласно их пакетам, сборкам или пространствам имен.

- минимизируйте количество пересекающихся линий, переломов на линиях.

- старайтесь размещать классы таким образом, чтобы уменьшить длину линий между классами.

- диаграммы классов предназначены для пояснения ключевых связей между классами. Поэтому показывать все поля или методы необязательно. Например, можно исключить из диаграммы конструкторы и деструкторы; закрытые поля, для которых предоставляется доступ через геттеры и сеттеры.

- Не забывайте оставлять текстовые комментарии к тем или иным классам. Комментарий на диаграмме может быть полезнее, чем отдельное текстовое описание. Например, если вы используете какой-либо паттерн проектирования, укажите его в виде комментария на диаграмме рядом с соответствующими классами. Это сэкономит много времени

Разумеется, выполнить все описанные правила сложно, а иногда невозможно. Поэтому ваша задача найти компромисс в расположении сущностей таким образом, что диаграмма была наиболее читаемой и понятной.

Сущности

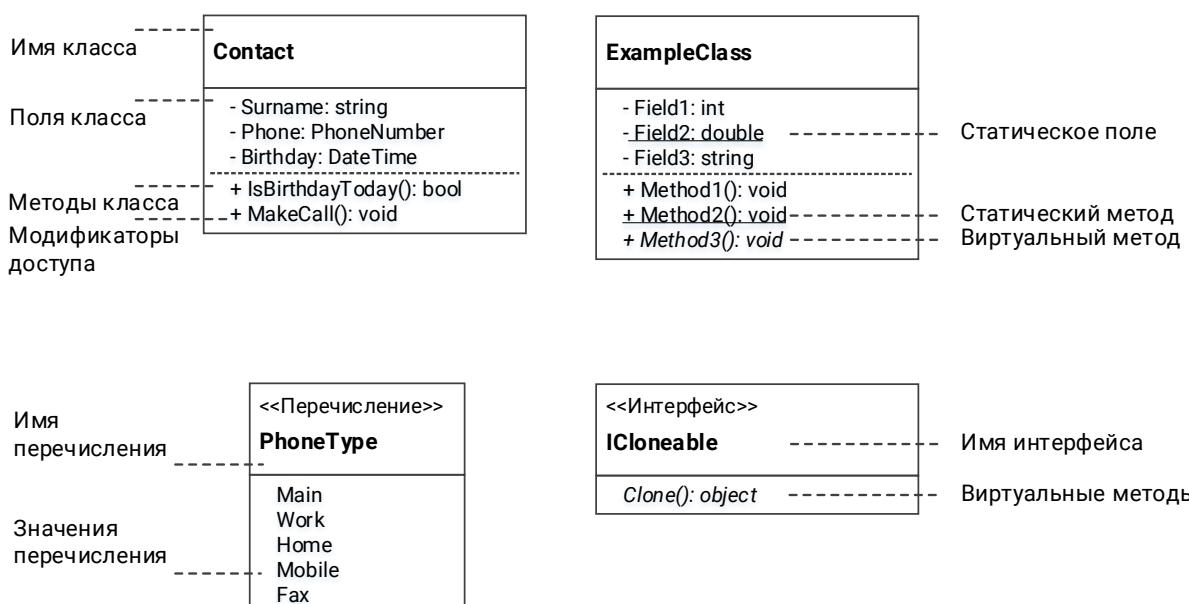


Рисунок 34 – Базовые элементы диаграмм классов

Связи

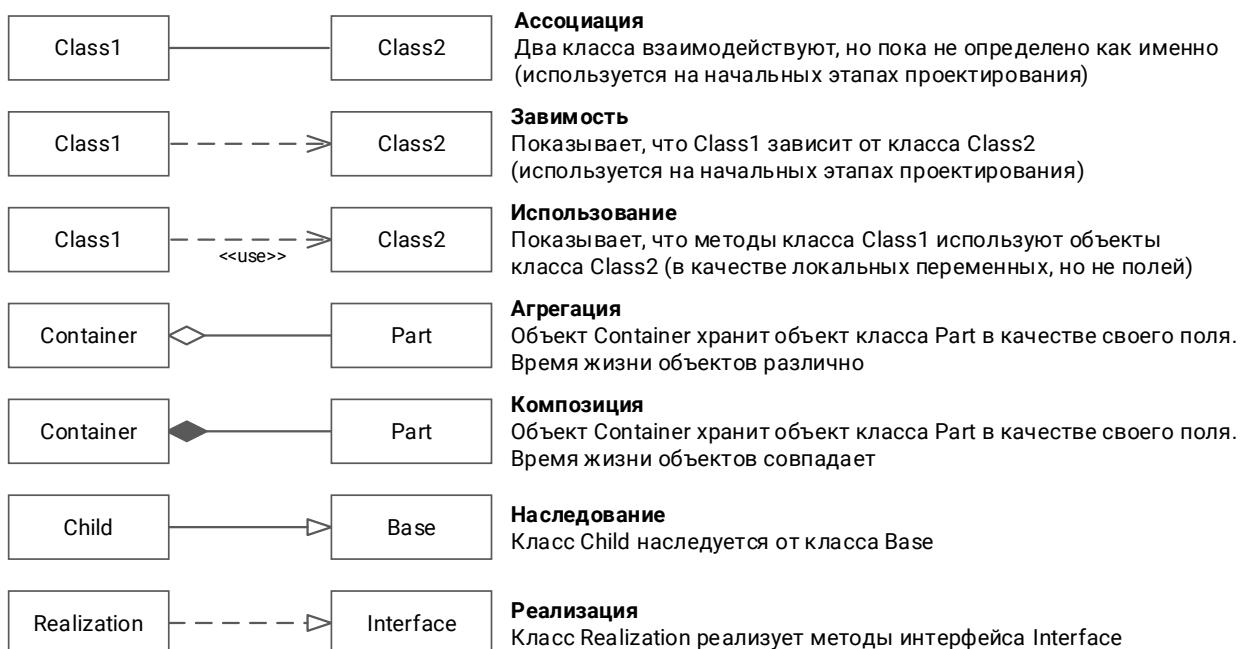


Рисунок 35 – Связи между сущностями на диаграммах классов

Обработка исключений в C# [1, 2, 3, 4]. Механизм обработки исключений необходим для раннего обнаружения ошибок в исходном коде и обеспечения целостности данных в классе. Например, класс Person, описывающий человека, может содержать поле возраста Age. Однако, если поле Age будет открытым, то в возраст может быть присвоено произвольное значение, в том числе и отрицательное:

```
public class Person
{
    public int Age;
}

public class Program
{
    public Main()
    {
        Person person = new Person();
        person.Age = 28; // Корректное значение для поля
        person.Age = -30; // Невозможное значение возраста,
                           // но программа не выдаст ошибки
        // Программа работает дальше, а найти такую ошибку
        // в тысячах строк кода очень сложно
    }
}
```

Для того, чтобы защитить классы от некорректных значений, используем механизм выбрасывания исключений для прерывания программы. Для этого сделаем поле закрытым, а для работы с полем вне класса сделаем метод-геттер и метод-сеттер:

```
public class Person
{
    private int _age;

    public int GetAge() {return _age;} // метод-геттер для поля _age

    public void SetAge(int age) // метод-сеттер поля _age
    {
        if (age < 0)
        {
            throw new ArgumentException("Возраст должен быть меньше 0, а был " + age);
        }
        else
            _age = age;
    }
}

public class Program
{
    public Main()
    {
        Person person = new Person();
        // person._age = 28; !Ошибка компиляции – нельзя обратиться к полю напрямую
        person.SetAge(28); // Значение будет присвоено в поле
        person.SetAge(-30); // Программа завершится во время выполнения этой строки
    }
}
```

```

    // и программист сразу увидит ошибку в этой строке
    // Благодаря аварийному завершению программы становится проще отлаживать про-
    // грамму, а поля класса становятся защищенными от неправильных значений
    }
}

```

Выбрасывание исключения выполняется с помощью ключевого слова `throw`. Оператор `throw` аналогичен оператору `return` – инструкции метода после этого оператора не выполняются, а управление возвращается в точку вызова. Однако, в отличие от `return`, если в точке вызова нет обработки исключения (операторов `try-catch`), то программа аварийно завершится.

Оператор `throw` может выбросить объект любого типа данных, в том числе `int`, `double`, `string` и т.д. Однако следует выбрасывать объекты класса `Exception` и его наследников. Стандартные наследники класса `Exception` это `ArgumentException`, `NullException`, `OutOfRangeException` и др. Но можно создать собственный вид исключений, отнаследовавшись от класса `Exception`.

Зачастую, аварийное завершение программы не требуется. Например, если некорректное значение было введено пользователем, то логичнее было бы запросить пользователя повторный ввод значения, вместо завершения программы и необходимости перезапуска всего приложения. Для того, чтобы сделать реакцию программы на выброс исключения, отличную от аварийного завершения, применяют блоки `try` и `catch`:

```

public class Program
{
    public Main()
    {
        Person person = new Person();
        Console.Write("Введите значение возраста");
        int age;
        Console.Read(age);
        try
        {
            person.SetAge(age);
        }
        catch (ArgumentException exception)
        {
            Console.WriteLine(exception.Message);
            Console.Write("Введите значение возраста");
            Console.Read(age);
            person.SetAge(age);
        }
    }
}

```

В блок `try` помещается код, в котором мы ожидаем выброса исключения. Например, мы не знаем, какое число ввел пользователь. Но поскольку оно может быть и отрицательным, мы ожидаем, что при присвоении возраста в экземпляр `Person` может вылететь исключение.

В блок `catch` помещается код, который будет выполняться в случае выброса исключения. Если исключения в блоке `try` не возникнет, то и весь блок `catch` не будет выполнен, и программа его пропустит. Так, если возраст будет отрицательным, в блоке `try` вылетит исключение, и начнет выполнение блок `catch`. В блоке `catch` мы выводим пользователю текст

ошибки и просим его повторить ввод. Но если пользователь ввел правильный положительный возраст, тогда блок catch не отработает.

В круглых скобках блока catch пишется тип исключения, для которого ниже написана обработка. Если в блоке try могут возникнуть два вида исключения, то мы можем написать последовательно два блока catch для каждого типа исключения со своей уникальной обработкой. Количество блоков catch неограничено.

Стоит помнить, что выбрасывание исключения throw и блоки try-catch должны находиться в разных методах или разных классах. Делать обработку исключения в том же методе, который и выбрасывает это исключение – как правило бессмысленно и усложняет читаемость кода. Подробнее с обработкой исключений можно ознакомиться в [1, 2, 3, 4].

Свойства класса в C# [1, 2, 3, 5]. Представленные выше примеры методов-геттеров и методов-сеттеров не удобны в использовании. Гораздо удобнее, когда работа с полем осуществляется через оператор присваивания. Сравните две формы работы с полем:

```
person.Age = 28; //Присвоение в поле через оператор присваивания  
person.SetAge(28); //Присвоение через метод-сеттер
```

Несмотря на защиту данных от неправильного ввода в сеттере, работа через оператор присваивания выглядит более привычной, естественной формой, и визуально воспринимается легче. Чтобы совместить преимущества обоих подходов, в язык C# были добавлены свойства. Свойства – это методы класса со специальным синтаксисом, предоставляющие доступ к значению поля класса. Исправим ранее описанный пример с обработкой исключений, где методы GetAge() и SetAge() заменим на свойство:

```
public class Person  
{  
    private int _age;  
  
    public int Age  
    {  
        get {return _age;}  
        set  
        {  
            if (value < 0)  
            {  
                throw new ArgumentException("Возраст должен быть меньше 0, а был " +  
value);  
            }  
            else  
                _age = value;  
        }  
    }  
}  
  
public class Program  
{  
    public Main()  
    {  
        Person person = new Person();  
        person.Age = 28; //Аналогично вызову person.SetAge(28)  
        Console.WriteLine(person.Age);  
        //Аналогично вызову Console.WriteLine(person.GetAge())  
    }  
}
```

Обратите внимание на вызов свойства. Обращение к свойству синтаксически выглядит как обращение к открытому полю класса, но при этом внутри свойства могут выполняться дополнительные проверки входных значений и выбрасывание исключений.

Внутри сеттера доступна локальная переменная `value`. Она хранит значение, которое пытаются присвоить свойству извне (в случае представленного примера, это число 28).

Если хотите, можно ограничить использование сеттера, указав перед ним модификатор доступа `private`:

```
public int Age {get{...} private set{...}}
```

В таком случае, сеттер можно будет вызвать только внутри класса, а геттер будет доступен в других классах. Фактически, свойство становится доступным только для чтения. Также можно создавать свойства без сеттера. Подробнее со свойствами, а также их разновидностью – автосвойствами – можно ознакомиться в [1, 2, 3, 5].

Полиморфизм и интерфейсы [1, 2, 3, 7, 8]. В языке C#, в отличие от Си++, появился новый вид сущности – интерфейс. Интерфейс используется для реализации полиморфизма и во многом похож на чисто абстрактные классы. Интерфейс объявляется с помощью ключевого слова `interface`:

```
//интерфейс геометрических фигур
public interface IGeometricFigure
{
    double GetArea(); //все методы в интерфейсе чисто виртуальные
                      //модификатор доступа указывать не нужно
}
```

Все методы в интерфейсе являются чисто виртуальными. Модификатор доступа для методов указывать не нужно – подразумевается, что класс, реализующий интерфейс, обязан реализовать все методы с модификатором доступа `public`:

```
// класс, реализующий интерфейс
public Rectangle: IGeometricFigure
{
    public X {get; set; }
    public Y {get; set; }

    public double GetArea() {return X*Y;}
}
```

Как было сказано ранее, интерфейсы предназначены для реализации полиморфизма и использования полиморфных объектов:

```
IGeometricFigure figure = null;
figure = new Rectangle();
Console.WriteLine(figure.GetArea());
figure = new Circle();
Console.WriteLine(figure.GetArea());
```

В данном примере мы вызываем метод `GetArea()` для объектов разных классов через их общий интерфейс. Реализация метода, который будет выполнен (расчет площади для прямоугольника или для круга), определяется динамически во время выполнения программы.

Интерфейсы имеют ряд отличий от абстрактных классов:

- 1) Все методы интерфейса должны быть реализованы с модификатором доступа `public` в дочерних классах. В абстрактных классах можно указать любой модификатор доступа для чисто виртуальных функций.

- 2) В интерфейсе не может быть никакой реализации. В абстрактных классах помимо чисто виртуальных функций можно объявить несколько методов с реализацией, а также создать поля, константы и т.д.
- 3) Любой класс может реализовывать неограниченное количество интерфейсов, однако может наследоваться только от одного абстрактного класса.

В силу перечисленных отличий, сложилась следующая практика: интерфейсы должны использоваться в том случае, если необходимо реализовать полиморфизм; абстрактные классы использовать, если необходимо вынести общую реализацию нескольких классов в один базовый класс для избавления от дублирования.

Обратите внимание, что в приведенном примере метод GetArea() в интерфейсе может быть заменен на свойство с геттером Area {get;}, что сделает использование метода более удобным для разработчиков.

Подробнее с интерфейсами можно ознакомиться в [\[1, 2, 3, 7, 8\]](#).

Разделение приложения на логику и пользовательский интерфейс.

2.2.2 Нотация оформления кода RSDN

Далее представлены основные положения нотации RSDN. С полным перечнем требований к оформлению можно ознакомиться в [\[9\]](#).

Общие правила.

- 1) Помните! Код чаще читается, чем пишется, поэтому не экономьте на понятности и чистоте кода ради скорости набора.
- 2) Не используйте малопонятные префиксы или суффиксы (например, венгерскую нотацию), современные языки и средства разработки позволяют контролировать типы данных на этапе разработки и сборки.
- 3) Не используйте подчеркивание для отделения слов внутри идентификаторов, это удлиняет идентификаторы и затрудняет чтение. Вместо этого используйте стиль именования Кемел или Паскаль. Единственным исключением являются обработчики событий контрололов на формах (см. именование методов).
- 4) Страйтесь не использовать сокращения лишний раз, помните о тех, кто читает код.
- 5) Страйтесь делать имена идентификаторов как можно короче (но не в ущерб читабельности). Помните, что современные языки позволяют формировать имя из пространств имен и типов. Главное, чтобы смысл идентификатора был понятен в используемом контексте. Например, количество элементов коллекции лучше называть Count, а не CountOfElementsInMyCollection.
- 6) Когда придумываете название для нового, общедоступного (public) класса, пространства имен или интерфейса, страйтесь не использовать имена, потенциально или явно конфликтующие со стандартными идентификаторами.
- 7) Предпочтительно использовать имена, которые ясно и четко описывают предназначение и/или смысл сущности.
- 8) Страйтесь не использовать для разных сущностей имена, отличающиеся только регистром букв. Разрабатываемые вами компоненты могут быть использованы из языков, не различающих регистр, и некоторые методы (или даже весь компонент) окажутся недоступными.

9) Страйтесь использовать имена с простым написанием. Их легче читать и набирать. Избегайте (в разумных пределах) использования слов с двойными буквами, сложным чередованием согласных. Прежде, чем остановиться в выборе имени, убедитесь, что оно легко пишется и однозначно воспринимается на слух. Если оно с трудом читается, и вы ошибаетесь при его наборе, возможно, стоит выбрать другое.

Сокращения.

Не используйте аббревиатуры или неполные слова в идентификаторах, если только они не являются общепринятыми. Например, пишите GetWindow, а не GetWin.

Не используйте акронимы, если они не общеприняты в области информационных технологий.

Если имеется идентификатор длиной менее трех букв, являющийся сокращением, то его записывают заглавными буквами, например, System.IO, System.Web.UI. Имена длиннее двух букв записывайте в стиле Паскаль или Кэмл, например, Guid, Xml, XmlDocument.

Не следует использовать сокращения длиной менее 3 букв, кроме редкого исключения (например, математические константы, Re, Im).

Не следует использовать сокращения короче других, более общепринятых сокращений (Ma вместо Mag).

Не используйте одно и то же имя для класса и пространства имен. Например, не используйте класс *Debug* и пространство имен *Debug*.

Пространства имен должно совпадать с расположением сборок по директориям в проекте, кроме пространства имен верхнего уровня (имя компании).

Все вложенные типы (размещаемые в пространствах имен или непосредственно в глобальном пространстве), за исключением делегатов, должны находиться в отдельных файлах.

Страйтесь объявлять вложенные типы в начале внешнего типа.

Страйтесь не делать излишней вложенности типов. Помните, что вложенными должны быть только тесно связанные типы.

Не используйте литеральные константы (магические числа, защищенные в код размеры буферов, времена ожидания и тому подобное). Лучше определите константу (если вы никогда не будете ее менять) или переменную только для чтения (если она может измениться в будущих версиях вашего класса).

Классы и структуры.

Используйте существительное (одно или несколько прилагательных и существительное) для имени класса.

Используйте стиль Паскаль для регистра букв.

Не используйте специальных префиксов, поясняющих, что это класс. Например, FileStream, а не CFileStream.

В подходящих случаях используйте составные слова для производных классов, где вторая часть слова поясняет базовый класс. К примеру, ApplicationException – вполне подходящее название для класса, унаследованного от Exception, поскольку

`ApplicationException` является наследником класса `Exception`. Не стоит, однако злоупотреблять этим методом, пользуйтесь им разумно. К примеру, `Button` – вполне подходящее название для класса, производного от `Control`. Общее правило может быть, например, таким: «Если производный класс незначительно меняет свойства, поведение или внешний вид базового, используйте составные слова. Если класс значительно расширяет или меняет поведение базового, используйте новое существительное, отражающее суть производного класса». `LinkLabel` незначительно меняет внешний вид и поведение `Label` и, соответственно, использует составное имя.

Используйте составное имя, когда класс принадлежит некоторой специфичной категории, например `FileStream`, `StringCollection`, `IntegrityException`. Это относится к классам, которые являются потоками (`Stream`), коллекциями (`Collection`, `Queue`, `Stack`), ассоциативными контейнерами (`Dictionary`), исключениями (`Exception`).

Для базового класса, предназначенного не для прямого использования, а для наследования, следует использовать суффикс `Base`. Например, `CollectionBase`. Такие классы также необходимо делать абстрактными.

Коллекциям (реализующим интерфейс `ICollection/IList`) нужно давать имя в виде `<ИмяЭлемента>Collection`. Переменным же этих типов лучше давать имена, являющиеся множественным числом от элемента. Например, коллекция кнопок должна иметь имя `ButtonCollection`, а переменная `buttons`.

Интерфейсы.

Используйте описывающее существительное, прилагательное или одно, или несколько прилагательных и существительное для идентификатора интерфейса. Например, `IComponent` – это описывающее существительное, `ICustomAttributeProvider` – это конкретизированное прилагательными существительное, а `IPersistable` – это характеризующее прилагательное.

Если интерфейс описывает поведение, а не объект, то следует использовать суффикс `-able`. (Класс может клонироваться – `ICloneable`)

Используйте стиль Паскаль для регистра букв.

Используйте префикс `I` (заглавная `i`) для интерфейсов, чтобы уточнить, что тип является интерфейсом. Страйтесь избегать интерфейсов с двумя `I` в начале, например, `Identifiable`. Попробуйте подобрать синоним, например, `IRecognizable`.

Для пары класс-интерфейс, в которой класс является некоторой стандартной или эталонной реализацией интерфейса, используйте одинаковые имена, отличающиеся только префиксом `I` для интерфейса. Например, `IConfigurationManager` и `ConfigurationManager`, при этом `ConfigurationManager` не должен быть абстрактным, в этом случае используйте суффикс `Base`.

Перечисления.

Используйте стиль Паскаль для регистра букв в названии и значениях перечисления.

Не используйте суффикс `Enum` в названии типа, вместо этого используйте более конкретный суффикс, например, `Style`, `Type`, `Mode`, `State`. Чтобы код легче читался, используйте следующее правило: «Название перечисления + `is` + значение должно образовывать простое предложение». Например: `BorderStyle.Single` (`Border style is single`, `ThreadState.Aborted --> Thread state is "aborted"`).

Записывайте значения перечисления на отдельных строках. Если элементы имеют определенный семантический порядок, описывайте их именно так, иначе используйте алфавитный порядок. Для каждого элемента необходимы xml-комментарии. Перед комментариями необходима пустая строка.

Имена членов перечисления не должны содержать имени перечисления и другой не относящейся к конкретному значению информации.

Поля и методы.

Непубличные поля (`private`, `protected` и `protected internal`) именуются в стиле Кэмел и начинаются с префикса `_`.

Публичных полей быть не должно, вместо этого следует использовать автосвойства (см. именование свойств).

Одна декларация должна содержать не более одного поля и должна располагаться на одной строке.

Все поля должны в обязательном порядке документироваться XML-комментариями.

Константные поля следует именовать с использованием стиля Паскаль, без префиксов, вне зависимости от модификаторов доступа.

Статические поля именуются в зависимости от модификатора доступа.

Поля только для чтения, должны быть приватными и именоваться согласно модификатору доступа.

Используйте глаголы или комбинацию глагола и существительных и прилагательных для имен методов.

Используйте стиль Паскаль для регистра букв (вне зависимости от области видимости метода).

Если метод является обработчиком события формы, то название метода отделяется от названия события символом подчерка (`«_»`). Например, `ShowFilePathsMenuItem_CheckedChanged`.

Свойства.

Используйте существительное или одно или несколько прилагательных и существительное для имени свойства.

Используйте стиль Паскаль для регистра букв.

В подходящих случаях используйте имя свойства, совпадающее с типом его значения. Одним из критериев для применения этого правила является наличие единственного свойства со значением некоторого (нетривиального) типа. Например, `public Color Color { get; set; }`.

Параметры методов.

Из имени и типа параметра должны быть понятны его назначение и смысл.

Используйте стиль Кэмел для регистра букв в имени параметра.

Старайтесь избегать указания типа в имени параметра.

Имена параметров не должны совпадать с именами членов класса. Это должно решаться префиксом `«_»` во всех полях (предполагается, что ВСЕ поля приватные).

Оформление.

Используйте табуляцию в 4 пробела, а не пробелы для отступов. Количество пробелов необходимо выставить в настройках IDE.

Избегайте строк длинной больше ширины экрана. Конечно данный совет зависит от форматности экрана и настроек IDE, но это примерно 100 символов в строку.

При переносе части кода инструкций и описаний на другую строку вторая и последующая строки должны быть отбиты вправо на один отступ (табуляцию).

Оставляйте запятую на предыдущей строке так же, как вы это делаете в обычных языках (русском, например).

Избегайте лишних скобок, обрамляющих выражения целиком. Лишние скобки усложняют восприятие кода и увеличивают возможность ошибки. Если вы не уверены в приоритете операторов, лучше загляните в соответствующий раздел документации.

Не размещайте несколько инструкций на одной строке. Каждая инструкция должна начинаться с новой строки.

Пустые строки.

Перед определением пространства имен в файле не должно быть строки.

После определения пространства имен две строки (перед директивами using).

После директив using должно быть 2 строки.

Одна пустая строка перед и после объявления поля, свойства, делегата, события, метода.

Одна пустая строка до и после директив #region #endregion (в объявление входят xml-комментарии).

Если переменные в методе объявляются отдельным блоком, используйте одну пустую строку между их объявлением и инструкцией, идущей за этим блоком.

Используйте одну пустую строку между логическими частями в методе. Например, условными блоками и циклами.

Перед комментариями внутри методов необходимо оставлять одну пустую строку.

Локальные переменные.

Объявляйте переменные непосредственно перед их использованием.

Если необходимы счетчики в циклах, не отражающие реальных объектов, традиционно называют i, j, k, l, m, n.

Инициализируйте переменные при объявлении, если это возможно.

При объявлении с инициализацией используйте ключевое слово var. Это же касается определений в инструкциях for, foreach, using. Название класса при объявлении допускается только если ожидается (или необходимо) приведение типа, или инициализация невозможна.

```
var list = new List<int>();
IEnumerable<int> list = new List<int>();
```

Комментарии.

Не используйте многострочные /*...*/ комментарии для описания классов и методов, используйте для этих целей xml-documentation-комментарии. Многострочные комментарии не могут быть вложенными, поэтому их использование может создать проблемы.

Должны быть заполнены все поля xml-комментариев (summary, parameters, return и т.д.)

Для описания сути некоторого участка кода, пояснений к алгоритму и другой важной информации используйте несколько подряд идущих односрочных комментариев (///...).

Отделяйте текст комментария одним пробелом «// Текст комментария.». В конце обязательна точка.

Помните, что экономить на комментариях нельзя. Однако не стоит также формально подходить к процессу создания комментариев. Задача комментария – упростить понимание кода. Есть немало случаев, когда сам код отличным образом себя документирует.

Определения классов тоже необходимо комментировать xml-комментариями.

Комментарии к строке кода всегда должны располагаться над ней, а не после «;». Это делается для того, чтобы при различных переименованиях или рефакторинге кода комментарий не вышел за границы рабочей области IDE, что привело бы к его неудобочитаемости.

2.2.3 Xml-комментарии

xml-комментарии – комментарии специального формата в исходном коде для автодокументирования кода [10, 11]. Термин «автодокументирование» означает, что документация по использованию и работе кода описана непосредственно в исходном коде.

До начала 2000-х годов документация по исходному коду представляла собой отдельный документ-руководство с перечислением всех классов, полей и методов в программе с примерами их использования для решения задач. Однако подход с отдельным документом приводил к проблеме актуализации документации – при активном развитии проекта исходный код меняется быстро, и зачастую разработчики забывают исправить соответствующие разделы.

Xml-комментарии стали изящным решением данной проблемы, совместив документацию с самим исходным кодом. Благодаря такому подходу, вы можете посмотреть комментарии и руководство к библиотеке непосредственно в среде разработки. Вся необходимая информация будет храниться в самой сборке.

Более того, современные среды разработки, такие как Visual Studio, показывают xml-комментарии во время автодополнения при наборе исходного кода или даже просто при наведении мыши на поле, метод или класс. Таким образом, во время набора кода при работе с классами сторонней библиотеки с помощью автодополнения мы можете не только оперативно посмотреть перечень членов класса, но и узнать более подробную документацию по каждому из методов. Особенно это удобно при множестве перегруженных методов в классе, где xml-комментарии описывают входные аргументы каждого из методов и позволяют выбрать наиболее подходящий вариант функции.

И третьим преимуществом xml-комментариев является возможность генерировать документацию к коду в отдельный документ. Если по какой-либо причине вам всё-таки понадобится отдельное руководство по вашему исходному коду, вы можете найти готовые плагины или программы для генерации руководства на основе вашего исходного кода с xml-комментариями. Примером таких автогенерированных руководств может служить сайт msdn.microsoft.com, где содержится полная документация по платформе .NET. Все таблицы с полями, методами, классами и их описанием генерированы автоматически на ос-

нове исходного кода. Ручное создание такого количества таблиц заняло огромное количество человеко-часов, а поддержание документации в актуальном состоянии было бы крайне затруднительно.

А значит, применение xml-комментариев может сэкономить вам много времени при чтении кода, написании кода, а также при создании проектной документации.

Однако, чтобы все эти преимущества были доступны, необходимо выработать привычку написания xml-комментариев сразу же во время написания кода. Рассмотрим примеры xml-комментариев.

```
/// <summary>
/// Класс аккаунта, хранящий информацию о логине, пароле
/// и аватаре зарегистрированного пользователя.
/// </summary>
public class Account
{

}
```

Как представлено в примере, xml-комментарий начинается с тройного слэша «///». После тройного слэша используются специальные xml-тэги. К обязательным тэгам в xml-комментариях относится тэг `<summary>` - тэг описания («summary» – с англ. «описание»). Тэг `<summary>` является открывающим тэгом, а `</summary>` - закрывающим. Весь текст, находящийся между этими двумя тэгами будет считаться описанием класса.

Теперь, если где-либо в коде вы напишите, например, следующую строку:

```
Account account = new Account();
```

То при наведении курсора мыши на название класса `Account` среда разработки покажет текст xml-комментария во всплывающей подсказке:

```
public MainForm()
{
    InitializeComponent();
    Account account = new Account();
}
```

The screenshot shows a code editor window with the following C# code:

```
public MainForm()
{
    InitializeComponent();
    Account account = new Account();
}
```

A tooltip is displayed over the word "Account". The tooltip contains the XML documentation:

Класс аккаунта, хранящий информацию о логине, пароле и аватаре зарегистрированного пользователя.

Рисунок 36 – Пример подсказки с xml-комментарием при наведении на название класса

Или во время набора названия класса, при автодополнении:

The screenshot shows a code editor window with the following C# code:

```
public MainForm()
{
    InitializeComponent();
    Acc
```

An auto-completion dropdown is open at the cursor position. It lists several items, including "Account". To the right of the dropdown, a tooltip displays the XML documentation:

Класс аккаунта, хранящий информацию о логине, пароле и аватаре зарегистрированного пользователя.

Рисунок 37 – Пример подсказки с xml-комментарием в автодополнении

Не нужно запоминать и вводить конструкцию xml-комментария каждый раз вручную. Как только вы наберете в среде разработки три слэша, среда разработки Visual Studio автоматически вставит на место курсора заготовку xml-комментария с тэгом <summary>.

Аналогично тэг <summary> пишется для полей, методов, интерфейсов, перечислений и т.д.:

```
/// <summary>
/// Класс аккаунта, хранящий информацию о логине, пароле
/// и аватаре зарегистрированного пользователя.
/// </summary>
public class Account
{
    /// <summary>
    /// Возвращает и задает имя пользователя.
    /// </summary>
    public string Username { get; set; }

    /// <summary>
    /// Возвращает и задает пароль пользователя.
    /// </summary>
    public string Password { get; set; }

    /// <summary>
    /// Возвращает и задает аватар пользователя.
    /// </summary>
    public Image Avatar { get; set; }

    /// <summary>
    /// Метод, выполняющий смену пароля пользователя.
    /// </summary>
    public void ChangePassword(string newPassword)
    {
    }
}
```

Первоначально использование и написание xml-комментариев кажется избыточным, так как на одну строку с объявлением поля или свойства приходится три новых строки с комментарием. Однако после некоторой практики вы привыкните читать код с xml-комментариями. Также сами xml-комментарии могут показаться самоочевидными. Для приведенного примера это действительно так. Однако в большинстве случаев назначение полей и методов самоочевидно только разработчику, написавшему класс. Когда же вашим классом захочет воспользоваться другой разработчик, без xml-комментариев (даже самоочевидных на первый взгляд) ему будет гораздо сложнее разобраться с тем, как именно вызывать методы вашего класса.

Помимо основного тэга <summary> в xml-комментариях есть еще множество дополнительных тэгов, таких как <example>, <exception>, <param>, <paramref>, <remarks>, <returns>, <value> и др. Например, тэг <param> позволяет описать входной аргумент метода, а тэг <returns> возвращаемое значение:

```
/// <summary>
/// Метод, выполняющий смену пароля пользователя.
/// </summary>
/// <param name="newPassword">Значение нового пароля. Новый пароль не должен совпадать со старым. </param>
/// <returns>Значение показывает, прошла ли смена пароля успешно.</returns>
public bool ChangePassword(string newPassword){}
```

Обратите внимание на описание в тэге <param>. Если назначение входного аргумента другому разработчику будет самоочевидно (newPassword совместно с именем метода даст понять, что нужно передать новый пароль), то информация о том, что новый пароль не должен совпадать со старым – важная и полезная информация для стороннего разработчика. Именно для такой дополнительной специфической информации и используются тэги <param> и <returns>. Например, если в качестве аргумента нельзя передавать пустые списки или null в качестве объекта – предупредите разработчика об этом в xml-комментарии.

Xml-комментарии методов также будут показываться при автодополнении и наведении курсора мыши на метод:

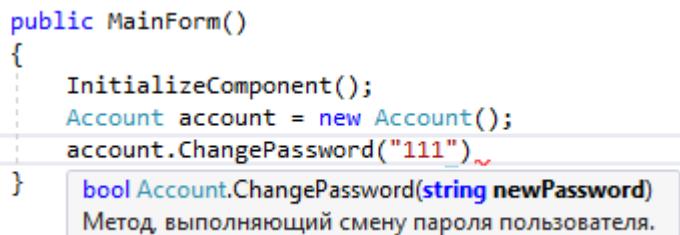


Рисунок 38 – Подсказка с xml-комментарием метода

Для примера посмотрите, насколько полезными могут быть xml-комментарии в случае перегрузки методов и множества входных аргументов:

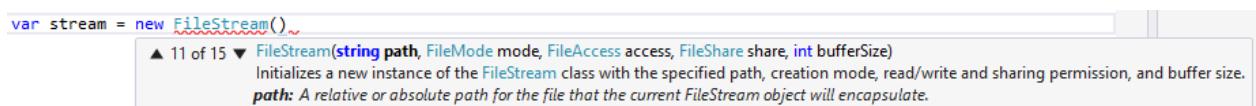


Рисунок 39 – При использовании тэга <param> автодополнение показывает комментарии к каждому входному аргументу

Стрелочки позволяют выбрать описание нужного варианта перегруженного метода (для стандартного класса `FileStream` существует 15 перегрузок конструктора), а в процессе написания входных аргументов курсивный текст во всплывающей подсказке будет показывать описание текущего входного аргумента.

В завершение данного раздела хотелось бы сказать, что требования к заполнению xml-комментариев отличаются в разных компаниях. В одних компаниях могут требовать описывать комментарии вплоть до генерируемых исключений и ссылок на используемые типы данных – и это будет проверять тимлид при проверке кода. В других же компаниях xml-комментариями могут быть желательными, но необязательными. В данном курсе лабораторных работ xml-комментарии *обязательны ко всем классам, интерфейсам, полям, методам, перечислениям*, но достаточны будут только тэги <summary>. Подробнее об использовании xml-комментариев можно узнать в [10, 11].

2.2.4 Добавление ссылок на проекты внутри решения

Для удобства разработки программы, а также оптимизации компиляции и работы приложения, исходный код принято делить на отдельные проекты-сборки. Как правило, можно выделить один проект, компилирующийся в исполняемый файл *.exe – проект с пользовательским интерфейсом, а также проекты бизнес-логики, компилирующиеся в *.dll. Дополнительно могут быть сборки с юнит-тестами, ресурсами и другие.

Для того, чтобы классы одного проекта можно было использовать в другом проекте решения, необходимо подключить ссылку на проект [12, 13].

Рассмотрим вариант решения, разделенного на два проекта: NoteApp – бизнес-логика приложения; и NoteAppUI – пользовательский интерфейс программы:

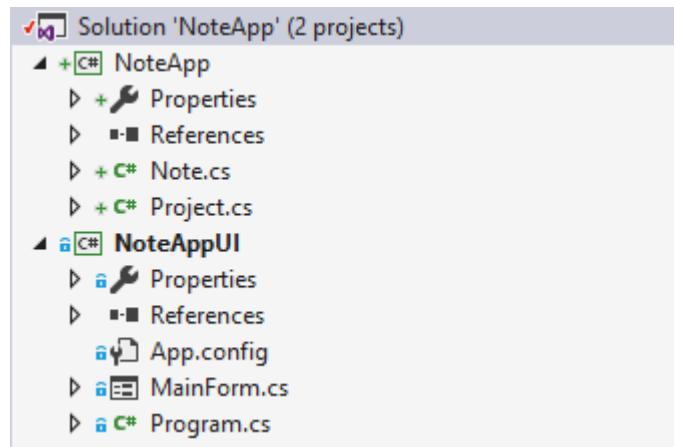


Рисунок 40 – Дерево решения с проектом пользовательского интерфейса и проектом логики

Для того, чтобы мы могли использовать классы из проекта NoteApp в проекте NoteAppUI, необходимо кликнуть правой кнопкой мыши по названию проекта NoteAppUI или пункту References («ссылки») внутри проекта:

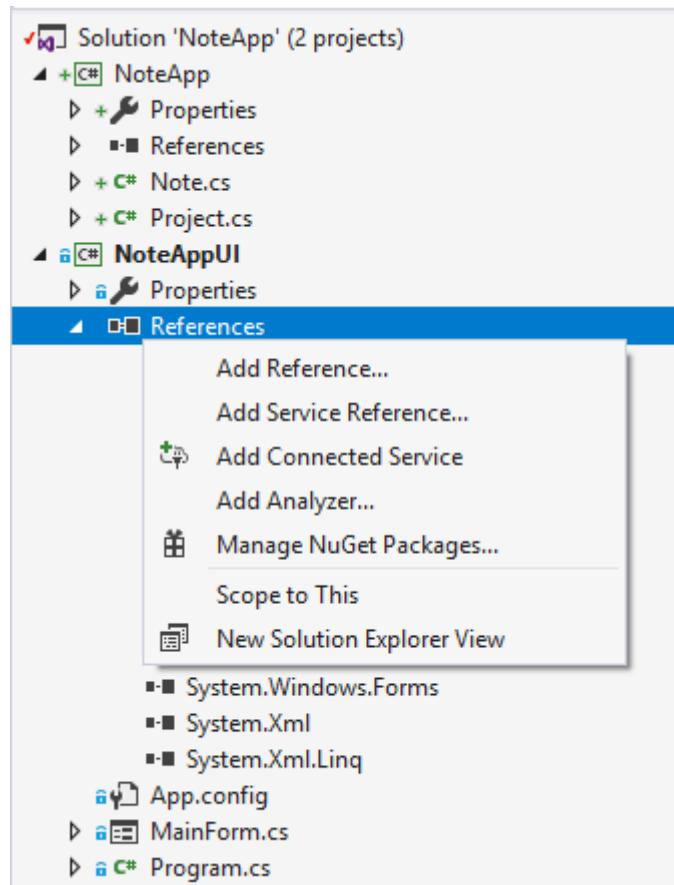


Рисунок 41 – Контекстное меню добавления ссылок на проект

В появившемся контекстом меню выберите пункт Add Reference. Появится следующее окно:

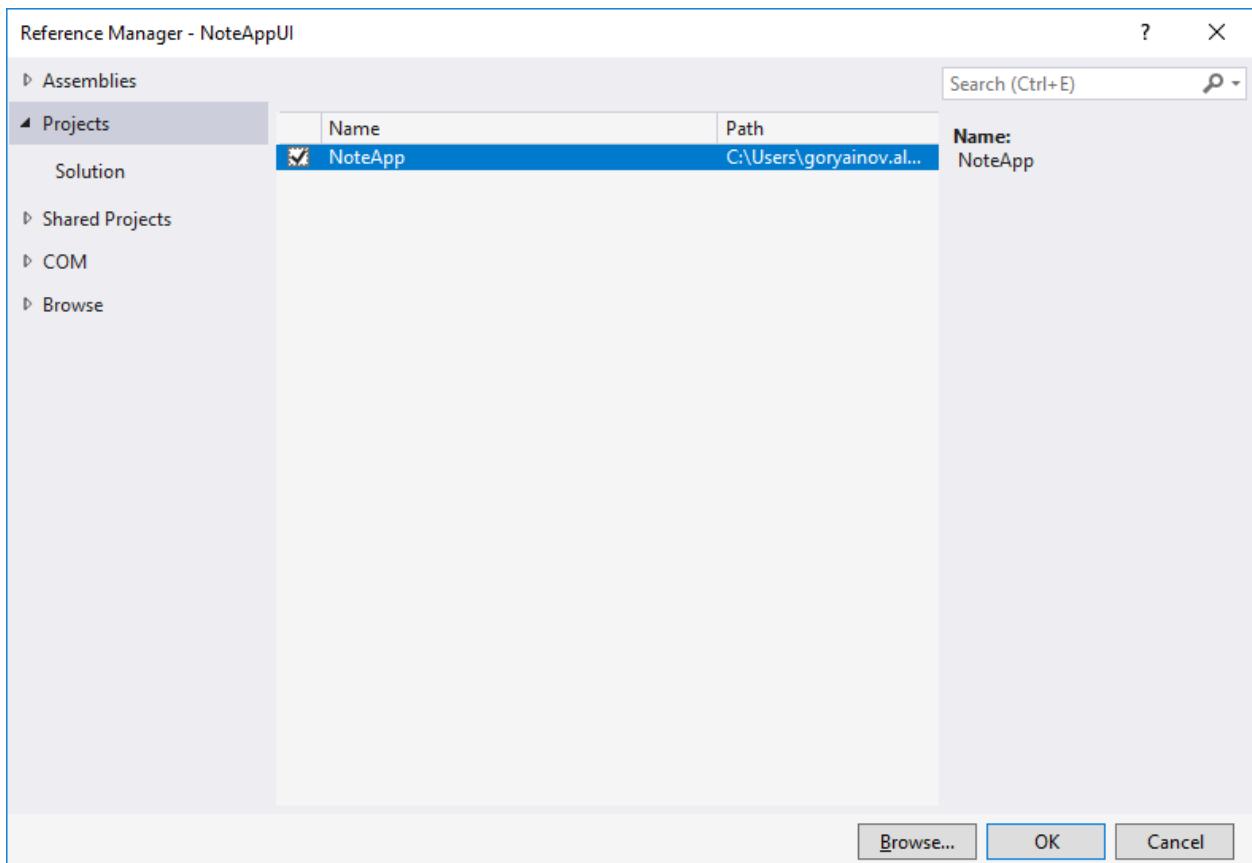


Рисунок 42 – Диспетчер управления ссылками проекта

В данном окне можно подключить различные библиотеки:

Assemblies – добавление стандартных библиотек платформы .NET и установленных сторонних расширений.

Projects – добавление ссылок на проекты, размещенные в текущем решении.

СОМ – добавление ссылок на зарегистрированные СОМ-библиотеки.

Browse – добавление ссылок на отдельные dll-файлы.

В текущем примере нас интересует вкладка Projects. Поставьте галочку напротив названия проекта, который вы хотите подключить, и нажмите OK. Если вы всё сделали правильно, то подключенный проект появится в списке библиотек References. Если же название библиотеки отсутствует или рядом с названием появился желтый предупреждающий значок – есть проблемы с подключением библиотеки. Для удаления подключенных библиотек достаточно выбрать нужную библиотеку в списке References и нажать клавишу Delete.

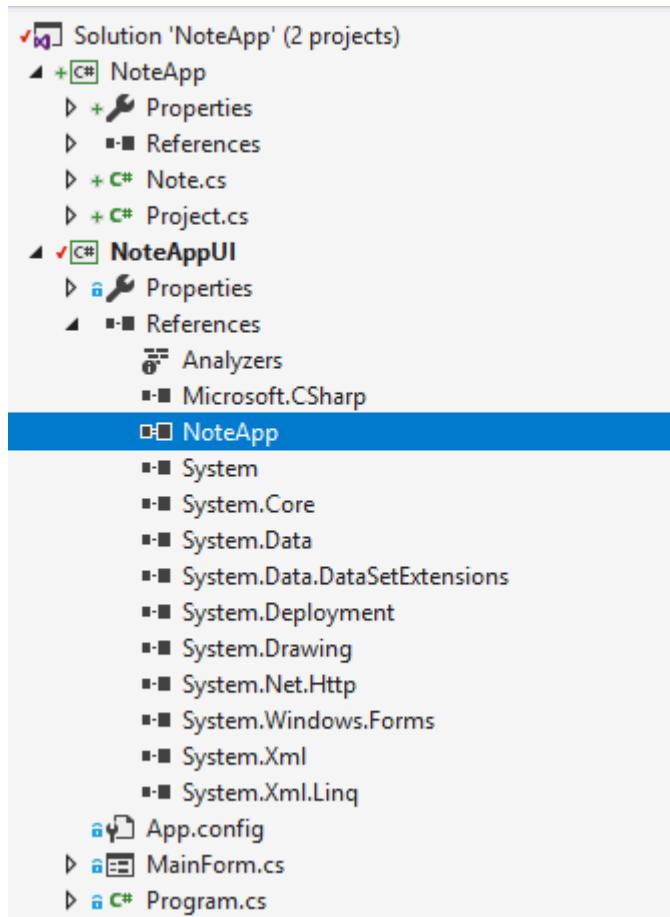


Рисунок 43 – Результат добавления ссылки на проект логики

Теперь мы можем использовать классы из проекта NoteApp в проекте NoteAppUI:

```
using NoteApp;
```

```
namespace NoteAppUI
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
            var project = new Project();
        }
    }
}
```

Обратите внимание, что для использования классов других проектов необходимо в начале файла подключить пространство имён, в котором находится используемый класс (директива `using` в примере кода). Также, для возможности использования классов в других проектах, класс должен быть объявлен с модификатором `public`:

```
namespace NoteApp
{
    public class Project {}
}
```

В противном случае, использовать класс в других проектах будет невозможно: компилятор сообщит об ошибке с правами доступа. Если же вы хотите, чтобы класс был доступен только внутри своего проекта, используйте модификатор доступа `internal`.

2.2.5 Добавление ссылок на сторонние библиотеки через NuGet

Стандартный способ подключения сторонних библиотек обладает своими недостатками. Например, если у сторонней библиотеки выходит обновление, то переподключение новой версии библиотеки приходится делать вручную. Это особенно важно в случае критических обновлений, обновлений библиотек связанных с безопасностью данных. Также иногда возникает потребность в откате версий используемых библиотек на более ранние. Для того, чтобы упростить перечисленные задачи, был разработан новый механизм подключение .NET-сборок – менеджер подключения пакетов NuGet [14].

Для подключения сборки в проект с помощью NuGet, кликните правой кнопкой по пункту References целевого проекта в окне Solution Explorer, и выберите в появившемся контекстом меню пункт Manage NuGet Packages («управление пакетами NuGet»):

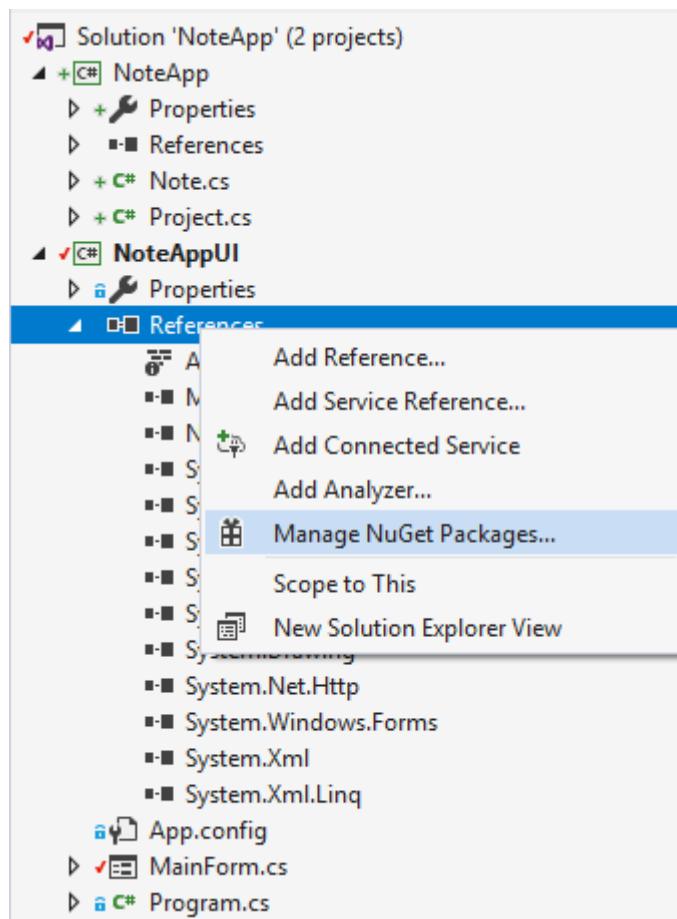


Рисунок 44 – Пункт меню для добавления ссылок на пакеты через NuGet

После выбора пункта меню появится вкладка менеджера пакетов (см. рисунок далее). В данной вкладке можно установить новые пакеты (вкладка Browse), так и управлять уже подключенными сборками (вкладка Installed). Для подключения сторонних сборок необходимо подключение к Интернету. Для каждой сборки перед установкой вы можете выбрать нужную версию (в том числе, ранние стабильные версии), а также прочитать описание. Скачивание библиотеки и установка может занять некоторое время.

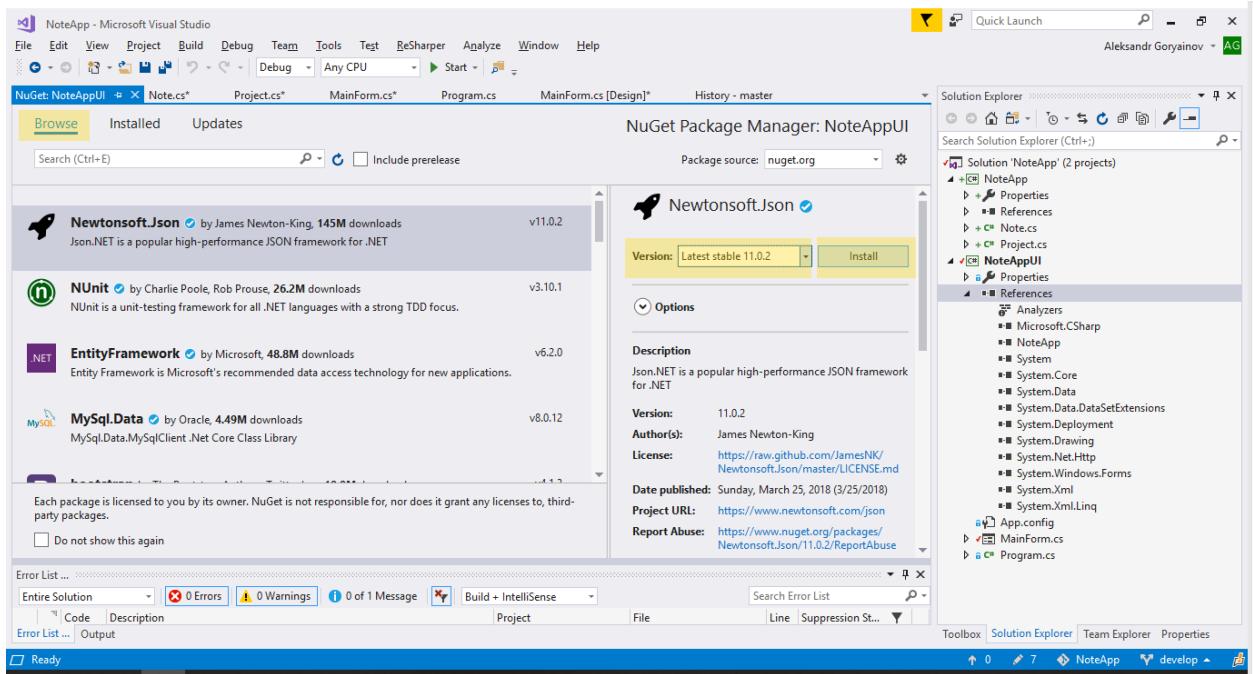


Рисунок 45 – Вкладка менеджера пакетов NuGet

После успешной установки библиотеки, ссылка добавится в проект, а в корне проекта появится файл packages.config:

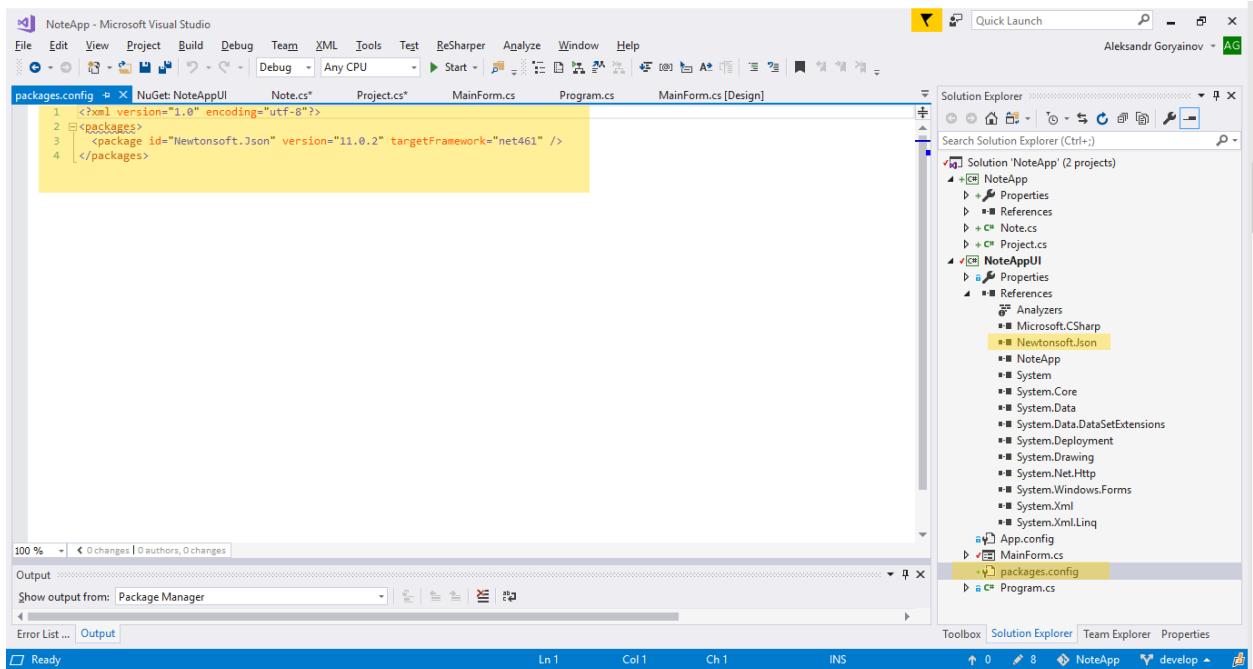


Рисунок 46 – packages.config хранит ссылки всех подключенных через NuGet пакетов

В данном файле хранится информация о подключенных через NuGet сборках, поэтому удалять данный файл не нужно.

Небольшое примечание к работе NuGet: при открытии вашего решения на другом компьютере, Visual Studio изначально не сможет найти подключенные сборки и обозначит библиотеки NuGet как отсутствующие (появятся соответствующие ошибки компилятора, а

сами сборки будут отмечены желтыми значками). Однако во время первой компиляции решения, среда разработки автоматически установит требуемые библиотеки, и программа будет работать корректно.

Менеджер NuGet автоматически вас предупредит в случае появления обновлений к используемым библиотекам, обновление библиотек можно будет выполнить в нажатие одной кнопки.

2.2.6 Библиотека сериализации Newtonsoft JSON.NET

Библиотека JSON.NET [\[15\]](#) от компании Newtonsoft предназначена для сериализации и десериализации объектов в текстовый формат json. Несмотря на наличие в .NET нескольких стандартных механизмов сериализации, библиотека от компании Newtonsoft получила более широкое распространение. Преимуществом сторонней библиотеки является возможность сериализации стандартных коллекций (массивов, списков и словарей), также сериализация объектов, хранящихся интерфейсных переменных.

Сериализация в формате json используется для передачи данных между частями системы (например, клиент-серверных приложений или распределенных систем), сохранения данных о самой системе, представления конфигурационных файлов.

Библиотека Newtonsoft JSON.NET доступна для подключения в проект через средства NuGet (см. 2.2.5, [\[14\]](#)). В случае, если подключить библиотеку через NuGet не представляется возможным, библиотека может быть скачана с официального сайта и подключена как обычная dll-сборка [\[12, 13\]](#).

Предположим, что у нас есть класс Product, описывающий товар интернет-магазина:

```
public class Product
{
    public string Name { get; set; }

    public double Price { get; set; }

    public string[] Sizes { get; set; }
}
```

И нам необходимо сериализовать экземпляр данного класса. Для начала создадим объект, который будем сериализовывать:

```
var product = new Product();
product.Name = "T-Shirt";
product.Price = 1100.0;
product.Sizes = new string[]
{
    "S",
    "M",
    "L",
    "XL"
};
```

Теперь используем библиотеку JSON.NET для сериализации:

```
//Создаём экземпляр сериализатора
JsonSerializer serializer = new JsonSerializer();

//Открываем поток для записи в файл с указанием пути
using (StreamWriter sw = new StreamWriter(@"c:\json.txt"))
using (JsonWriter writer = new JsonTextWriter(sw))
{
    //Вызываем сериализацию и передаем объект, который хотим сериализовать
    serializer.Serialize(writer, product);
}
```

Результатом работы кода будет файл json.txt следующего содержания:

```
{  
    "Name": "T-Shirt",  
    "Price": 1100.0,  
    "Sizes": [  
        "S",  
        "M",  
        "L",  
        "XL"  
    ]  
}
```

Аналогично выполняется десериализация:

```
//Создаём переменную, в которую поместим результат десериализации  
Product product = null;
```

```
//Создаём экземпляр сериализатора  
JsonSerializer serializer = new JsonSerializer();  
  
//Открываем поток для чтения из файла с указанием пути  
using (StreamReader sr = new StreamReader(@"c:\json.txt"))  
using (JsonReader reader = new JsonTextReader(sr))  
{  
    //Вызываем десериализацию и явно преобразуем результат в целевой тип данных  
    product = (Product)serializer.Deserialize<Product>(reader);  
}
```

Обратите внимание на необходимость явного преобразования типов при десериализации.

Класс JsonSerializer содержит ряд полей-настроек, используемых для форматирования результирующего файла, а также указания настроек самой сериализации. Из наиболее значимых настроек можно выделить:

JsonSerializer.NullValueHandling – настройка, указывающая как записывать поля, принимающие значение null.

JsonSerializer.TypeNameHandling – настройка, указывающая необходимо ли записывать названия типов данных для сериализуемых объектов. Настройку необходимо использовать при сериализации полей интерфейсного типа данных, а также шаблонных коллекций.

JsonSerializer.Formatting – настройка, указывающая форматирование в итогом файле. Например, автоматическая табуляция для вложенных типов данных.

JSON.NET может сериализовать объекты любой сложности. Если вы сериализуете класс, агрегирующий другие типы данных, то все агрегируемые объекты также будут сериализованы. Однако при сериализации стоит избегать сериализации крупных объектов (например, сериализовать экземпляр главного окна программы MainForm будет плохой идеей), а также избегать сериализации объектов с циклическими ссылками (когда объект A агрегирует объект B, в котором хранится ссылка на объект A).

Подробная документация по работе с библиотекой находится на официальном сайте компании [\[15\]](#). На сайте также есть обширные примеры исходного кода, и форум с обратной связью, где можно задать вопросы по работе библиотеки разработчикам.

2.2.7 Задание

1. Реализовать классы логики:

- 1.1. Добавить в проект логики классы и другие сущности согласно примечаниям руководителя, описанных в варианте задания (см. вариант задания). Каждый созданный

тип данных должен быть в отдельном файле расширения .cs, имя файла должно совпадать с именем созданного типа данных. Убедиться, что классы находятся под модификаторами доступа public.

- 1.2. Создать поля и свойства для этих полей внутри каждого класса. Убедиться, что все поля класса находятся под модификатором private (protected при необходимости), а свойства – под модификатором public.
- 1.3. Создать конструкторы в классах, если это необходимо.
- 1.4. Добавить методы прямого назначения класса и реализовать их логику.
- 1.5. Внутри свойств реализовать проверку правильности вводимых значений. Ограничения на присваиваемые значения должны быть определены согласно описанию программы в индивидуальном варианте или здравому смыслу.
- 1.6. Если присваиваемое значение не подходит под ограничения, свойство должно выбросить исключение типа ArgumentException (предполагается, что блоки try-catch будут в пользовательском интерфейсе).

2. Проверить правильность работы созданных классов:

- 2.1. Убедиться, что названия классов, перечислений, полей и методов соответствуют нормативии оформления кода RSDN [\[9\]](#).
- 2.2. Добавить xml-комментарии к классам, их полям, методам и другим типам данных.
- 2.3. В исходном коде любой формы проекта пользовательского интерфейса создать экземпляры классов логики, проинициализировать их значениями, вызвать их методы. Результаты вызова методов отобразить на экране (в текстовом поле, всплывающем сообщении или любым другим способом).
- 2.4. Если логика приложения работает правильно и без ошибок, сделайте коммит «Разработана логика приложения» и через двоеточие перечислите классы, которые были реализованы.

3. Реализовать сохранение и загрузку данных программы с помощью механизма сериализации:

- 3.1. Через встроенную службу пакетов NuGet добавить ссылку на библиотеку Newtonsoft JSON.NET в проект логики. Перекомпилируйте решение для завершения добавления библиотеки.
- 3.2. Добавить в проект логики класс, отвечающий за сериализацию. Какие данные должен сериализовать класс, описано в индивидуальном варианте.
- 3.3. Созданный класс должен быть статическим, и иметь два статических метода для сохранения и загрузки данных:

```
public static void SaveToFile(<type> data, string filename);
public static <type> LoadFromFile(string filename);
```

где <type> - тип данных, который требуется сериализовать согласно варианту.
- 3.4. Убедитесь, что сериализация работает верно. Для этого напишите проверочный код в форме проекта пользовательского интерфейса. Проверочный код должен создавать и инициализировать объект для сериализации, выполнять сохранение данных в указанный файл с помощью написанного класса, а затем загружать сохраненный объект из файла.
- 3.5. Проверьте правильность оформления кода согласно требованиям RSDN, а также наличие xml-комментариев в классе сериализации.
- 3.6. Если проект компилируется без ошибок, код оформлен правильно и сериализация работает верно, выполните коммит с комментарием «Добавлена библиотека

Newtonsoft JSON.NET. Добавлен класс сериализации». Синхронизируйте коммиты с удаленным репозиторием.

4. Проверить, что задание лабораторной работы выполнено верно:

- 4.1. Зайдите в свой аккаунт GitHub и откройте репозиторий проекта. Переключитесь на ветку develop для просмотра её текущего состояния. Убедитесь, что новые классы действительно были синхронизированы с удаленным репозиторием.
- 4.2. Зайдите в историю коммитов ветки develop. Убедитесь, что в истории ветки develop на GitHub появились ваши последние коммиты.
- 4.3. Если история коммитов отличается – это значит, вы не выполняли требуемые коммиты или не синхронизировались с удаленным репозиторием.
- 4.4. Составьте UML-диаграмму классов ^[16, 17] проекта логики с помощью программы Enterprise Architect ^[18] или Microsoft Visio.
- 4.5. Если все задания выполнены верно, ответьте на вопросы для самоподготовки и переходите к защите лабораторной работы. На защите продемонстрируйте работу программы, структуру файлов в удаленном репозитории и историю коммитов.

2.2.8 Вопросы для самоподготовки

1. Зачем необходимо разделять программу на проект логики и проект пользовательского интерфейса?
2. Что такое свойства класса? В чём преимущество их использования по сравнению с обычными полями класса?
3. Для чего нужен механизм обработки исключений? Какие ключевые слова используются для генерации и обработки исключений?
4. Что такое полиморфизм?
5. В чём отличие интерфейса от класса?
6. В чём преимущества сторонней библиотеки сериализации JSON.NET от компании Newtonsoft перед стандартными библиотеками сериализации?
7. Как на UML-диagramмах обозначаются агрегация, использование, наследование?
8. Как на UML-диagramмах обозначаются виртуальные функции? Как обозначаются статические методы?
9. Что такое стереотип в UML?

2.2.9 Содержание отчета

Отчет по лабораторной работе должен содержать:

- Титульный лист, оформленный согласно требованиям ОС ТУСУР ^[19].
- UML-диаграмма классов проекта логики с пояснением назначения классов.
- Пример исходного кода, демонстрирующий создание и инициализацию экземпляров каждого класса программы, а также вызов их методов.
- Текущая история коммитов ветки develop (допустимо в виде снимка экрана соответствующей страницы GitHub или VisualStudio).

Отчёт сдается с указанием даты защиты лабораторной и подписью студента.

2.2.10 Список источников

Язык программирования C#

1. Введение в C#: язык C# и платформа .NET. [Электронный ресурс]. / Metanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/tutorial/1.1.php> (дата обращения 6.08.2018).
2. Шилдт Г. C# 4.0: полное руководство.: пер. с англ. Бернштейн И.В. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.: ил.
3. [Видео] Видеокурс по языку программирования C#/ Eugene Popov: Youtube-канал. — URL: https://www.youtube.com/watch?v=GeUrbwSYFvM&list=PLL-k0Ff5RfqXGhAooRkUpzMLd6_Fpr13I&index=3 (дата обращения 27.08.2018).
4. [Видео] Обработка исключений в языке программирования C#: Видеокурс по языку программирования C#/ Eugene Popov: Youtube-канал. — URL: https://www.youtube.com/watch?v=A0lccpS6P8s&list=PLL-k0Ff5RfqXGhAooRkUpzMLd6_Fpr13I&index=19 (дата обращения 27.08.2018).
5. [Видео] Свойства в языке программирования C#: Видеокурс по языку программирования C#/ Eugene Popov: Youtube-канал. — URL: https://www.youtube.com/watch?v=mdfbFJo87bs&index=26&list=PLL-k0Ff5RfqXGhAooRkUpzMLd6_Fpr13I (дата обращения 27.08.2018).
6. [Видео] Статические классы и члены классов в языке программирования C#: Видеокурс по языку программирования C#/ Eugene Popov: Youtube-канал. — URL: https://www.youtube.com/watch?v=fd4k8GrH0Yc&list=PLL-k0Ff5RfqXGhAooRkUpzMLd6_Fpr13I&index=28 (дата обращения 27.08.2018).
7. [Видео] Полиморфизм и переопределение методов: Видеокурс по языку программирования C#/ Eugene Popov: Youtube-канал. — URL: https://www.youtube.com/watch?v=ZOWBZWXYPq0&index=32&list=PLL-k0Ff5RfqXGhAooRkUpzMLd6_Fpr13I (дата обращения 27.08.2018).
8. [Видео] Интерфейсы в языке программирования C#: Видеокурс по языку программирования C#/ Eugene Popov: Youtube-канал. — URL: https://www.youtube.com/watch?v=tVbOmPm78-E&index=39&list=PLL-k0Ff5RfqXGhAooRkUpzMLd6_Fpr13I (дата обращения 27.08.2018).

Оформление кода

9. Соглашение по оформлению кода команды RSDN [Электронный ресурс]. / RSDN (Russian Software Developers Network): сайт о программировании. — URL: <https://rsdn.org/article/mag/200401/codestyle.XML> (дата обращения 27.08.2018).
10. Комментарии к XML-документации (Руководство по программированию на C#) [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/xmldoc/xml-documentation-comments> (дата обращения 27.08.2018).
11. Практическое руководство. Использование XML-документации [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/xmldoc/how-to-use-the-xml-documentation-features> (дата обращения 27.08.2018).

Подключение сторонних библиотек

12. Управление ссылками в проекте [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/ru-ru/visualstudio/ide/managing-references-in-a-project> (дата обращения 27.08.2018).
13. Практическое руководство. Добавление и удаление ссылок с помощью диспетчера ссылок [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/ru-ru/visualstudio/ide/how-to-add-or-remove-references-by-using-the-reference-manager> (дата обращения 27.08.2018).
14. Менеджер подключения пакетов NuGet (NuGet Package Manager UI) [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/en-us/nuget/tools/package-manager-ui> (дата обращения 27.08.2018).
15. Сериализация и десериализация JSON библиотеки Newtonsoft JSON.NET (Serializing and Deserializing JSON). [Электронный ресурс]. / newtonsoft.com: официальный сайт компании Newtonsoft. — URL: <https://www.newtonsoft.com/json/help/html/SerializingJSON.htm> (дата обращения 27.08.2018).

UML-диаграммы классов

16. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд. / Г. Буч, Д. Рамбо, И. Якобсон; пер. с англ. Мухин Н. – М.: ДМК Пресс, 2006. – 496 с.: ил.
17. Фаулер М. UML. Основы, 3-е издание. – пер. с англ. Петухов А. – СПб: Символ-Плюс, 2004. – 192 с.: ил.

UML-диаграммы классов в Sparx Enterprise Architect

18. [Видео] Диаграммы классов в Enterprise Architect – пошаговое руководство (Class diagram in Enterprise Architect – Step by step guide) / Sparx Systems India: Youtubekanal. — URL: <https://www.youtube.com/watch?v=SFgSHpqJq1U> (дата обращения 27.08.2018).

Требования к оформлению отчета

19. ОС ТУСУР 01-2013. Работы студенческие по направления подготовки и специальностям технического профиля. Общие требования и правила оформления. – Томск: ТУСУР, 2013. – 57с. – URL: https://storage.tusur.ru/files/40668/rules_tech_01-2013.pdf (дата обращения 27.08.2018)

2.3 Разработка пользовательского интерфейса

Цель работы: изучить разработку адаптивного пользовательского интерфейса десктоп-приложения.

Задачи:

- 1) Изучить использование фреймворка пользовательского интерфейса Windows Forms и его компоненты.
- 2) Освоить компоненты и свойства для создания адаптивного дизайна.
- 3) Освоить обработку событий компонентов Windows Forms.
- 4) Реализовать передачу данных между разрабатываемыми формами и пользовательскими элементами управления.
- 5) Реализовать защиту от некорректного ввода.

2.3.1 Формы и элементы управления Windows Forms

Windows Forms – это набор библиотек для создания десктоп-приложений для операционной системы Windows. Windows Forms, или чаще WinForms, в настоящий момент является устаревшей технологией – ей на смену пришел новый набор библиотек – **Windows Presentation Foundation** (WPF). Как и Windows Forms, WPF предназначен в первую очередь для разработки десктоп-приложений для Windows, в то время как для мобильных и веб-приложений используются другие подходы и технологии. Таким образом, в настоящее время Windows Forms редко используется в новых проектах и больше связан с поддержкой старых приложений, где переход на WPF или другие технологии затруднителен.

Однако, несмотря на свои недостатки и неактуальность, Windows Forms хорошо подходит для изучения принципов разработки графических пользовательских интерфейсов. По этой причине, мы рекомендуем начать изучение разработки приложений с использованием Windows Forms, а после перейти к изучению современных пользовательских интерфейсов.

Два базовых понятия Windows Form – это элементы управления (Controls) и формы (Forms). Через элементы управления пользователь взаимодействует с программой: это кнопки, текстовые поля, фляжки, выпадающие списки, таблицы, графики и т.п. Элементы управления размещаются на формах – окнах. Формы размещают на себе элементы управления, а также хранят логику их взаимодействия между собой.

В более сложных, многооконных приложениях, формы могут создавать экземпляры друг друга и показывать их пользователю в зависимости от логики приложения. Более того, помимо стандартных элементов управления, часть которых перечислена выше, разработчик может создавать собственные элементы управления. Собственные элементы управления могут быть как составными – состоять из других элементов управления –, так и разработаны «с нуля», описывая собственную логику работы и отрисовки на экране.

При работе с формами и элементами управления нужно помнить, что они представляют собой точно такие же классы и объекты, как и любые другие типы данных в ООП. То есть, у классов форм и элементов также есть конструкторы, открытые и закрытые поля, методы. Формы могут агрегировать другие объекты, и все формы, которые вы создадите у себя в приложении, наследуются от базового класса Form. По существу, разработка пользовательского интерфейса заключается в размещении элементов управления на форме, а

затем написании методов в классе формы, в которых происходит передача и обработка данных.

Для создания пользовательского интерфейса в среде Visual Studio есть шаблон проекта Windows Forms Application. После создания такого проекта, в дереве проекта можно будет увидеть автоматически созданный класс Program и класс формы Form1, и также обязательные для любого проекта подпункты Properties (Свойства) и References (Ссылки). Также в главной части среды Visual Studio откроется так называемый дизайнер форм с пустой формой Form1:

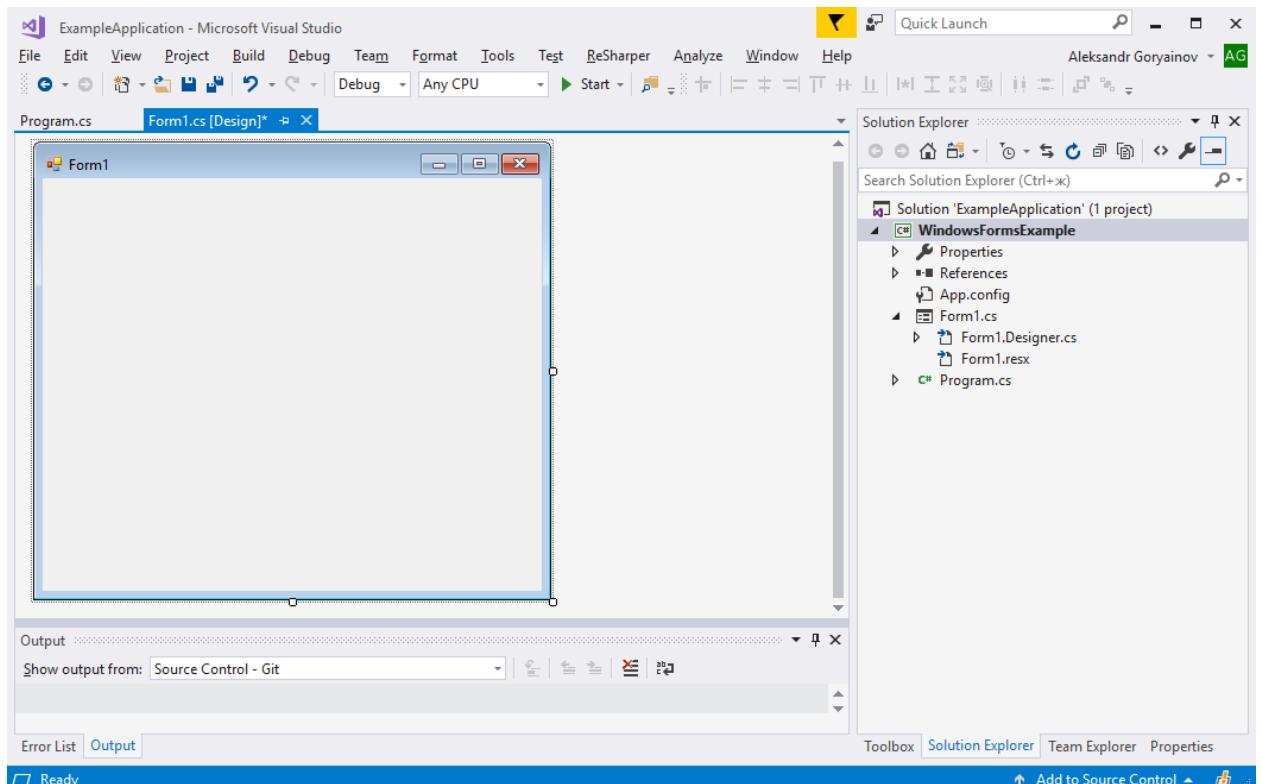


Рисунок 47 – Окно дизайнера формы после создания проекта Windows Forms

Изначально форма Form1 пустая, и здесь вы уже можете размещать элементы управления с помощью панели Toolbox (Панель инструментов). Если панель Toolbox у вас не открыта по умолчанию, её можно открыть через меню View в главном меню Visual Studio. О дизайнере форм и размещении элементов будет рассказано позже.

После компиляции и запуска проекта отобразится пустая форма:

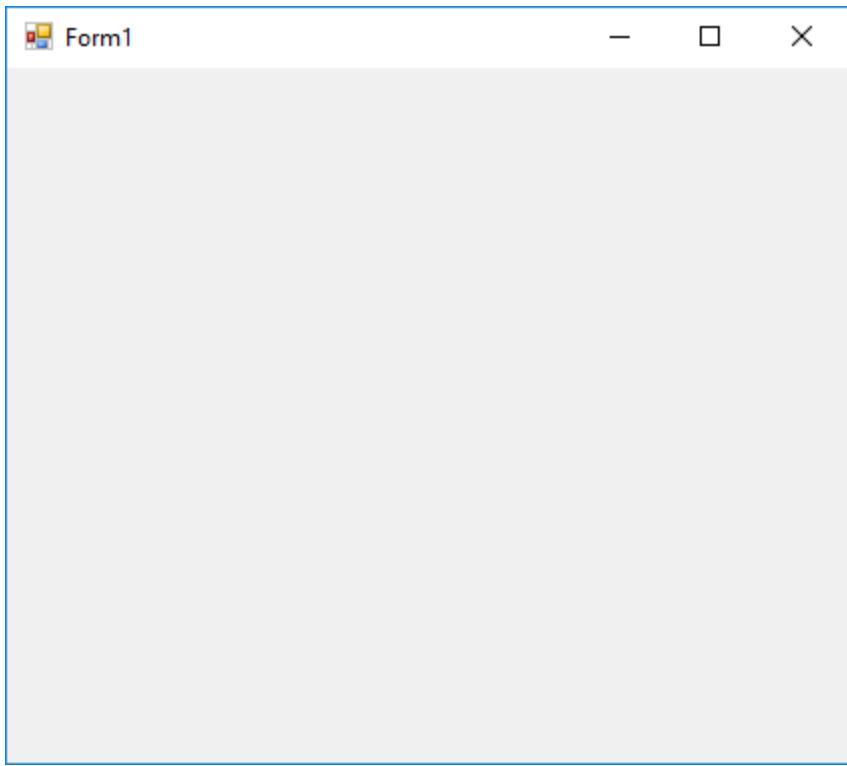


Рисунок 48 – Результат запуска созданного проекта Windows Forms

Главным исполняемым файлом проекта является файл Program.cs, содержащую метод Main():

```
using System;
using System.Windows.Forms;

namespace WindowsFormsExample
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Задача функции Main() – создать экземпляр формы и запустить её для отображения. Данный файл является стандартным, и что-либо менять в классе Program не требуется. Вместо этого стоит открыть файл Form1.cs:

```
using System.Windows.Forms;

namespace WindowsFormsExample
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
}
```

```
    }  
}
```

Для открытия файла необходимо нажать правой кнопкой мыши по названию файла и в контекстном меню выбрать пункт Source Code (Показать исходный код). Исходный код формы представляет класс со стандартным генерированным конструктором.

В исходном коде формы следует обратить внимание на следующие моменты:

- 1) модификатор partial в объявлении класса указывает на то, что класс Form1 разделен на два файла. И действительно, вторая часть класса находится в файле Form1.Designer.cs. Данное деление будет пояснено чуть позже. Сейчас же стоит отметить, что во втором файле находится метод InitializeComponent(), который вызывается в конструкторе формы.
- 2) класс Form1 наследуется от класса Form. Это наследование обязательно для всех форм Windows Forms. С помощью наследования, Form1 приобретает множество готовых полей и методов, необходимых для работы графического интерфейса.

Добавим код, который изменит заголовок формы, а также её размер. Код необходимо добавлять в конструктор и обязательно после вызова метода InitializeComponent():

```
namespace WindowsFormsExample  
{  
    public partial class Form1 : Form  
    {  
        public Form1()  
        {  
            InitializeComponent();  
            this.Text = "Главное окно программы";  
            this.Size = new Size(400, 250);  
        }  
    }  
}
```

Слово this это указатель на объект класса, в котором этот указатель вызван. Таким образом, через указатель this мы можем обратиться к экземпляру формы Form1. Через указатель this мы обращаемся к полям формы Text и Size. Text – это свойство типа string, хранящее текст заголовка окна. Свойство Size – это свойство, задающее размер окна. Для указания размера используется одноименная структура данных Size. Свойство Size есть как у форм, так и у элементов управления. Запустим приложение и убедимся, что текст и размер действительно изменены:

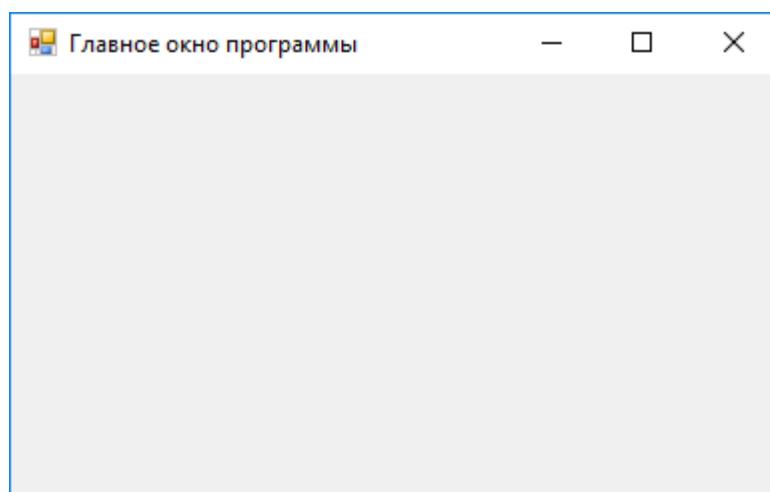


Рисунок 49 – Вид окна после изменения свойств Text и Size в конструкторе класса

Далее разместим на форме какой-нибудь элемент управления. Например, кнопку:

```
namespace WindowsFormsExample
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            this.Text = "Главное окно программы";
            this.Size = new Size(400, 250);

            //Создаем кнопку
            var button = new Button();
            button.Text = "Сменить заголовок окна";
            button.Size = new Size(150, 25);
            button.Location = new Point(150, 150);

            //Помещаем кнопку на форму
            this.Controls.Add(button);
        }
    }
}
```

Помимо свойств Text и Size, у элементов управления есть свойство Location, указывающее координату, где этот элемент должен располагаться на форме. Свойство Location имеет тип данных Point.

После создания экземпляра кнопки и инициализации его свойств, кнопку надо добавить на форму. Для этого кнопка добавляется в список элементов управления Controls, который есть в каждой форме. Без добавления элемента в этот список он не отобразится на экране. Запустим программу:

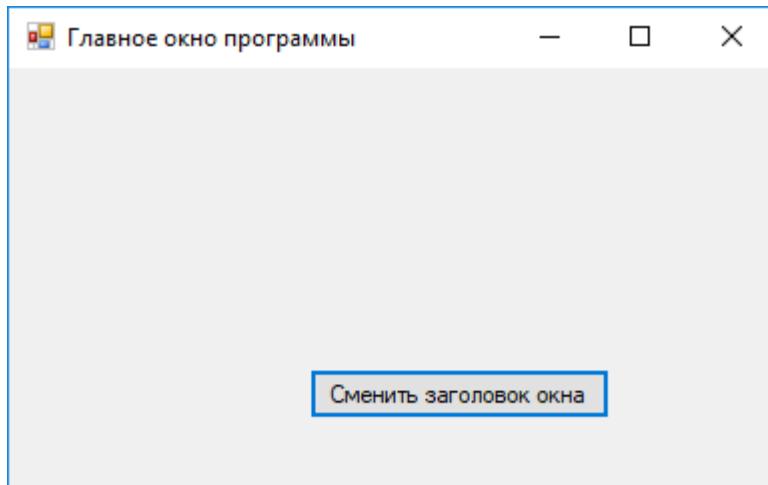


Рисунок 50 – Результат программного добавления кнопки на форму

Кнопка добавлена на форму, но пока она ничего не делает. Для того, чтобы кнопка реагировала на нажатие, необходимо сделать следующее:

- 1) Внутри формы написать метод, который будет выполняться при нажатии на кнопку.
- 2) Добавить в кнопку ссылку на этот метод, или, как это правильнее называется, подписать кнопку на обработчик события. Событие в данном случае будет нажатие мыши, а обработчиком события называется метод, который мы напишем.

Обработчики событий обладают определенным синтаксисом, отличающимся специфическими входными аргументами. Напишем обработчик события для кнопки, выполняющий смену названия окна:

```
namespace WindowsFormsExample
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            this.Text = "Главное окно программы";
            this.Size = new Size(400, 250);

            //Создаем кнопку
            var button = new Button();
            button.Text = "Сменить заголовок окна";
            button.Size = new Size(150, 25);
            button.Location = new Point(150, 150);

            //Подписываем кнопку на обработчик
            button.Click += Button_Click;

            //Помещаем кнопку на форму
            this.Controls.Add(button);
        }

        //Обработчик события нажатия кнопки
        private void Button_Click(object sender, EventArgs e)
        {
            //Здесь пишем код, который должен выполняться
            // каждый раз при нажатии на кнопку.
            this.Text = "Новый заголовок";
        }
    }
}
```

Обработчик события может быть назван любым именем, но для упрощения работы с классом принято называть обработчики с названием элемента управления, а также самого события, на который этот обработчик должен сработать. В противном случае, при разработке сложного окна, где количество обработчиков может легко превысить двадцать методов, будет сложно ориентироваться в коде.

Наш обработчик содержит только одну строку кода и выполняет смену заголовка формы. Для того, чтобы обработчик срабатывал при нажатии на кнопку, в конструкторе формы перед добавлением кнопки в список Controls, мы подписываем кнопку на этот обработчик следующей конструкцией:

```
button.Click += Button_Click;
```

Где в левой части идет обращение к так называемому событию Click кнопки button, а в правой части указывается имя метода – обработчика события. Обратите внимание, что имя метода пишется без круглых скобок. То есть в данном случае, мы не вызываем метод Button_Click для выполнения, а лишь помещаем указатель на этот метод в событие кнопки.

Запустив приложение и нажав на кнопку, заголовок окна действительно сменится:

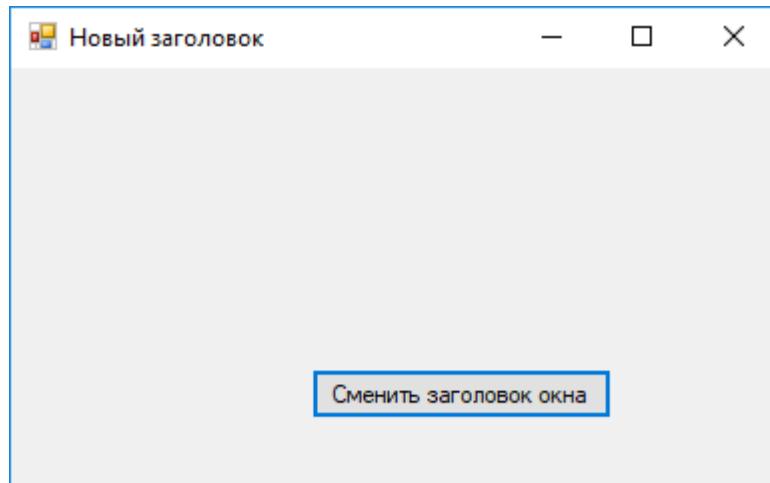


Рисунок 51 – После подписывания кнопки на обработчик события, при нажатии на кнопку происходит смена заголовка

Повторное нажатие на кнопку не приведет к каким-либо изменениям в форме, но это только на первый взгляд. На самом деле, при каждом нажатии кнопки будет выполняться метод Button_Click. В этом можно убедиться, поставив точку останова внутри метода. Но так как при повторном нажатии заголовок будет меняться на одну и ту же строку «Новый заголовок», для пользователя визуально ничего не изменится.

Внутри обработчика может быть написан любой код, не только изменяющий заголовок окна. Например, в обработчике можно создавать экземпляры классов логики приложения, вызывать их методы и выводить результат их работы пользователю.

Описанный способ добавления элементов управления на форму называется *программным*, потому что весь код инициализации и добавления элемента надо писать программно вручную. Однако Visual Studio содержит более удобный, визуальный способ добавления элементов управления на форму. Для этого снова откроем дизайнер форм, предварительно удалив код по добавлению кнопки:

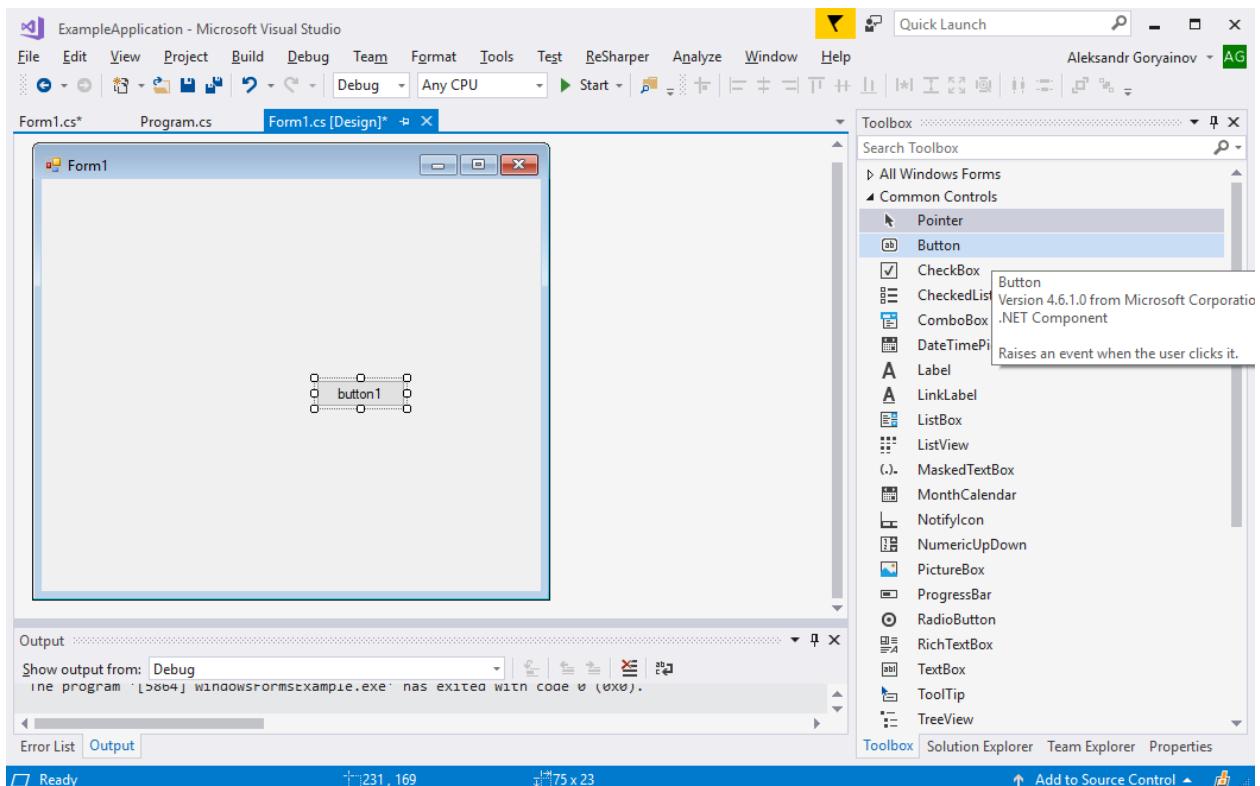


Рисунок 52 – Размещение элементов управления с помощью дизайнера форм

Дизайнер форм позволяет добавлять элементы на форму с помощью мыши без написания кода. Весь необходимый код будет сгенерирован автоматически. Откройте панель Toolbox и перетащите любой из стандартных элементов управления на форму, например, кнопку. Обратите внимание, как много готовых элементов управления реализовано в библиотеке Windows Forms.

После перетаскивания кнопки на форму она отобразится с текстом button1. Кнопку можно перетащить на нужную координату окна, или изменить её размер. Однако, чтобы увидеть все свойства кнопки, доступные для редактирования, следует открыть панель Properties (Свойства):

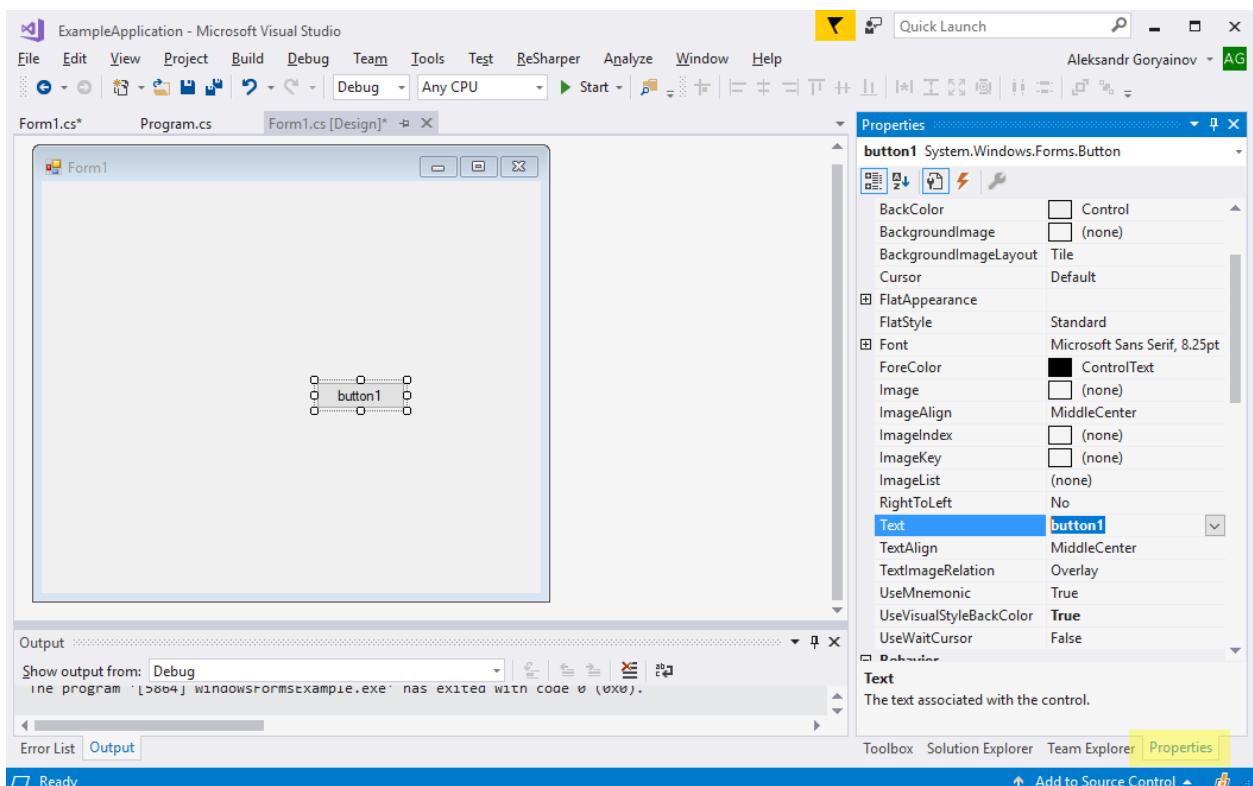


Рисунок 53 – Управление свойствами выбранного элемента управления

Панель Properties позволяет видеть практически все свойства выбранного элемента управления или формы. Например, можно задать другой текст кнопки, другой шрифт, цвет фона или фоновое изображение, выравнивание текста, видимость и т.д. Ознакомьтесь со списком свойств элемента управления. Стоит заметить, что большинство свойств для разных элементов управления одинаковы, однако есть свойства, уникальные для каждого элемента.

Таким образом, для создания пользовательского интерфейса, необходимо две дополнительных панели. Это панель Toolbox, с помощью которой можно размещать элементы на форме. И панель Properties, с помощью которой можно настраивать эти элементы.

Сделайте двойное нажатие мыши по кнопке button1. По двойному нажатию будет автоматически сгенерирован обработчик события нажатия, а среда разработки откроет исходный код формы:

```
namespace WindowsFormsExample
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {

        }
    }
}
```

Теперь внутри обработчика button1_Click остается только написать код, который должен срабатывать при нажатии на кнопку. Другими словами, дизайнер форм значительно упрощает создание пользовательского интерфейса, не требуя написания кода.

Возможно, вы обратили внимание, что в исходном коде файла Form1.cs нет кода по созданию кнопки, подписки на событие и добавление кнопки в список Controls формы. Дело в том, что дизайнер форм размещает весь этот код во втором файле класса Form1.Designer.cs. А точнее, в методе InitializeComponent(). Если вы откроете исходный код файла Form1.Designer.cs, то убедитесь в этом:

```
partial class Form1
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.button1 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(231, 169);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(75, 23);
        this.button1.TabIndex = 0;
        this.button1.Text = "button1";
        this.button1.UseVisualStyleBackColor = true;
        this.button1.Click += new System.EventHandler(this.button1_Click);
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(420, 347);
        this.Controls.Add(this.button1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
    }
}
```

```
}

#endregion

private System.Windows.Forms.Button button1;
}
```

Сделано это для упрощения навигации по коду форм. Так как инициализация элементов управления занимает много строк кода, читать такой код становится неудобным. По этому, весь код с инициализацией элементов (верстка) размещается в файле дизайнера, а обработчики событий (логика формы) размещается в первом файле класса.

Важно: файлы дизайнера содержат автогенерированный код, и потому его ручное редактирование нежелательно. В случае возникновения ошибок в этом файле, попытка открыть дизайнер формы приведет к перегенерации пустого файла дизайнера и потере **всех элементов управления**, размещенных на форме. Это веская причина для того, чтобы не править файл вручную и как можно чаще делать коммиты в системе версионного контроля.

Обратите внимание, что дизайнер форм не только помещает элементы управления в список Controls, но делает каждый элемент полем класса формы:

```
private System.Windows.Forms.Button button1;
```

Это значит следующее:

- 1) Любая форма композирует элементы, размещенные на ней с помощью дизайнера форм – то есть элементы являются обязательной частью формы, они создаются одновременно с формой и одновременно с формой уничтожаются.
- 2) К любому элементу формы можно обратиться внутри класса как к обычному полю. Например, кнопка button1 может быть вызвана из любого обработчика событий или другого метода класса Form1.

Также для каждой формы генерируется третий файл, например, Form1.resx. Данные файлы хранят так называемые ресурсы для каждой формы. Ресурсами формы могут быть ссылки на используемые изображения, или текстовые строки для всплывающих подсказок или локализации. Файл ресурсов формы также не следует редактировать вручную, и работать с данным файлом в рамках лабораторных работ вам не придется.

Если вы вернетесь в дизайнер формы и откроете панель Properties, то можете заметить на ней значок желтой молнии. При нажатии на этот значок, панель Properties будет показывать не свойства выбранного элемента, а его события. Событие Click далеко не единственное событие кнопки. В частности, можно создать обработчики событий, которые будут срабатывать при наведении курсора мыши на кнопку, при зажатии кнопки мыши, при изменении её размера, при изменении текста в кнопке и так далее. Учтите, что речь идет об изменении текста в кнопке или изменении её размера **в ходе выполнения приложения**, а не при работе в дизайнере форм. Чтобы вернуться к просмотру свойств выбранного элемента, нажмите на значок гаечного ключа на фоне документа.

Более подробно с созданием форм, дизайнером форм, а также описанием стандартных элементов управления можно ознакомиться в [1]. В частности, необходимо ознакомиться с наиболее часто используемыми элементами управления: кнопкой, текстовым полем TextBox, выпадающим списком ComboBox, флажком CheckBox, переключателем RadioButton, списком ListBox и таблицей DataGridView.

2.3.2 Получение данных со стандартных элементов управления

Разместим на нашей форме следующие элементы управления: TextBox, ComboBox, CheckBox. Через панель Properties дадим им имена NumberTextBox, ColorComboBox, VisibilityCheckBox соответственно. Также сразу же переименуем наш класс в MainForm. Именование элементов управления и форм также попадает под требования RSDN [6]. Элементы управления на форме должны содержать в своём имени название того элемента, экземпляром которого они являются, а также приставку, отражающую их назначение на данной конкретной форме. Например, название NumberTextBox выбрано потому, что само текстовое поле является экземпляром стандартного элемента TextBox, а приставка Number подразумевает, что мы будем вводить сюда число. Согласитесь, название NumberTextBox более содержательно, чем textbox1.

Аналогично формы должны содержать в своём названии слово Form и приставку, отражающую их назначение в проекте. Например, MainForm подразумевает, что это главное окно программы, а значит это же окно будет стартовым для пользователя. Переименовывать компоненты и формы стоит до того, как вы автоматически сгенерируете для них обработчики событий. В противном случае, названия обработчиков будут выглядеть как button1_Click(), button2_Click(), button3_Click() – то есть будут основаны на автоматически сгенерированном имени элемента управления. В таких названиях легко запутаться и в итоге тратится много времени, чтобы разобраться в логике формы.

После размещения элементов, форма должна выглядеть следующим образом:

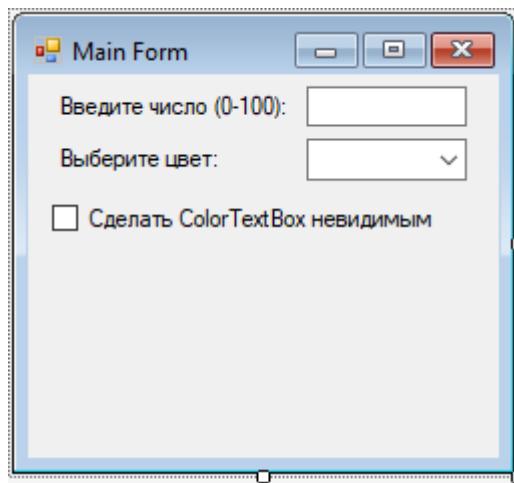


Рисунок 54 – Пример верстки главного окна

Обратите внимание, что помимо самих элементов управления, на форме следует размещать подписи (Label) для того, чтобы пользователь понимал назначение элементов. В подписях необходимо описывать что нужно ввести, а также ограничения на ввод. Чем понятнее будут подписи, тем меньше ошибок пользователь сделает при работе с вашей программой.

Текстовое поле TextBox. Двойной клик по NumberTextBox приведет к созданию обработчика события:

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }
}
```

```

private void NumberTextBox_TextChanged(object sender, EventArgs e)
{
}

```

В отличие от кнопки, обработчик NumberTextBox_TextChanged() будет срабатывать каждый раз, когда пользователь будет вводить текст в текстовое поле. Чтобы забрать введенный пользователем текст, необходимо обратиться к полю Text элемента NumberTextBox:

```

//Получаем текст из текстового поля
// в переменную типа string
string text = NumberTextBox.Text;

```

Что делать дальше с полученным текстом зависит от назначения программы. Мы же для примера присвоим полученный текст в название заголовка:

```

private void NumberTextBox_TextChanged(object sender, EventArgs e)
{
    string text = NumberTextBox.Text;
    this.Text = text;
}

```

Теперь, если вы запустите программу и начнете вводить текст в текстовое поле, то при каждом нажатии клавиши заголовок формы будет обновляться:

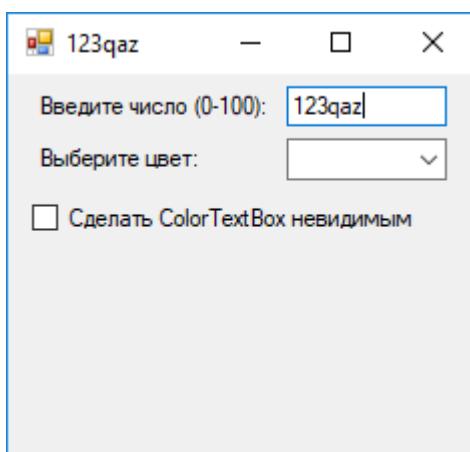


Рисунок 55 – Пример обработки ввода текста в окне. Введенный текст присваивается в заголовок окна

Зачастую необходимо получить от пользователя не текстовые данные, а данные другого типа, например, численные. Для того, чтобы преобразовать текст из текстового поля в число, можно использовать стандартный статический класс Convert или статические методы Parse стандартных типов данных. Например, для преобразования текста в целое число, можно использовать следующую конструкцию:

```

string text = NumberTextBox.Text;
int number = int.Parse(text);

```

Метод Parse попробует сконвертировать любой переданный в него текст в целое число. В случае, если текст можно преобразовать (например, текст «15»), то метод вернет это число. Если же текст нельзя преобразовать в число (например, «qaz» или «1t5»), то метод сгенерирует исключение. Исключение можно либо поймать с помощью блока try-catch. Однако на практике удобнее использовать метод TryParse():

```

string text = NumberTextBox.Text;
int number;
int.TryParse(text, out number);

```

Отличие метода TryParse() от Parse() заключается в том, что вместо генерации исключения он возвращает логическое значение true или false в зависимости от того, удалось ли преобразовать текст в число. Если текст удалось преобразовать, то значение поместится в переданную переменную number. При этом переменную number следует передавать с помощью модификатора доступа out. Подробнее о модификаторе доступа out можно прочитать в [7].

Исправим наш пример таким образом, чтобы заголовок формы менялся только в том случае, если текст, введенный в поле является целым числом:

```
private void NumberTextBox_TextChanged(object sender, EventArgs e)
{
    string text = NumberTextBox.Text;
    int number;
    if (int.TryParse(text, out number))
    {
        this.Text = number.ToString();
    }
}
```

В данном случае if получает результат вызова метода TryParse(), и если конвертирование прошло успешно, заголовок будет изменен. Запустите программу и убедитесь, что заголовок действительно игнорирует нечисловые значения в текстовом поле.

Более того, мы можем добавить проверку полученного числа на вхождение в диапазон, добавив еще одно условие:

```
private void NumberTextBox_TextChanged(object sender, EventArgs e)
{
    string text = NumberTextBox.Text;
    int number;
    if (int.TryParse(text, out number))
    {
        if (number >= 0 && number <= 100)
        {
            this.Text = number.ToString();
        }
        else
        {
            this.Text = "Число не входит в диапазон";
        }
    }
    else
    {
        this.Text = "Не число";
    }
}
```

Запустите программу и проверьте правильность её работы.

При необходимости вы можете не только получать значение из текстового поля, но и присваивать собственный текст в текстовое поле программно:

```
NumberTextBox.Text = "15";
```

Так вы можете, например, задать значение текстового поля по умолчанию или вывести пользователю какую-либо информацию.

Выпадающий список ComboBox. Работа с выпадающим списком ComboBox несколько отличается от работы с текстовым полем. Разница заключается в том, что ComboBox хранит в себе список всех допустимых значений в собственном поле Items, и одно из значений является выбранным (SelectedItem). ComboBox, в частности, очень подходит, когда пользователю нужно выбрать значение из перечисления.

Для работы ComboBox необходимо: 1) в конструкторе формы добавить допустимые значения в поле Items; 2) написать обработчик события, который будет срабатывать при изменении выбранного значения. Рассмотрим работу с выпадающим списком на примере ColorComboBox и перечисления Color.

Color является стандартным перечислением .NET и описывает порядка нескольких со-тей различных цветов:

```
public enum Color
{
    Red,
    Green,
    Blue,
    White,
    Black,
    Yellow,
    ...
}
```

Для того, чтобы в выпадающем списке были отображены значения, их необходимо добавить. Добавление должно выполняться в конструкторе формы:

```
public MainForm()
{
    InitializeComponent();
    ColorComboBox.Items.Add(Color.Red);
    ColorComboBox.Items.Add(Color.Green);
    ColorComboBox.Items.Add(Color.Blue);
    ColorComboBox.Items.Add(Color.White);
}
```

После двойного клика по ColorComboBox в дизайнере форм автоматически сгенери-руется обработчик события ColorComboBox_SelectedIndexChanged(). В него добавим код, который будет менять фоновый цвет главного окна на выбранный в выпадающем списке:

```
private void ColorComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    Color selectedColor;
    selectedColor = (Color)ColorComboBox.SelectedItem;
    this.BackColor = selectedColor;
}
```

Для этого мы сначала создадим переменную selectedColor того типа данных, который мы хотим получить из выпадающего списка – Color. Во второй строке мы забираем значе-ния цвета из выпадающего списка через поле SelectedItem. Однако SelectedItem хранит все значения в обобщенном типе данных object, и чтобы сохранить его в переменной, необхо-димо явно преобразовать значение в требуемый тип данных. Для этого во второй строке перед ColorComboBox.SelectedItem указаны скобки с названием типа данных для преобра-зования - (Color).

Третья строка выполняет лишь подстановку считанного из выпадающего списка цвета в поле BackColor формы, что заменит её фоновый цвет. Запустите форму и убедитесь, что она работает верно:

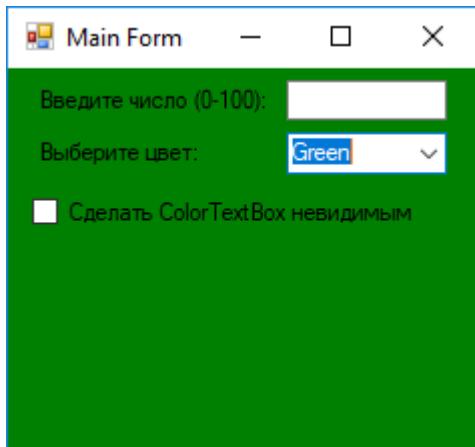


Рисунок 56 – Пример обработки значений из выпадающего списка

Фоновый цвет, также как и цвет шрифта, можно изменить не только у формы, но и у любого элемента управления на форме. Трюк с изменением фонового цвета можно использовать для того, чтобы сообщать пользователю о неправильно введенном значение. Например. Исправим пример с NumberTextBox на следующий обработчик:

```
private void NumberTextBox_TextChanged(object sender, EventArgs e)
{
    string text = NumberTextBox.Text;
    int number;
    if (int.TryParse(text, out number))
    {
        if (number >= 0 && number <= 100)
        {
            NumberTextBox.BackColor = Color.White;
            this.Text = number.ToString();
        }
        else
        {
            NumberTextBox.BackColor = Color.LightSalmon;
        }
    }
    else
    {
        NumberTextBox.BackColor = Color.LightSalmon;
    }
}
```

Теперь, в случае ввода неправильного значения в текстовое поле, поле будет подсвеченено светло-красным цветом:

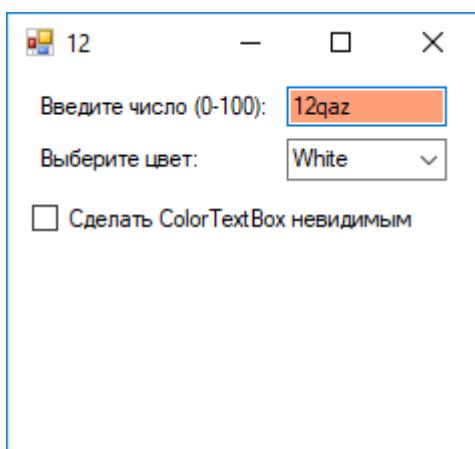


Рисунок 57 – Подсветка элемента управления сообщает пользователю о неправильно введенном значении

Другим способом работы с ComboBox является поле SelectedIndex. В отличие от SelectedItem, данное поле возвращает числовой индекс выбранного элемента. Индекс не должен превышать количество значений в выпадающем списке Items.Count. Также у поля SelectedIndex есть уникальное значение -1, означающее, что ни одно значение из выпадающего списка не выбрано.

Присвоив в конструкторе полю SelectedIndex некоторое значение, вы можете указать значение из выпадающего списка, которое будет выбрано по умолчанию:

```
public MainForm()
{
    InitializeComponent();
    ColorComboBox.Items.Add(Color.Red);
    ColorComboBox.Items.Add(Color.Green);
    ColorComboBox.Items.Add(Color.Blue);
    ColorComboBox.Items.Add(Color.White);
    //Задать зеленый цвет по умолчанию
    ColorComboBox.SelectedIndex = 1;
}
```

Любое изменение SelectedIndex приводит к выполнению обработчика события SelectedIndexChanged(), поэтому при запуске программы форма сразу же станет зеленой.

В нашем примере не учтена одна ситуация, которая приведет к аварийному завершению программы. Эта ситуация может возникнуть, если мы попытаемся программно присвоить в поле SelectedIndex значение -1.

Дело в том, что после присвоения индекса происходит выполнение обработчика ColorComboBox_SelectedIndexChanged(). Во второй строке этого обработчика происходит обращение к полю SelectedItem, однако оно имеет значение null, так как присвоение индекса -1 в SelectedIndex сбросит выбранное значение выпадающего списка. При попытке явного преобразования null в Color программа генерирует системное исключение, которое и завершит программу. Попробуйте запустить такое приложение и убедитесь, что оно действительно завершается с ошибкой.

Для того, чтобы приложение не завершалось в такой ситуации, необходимо либо: а) обвернуть соответствующую строку блоком try-catch; б) в начале обработчика добавить проверку на текущий индекс выпадающего списка. Реализация второго подхода более лаконична и приведена ниже:

```
private void ColorComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    if (ColorComboBox.SelectedIndex == -1)
    {
        // Если ничего не выбрано, завершаем обработчик
        return;
    }
    Color selectedColor;
    selectedColor = (Color)ColorComboBox.SelectedItem;
    this.BackColor = selectedColor;
}
```

Таким образом, мы исключаем ситуацию, в которой происходит обращение к null-объекту.

Разумеется, выпадающий список можно использовать для любых перечислений, в том числе созданных вами. А также и для иных значений, необязательно объединенных в перечисление.

Флажок CheckBox. Для работы с логическими переключателями используется стандартный элемент CheckBox. Для получения текущего состояния флашка используется поле Checked, принимающее значения true или false.

Например, реализуем логику управления видимостью ранее созданного выпадающего списка ColorComboBox с помощью VisibilityCheckBox. Для создания обработчика события сделаем двойной клик по флашку в дизайнере форм:

```
private void VisibilityCheckBox_CheckedChanged(object sender, EventArgs e)
{
}
```

Чтобы получить текущее состояние CheckBox, необходимо обратиться к его логическому полю Checked. Также через поле Checked можно установить значение флашка:

```
bool isChecked = VisibilityCheckBox.Checked; //Забрать текущее значение из флашка
VisibilityCheckBox.Checked = true; //Флашок на интерфейсе станет отмеченным
```

Полученное значение можно сохранить в переменной или в поле другого объекта. Для примера использования полученного значения, изменим видимость выпадающего списка:

```
private void VisibilityCheckBox_CheckedChanged(object sender, EventArgs e)
{
    bool isChecked = VisibilityCheckBox.Checked;
    ColorComboBox.Visible = isChecked;
}
```

Теперь при нажатии на флашок в интерфейсе, будет появляться или исчезать выпадающий список. Запустите программу и убедитесь, что логика управления видимостью работает.

Отметим, поле Visible есть у всех элементов управления, и можно управлять видимостью любого элемента. Также у всех элементов управления есть поле Enabled, которое позволяет включить или отключить для пользователя возможность взаимодействия с этим элементом. В случае отключения (Enabled принимает значение false) элемент будет отображаться серым цветом.

Работа с другими элементами управления происходит аналогичным с представленными элементами способом. Ознакомьтесь с их работой самостоятельно [\[1\]](#).

2.3.3 Стандартные формы MessageBox, SafeFileDialog, OpenFileDialog

Помимо стандартных элементов управления в Windows Forms есть также ряд стандартных окон. Разберем несколько часто используемых из них.

Форма MessageBox предназначена для показа пользователю небольших текстовых сообщений, например, предупреждений, ошибок или подсказок. Данная форма имеет статический метод Show с множеством различных перегрузок. В качестве аргументов метода необходимо передать строку с текстом сообщения, заголовком, и дополнительно пиктограммой и набором кнопок, которые нужно показать пользователю:

```
MessageBox.Show("Сообщение для пользователя");
MessageBox.Show("Сообщение", "Заголовок");
MessageBox.Show("Сообщение", "Заголовок",
    MessageBoxButtons.OKCancel);
MessageBox.Show("Сообщение", "Заголовок",
    MessageBoxButtons.OKCancel,
    MessageBoxIcon.Information);
```

Выше показаны четыре различных варианта вызова метода Show для MessageBox. Результатом их выполнения будут следующие окна:

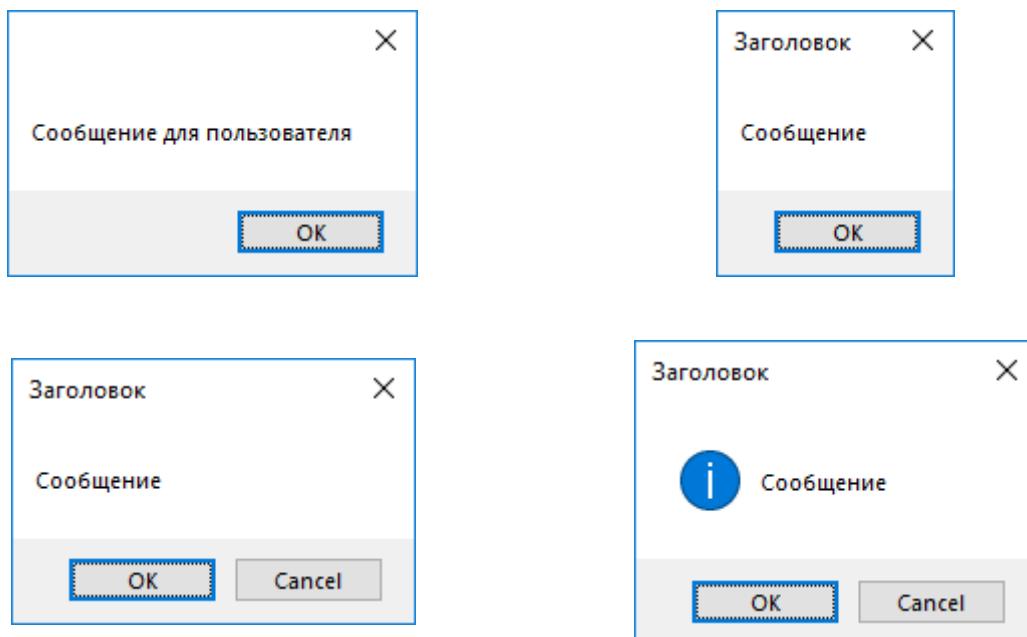


Рисунок 58 – Варианты MessageBox

С помощью перечисления MessageBoxButtons можно задать перечень кнопок, которые необходимо отобразить в окне. MessageBoxButtons принимает следующие значения:

- AbortRetryIgnore
- OK
- OKCancel
- RetryCancel
- YesNo
- YesNoCancel

С помощью перечисления MessageBoxIcon можно задать пиктограмму, которую необходимо отобразить в окне. MessageBoxIcon принимает следующие значения:

- Error (ошибка)
- Information (информация)
- None (без пиктограммы)
- Question (вопрос)
- Warning (предупреждение)

Так как окно сообщения может закрыться нажатием одной из нескольких кнопок (например, OK или Cancel), результат дальнейшей работы программы будет зависеть от того, какая именно кнопка была нажата. Для того, чтобы узнать, какая кнопка была нажата пользователем, используется специальный тип данных DialogResult. DialogResult, это перечисление, которое возвращает метод Show(), и принимает следующие значения:

- Abort
- Cancel
- Ignore
- No

- None
- OK
- Retry
- Yes

Следующий код показывает пользователю сообщение с запросом на сохранение файла и сохраняет только в том случае, если пользователь нажал кнопку OK:

```
string filename = "C:\file.txt";
DialogResult result = MessageBox.Show("Сохранить файл?\n" + filename,
                                      "Сохранение файла",
                                      MessageBoxButtons.OKCancel,
                                      MessageBoxIcon.Question);

if (result == DialogResult.OK)
{
    SaveFile(filename);
}
```

где метод SaveFile() выполняет сохранение в файл некоторых данных – его реализация не важна для примера. Аналогичным образом можно реализовать поведение программы на нажатие других кнопок: Cancel, Yes/No, Retry/Abort. Механизм обработки DialogResult является стандартным и для других форм.

Формы SaveFileDialog и OpenFileDialog реализуют окна, предназначенные для выбора файла в файловой системе компьютера:

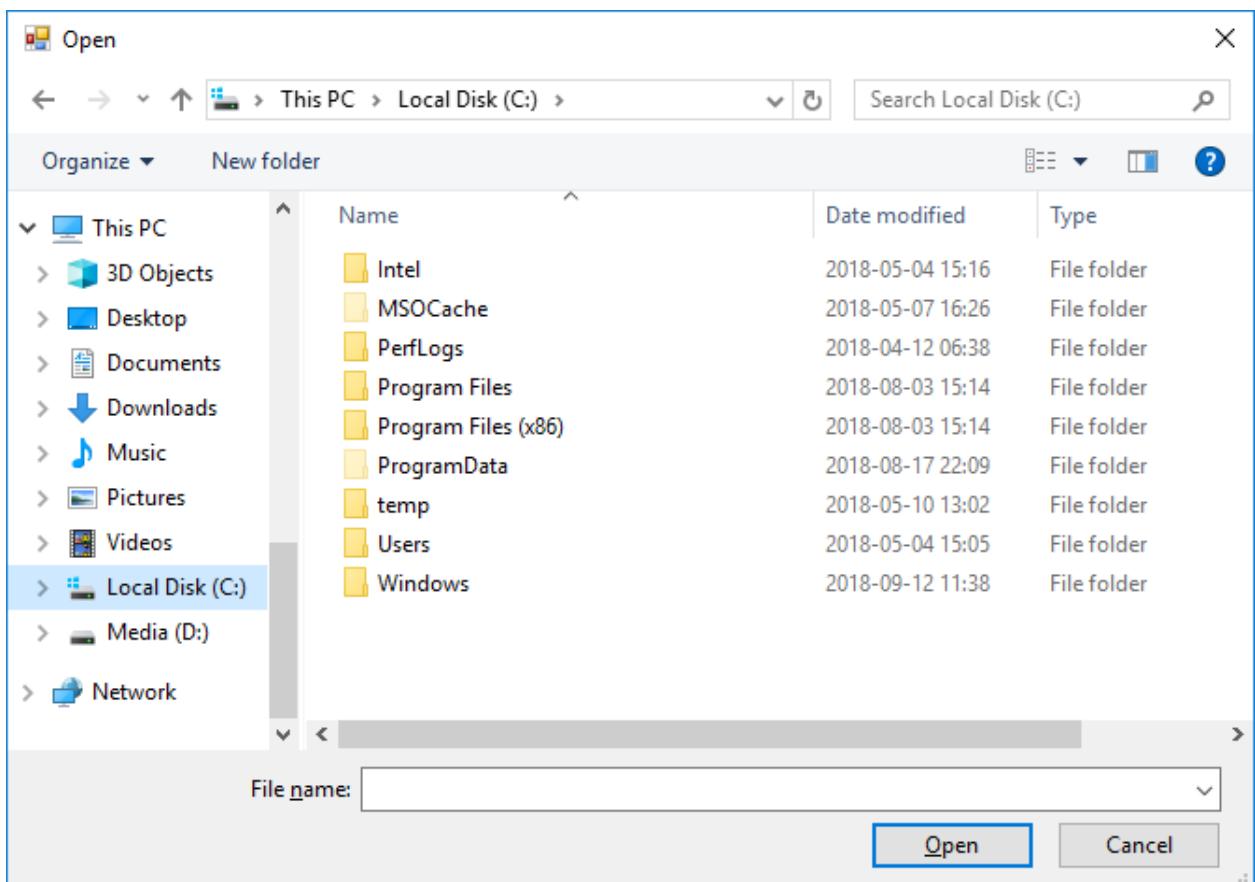


Рисунок 59 – Пример вызова стандартного окна OpenFileDialog

В отличие от MessageBox – статического класса -, для работы с OpenFileDialog необходимо создавать экземпляры класса:

```
// Создаем экземпляр формы
 OpenFileDialog dialog = new OpenFileDialog();
```

```

//Показываем её пользователю в диалоговом режиме
dialog.ShowDialog();

//Забираем имя файла из формы после её закрытия
string filename = dialog.FileName;

//Вызываем собственную функцию для чтения файла
OpenFile(filename);

```

Форма OpenFileDialog показывается только в диалоговом режиме. Диалоговой режим, как говорилось ранее, это такой режим показа окна, при котором нельзя переключиться к главному окну, пока не будет закрыто диалоговое окно.

Формы SaveFileDialog и OpenFileDialog не занимаются сохранением или загрузкой файлов. Они предназначены только для выбора имён файлов. Реализация сохранения и загрузки в выбранные файлы выполняется самим разработчиком в зависимости от задачи. Например, с помощью библиотеки сериализации Newtonsoft JSON.NET, рассмотренной в предыдущей лабораторной работе.

OpenFileDialog и SaveFileDialog имеют ряд дополнительных настроек. Например, через свойства формы можно указать фильтр имен файлов, возможность одновременного выбора нескольких файлов, выполнение проверки на существование файла и т.д.:

```

dialog.Filter = "Только текстовые файлы (*.txt) | *.txt";
dialog.Multiselect = true;
dialog.CheckFileExists = true;

```

Список всех настроек можно посмотреть с помощью автодополнения в VisualStudio или на сайте forum.msdn.com. Аналогично работе с MessageBox, следует проверять результат завершения окна выбора файла:

```

 OpenFileDialog dialog = new OpenFileDialog();
dialog.Filter = "Только текстовые файлы (*.txt) | *.txt";
dialog.CheckFileExists = true;

DialogResult result = dialog.ShowDialog();
if (result == DialogResult.OK)
{
    string filename = dialog.FileName;
    if (filename == string.Empty)
    {
        OpenFile(filename);
    }
}

```

В противном случае, вы можете загрузить файл, хотя пользователь уже передумал и нажал кнопку отмены. Также обратите внимание на проверку непустой строки перед загрузкой файла.

2.3.4 Использование проекта бизнес-логики в пользовательском интерфейсе

В случае разработки сложных приложений, исчисляющихся сотнями тысяч строк кода, существуют специальные архитектурные паттерны для связи пользовательского интерфейса с бизнес-логикой. Эти паттерны описывают взаимодействие интерфейса и логики на уровне пакетов и общих рекомендаций, и во многом зависят от платформы, для которой разрабатывается приложение. MVC и MVVM – примеры таких паттернов.

В небольших приложениях взаимодействие интерфейса и логики можно организовать проще. В первую очередь, для удобства навигации и уменьшения связности в проекте необходимо разделить программу на две части – логику и интерфейс. В самом простом случае, логика и интерфейс представляются двумя отдельными сборками, что позволяет использовать логику без интерфейса – через API или для автоматизации тестирования.

Второй шаг – это использование или агрегирование классов логики внутри классов интерфейса.

Например, предположим, что мы разрабатываем приложение с базой контактов пользователей. Тогда в проекте логики у нас наверняка будет класс или структура данных, описывающая один контакт:

```
public class Contact
{
    public string Name { get; set; }

    public uint Phone { get; set; }

    public string Email { get; set; }
}
```

Для возможности использования данного класса логики в интерфейсе, необходимо подключить ссылку на проект логики в проект интерфейса, а также убедиться, что класс Contact объявлен с модификатором public (public class Contact).

Предположим, что где-то в интерфейсе нам необходимо вывести список контактов на экран в виде списка и дать возможность добавлять и удалять контакты:

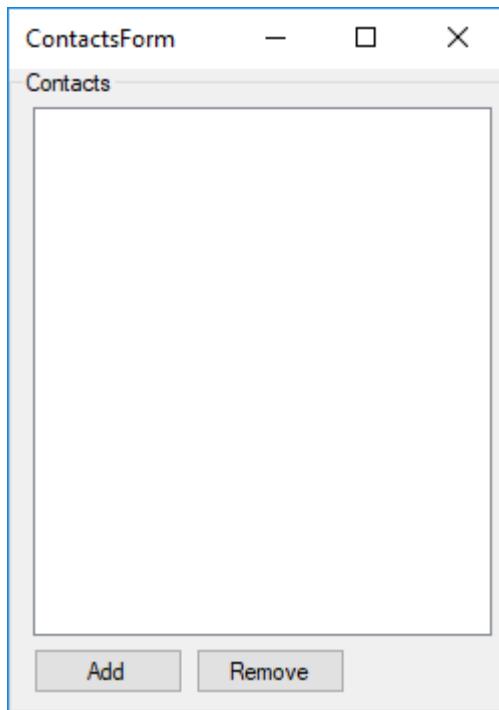


Рисунок 60 – Окно вывода списка контактов

Так как любая форма – это обычный класс, в котором мы можем создавать собственные поля и методы, создадим поле контактов внутри ContactsForm:

```
public partial class ContactsForm : Form
{
    private List<Contact> _contacts = new List<Contact>();

    public ContactsForm()
```

```
{  
    InitializeComponent();  
}  
}
```

Теперь мы можем работать со списком контактов внутри нашей формы – обращаться к ней из любого обработчика событий, добавлять и удалять контакты, редактировать их. Например, реализуем добавление контактов. Для этого двойным кликом по кнопке Add создадим обработчик события кнопки:

```
public partial class ContactsForm : Form  
{  
    private List<Contact> _contacts = new List<Contact>();  
  
    public ContactsForm()  
    {  
        InitializeComponent();  
    }  
  
    private void addButton_Click(object sender, EventArgs e)  
    {  
    }  
}
```

В этом обработчике необходимо написать код, который:

- 1) создаёт экземпляр контактов;
- 2) добавляет новый контакт в список контактов внутри формы;
- 3) добавляет новый контакт в элемент ListBox для отображения контакта пользователю.

Реализация может выглядеть следующим образом:

```
private void addButton_Click(object sender, EventArgs e)  
{  
    Contact newContact = new Contact();  
    newContact.Name = "Смирнов";  
    newContact.Phone = 55512345678;  
    newContact.Email = "u.smirnov@fake.mail";  
  
    _contacts.Add(newContact);  
    ContactsListBox.Items.Add(newContact.Name);  
}
```

В примере выше экземпляр контакта создается программно, и при каждом нажатии всегда будут создаваться новые контакты с одними и теми же данными. В реальной программе вам необходимо запросить имя, номер телефона и почту у пользователя, а затем введенные данные поместить в поля контакта.

Обратите внимание, что добавлять нужно *и в список* _contacts, *и в элемент* ListBox. Противном случае, ваш интерфейс не будет отражать реальных данных, хранящихся в вашей программе.

В обработчике для удаления контакта, соответственно, необходимо написать код, который:

- 1) определяет индекс текущего выбранного элемента в ListBox;
- 2) удаляет элемент по текущему индексу из ListBox;
- 3) удаляет контакт по текущему индексу из списка контактов _contacts.

В качестве полей формы вы можете создавать любые объекты вашей бизнес-логики. Если у вас есть класс проекта Project, вы можете создать его объект в качестве поля главной формы. Если у вас есть класс, отвечающий за сериализацию данных, вы можете вызывать

его методы в обработчиках событий. Например, по нажатию кнопки Open File вы можете сначала запросить у пользователя имя файла для открытия с помощью диалога OpenFileDialog, а затем открыть этот файл, вызвав методы сериализации из бизнес-логики приложения.

2.3.5 Передача данных между формами

Передача данных между формами осуществляется также, как и передача данных между двумя обычными классами. Единственная разница заключается в том, что при передаче данных между формами необходимо обновить отображаемые на экране данные, если в результате передачи они поменялись.

Сначала рассмотрим пример передачи данных между двумя обычными классами. В этом примере у нас есть три класса: 1) Data – класс с условными данными, которые надо передавать между двумя классами; 2) Outer – внешний класс, который будет передавать данные во внутренний класс Inner и забирать их после обработки; 3) Inner – внутренний класс, который будет получать данные из внешнего класса Outer, обрабатывать их и возвращать обратно. Далее представлен код этих классов:

```
//Внешний класс
public class Outer
{
    //Данные, которые будут передаваться
    private Data _data = new Data();

    public void DoSomething()
    {
        //При необходимости инициализируем данные
        _data.Text = "Some text";
        _data.LastUpdate = DateTime.Now;

        var inner = new Inner(); //Создаем объект внутреннего класса
        inner.Data = _data; //Передаем ему данные из внешнего класса
        inner.UpdateData(); //Говорим внутреннему классу обработать данные
        _data = inner.Data; //После обработки забираем данные из внутреннего класса
    }
}

//Внутренний класс
public class Inner
{
    //Поле для временного хранения переданных данных
    private Data _data;

    //Свойство, через которое будут передаваться данные извне
    public Data Data
    {
        get
        {
            return _data;
        }
        set
        {
            _data = value;
        }
    }

    //Обрабатывает переданные в класс данные
    public void UpdateData()
    {
```

```

        _data.Text = "Text is updated";
        _data.LastUpdate = DateTime.Now;
    }

//Класс может содержать любые данные,
// в зависимости от целей вашей программы
public class Data
{
    public string Text;

    public DateTime LastUpdate;
}

```

В качестве данных структура Data будет хранить строку Text и время последнего обновления строки LastUpdate. В вашей программе вместо класса Data может быть любой другой класс.

Для того, чтобы в класс Inner можно было передать данные, в нём должно быть свойство с открытым сеттером. Чтобы из класса Inner можно было забрать данные, в нём должно быть свойство с открытым геттером. В самом общем случае, входные и выходные данные могут быть представлены разными классами, например, Data1 и Data2.

Весь процесс обмена реализован в методе DoSomething() класса Outer. Класс Outer обращается к экземпляру класса Inner (который может создаваться внутри метода или быть агрегирован классом Outer), и через свойство Data передает собственный объект данных _data со строкой “Some text”. Теперь данные хранятся и в Outer, и в Inner.

Предположим, что внутри класса Inner эти данные поменялись. Например, был выполнен метод UpdateData(). Чтобы забрать данные из Inner, класс Outer обращается к геттеру _inner.Data и получает обновленный экземпляр данных со строкой “Text is updated” и новым значением времени в LastUpdate.

Теперь предположим, что вместо классов Outer и Inner нам необходимо передать данные между формами OuterForm и InnerForm. При этом, в внешней форме теперь будет хранится список данных Data, отображающихся в списке ListBox. А во внутренней форме InnerForm переданный экземпляр выводится на экран, где пользователь самостоятельно может ввести произвольный текст с помощью текстового поля:

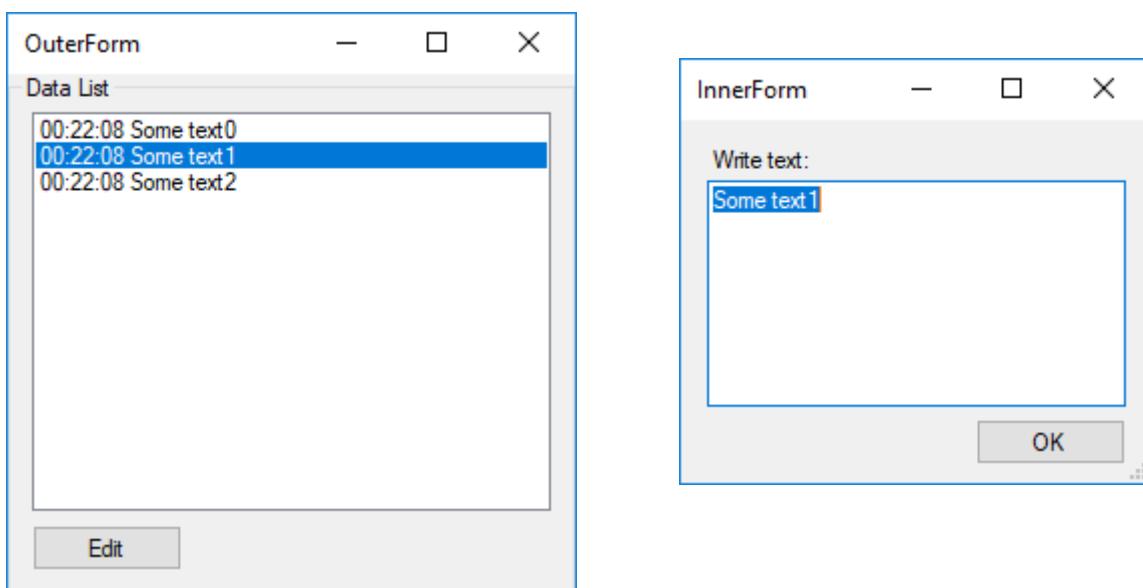


Рисунок 61 – Результат передачи данных из внешней формы через свойство формы

Код OuterForm:

```
public partial class OuterForm : Form
{
    private List<Data> _data = new List<Data>();

    public OuterForm()
    {
        InitializeComponent();
        FillListBoxByTestData();
    }

    private void EditButton_Click(object sender, EventArgs e)
    {
        //Получаем текущую выбранную дату
        var selectedIndex = DataListBox.SelectedIndex;
        var selectedData = _data[selectedIndex];

        var inner = new InnerForm(); //Создаем форму
        inner.Data = selectedData; //Передаем форме данные
        inner.ShowDialog(); //Отображаем форму для редактирования
        var updatedData = inner.Data; //Забираем измененные данные

        //Осталось удалить старые данные по выбранному индексу
        // и заменить их на обновленные
        DataListBox.Items.RemoveAt(selectedIndex);
        _data.RemoveAt(selectedIndex);

        _data.Insert(selectedIndex, updatedData);
        var time = updatedData.LastUpdate.ToString("yyyy-MM-dd HH:mm:ss");
        var text = updatedData.Text;
        DataListBox.Items.Insert(selectedIndex, time + " " + text);
    }

    //Генерирует начальные тестовые данные для удобства тестирования
    // (не обязательен для примера)
    private void FillListBoxByTestData(int dataCount = 3)
    {
        for (int i = 0; i < 3; i++)
        {
            var data = new Data()
            {
                Text = "Some text" + i,
                LastUpdate = DateTime.Now
            };
            _data.Add(data);
            var time = data.LastUpdate.ToString("yyyy-MM-dd HH:mm:ss");
            var text = data.Text;
            DataListBox.Items.Add(time + " " + text);
        }
    }
}
```

Если сравнить код OuterForm с классом Outer из предыдущего примера, то разница заключается лишь в обновлении интерфейса (пересоздаем элемент списка ListBox). Не считая метода FillListBoxByTestData(), написанного только для генерации тестовых данных, в код формы добавилось лишь 8 строчек кода. Исходный код InnerForm:

```
public partial class InnerForm : Form
{
    private Data _data;

    public Data Data
    {
        get
    }
```

```

    {
        return _data;
    }
    set
    {
        _data = value;
        if (_data != null)
        {
            DataTextBox.Text = _data.Text;
        }
    }
}

public InnerForm()
{
    InitializeComponent();
}

private void DataTextBox_TextChanged(object sender, EventArgs e)
{
    _data.Text = DataTextBox.Text;
    _data.LastUpdate = DateTime.Now;
}

private void OkButton_Click(object sender, EventArgs e)
{
    DialogResult = DialogResult.OK;
    this.Close();
}
}

```

Опять же, реализация InnerForm мало отличается от класса Inner. В данном случае, изменение данных происходит не при вызове метода UpdateData(), а при действиях пользователя. Пользователь вводит новые данные в текстовое поле, после чего запускается обработчик события DataTextBox_TextChanged(), который и обновляет поля объекта _data. По нажатию на кнопку OK, внутренняя форма закрывается и управление возвращается в обработчик события EditButton_Click() формы OuterForm. Внешняя форма тут же забирает обновленные данные updatedData из уже закрытой внутренней формы, после чего обновляет список данных Data и пользовательский интерфейс. В данном случае, для обновления интерфейса, нам необходимо удалить строку в ListBox, которая была связана со старым объектом, и вставить на её место строку, сформированную из новых данных.

В результате работы получим:

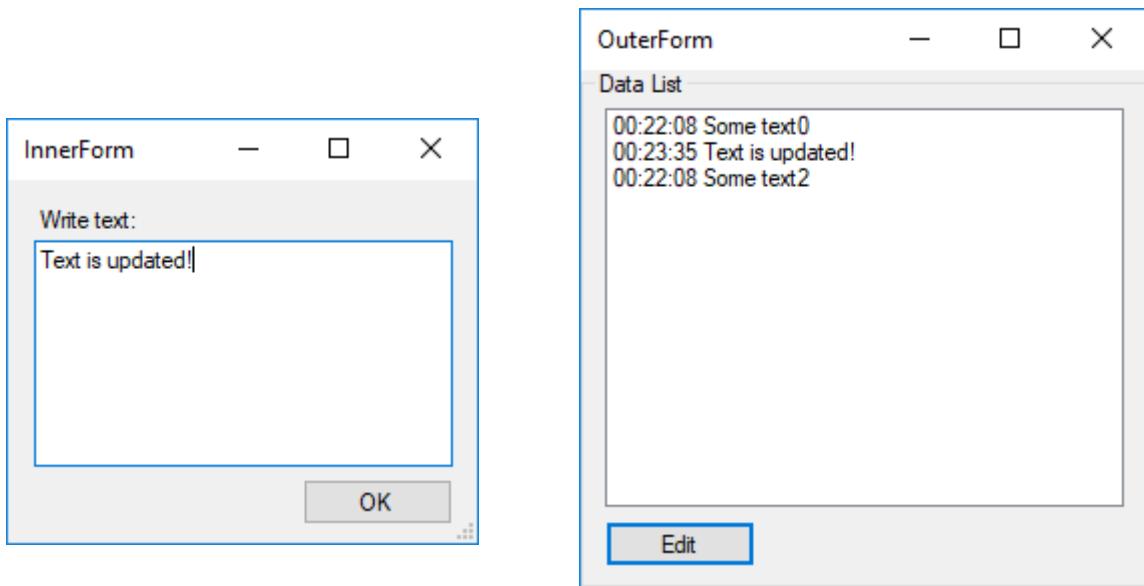


Рисунок 62 – Пример передачи данных из внутренней формы во внешнюю. Данные внешней формы обновились

То есть форма обновилась, и выделенная вторая строка теперь показывает новый введенный текст.

Для закрепления примера повторите реализацию этих форм. Для практики добавьте кнопку Cancel на InnerForm, отменяющую изменения текущего объекта. Подумайте самостоятельно, как её реализовать – задача не такая простая, как может показаться на первый взгляд.

На практике студенты часто используют другие способы передачи данных между формами. Сразу отметим, что эти способы неправильные:

- 1) Поместить текстовое поле DataTextBox внутренней формы под модификатор public, чтобы к нему можно было обратиться из внешней формы. Данный подход нарушает инкапсуляцию, что усложнит возможную модификацию внутренней формы.
- 2) Передать в качестве свойства внутренней формы не экземпляр Data, а экземпляр OuterForm, где список данных _data или ListBox помещены под модификатор доступа public. Этот способ еще более глупый, так как помимо нарушения инкапсуляции OuterForm, делает невозможным использование InnerForm без экземпляра OuterForm. Поддержка программы становится еще сложнее.
- 3) Передать экземпляр OuterForm в конструктор InnerForm. Аналогичен предыдущему способу.
- 4) Передать в InnerForm список всех данных _data вместо передачи только одного выбранного объекта. Данный подход опять же нарушает инкапсуляцию. Более того, подход заставляет передавать во внутреннюю форму больше данных, чем ей нужно для работы. Передавать нужно только те данные, которые реально нужны для работы формы, передача же лишних данных может привести к их повреждению или потере.

2.3.6 Верстка

Пользовательский интерфейс – это то, с чем будет работать конечный пользователь. От качества пользовательского интерфейса, от его удобства, красоты и аккуратности во многом зависит, будет ли пользователь работать в вашей программе и купит ли он её. Есть множество примеров, когда программы проваливались в продаже из-за плохого или неаккуратного пользовательского интерфейса. Неаккуратный пользовательский интерфейс создает ощущение школьной поделки, непроверенной и не оттестированной программы. Пользователь не хочет платить деньги за то, что не создает ощущения уверенности – это особенно актуально, когда речь идет об корпоративных enterprise-системах или САПР.

Проектирование пользовательского интерфейса задача UX-специалистов (User eXperience), красота и аккуратность – задача дизайнеров. Разработчик лишь должен в точности повторить предоставленный дизайнерами макет. Однако любой разработчик должен понимать основные понятия дизайна интерфейсов, а также общие принципы построения интерфейса:

- 1) Расположение элементов в порядке важности их функционального назначения или в порядке работы пользователя с ними.
- 2) Определение размеров элементов для комфортной работы пользователя.
- 3) Выравнивание элементов относительно друг друга.
- 4) Соблюдение расстояний (padding&margins) между элементами.

Также, для удобства пользователей интерфейсы десктоп- и веб-приложений должны быть адаптивными – то есть интерфейс должен реагировать на изменение размера окна, пропорционально изменения размеры внутренних компонентов. Учитывая, что размеры мониторов у пользователей могут сильно отличаться, адаптивность интерфейса по возможности надо закладывать в реализацию на более ранних этапах.

Для аккуратного расположения элементов относительно друг друга и краёв формы, в дизайнере форм есть направляющие. Направляющие – это синие и розовые линии, появляющиеся в дизайнере форм, когда вы двигаете на нем элементы управления. Синие линии соответствуют рекомендованным отступам (padding) между элементами – старайтесь располагать элементы согласно направляющим.

Розовые линии – линии нижней границы текста. Дело в том, что у подписи Label и текстового поля TextBox различная высота. В итоге, если выравнивать эти элементы относительно друг друга по верхней или нижней грани, текст в них будет располагаться на разных высотах, что выглядит неаккуратно, «криво». Для того, чтобы пользователь воспринимал элементы аккуратно, правильнее будет выравнивать элементы по нижней линии текста внутри элементов. Тогда текст будет располагаться на одной линии и общее впечатление будет лучше. Можете убедиться в этом, разместив элементы по розовой линии и не без расположения по розовой линии. Соблюдение одинаковых отступов (одинаковых между всеми элементами или блоками элементов), а также выравнивание по направляющим, являются базовыми правилами для создания визуально аккуратного интерфейса. Но главный критерий для разработчика – это утвержденный макет интерфейса. Интерфейс, который вы сверстаете должен полностью повторять макет, в отдельных случаях – с точностью до пикселя. Соблюдение макета «пиксель в пиксель» также называется «Pixel Perfect».

Основные свойства элементов Windows Forms, использующиеся для правильной верстки интерфейса, описаны в [\[4\]](#). К таким свойствам относятся Anchor и Dock.

Свойство Anchor (с англ. «якорь») отвечает за привязку элемента управления к краям родительской формы. В случае перемещения краев, например, при изменении размеров формы, элемент перемещается вместе с тем краем, к которому он прикреплен. Элемент может быть прикреплен как к одному краю, так и к двум или четырем, так и не быть прикрепленным ни к одному из краёв. По умолчанию все элементы привязываются к верхнему левому краю. При таком прикреплении, если форма меняет свой размер, элементы остаются на своих местах (свойство Location не меняет значений). Если же прикрепить элемент к нижнему правому углу, тогда элемент будет перемещаться вместе с краями формы.

Для того, чтобы понять, как именно работает свойство Anchor в разных комбинациях, создайте следующие две формы с девятью кнопками, где каждой кнопке назначена своя уникальная комбинация якорей:

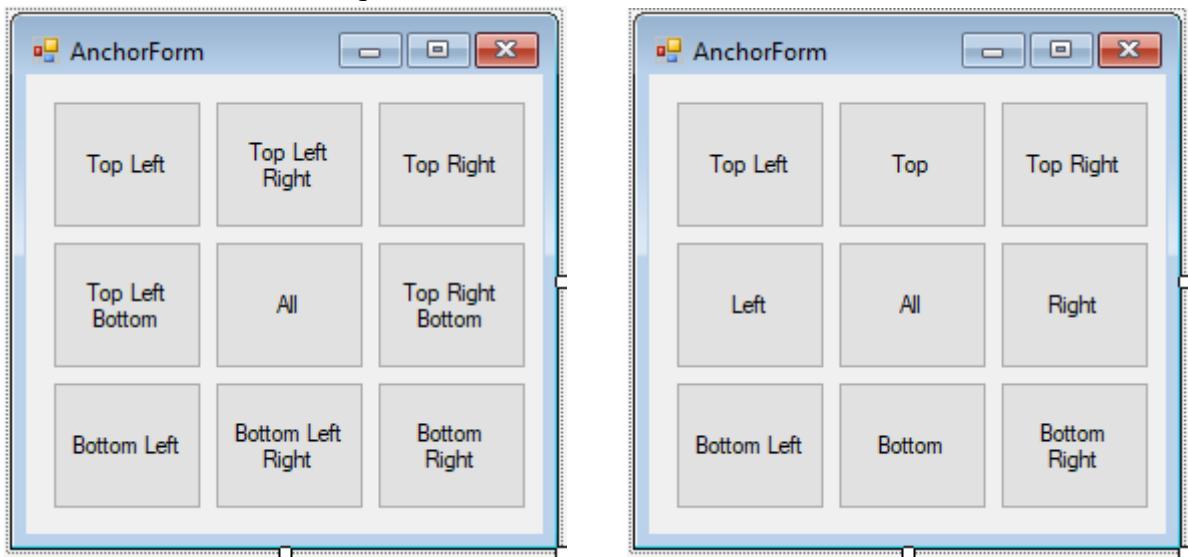


Рисунок 63 – Пример верстки окна для изучения влияния свойства Anchor

Запустите программу с каждым из этих окон и попробуйте растягивать и сжимать окно. Обратите внимание как именно поведение элементов зависит от размера окна.

Также в Windows Forms есть элементы-контейнеры. Это рамка GroupBox, табличная верстка TableLayoutPanel, это разделенный контейнер с регулируемой шириной SplitContainer. Они предназначены для создания более сложной и адаптивной верстки. Особенность элементов-контейнеров заключается в том, что элементы управления, расположенные на них, фактически размещаются не на форме, а именно в контейнере. Таким образом, свойства Location или Anchor будут зависеть не от родительской формы, а от элемента-контейнера.

GroupBox предназначен для группировки элементов на форме. Имея заголовок, GroupBox помогает визуально разделить окно на блоки и упростить навигацию в программе.

SplitContainer состоит из двух панелей (левой и правой, либо верхней и нижней), между которыми расположен так называемый разделитель Splitter. Во время работы приложения пользователь может двигать разделитель, тем самым меняя пропорции между двумя панелями. SplitContainer позволяет достаточно просто реализовать двухколоночную верстку, разделив окно на основную часть и дополнительную панель.

TableLayoutPanel позволяет создать таблицу, каждая ячейка которой является отдельным контейнером. В настройках TableLayoutPanel можно указать поведение строк и столбцов при изменении размеров окна. Можно задать либо абсолютный размер для столбца или строки,

либо задать его относительный размер в процентах. Как итог можно создавать трёхколоночную верстку, где, например, один столбец фиксированной величины, а два других автоматически занимают оставшееся пространство.

Также нужно помнить, что все элементы-контейнеры могут комбинироваться между собой. Например, в любую ячейку TableLayout можно поместить SplitContainer или еще один TableLayout, сделав еще более мелкое разбиение ячейки. Всё это позволяет создавать достаточно сложные адаптивные интерфейсы программ.

Так, задание данной лабораторной предполагает, что верстку главного окна программы вы будете реализовывать с помощью TableLayout или SplitContainer, а отдельные элементы группировать с помощью GroupBox.

2.3.7 Задание

- 1. Сверстать формы и пользовательские элементы управления согласно макетам:**
 - 1.1. Добавить класс форм согласно макетам пользовательского интерфейса, представленных в варианте технического задания.
 - 1.2. Разместить элементы управления на формах. Элементы должны быть размещены с помощью дизайнера Visual Studio с учетом стандартных отступов и направляющих.
 - 1.3. Реализовать адаптивную верстку форм, используя свойства Anchor, Dock, MinimumSize, MaximumSize, а также используя Panel, SplitContainer и TableLayout в качестве компонентов-контейнеров.
 - 1.4. Создать и сверстать собственные пользовательские элементы управления, если этого требует техническое задание. Разместить собственные элементы управления на ранее созданных формах.
 - 1.5. Во всех созданных формах переименовать размещенные на них элементы управления согласно RSDN. Например, вместо компонента textbox1 должен быть TitleTextBox – название, отражающее назначение компонента на форме. Именование в стиле Паскаль.
 - 1.6. Скомпилировать программу и запустить. Убедиться, что адаптивная верстка работает так, как того требует техническое задание.
 - 1.7. Если верстка работает верно, сделать коммит в локальный репозиторий. Не забудьте написать правильный комментарий к коммиту: а) что было сделано в коммите; б) какие классы были добавлены; в) какие классы были изменены или удалены.
- 2. Реализовать логику работы пользовательского интерфейса:**
 - 2.1. В главном окне создайте поле, которое будет хранить экземпляр класса Project (класс из проекта логики, разработанный в предыдущей лабораторной работе). Поле должно инициализироваться экземпляром в конструкторе главного окна.
 - 2.2. Реализуйте обработчики событий в формах, которые позволят работать с проектом: добавлять записи в проект, удалять и редактировать записи.
 - 2.3. Так как создание и редактирование записей в проекте Project происходит в дополнительном (не главном) окне приложения, реализуйте механизм передачи данных из главного окна во второстепенное. Также реализуйте механизм передачи данных в созданный вами элемент управления, если он предусмотрен техническим заданием.
 - 2.4. Реализуйте логику сохранения и загрузки Project согласно техническому заданию.

- 2.5. Если логика приложения работает верно, сделайте коммит в локальный репозиторий. В комментарии к коммиту напишите, что вы добавили или изменили в программе.
- 3. Реализовать защиту от неправильного ввода:**
 - 3.1. Для всех элементов управления, в которые пользователь осуществляет ввод данных, реализуйте механизм защиты от неправильного ввода.
 - 3.2. Механизм защиты должен быть двухуровневым: а) запрет ввода некорректных символов или некорректного количества символов в элемент управления; б) проверка введенного значения после потери фокуса элемента. В случае нарушения проверки введенного значения программа должна подсветить элемент управления красным цветом до ввода корректного значения.
 - 3.3. Если защита от неправильного ввода работает верно, сделайте коммит в локальный репозиторий с соответствующим комментарием. Синхронизируйте локальный репозиторий с удаленным репозиторием.
- 4. Проверить, что задание лабораторной работы выполнено верно:**
 - 4.1. Зайдите в свой аккаунт GitHub и убедитесь, что синхронизация прошла успешно.
 - 4.2. Сделайте снимки экрана работы вашего приложения с реальными данными. Выполните снимки для всех созданных форм и компонентов. Сделайте снимок, демонстрирующий работу механизма защиты от неправильного ввода в случае, когда пользователь ввёл неправильные данные. На основе сделанных снимков экрана составьте отчет к лабораторной работе.
 - 4.3. Если все задания выполнены верно, ответьте на вопросы для самоподготовки и переходите к защите лабораторной работы. На защите продемонстрируйте работу программы, структуру файлов в удаленном репозитории и историю коммитов.

2.3.8 Вопросы для самоподготовки

1. Чем отличаются понятие формы Form от элемента управления Control?
2. Что значат понятия родительский и дочерний элементы управления?
3. В чем разница между методами Show() и ShowDialog()?
4. Как работать со стандартными окнами MessageBox, SafeFileDialog, OpenFileDialog?
5. Для чего нужны стандартные элементы управления SplitContainer, TableLayoutPanel, FlowLayoutPanel?
6. Как сделать двухколоночную верстку, где обе колонки пропорционально меняют свою ширину при изменении ширины окна? Как сделать трехколоночную верстку, где одна колонка при изменении ширины окна остается фиксированного размера, а две колонки меняются в пропорциях ширины 40/60?
7. Что делают свойства Anchor и Dock у каждого элемента управления?
8. Как сделать элемент управления, автоматически меняющий ширину под размер родительского элемента? Как сделать элемент управления, автоматически меняющий высоту под размер родительского элемента?
9. Что означают свойства Location и Size у элемента управления? Как сделать элемент управления, меняющий свой размер в зависимости от размера родителя, но ограниченный в максимальных или минимальных размерах?
10. В чем разница между свойствами Enabled и Visible у элементов управления?
11. За что отвечает свойство TabIndex у элементов управления?

12. Когда срабатываются события MouseDown, MouseEnter, MouseMove, MouseLeave?
13. В чем разница между событиями Leave и MouseLeave? Enter и MouseEnter? Когда срабатывает каждое из них? Для чего их можно использовать?

2.3.9 Содержание отчета

Отчет по лабораторной работе должен содержать:

- Титульный лист, оформленный согласно требованиям ОС ТУСУР [11].
- Снимки экрана всех разработанных окон и компонентов с текстовыми пояснениями о работе пользователя в окне.
- Описание свойств разработанного элемента управления. Пример исходного кода создания экземпляра компонента, его инициализации, передачи данных в компонент и получение данных из компонента.
- Текущая история коммитов ветки develop (допустимо в виде снимка экрана соответствующей страницы GitHub или VisualStudio).

Отчёт сдается с указанием даты защиты лабораторной и подписью студента.

2.3.10 Список источников

Верстка пользовательского интерфейса на Windows Forms

1. Руководство по программированию в Windows Forms [Электронный ресурс]. / matanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/windowsforms/> (дата обращения 27.08.2018).
2. Основные свойства форм [Электронный ресурс]. / matanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/windowsforms/2.2.php> (дата обращения 27.08.2018).
3. Добавление форм. Взаимодействие между формами [Электронный ресурс]. / matanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/windowsforms/2.3.php> (дата обращения 27.08.2018).
4. Размеры элементов и их позиционирование в контейнере [Электронный ресурс]. / matanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/windowsforms/3.4.php> (дата обращения 27.08.2018).
5. OpenFileDialog и SaveFileDialog [Электронный ресурс]. / matanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/windowsforms/4.20.php> (дата обращения 27.08.2018).
6. Соглашение по оформлению кода команды RSDN [Электронный ресурс]. / RSDN (Russian Software Developers Network): сайт о программировании. — URL: <https://rsdn.org/article/mag/200401/codestyle.XML> (дата обращения 27.08.2018).

Разработка логики пользовательского интерфейса в Windows Forms

7. Шилдт Г. C# 4.0: полное руководство.: пер. с анг. Бернштейн И.В. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.: ил.
8. События в Windows Forms. События формы [Электронный ресурс]. / matanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/windowsforms/2.4.php> (дата обращения 27.08.2018).
9. Событие Control.KeyPress [Электронный ресурс]. / Официальный портал Microsoft. — URL: [https://msdn.microsoft.com/ru-ru/library/system.windows.forms.control.keypress\(v=vs.110\).aspx](https://msdn.microsoft.com/ru-ru/library/system.windows.forms.control.keypress(v=vs.110).aspx)

- [ru/library/system.windows.forms.control.keypress\(v=vs.110\).aspx](https://msdn.microsoft.com/ru-ru/library/system.windows.forms.control.keypress(v=vs.110).aspx) (дата обращения 27.08.2018).
10. Событие Control.Leave [Электронный ресурс]. / Официальный портал Microsoft. — URL: [https://msdn.microsoft.com/ru-ru/library/system.windows.forms.control.leave\(v=vs.110\).aspx](https://msdn.microsoft.com/ru-ru/library/system.windows.forms.control.leave(v=vs.110).aspx) (дата обращения 27.08.2018).

Требования к оформлению отчета

11. ОС ТУСУР 01-2013. Работы студенческие по направления подготовки и специальностям технического профиля. Общие требования и правила оформления. – Томск: ТУСУР, 2013. – 57с. – URL: https://storage.tusur.ru/files/40668/rules_tech_01-2013.pdf (дата обращения 27.08.2018)

2.4 Юнит-тестирование

Цель работы: изучить организацию тестирования в разработке ПО, и получить умения написания юнит-тестов.

Задачи:

1. Изучить организацию процесса тестирования в разработке ПО, виды тестирования, сроки их проведения и ответственных исполнителей.
2. Изучить основные атрибуты и классы библиотеки NUnit для написания юнит-тестов.
3. Научиться рассчитывать цикломатическую сложность методов, классов и проектов, оценивать степень покрытия кода тестами.
4. Написать юнит-тесты для классов логики приложения с использованием библиотеки NUnit.

2.4.1 Организация тестирования в проекте

Любой инженер (а программист – это инженерная специальность) обязан проверять и гарантировать качество своих решений. Поэтому прежде, чем отдать программу заказчику или пользователям, разработчик ПО обязан выполнить тестирование. И только в том случае, если программа работает корректно, проект можно считать законченным.

Тестирование – процесс проверки соответствия объекта заявленным требованиям [1, 2, 3]. Таким образом, для проведения тестирования нужно: 1) тестируемый объект; 2) требования – техническое задание. Без четкого технического задания нельзя определить, соответствует ли объект требуемому качеству.

Тестирование сложно назвать отдельным этапом разработки ПО, так как тестирование выполняется параллельно реализации, а подготовка к тестированию начинается после утверждения ТЗ, на этапах макетирования и проектирования.

Тестирование требует особых навыков. По этой причине, чтобы ускорить разработку в команде всегда выделяют нескольких инженеров-тестировщиков. Тестировщик, он же специалист по контролю качества, он же QA-специалист (от англ. «Quality Assurance» – обеспечение качества).

Классификация тестов насчитывает свыше 20 видов, все они в той или иной мере выполняются в ходе разработки ПО [1]. Однако хотелось бы выделить 4 вида тестов:

- 1) Тестирование новой функциональности.
- 2) Регрессионное тестирование.
- 3) Юнит-тестирование.
- 4) Приёмочное тестирование.

Тестирование новой функциональности – при разработке новой версии приложения в программе появляются новые функции. Задача данного вида тестирования проверить правильность работы новой функциональности. Как правило, выполняется вручную сначала разработчиками (в процессе разработки), а затем тестировщиками, затем покрывается автоматизированными тестами. В отличие от приёмочного тестирования, тестирование новой функциональности предполагает тестирование каждой отдельной новой функции независимо от остальной системы. Тестирование проводится по мере готовности новых функций программы.

Регрессионное тестирование – тестирование ранее разработанной функциональности. В ходе разработки новой функциональности могут быть переписаны части кода старой системы, что приведет к появлению ошибок в ранее созданной функциональности. Как правило, разработчики, которые создают новую функциональность, при разработке концентрируются только на тестировании собственной задачи, не думая о том, что могли что-то повредить в другой части программы. Для того, чтобы гарантировать работоспособность старых функций, необходимо проводить их периодическое тестирование.

Логично, что с разработкой программы старых функций становится всё больше и больше, а значит и объём тестирования будет постоянно расти. Чтобы время тестирования не разрасталось, именно регрессионное тестирование подлежит максимальной автоматизации. Автоматизированные тесты составляются тестировщиками с помощью специализированных фреймворков. Когда выполнение всех автоматизированных тестов начинает занимать продолжительное время, в компании для их выполнения внедряют сервер (серверы) тестирования.

Юнит-тестирование – тестирование минимальных модулей архитектуры, максимально изолированных друг от друга. Так как минимальными модулями архитектуры как правило являются классы, юнит-тестирование фактически является тестированием исходного кода. В отличии от регрессионного тестирования, юнит-тесты пишутся на языке программирования программы в том же проекте. Написание юнит-тестов – обязанность разработчиков [4, 5, 6].

В отличии от других видов тестов, юнит-тесты как правило составляются на основе разработанных алгоритмов, а не на основе технического задания. Разработчик, написавший некоторый алгоритм, определяет необходимое количество тестов, проверяющих все возможные варианты исходных данных и результатов выполнения. Степень покрытия исходного кода юнит-тестами в различных компаниях может отличаться и зависит от многих организационных факторов.

Приёмочное тестирование – тестирование завершенной разработанной версии программы перед передачей разработки заказчику. Приёмочное тестирование выполняется при представителе заказчика и фактически является завершением проекта или этапа разработки. В конце приёмочного тестирования представитель заказчика подписывает акт приёма-передачи проекта, либо составляется перечень обнаруженных недостатков. В отличие от тестирования новой функциональности, приёмочное тестирование проводится комплексно, на всей системе целиком. Процедура проведения приёмочного тестирования может быть описана в техническом задании; может заключаться как в полной проверке соответствия проекта техническому заданию, так и проверке отдельных ключевых функций программы.

Как говорилось ранее, в процессе разработки продукта, объёмы тестирования будут постоянно расти. Поэтому если при разработке первых версий приложения допустимо проведения только ручного тестирования, то уже через пару месяцев необходима автоматизация этого процесса. Главная проблема автоматизации тестирования заключается в том, что автоматизированные тесты по существу тоже являются кодом. А это значит, что этот код также надо поддерживать, и этот код также может содержать ошибки. Более того, количество кода, необходимого для тестирования какой-либо функции может в разы превышать количество кода самой функции. Чтобы избежать постоянного переписывания кода тестов, необходимо, что команда обладала достаточным опытом в тестировании систем. Умение тестировать код заставляет разработчика заранее задумываться об архитектуре системы.

Архитектуры становятся проще, удобнее в использовании и тестировании. Поэтому целью данной лабораторной работы является формирование навыков в написании юнит-тестов.

2.4.2 Общие принципы написания юнит-тестов

Юнит-тестирование – тестирование программы в виде отдельных, изолированных друг от друга минимальных модулей [4, 5, 6]. Условие изоляции тестируемого модуля от других необходимо для того, чтобы в случае обнаружения ошибки быть уверенным, что ошибка возникла именно в тестируемом модуле. В противном случае найти место возникновения ошибки будет гораздо сложнее.

В процедурном программировании минимальными модулями для юнит-тестирования являются функции или процедуры. В ООП минимальными модулями считаются классы. Именно классы, а не методы, так как поведение методов может зависеть от текущего состояния экземпляра класса. Таким образом, юнит-тесты в ООП это тесты, выполняющие тестирование отдельных классов с максимально возможной изоляцией от поведения других классов.

Юнит-тестированию подлежит поведение класса, находящееся под модификаторами доступа public или protected. Закрытая реализация класса (private) напрямую тестируемому не подвергается, только опосредованно через вызов открытых методов класса. Это обусловлено тем, что юнит-тесты фактически имитируют случаи реального использования классов логики, а, следовательно, не должны нарушать инкапсуляцию тестируемого класса.

Рассмотрим тестирование следующего класса:

```
public class Contact
{
    private string _surname;

    public string Surname
    {
        get
        {
            return _surname;
        }
        set
        {
            if (value.Length > 40)
            {
                var name = nameof(Surname);
                throw new ArgumentException($"Длина {name} равна {value.Length}, " +
                    $"а должно быть меньше 40.", name);
            }
            _surname = value;
        }
    }
}
```

Класс Contact содержит одно закрытое поле _surname и одно открытое свойство Surname, описывающие фамилии некоторого контакта. Закрытое поле тестируемому не подлежит, необходимо протестировать только свойство Surname. Однако не стоит забывать, что свойство фактически представляет собой два метода get и set, каждое из которых должно быть протестировано.

Метод set содержит внутри себя условие, проверяющее длину фамилии. Если присваиваемое значение длины превысит 40 символов, то будет сгенерировано исключение, и программа аварийно завершится. Если же фамилия короче 40 символов, то новое значение фамилии будет присвоено в поле _surname.

Таким образом, логика метода set может завершиться одним из двух вариантов: присвоение корректной фамилии или генерацией ошибки. А значит, необходимо написать два теста для метода set, проверяющих каждый результат его вызова. Первый тест – проверка присвоения корректной фамилии – может выглядеть следующим образом:

```
public void Test_Surname_Set_CorrectValue()
{
    try
    {
        Contact contact = new Contact();
        contact.Surname = "Смирнов";
        Console.WriteLine("Тест Surname присвоение корректной фамилии: пройден");
    }
    catch (ArgumentException exception)
    {
        Console.WriteLine("Тест Surname присвоение корректной фамилии: провален");
    }
}
```

Наш тест создает экземпляр контакта, после чего выполняет присвоение фамилии. Если в процессе присвоения произойдет генерация исключения, тогда выполнение перейдет в блок catch, и метод выведет на экран текст, что тест провален. Если же исключения не сгенерировано, на экран выводится текст, что тест пройден.

Сразу стоит обратить внимание, что вывод результатов тестирования в консоль на практике не применяется. Вместо этого используются специализированные сторонние библиотеки юнит-тестирования, такие как nUnit, выполняющие автоматический подсчет выполненных и проваленных тестов с формированием отчета. Об использовании библиотеки nUnit будет рассказано позже в п. 2.4.4. В этом же разделе будет использоваться консоль исключительно для примера работы юнит-тестов.

Аналогично первому тесту можно составить тест для проверки присвоения некорректной фамилии:

```
public void Test_Surname_Set_UncorrectValue()
{
    try
    {
        Contact contact = new Contact();
        contact.Surname = "Смирнов-Смирнов-Смирнов-Смирнов-Смирнов-Смирнов";
        Console.WriteLine("Тест Surname присвоение фамилии больше 40 символов: провален");
    }
    catch (ArgumentException exception)
    {
        Console.WriteLine("Тест Surname присвоение фамилии больше 40 символов: пройден");
    }
}
```

Разница заключается лишь в том, что тест будет считаться пройденным только в том случае, если будет сгенерировано исключение. Если же исключение на длинную фамилию не сгенерируется, тест будет провален.

Первый тест относится к так называемым **позитивным тестам**. Позитивный тест – тест, в котором проверяется работа метода при корректных исходных данных. Второй тест

относиться к **негативным тестам**. Негативный тест – тест, в котором проверяется обработка методом заведомо некорректных исходных данных.

Негативные тесты играют важную роль в юнит-тестировании. Именно негативные тесты позволяют проверить, защищает ли класс себя от попытки передачи ему неправильных данных. То есть, действительно ли класс сигнализирует об ошибке, если его используют неправильно. Рассмотрим смысл негативного теста подробнее.

В нашем тесте мы создали фамилию длиннее 40 символов. Мы знаем, что такая фамилия некорректна для нашего класса. Более того, ограничение в 40 символов скорее всего будет прописано в техническом задании при разработке реального проекта.

Так как мы передаем в сеттер класса неправильную фамилию, мы ожидаем, что класс должен сгенерировать исключение. Если же класс этого исключения не генерирует, это означает, что в поле `_surname` помещена фамилия длиннее 40 символов – а это нарушение технического задания. Как результат, некорректное значение в поле `_surname` может привести к сбоям в остальных частях программы, которые будут обращаться к этому некорректному полю.

Следовательно, мы должны передать заведомо некорректное значение в сеттер класса и убедиться, что исключение нужного типа действительно будет генерировано. Если класс реагирует исключением на некорректные данные, он защищает себя от неправильного использования.

Два написанных теста покрывают весь код метода `set` свойства `Surname` и все возможные случаи. Теперь напишем тест для `get`:

```
public void Test_Surname_Get_CorrectValue()
{
    //Подготовка объекта к тестированию
    Contact contact = new Contact();
    contact.Surname = "Смирнов";

    if (contact.Surname == "Смирнов")
    {
        Console.WriteLine("Тест Surname возврата корректной фамилии: пройден");
    }
    else
    {
        Console.WriteLine("Тест Surname возврата корректной фамилии: провален");
    }
}
```

Суть теста `get` заключается в присвоении в свойство некоторого значения, а затем проверки, действительно ли `get` возвращает ту фамилию, которую мы присвоили. Если фамилия из метода `get` совпадает с исходной, значит тест пройден. Если же фамилия отличается, значит метод `get` работает некорректно.

Так как реализация метода `get` состоит только из одной строки:

```
get
{
    return _surname;
}
```

и не содержит никаких условий, циклов, управления процессом выполнения (`continue`, `break`, `goto`), и не генерирует исключения, для его тестирования достаточно только одного теста.

Мы написали тесты, которые полностью покрывают класс Contact и позволяют не только узнать об ошибке в коде, но и быстро найти место ошибки. Теперь, в случае модернизации класса Contact, запуск написанных тестов позволит быстро определить, не появились ли новые ошибки.

Для удобного запуска всех тестов можно написать дополнительный метод Test_Contact_Suite():

```
public void Test_Contact_Suite()
{
    Test_Surname_Set_CorrectValue();
    Test_Surname_Set_UncorrectValue();
    Test_Surname_Get_CorrectValue();
}
```

Вызов метода позволит сразу же выполнить все тесты класса Contact, а при написании новых тестов, их вызов нужно будет добавить в этот метод.

С точки зрения программной архитектуры, тесты располагаются в отдельных классах отдельного проекта. Например, для наших трёх тестов мы должны создать класс ContactTest в отдельном проекте UnitTests. Хранение же тестов непосредственно в классе Contact будет архитектурной ошибкой, усложняющей чтение кода класса Contact, а также сильно раздувая размер компилируемой dll:

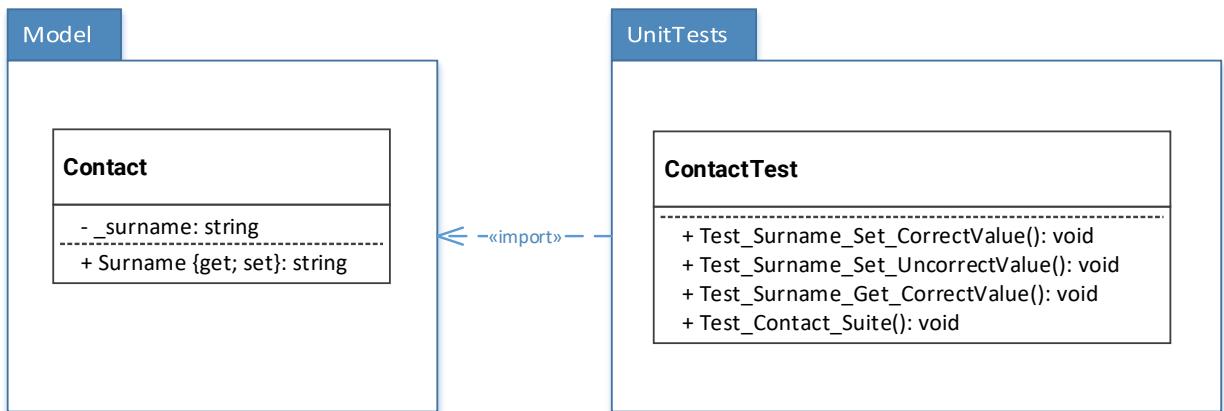


Рисунок 64 – Юнит-тесты, как правило, выделяются в отдельный пакет/проект, к которому в качестве ссылки подключается тестируемый проект бизнес-логики. На каждый класс бизнес-логики создается отдельный класс тестов в проекте юнит-тестов

Еще один нюанс, на который вы возможно обратили внимание, это то, что тест метода get содержит в себе вызов метода set. Это логично, так как прежде чем получить значение из закрытого поля, мы сначала должны это значение как-то туда поместить. Однако, если тест будет провален, то нельзя достоверно определить, была ли это ошибка метода get или всё-таки ошибка возникла еще на этапе присвоения в методе set. К сожалению, такие неоднозначности могут возникать при написании методов. Более того, ошибка может даже находиться в конструкторе класса (который также необходимо было бы протестировать, если бы он был реализован в классе Contact). Фактически, в случае падения подобных тестов, мы сможем узнать, что в коде есть ошибки, но не сможем быстро определить их причину и исправить. Разрешение подобных неоднозначностей зависит от конкретного случая. В случае с нашими тестами место ошибки можно будет однозначно локализовать, просмотрев результаты выполнения остальных тестов. Так, если тест Test_Surname_Get_CorrectValue()

будет провален, а тест Test_Surname_Set_CorrectValue() пройден – значит ошибка однозначно в реализации get. Если же оба теста будут провалены, то вероятнее всего ошибка в методе set, так как именно он вызывается в обоих тестах.

2.4.3 Цикломатическая сложность и определение требуемых тестов

Теперь пришло время поговорить о том, как же определить все требуемые тесты для того или иного класса. Мы уже определили, что для полного покрытия класса Contact необходимо три теста, однако на практике классы имеют куда более сложную реализацию и требуют большего числа тестов. Так, для одного класса может потребоваться порядка пятидесяти или даже сотни тестов. Как же определить количество требуемых тестов и их исходные данные, позволяющие полностью покрыть исходный код тестами?

Определение тестовых случаев основывается на понятии цикломатической сложности. Строго говоря, цикломатическая сложность алгоритма (функции, метода) — количество линейно независимых маршрутов через программный код [7]. Например, если исходный код не содержит никаких точек ветвления или циклов, то сложность равна единице, поскольку есть только единственный маршрут через код. Если код имеет единственный оператор if, содержащий простое условие, то существует два пути через код: один если условие оператора if имеет значение true и один — если false. Определить значение цикломатической сложности можно с помощью блок-схемы или ориентированного графа алгоритма.

Например, метод get свойства Surname не имеет никаких точек ветвления, а потому цикломатическая сложность метода равна 1. Следовательно, требуется только один тест для полного покрытия этого метода тестами. Метод set в свою очередь содержит оператор if с простым условием (value.Length > 40), и необходимо два теста, проверяющих каждую ветку условного оператора:

```
get
{
    return _surname;
}

set
{
    if (value.Length > 40)
    {
        throw new ArgumentException();
    }
    _surname = value;
}
```

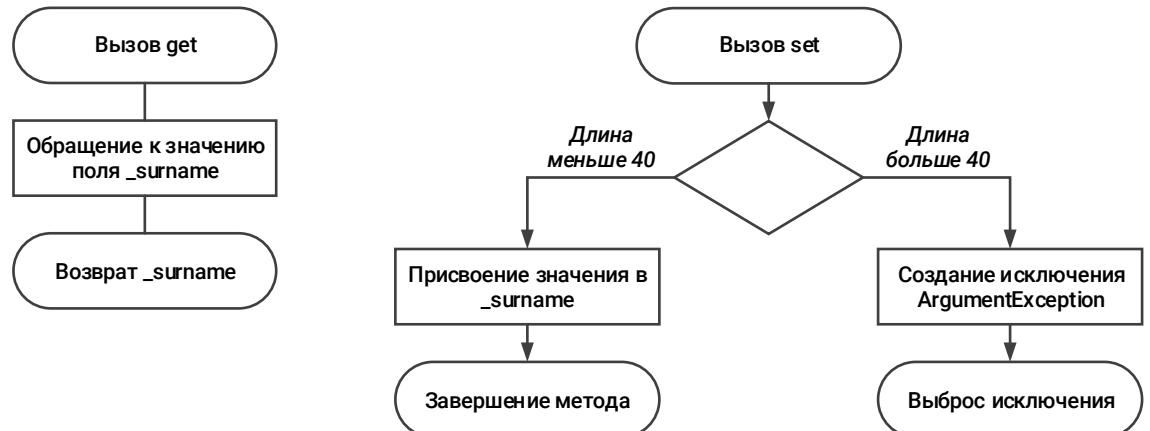


Рисунок 65 – Блок-схемы алгоритмов с разной цикломатической сложностью на примере оператора ветвления. Правая блок-схема имеет две ветки выполнения, а следовательно, её цикломатическая сложность равна 2.

Цикломатическая сложность класса в свою очередь определяется суммой цикломатических сложностей всех открытых методов класса. Например, сложность класса Contact равна 3 (сумме сложности get и set).

В случае последовательного расположения двух операторов if необходимо считать ветвление для каждого условия. Например, если в метод set добавить еще один условный оператор, его сложность составит 3:

```
set
{
    if (value.Length == 0) // Первое ветвление
    {
        throw new ArgumentException("Фамилия не должна быть пустой");
    }
    if (value.Length > 40) // Второе ветвление
    {
        var name = nameof(Surname);
        throw new ArgumentException($"Длина {name} равна {value.Length}, " +
            $"а должно быть меньше 40.", name);
    }
    _surname = value; // Третье ветка ветвления
}
```

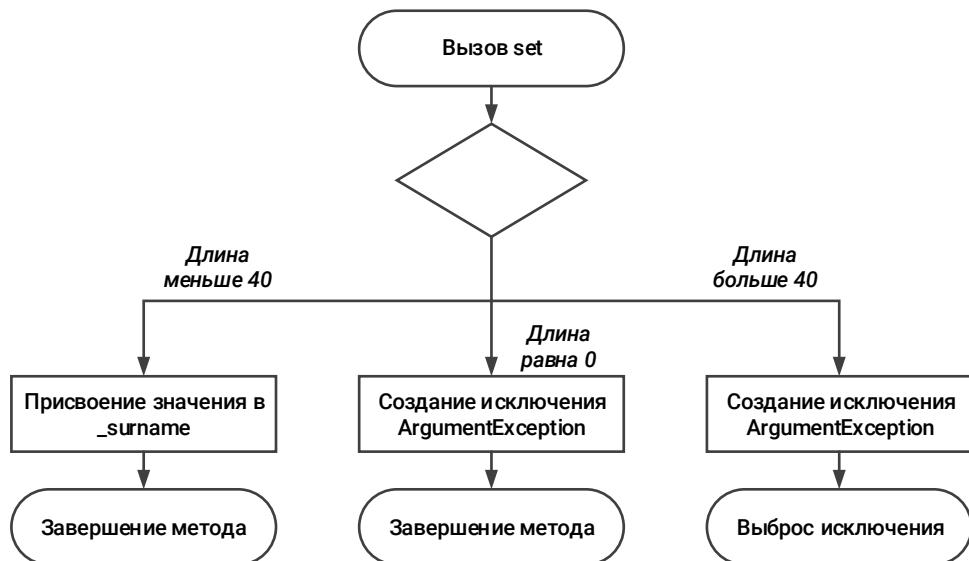


Рисунок 66 – Каждый оператор ветвления или составное условие увеличивает цикломатическую сложность

Аналогично цикломатическая сложность увеличивается, если условие под оператором if становится составным:

```
if (value.Length == 0 || value.Length > 40)
{
    //...
}
```

При составных условиях необходимо учесть все возможные комбинации входных параметров, при которых обеспечивается попадание под условие. Недостаточно будет написать тест, который входит в составное условие только один раз. Необходимо написать такое количество тестов, которые входили бы в условие при каждом простом условии внутри составного.

Операторы цикла также увеличивают цикломатическую сложность на единицу.

Например:

```
//Возвращает сумму списка чисел
public double Sum(List<double> numbers)
{
    double sum = 0.0;
    foreach (var number in numbers)
    {
        sum += number;
    }
    return sum;
}
```

Для метода Sum() цикломатическая сложность равна 2, и необходимо написать два теста – при котором алгоритм попадает внутрь цикла (когда входной список не пустой), и при котором алгоритм не входит в цикл (при пустом входном списке).

При определении цикломатической сложности метода учитывается всё – составные условия, наличие циклов, операторов break и continue, вызов других методов. Если внутри тестируемого метода вызывается другой метод, то это необходимо учитывать в значении цикломатической сложности:

```
public bool OuterMethod(double value)
{
    if /*condition1*/
    {
        //...
    }
    else
    {
        return Goo(value);
    }
}

private bool InnerMethod(double value)
{
    if /*condition2*/
    {
        //...
    }
    else if /*condition3*/
    {
        //...
    }
    else //...
}
```

Метод OuterMethod() содержит один условный оператор, и можно подумать, что его цикломатическая сложность равна 2. Однако под оператором else происходит вызов метода InnerMethod(), цикломатическая сложность которого равна 3. Чтобы определить цикломатическую сложность метода OuterMethod() попробуйте мысленно подставить код InnerMethod() в точку его вызова. Тогда мы получим лестничную структуру условий, цикломатическая сложность которых равна 4. Таким образом, реальная цикломатическая сложность метода OuterMethod() равна 4, и требуется 4 теста для полного покрытия кода.

Можно понять, что цикломатическая сложность будет расти при большей иерархии вызова методов внутри класса, а также при вызове методов агрегируемых объектов. Чем выше класс стоит в иерархии агрегирования, тем большую цикломатическую сложность (теоретически) имеют его методы. Это отражается в необходимости большого числа тестов

для качественного и полного тестирования кода. Однако для борьбы с ростом цикломатической сложности необходимо:

- 1) инкапсулировать как можно больше логики внутри класса, оставляя меньше открытых методов и с меньшим количеством входных и выходных аргументов.
- 2) по возможности декомпозировать систему на небольшие классы, с тремя-пятью открытыми методами.
- 3) агрегировать объекты не через конкретный тип данных, а через интерфейс – с возможным использованием паттерна Dependency Injection [4, 5]. Изучение паттерна Dependency Injection оставьте после прохождения данного учебного курса.

Еще раз подчеркнем, что в рамках класса тестированию подлежат все открытые и защищенные методы, в том числе конструкторы, деструкторы, свойства, целевые методы. Для тестирования защищенных методов класс тестов может наследоваться от тестируемого класса и вызывать защищенные методы как часть собственной реализации. Этот же подход применяется для тестирования базовых абстрактных классов. Однако данный подход к покрытию кода тестами в данном учебном курсе не рассматривается.

2.4.4 Написание юнит-тестов с использованием NUnit

Если посмотреть на тест метода get, то можно определить его ключевые элементы:

```
public void Test_Surname_Get_CorrectValue()
{
    //Setup - Подготовка объекта к тестированию
    Contact contact = new Contact();
    contact.Surname = "Смирнов"; //Исходные данные - setup

    //Выполнение тестируемого метода и сравнение с ожидаемым значением
    if (contact.Surname == "Смирнов")
    {
        // Test Description + Test Result
        Console.WriteLine("Тест Surname возврата корректной фамилии: пройден");
    }
    else
    {
        Console.WriteLine("Тест Surname возврата корректной фамилии: провален");
    }
}
```

Во-первых, у каждого теста есть собственное описание TestDescription. Оно содержит информацию о том, что именно проверяется данным тестом. Наличие понятного описания – важное условие для легкости поддержки юнит-тестов в будущем.

Далее, любой тест начинается с подготовки тестируемого объекта. Данный этап называется Setup. В нашем случае, подготовка заключается в создании объекта Contact и присвоения в его поле значения «Смирнов». «Смирнов» в данном случае, это исходные данные для теста – setupData, setupValues или просто setup.

Также строка «Смирнов» является ожидаемым результатом теста, т.е. мы ожидаем, что при правильной работе метода get вернется значение «Смирнов». Ожидаемое значение в тестах называется expected.

В условии оператора if у нас сразу выполняются два действия: вызов тестируемой функции и сравнение полученного результата с ожидаемым значением. Полученный из тестируемого метода результат принято называть actual. Таким образом, цель любого теста – это сравнить actual и expected. Если они равны, метод или класс работает верно.

Приведем тест к каноническому виду:

```

public void Test_Surname_Get_CorrectValue()
{
    //Description: Описание теста
    var description = "Тест Surname возврата корректной фамилии";

    //Setup: Подготовка объекта к тестированию
    var setup = "Смирнов";
    Contact contact = new Contact();
    contact.Surname = setup;

    //Testing: Вызов тестируемого метода
    var expected = "Смирнов";
    var actual = contact.Surname;

    //Assert: Сравнение результата
    var result = (actual == expected) ? "пройден" : "провален";
    Console.WriteLine($"{description}: {result}");

    //Teardown: действия после теста
    //... например, отключение от БД, удаление созданных в тесте файлов и т.д.
}

```

Такое представление теста является наиболее комфортным для чтения и удобным в поддержке. Обратите внимание, что для всех данных создаются отдельные переменные: description, setup, actual, expected, result. Благодаря именованию переменных можно легко разобраться в логике теста, а отсутствие условных операторов упрощает чтение. Придерживайтесь такого оформления кода тестов для повышения читаемости кода.

Также в teste может быть блок Teardown – действия, которые надо сделать после выполнения теста. Такими действиями могут быть отключение от базы данных (если подключение требовалось при выполнении теста), удаление данных из БД (если при выполнении теста в БД необходимо было добавить данные), удаление файлов (если тест тестирует сохранение файлов или сериализацию). Однако в нашем teste Surname таких действий нет, поэтому блок остается пустым.

Теперь, когда мы изучили, как определять тестовые случаи, и разобрались с общей структурой теста, рассмотрим специализированную библиотеку NUnit для написания юнит-тестов.

Подключение NUnit. Библиотека NUnit [9] предназначена для написания юнит-тестов приложений для платформы .NET и доступна для подключения в проект как пакет NuGet (п. 2.2.5) [16]. Для работы с платформой необходимо подключить два пакета:

1. NUnit – непосредственно фреймворк для автоматизации тестирования;
2. NUnit3TestAdapter – дополнительный пакет, позволяющий запускать юнит-тесты в среде Visual Studio. Без данного пакета запуск юнит-тестов придется выполнять в отдельном консольном приложении.

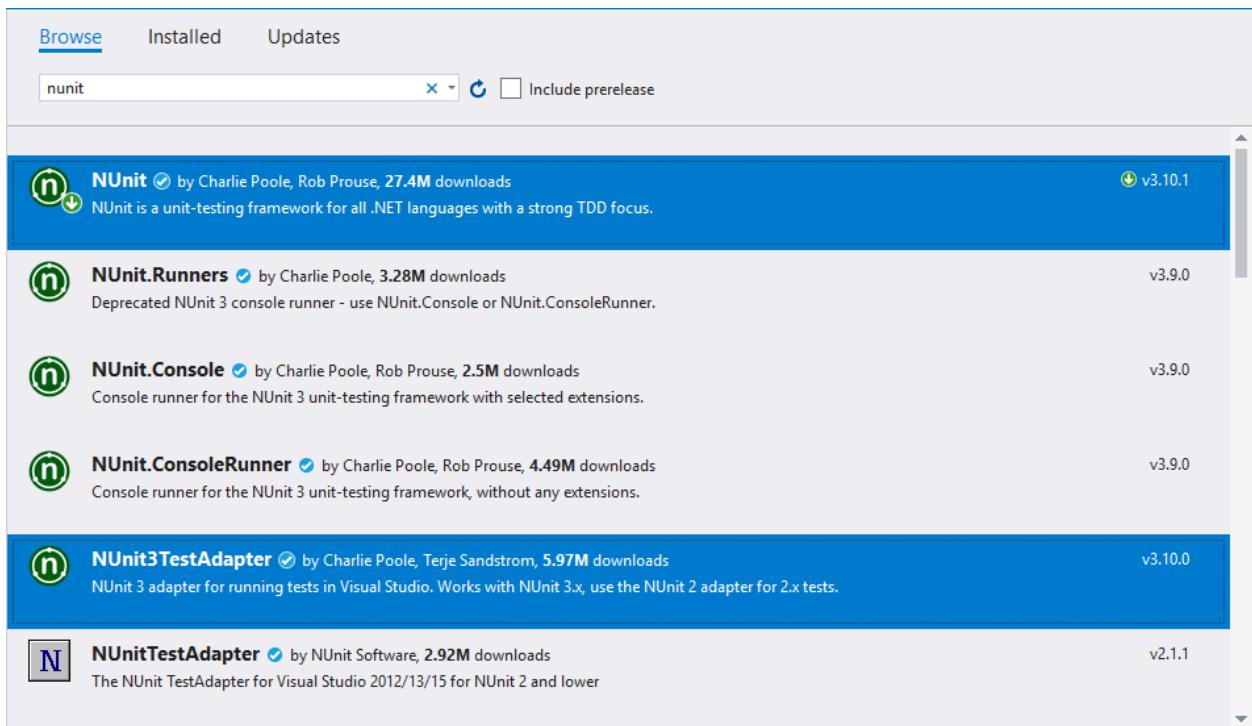


Рисунок 67 – Для работы с библиотекой NUnit необходимо подключить через NuGet следующие пакеты

После подключения библиотеки к проекту юнит-тестов (например, ContactApp.UnitTests), в проекте будут доступны классы и инструменты данной библиотеки. Nunit.framework отобразится в качестве библиотеки в ссылках проекта, nunit.adapter в ссылках проекта не отображается:

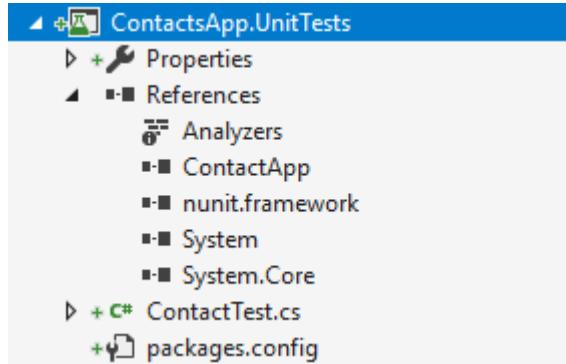


Рисунок 68 – Результат подключения пакетов в проект юнит-тестов

Написание простого теста. Работа с библиотекой основана на использовании класса Assert для сравнения ожидаемых и полученных данных, и использовании так называемых атрибутов [8]. Напишем класс тестов для тестирования Contact:

```
using NUnit.Framework;
using ContactApp;
```

```
namespace ContactsApp.UnitTests
{
    [TestFixture]
    public class ContactTest
    {
        [Test(Description = "Позитивный тест геттера Surname")]
        public void TestSurnameGet_CorrectValue()
        {
            var expected = "Смирнов";
```

```

        var contact = new Contact();
        contact.Surname = expected;
        var actual = contact.Surname;

        Assert.AreEqual(expected, actual, "Геттер Surname возвращает неправильную фа-
милию");
    }
}

```

В первую очередь, для работы с фреймворком в классе ContactTest необходимо подключить соответствующее пространство имен (using NUnit.Framework). Далее, для того, чтобы подключенный адаптер и Visual Studio могли отличить классы с тестами от обычных классов, перед объявлением класса прописывается специальный атрибут [TestFixture]. Благодаря этому атрибуту Visual Studio сможет автоматически определять классы тестов и предлагать их для запуска [10].

Перед методом теста TestSurnameGet_CorrectValue() также необходимо указать атрибут [Test]. Внутри атрибута Test можно задать значение для свойства Description и дать описание данного теста. Наличие описания также упрощает навигацию в юнит-тестах, а потому обязательно.

Далее внутри самого теста всё аналогично ранее написанным тестам: подготовка данных (setup), запуск тестируемой функциональности (testing), проверка результата (assert). Отличие составляет проверка результата – для этого используется метод AreEqual специального класса Assert [12]. В этот метод необходимо передать 3 аргумента: ожидаемое значение expected, реальное значение actual, и текст сообщения ошибки в случае провала теста. Использование класса Assert позволяет избавиться от громоздких конструкций с ветвлением или обработкой исключений try-catch, сократив сравнение двух объектов до одной простой читаемой строчки кода.

Текст ошибки должен быть написан достаточно понятно, чтобы разработчику было проще найти участок кода с ошибкой и исправить его. Очень часто разработчики формально относятся к составлению описания теста Description и текста ошибки в Assert, за счёт чего поиск и исправление ошибки становится непростой задачей. Возможно, данная проблема не очевидна в случае одного теста, но в реальных системах, когда количество тестов может превышать несколько тысяч, это становится большой проблемой.

Запуск тестов. Для запуска теста необходимо включить окно Test Explorer (Обозреватель тестов). Включить его можно через меню Test -> Windows -> Test Explorer, после чего появиться следующая панель:

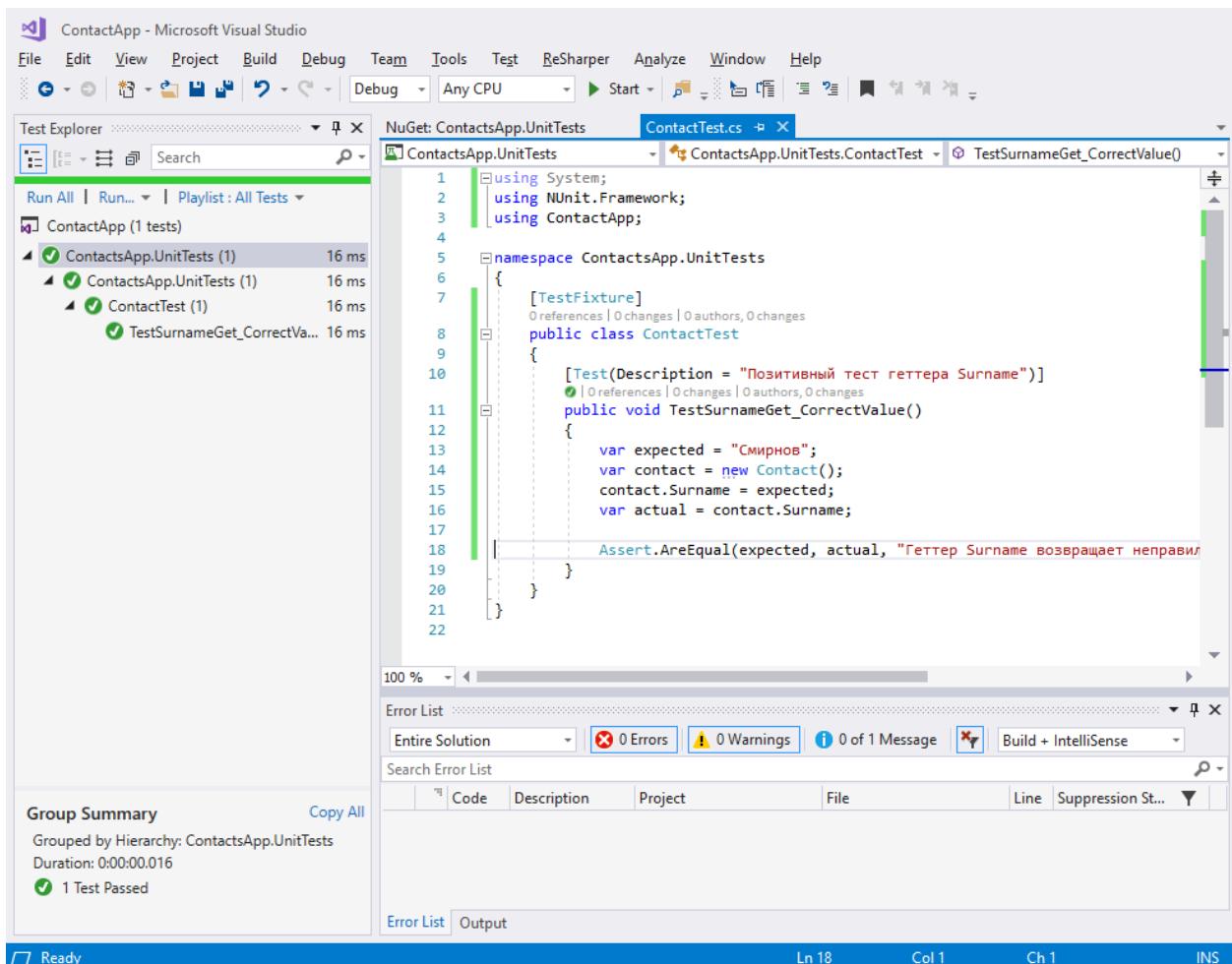


Рисунок 69 – Панель Test Explorer содержит дерево всех обнаруженных юнит-тестов решения

При открытии этой панели, Visual Studio начнет сканирование всего решения на наличие тестов. Для определения тестов, как говорилось ранее, Visual Studio ищет все атрибуты [TestFixture] и [Test] – это обязательное требование для обнаружения тестов. Второе обязательное требование – тестовый класс и методы тестов должны обязательно быть под модификатором доступа public. Все тесты будут сгруппированы по проектам и тестовым классам, что упрощает навигацию.

Нажмите кнопку Run All, для запуска всех тестов. После нажатия кнопки начнется последовательное выполнение тестов, по мере их выполнения для тестов будет указываться время их выполнения, а также успешность их выполнения. Если какой-либо из тестов будет провален, то тест будет отмечен красным цветом, и для него будет выведено сообщение об ошибке из метода Assert.

Помимо запуска всех тестов есть возможность запустить конкретный выбранный тест, запустить только проваленные с прошлого запуска теста, запустить тесты запущенные в прошлый раз, а также запустить тесты в режиме отладки (Debug Last Run). Запуск тестов в режиме отладки делает доступными точки останова внутри кода тестов и тестируемой функциональности. Поставьте точку останова внутри теста и запустите тест в режиме отладки. Режим отладки полезен в тех случаях, когда причина возникновения ошибки и провала теста неочевидна, и необходимо отслеживать поведение методов шаг за шагом.

Негативные тесты в NUnit. Для написания негативных тестов в NUnit используется метод Throws класса Assert. Напишем негативный тест для сеттера Surname, определяющего генерацию исключения:

```
[Test(Description = "Присвоение неправильной фамилии больше 40 символов")]
public void TestSurnameSet_Longer40Symbols()
{
    var wrongSurname = "Смирнов-Смирнов-Смирнов-Смирнов-Смирнов-Смирнов";
    var contact = new Contact();

    Assert.Throws<ArgumentException>(
        () => { contact.Surname = wrongSurname; },
        "Должно возникать исключение, если фамилия длиннее 40 символов");
}
```

Throws это шаблонный метод, при вызове которого необходимо указать:

- 1) Тип ожидаемого исключения в угловых скобках перед передачей аргументов.
- 2) Код, в котором ожидается генерация исключения, заключенный в анонимный метод () => { тестируемый код }, в качестве первого аргумента метода.
- 3) Текст сообщения об ошибке.

Таким образом, если в ходе выполнения теста, заключенный в метод Throws код не сгенерирует исключение указанного типа, то тест будет считаться проваленным. Запись теста с использованием метода Throws более лаконична и понятна, чем использование блоков try-catch, однако необходимо запомнить конструкцию написания негативных тестов.

Атрибут TestCase [11]. Как правило, большинство юнит-тестов, написанных для одного и того же метода, одинаковые и отличаются только набором начальных данных и ожидаемым значением. Например, второй негативный тест для сеттера фамилии будет выглядеть так:

```
[Test(Description = "Присвоение пустой строки в качестве фамилии")]
public void TestSurnameSet_EmptyString()
{
    var wrongSurname = string.Empty;
    var contact = new Contact();

    Assert.Throws<ArgumentException>(
        () => { contact.Surname = wrongSurname; },
        "Должно возникать исключение, если фамилия - пустая строка");
}
```

Как видно, тесты отличаются только значением wrongSurname и текстом ошибок. Фактически, это дублирование кода, а, следовательно, поддерживать дублирующийся код будет сложнее. Так, в случае изменения реализации класса Contact нам придется исправлять больше кода тестов. Для избавления от дублирования в библиотеке NUnit есть атрибут TestCase. Данный атрибут позволяет использовать один метод теста для разных исходных данных. Перепишем тесты сеттера Surname с использованием данного атрибута:

```
[TestCase("", "Должно возникать исключение, если фамилия - пустая строка",
    TestName = "Присвоение пустой строки в качестве фамилии")]
[TestCase("Смирнов-Смирнов-Смирнов-Смирнов-Смирнов-Смирнов", "Должно возникать исключение, если фамилия длиннее 40 символов",
    TestName = "Присвоение неправильной фамилии больше 40 символов")]
public void TestSurnameSet_ArgumentException(string wrongSurname, string message)
{
    var contact = new Contact();

    Assert.Throws<ArgumentException>(
        () => { contact.Surname = wrongSurname; },
        message);
}
```

Первое, что стоит отметить, что теперь у теста появились входные аргументы wrongSurname и message. Так как их значения отличаются в разных тестах, теперь значения этих переменных будут задаваться извне.

Значения для данных аргументов будут приходить из атрибута [TestCase]. [TestCase] описывает так называемый тестовый случай. Тестовый случай – это набор исходных данных и ожидаемого значения для теста. У одного теста может быть любое количество тестовых случаев. В данном примере, у теста TestSurnameSet_ArgumentException() два тестовых случая.

Внутри скобок атрибута [TestCase] указываются значения для входных аргументов теста в порядке их объявления. Их тип данных должен совпадать со значениями объявленных аргументов. В данном примере, в метод передаются исходная фамилия для присвоения и текст ошибки, но также в тесты можно передавать ожидаемое значение expected (для позитивных тестов) или тип исключения, который мы ожидаем поймать (для негативных тестов).

Последним в атрибуте [TestCase] задается свойство TestName, благодаря которому тестам можно дать осмысленные имена, отображающиеся в панели TestExplorer. Значения всех TestName должны отличаться друг от друга, в противном случае на панели TestExplorer будут отображаться не все тестовые случаи.

Атрибуты SetUp и TearDown [15]. В teste может быть подготовительная часть кода (setup), и часть кода, которую необходимо выполнить после проверки Assert (teardown), например, разорвать соединение с БД. Зачастую подготовительная часть кода одинаковая для тестов. Например, в наших тестах всегда есть код, создающий экземпляр класса Contact:

```
var contact = new Contact();
```

В данном случае, это только одна строка, но на практике это может быть несколько десятков строк.

Чтобы не дублировать этот код, его можно вынести в отдельный метод и пометить его атрибутом [Setup]:

```
[TestFixture]
public class ContactTest
{
    private Contact _contact;

    [SetUp]
    public void InitContact()
    {
        _contact = new Contact();
    }

    [TestCase("", "Должно возникать исключение, если фамилия - пустая строка",
              TestName = "Присвоение пустой строки в качестве фамилии")]
    [TestCase("Смирнов-Смирнов-Смирнов-Смирнов-Смирнов", "Должно возникать исключение, если фамилия длиннее 40 символов",
              TestName = "Присвоение неправильной фамилии больше 40 символов")]
    public void TestSurnameSet_ArgumentException(string wrongSurname, string message)
    {
        //var contact = new Contact(); // Теперь эта строка не нужна
        Assert.Throws<ArgumentException>(
            () => { _contact.Surname = wrongSurname; },
            message);
    }
}
```

Метод, помеченный атрибутом [SetUp], будет выполняться перед каждым запуском каждого теста. Таким образом, объект _contact будет пересоздаваться для каждого тестового случая. Аналогично, метод, помеченный атрибутом [Teardown] будет выполняться по завершении тестов.

Также обратите внимание, что теперь для передачи контакта из метода InitContact() в тесты, мы создали поле _contact. Это вполне допустимый приём – так как ContactTest является классом, мы можем создавать в нём поля, константы, свойства, вспомогательные закрытые методы и т.д. То есть, если какие-либо механизмы ООП или C# могут упростить написание ваших тестов – вы можете их использовать.

Атрибут Ignore [15]. При долгосрочной поддержке приложений часто меняется функциональность, написанная в предыдущих версиях программы. Программа постоянно меняется, дописываются новые функции, старые функции переписываются. Изменение классов бизнес-логики может сделать часть юнит-тестов неактуальной.

Может возникнуть ситуация, когда часть бизнес-логики была изменена, но нет времени для актуализации юнит-тестов. То есть, часть юнит-тестов завершаются провалом, но у нас нет времени для их исправления.

Самое простое решение, которое приходит на ум, это закомментировать проваливаемые тесты до тех пор, пока не появится время для их исправления. Но, как показывает практика, чаще всего, эти тесты забываются, и когда по прошествию некоторого времени закомментированные тесты увидит другой разработчик, он их просто удалит.

Чтобы не терять неактуальные тесты, можно использовать атрибут [Ignore]. Он указывается перед тестом или в качестве свойства в атрибуте [TestCase]. Тест или тестовый случай, помеченный данным атрибутом, будет игнорироваться при запуске тестов и будет помечаться специальным цветом:

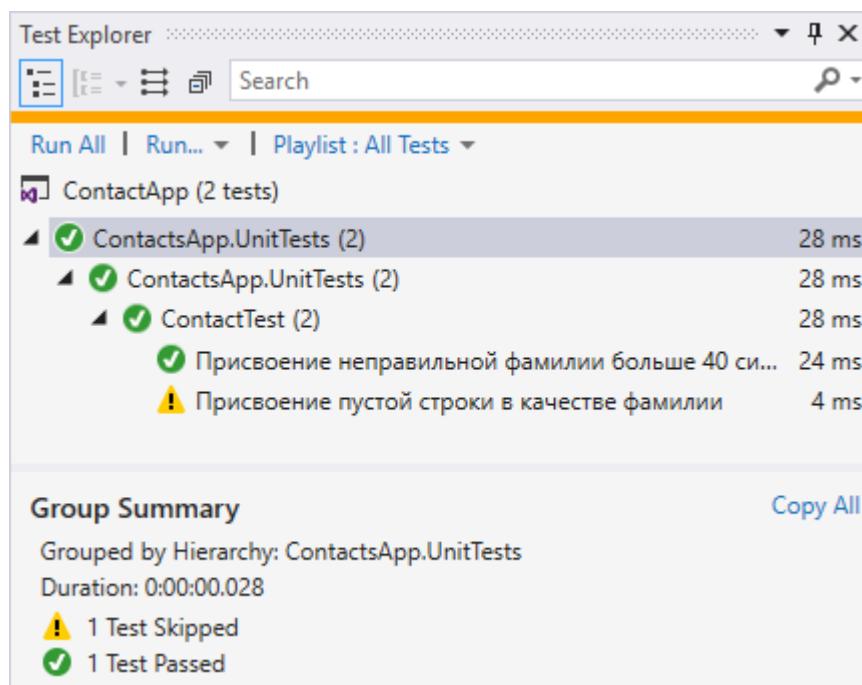


Рисунок 70 – Проигнорированные тесты будут отмечены восклицательным знаком в дереве тестов, а общее количество будет отображено на нижней информационной панели

Главное отличие использования [Ignore] от комментирования кода юнит-тестов заключается в том, что все проигнорированные тесты продолжают отображаться на панели

TestExplorer. Для команды разработчиков это будет постоянным напоминанием, что часть тестов неактуальна и их надо исправить. При комментировании кода юнит-тестов они на панели TestExplorer отображаться не будут, и через некоторое время разработчики о них забудут.

Использование атрибута [Ignore] внутри [TestCase] выглядит следующим образом:

```
[TestCase("", "Должно возникать исключение, если фамилия - пустая строка",
    Ignore = "Тест неактуален",
    TestName = "Присвоение пустой строки в качестве фамилии")]
```

В качестве строки для Ignore указывается причина игнорирования теста.

Другие возможности NUnit. Описанных атрибутов достаточно для того, чтобы автоматизировать тестирование бизнес-логики в своём приложении. Однако в библиотеке NUnit есть много других возможностей, изучение которых может повысить качество тестирования или упростить поддержку тестов.

В первую очередь хотелось бы отметить другие методы класса Assert. Класс Assert также содержит методы для сравнения коллекций (массивов, списков, словарей), определения принадлежности объекта конкретному типу данных, сравнения вещественных чисел с заданной точностью, тестирования асинхронных методов, составления сложных условий проверки (multiple assertions).

Также есть множество дополнительных атрибутов [15]. Вот перечень некоторых из них:

- [Category] – позволяет указать категорию юнит-тестов, например, для раздельного запуска различных тестов на тестовом сервере.
- [Combinatorial] – позволяет перебирать все возможные комбинации входных аргументов из заранее заданного массива значений. Благодаря этому, в отдельных случаях позволяет заменить множество TestCase одной строкой Combinatorial. Также имеет аналог PairWise, составляющий «умную» выборку комбинаций.
- [TestCaseSource] – позволяет указывать источник тестовых случаев, например, из поля класса или базы данных. Хранение тестовых случаев в отдельном поле или базе данных может упростить поддержку, а также чтение кода юнит-тестов.

Более полно тема качественного написания юнит-тестов раскрыта в [4, 5]. В частности, рассмотрены вопросы тестирования базовых абстрактных классов, защищенных методов, а также тестирования через mock-объекты (или фальшивые объекты) и методология Test-Driven-Development (разработка через тестирование). Развитие навыков написания юнит-тестов также способствует более грамотному планированию архитектур приложений и классов, удобных в использовании.

2.4.5 Задание

1. Создать проект юнит-тестов:

- 1.1. Создайте проект библиотеки классов с названием <имя проекта логики>.UnitTests.
- 1.2. Через встроенную службу пакетов NuGet добавьте ссылку на библиотеку NUnit в проект юнит-тестов. Перекомпилируйте решение для завершения добавления библиотеки.

1.3. Для каждого класса проекта логики создайте соответствующий класс в проекте юнит-тестов. Название класса юнит-тестов должно повторять название класса, который он будет тестировать, с добавлением приставки Test. Например, если класс логики называется Project, то класс с юнит-тестами для него будет называться ProjectTest. Такое именование упрощает навигацию в проекте юнит-тестов.

1.4. Сделайте коммит с соответствующим комментарием.

2. Написать тесты для классов логики проекта:

2.1. Рассчитайте цикломатическую сложность проекта логики. Для этого сначала рассчитайте цикломатическую сложность **каждого** метода во **всех** классах логики, затем просуммируйте цикломатическую сложность для каждого класса, а затем и для всего проекта. Представьте цикломатическую сложность проекта в виде дерева или на UML-диаграмме классов, например:

Проект NoteApp (цикл. сложность проекта = 32)

1. Класс Note (цикл. сложность класса = 22)

Свойство Title (цикл. сложность = 4)

Свойство CreatedDate (цикл. сложность = 4)

Свойство ModifiedDate (цикл. сложность = 6)

...

2. Класс Project (цикл. сложность класса = 2)

...

3. Класс ProjectManager (цикл. сложность класса = 8)

...

(цифры приведены для примера и могут отличаться от реальных значений)

2.2. На основе цикломатической сложности определите набор необходимых тестовых случаев для полного покрытия классов бизнес-логики тестами.

2.3. Напишите юнит-тесты для тестирования классов бизнес-логики. Бизнес-логика должна быть полностью покрыта юнит-тестами. Если цикломатическая сложность бизнес-логики превышает 50 тестовых случаев, ограничьтесь написанием только 50 тестов.

2.4. Если юнит-тесты работают верно, сделайте коммит с соответствующим комментарием.

3. Написать тесты для сериализации и десериализации:

3.1. Напишите юнит-тесты для тестирования сериализации и десериализации.

3.2. Если юнит-тесты написаны верно и работают, сделайте коммит в локальный репозиторий с соответствующим комментарием. Синхронизируйте локальный репозиторий с удаленным репозиторием.

4. Проверить, что задание лабораторной работы выполнено верно:

4.1. Зайдите в свой аккаунт GitHub и убедитесь, что синхронизация прошла успешно.

4.2. Оформите рассчитанное дерево цикломатической сложности проекта логики в документе Microsoft Word. Рассчитайте степень покрытия кода тестами для всего проекта (в виде процентов или дроби), укажите время выполнения всех юнит-тестов.

4.3. Если все задания выполнены верно, ответьте на вопросы для самоподготовки и переходите к защите лабораторной работы. На защите продемонстрируйте работу юнит-тестов, структуру файлов в удаленном репозитории и историю коммитов.

2.4.6 Вопросы для самоподготовки

1. Какие виды тестирования можно выделить в процессе разработки ПО? Когда выполняется каждый из видов?

2. Что такое юнит-тестирование? Когда оно проводится? Кем? Что является объектом юнит-тестирования?
3. Когда выполняется запуск юнит-тестов? Зачем делить юнит-тесты на категории?
4. Как определить количество требуемых тестовых случаев для метода? Для класса? Как рассчитать степень покрытия кода тестами?
5. Что такое позитивный и негативный тестовые случаи? Почему важно писать негативные тестовые случаи?
6. Каково назначение следующих атрибутов библиотеки NUnit: TestFixture, Test, TestCase, Category, Ignore, Setup, Teardown? Напишите код, демонстрирующий работу с этими атрибутами.
7. Назначение класса Assert. Что делают методы AreEqual(), Throws<>(), Pass(), Fail()? Напишите код, демонстрирующий работу с этими методами.
8. Какие преимущества помимо контроля качества ПО, даёт юнит-тестирование? Какие недостатки юнит-тестирования вы можете назвать?

2.4.7 Содержание отчета

Отчет по лабораторной работе должен содержать:

- Титульный лист, оформленный согласно требованиям ОС ТУСУР [17].
- Дерево цикломатической сложности методов и классов проекта логики. В дереве указать, количество написанных юнит-тестов.
- Общая цикломатическая сложность проекта логики и степень покрытия кода тестами.
- Исходный код любого написанного юнит-теста в качестве примера использования библиотеки NUnit. Работу кода пояснить сопроводительным текстом, прокомментировать используемые атрибуты и методы класса Assert.
- Время выполнения всех тестовых случаев.
- Текущая история коммитов ветки develop (допустимо в виде снимка экрана соответствующей страницы GitHub или VisualStudio).

Отчёт сдается с указанием даты защиты лабораторной и подписью студента.

2.4.8 Список источников

Юнит-тестирование

1. Калбертон Р., Браун К., Кобб Г. Быстрое тестирование. - М.: Вильямс, 2002. – 384 с.
2. Тестирование. Фундаментальная теория [Электронный ресурс]. / habr.com: портал корпоративных блогов, блог пользователя Gennadii_M. — URL: <https://habr.com/post/279535/> (дата обращения 27.08.2018).
3. Что такое качество кода и зачем его измерять [Электронный ресурс]. / habr.com: портал корпоративных блогов, блог пользователя WingeD. — URL: <https://habr.com/post/205342/> (дата обращения 27.08.2018).
4. Osherove R. The Art of Unit Testing. 2nd Edition with examples in C# / Osherove R., ed. Miller N. – NY.: Manning Publication, 2014. – 294 p.
5. Ошероув Р. Искусство автономного тестирования с примерами на C#. 2-е издание / пер. с англ. Слинкин А.А. – М.: ДМК Пресс, 2014. – 360 с.

6. [Видео] Unit тестирование в C#/ ITVDN: IT-видеокурсы по программированию. — URL: <https://itvdn.com/ru/video/unit-testing-csharp> (дата обращения 27.08.2018).
7. Цикломатическая сложность [Электронный ресурс]. / Википедия: онлайн-энциклопедия. — URL: https://ru.wikipedia.org/wiki/Цикломатическая_сложность (дата обращения 27.08.2018).
8. Шилдт Г. С# 4.0: полное руководство.: пер. с анг. Бернштейн И.В. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.: ил. (глава 17)

Библиотека юнит-тестирования NUnit

9. NUnit: официальная документация [Электронный ресурс]. / GitHub.com: сервис размещения удаленных репозиториев, репозиторий пользователя nunit. — URL: <https://github.com/nunit/docs/wiki/NUnit-Documentation> (дата обращения 27.08.2018).
10. Атрибут TestFixture библиотеки NUnit: официальная документация [Электронный ресурс]. / GitHub.com: сервис размещения удаленных репозиториев, репозиторий пользователя nunit. — URL: <https://github.com/nunit/docs/wiki/TestFixture-Attribute> (дата обращения 27.08.2018).
11. Атрибут TestCase библиотеки NUnit: официальная документация [Электронный ресурс]. / GitHub.com: сервис размещения удаленных репозиториев, репозиторий пользователя nunit. — URL: <https://github.com/nunit/docs/wiki/TestCase-Attribute> (дата обращения 27.08.2018).
12. Класс Assert библиотеки NUnit: официальная документация [Электронный ресурс]. / GitHub.com: сервис размещения удаленных репозиториев, репозиторий пользователя nunit. — URL: <https://github.com/nunit/docs/wiki/Assertions> (дата обращения 27.08.2018).
13. Ограничение ThrowsConstraint библиотеки NUnit: официальная документация [Электронный ресурс]. / GitHub.com: сервис размещения удаленных репозиториев, репозиторий пользователя nunit. — URL: <https://github.com/nunit/docs/wiki/ThrowsConstraint> (дата обращения 27.08.2018).
14. Список всех атрибутов библиотеки NUnit: официальная документация [Электронный ресурс]. / GitHub.com: сервис размещения удаленных репозиториев, репозиторий пользователя nunit. — URL: <https://github.com/nunit/docs/wiki/Attributes> (дата обращения 27.08.2018).

Подключение сторонних библиотек

15. Менеджер подключения пакетов NuGet (NuGet Package Manager UI) [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/en-us/nuget/tools/package-manager-ui> (дата обращения 27.08.2018).

Требования к оформлению отчета

16. ОС ТУСУР 01-2013. Работы студенческие по направления подготовки и специальностям технического профиля. Общие требования и правила оформления. – Томск: ТУСУР, 2013. – 57с. – URL: https://storage.tusur.ru/files/40668/rules_tech_01-2013.pdf (дата обращения 27.08.2018)

2.5 Функциональное расширение и релиз проекта

Цель работы: провести вторую итерацию разработки приложения, расширив её дополнительными функциональными возможностями, а также получить умения сборки установочных пакетов.

Задачи:

- 1) Изучить процессы сборки установочного пакета и сдачи проекта по окончанию работ.
- 2) Разработать дополнительную функциональность в проекте согласно техническому заданию.
- 3) Собрать установочный пакет приложения с использованием программы InnoSetup.
- 4) Провести внутреннее приёмочное тестирование установленного приложения.
- 5) Провести приёмочное тестирование с руководителем для сдачи завершенной программы.

2.5.1 Сборка установочного пакета

В зависимости от целевой платформы (веб, мобильные, десктоп) и продукта, способы доставки приложения пользователю будут отличаться. Так, в случае веб-сервисов нет необходимости собирать установщик приложения – достаточно развернуть скомпилированный сервис на рабочем сервере с выходом в Интернет или локальную сеть, через который будет предоставляться доступ к сервису пользователям. В случае десктоп-приложений необходимо создать установочный пакет – исполняемый файл, содержащий в себе архив вашего приложения, и выполняющий установку приложения на компьютер пользователя.

Установочный пакет — это не только архив с вашей программой, он может выполнять ряд дополнительных задач:

- 1) контролировать целевую папку установки приложения;
- 2) зарегистрировать приложение в реестре Windows;
- 3) выполнить проверку наличия у пользователя лицензионного ключа;
- 4) создать ярлыки и подпапки в меню «Пуск» для вашего приложения;
- 5) установить ассоциации для расширений файлов;
- 6) предоставить пользователю информацию о лицензионном соглашении;
- 7) и др.

В установочный пакет помещаются файлы, необходимые для работы вашего приложения. Также могут помещаться иконки, руководства пользователя, вспомогательные файлы примеров и т.д. – любые файлы, которые вы считаете нужными для работы приложения у пользователя. **Разумеется, в установочный пакет не стоит помещать файлы исходного кода** – только скомпилированное приложение.

Например, в папке вашего решения NoteApp после успешной компиляции будет располагаться папка bin, содержащая подпапки Debug и Release. Они содержат версии скомпилированного приложения. В такую именно подпапку будет скомпилирована программа, можно управлять в среде Visual Studio через стандартную панель под главным меню:



С помощью выпадающего списка вы можете выбрать режим Debug или Release. Это называется конфигурация сборки. Конфигурации Debug и Release отличаются не только центральной папкой для компиляции – так при компиляции в конфигурации Debug, среда разработки разместит в компилируемой программе дополнительный код, который необходим для отладки. Например, в конфигурации Debug вам будут доступны точки останова (breakpoints), профилировщики (контролируют нагрузку на процессор и оперативную память в процессе работы программы).

Конечно, дополнительный код в конфигурации Debug приведет к более большому размеру скомпилированных файлов, а также к более медленной работе программы (скорость работы может ухудшиться на 30% и более). В конфигурации Release инструменты отладки работать не будут, однако и дополнительный код не будет замедлять работу приложения.

Таким образом, во время разработки приложения необходимо компилировать программу в конфигурации Debug, однако для сборки установочного пакета и поставки пользователю необходимо компилировать приложения в конфигурации Release.

Скомпилировав программу в Release, в папке решения (например, NoteApp\bin\Release) появятся файлы скомпилированного приложения. Если ваше решение разбито на множество взаимодействующих проектов, то каждому проекту в этой папке будут соответствовать свои файлы. Рассмотрим эти файлы:

NoteAppUI.exe	- скомпилированный исполняемый файл проекта NoteAppUI
NoteApp.dll	- скомпилированная библиотека проекта бизнес-логики
Newtonsoft.json.dll	- библиотека, необходимая для работы сериализации в бизнес-логике
NoteAppUI.pdb	- побочный файл компиляции, для работы приложения не нужен
NoteApp.pdb	- побочный файл компиляции, для работы приложения не нужен
NoteAppUI.exe.config	- побочный файл конфигурации компиляции, для работы приложения не нужен

Таким образом, для работы приложения в установочный пакет необходимо поместить файлы с расширением *.exe и *.dll. Остальные файлы (*.pdb, *.config, *.manifest, *.xml и другие возможные файлы) для работы приложения не нужны.

Программа сборки установочных пакетов InnoSetup. Программа InnoSetup позволяет создавать сценарии для сборки установочных пакетов и компилировать установщики. Сценарии сохраняются в формате *.iss (inno setup script), и имеют специальный синтаксис:

```

#define MyAppURL "http://www.example.com/"
#define MyAppExeName "MyProg.exe"

[Setup]
; NOTE: The value of AppId uniquely identifies this application.
; Do not use the same AppId value in installers for other applications.
; (To generate a new GUID, click Tools | Generate GUID inside the IDE.)
AppId={{E1C20560-50A8-4E85-9578-8E4581EDCD15}
AppName=#MyAppName
AppVersion=#MyAppVersion
;AppVerName=#MyAppName (#MyAppVersion)
AppPublisher=#MyAppPublisher
AppPublisherURL=#MyAppURL
AppSupportURL=#MyAppURL
AppUpdatesURL=#MyAppURL
DefaultDirName={pf}\{#MyAppName}
DisableProgramGroupPage=yes
OutputBaseFilename=setup
Compression=lzma
SolidCompression=yes

[Languages]
Name: "english"; MessagesFile: "compiler:Default.isl"

[Tasks]
Name: "desktopicon"; Description: "{cm>CreateDesktopIcon}"; GroupDescription: "{cm}Custom"
; NOTE: Don't use "Flags: ignoreversion" on any shared system files

[Icons]
Name: "{commonprograms}\{#MyAppName}"; Filename: "{app}\{#MyAppExeName}"
Name: "{commondesktop}\{#MyAppName}"; Filename: "{app}\{#MyAppExeName}"; Tasks: desktopicon

[Run]

```

Сценарий разделен на область определения переменных, секции с инструкциями, а также может содержать код на языке Delphi для создания дополнительных, нестандартных окон в установщике.

В начале сценария можно объявить различные константы с помощью директивы `#define`. После директивы `#define` сначала указывается имя константы и через пробел её значение. В дальнейшем объявленную константу можно использовать в сценарии – например, для подстановки имени программы, версии программы, адреса почты или веб-сайта, пути программы и др. При дальнейшей поддержке программы, в случае изменения, например, адреса почты достаточно будет исправить только значение переменной вместо исправления значений во всём сценарии.

Секция [Setup] задаёт основные настройки установщика:

`AppId` – уникальный идентификационный номер приложения для регистрации в реестре Windows.

`AppName` – название приложения, в том числе и для регистрации в реестре.

`AppVersion` – версия приложения.

`Compression` – алгоритм сжатия (архивирования) файлов, помещаемых в установщик.

`OutputBaseFilename` – имя выходного файла – файла установщика.

и др. С полным перечнем параметров и допустимых значениях можно ознакомиться в руководстве программы.

Секция [Tasks] позволяет создавать дополнительные окна в установщике с опциями, которые сможет установить пользователь. Например, строка:

```
Name: "desktopicon"; Description: "{cm>CreateDesktopIcon}"; GroupDescription: "{cm:AdditionalIcons}"; Flags: unchecked
```

создаст опцию с именем desktopicon. В дальнейшем имя desktopicon можно будет указывать в других секциях сценария. Для пользователя при запуске установщика в одном из окон будет предложена опция «Установить ярлык программы на рабочий стол» (данное описание является стандартным и берется из стандартной переменной {cm>CreateDesktopIcon} для параметра Description). Однако при желании, вы можете задать другой текст, указав его в кавычках после Description. Значение unchecked для параметра Flags указывает, что по умолчанию данная опция в установщика будет отключена.

Аналогично можно создавать другие опции в установщике, отмечаемые пользователем с помощью галочек – например, настроить ассоциации файлов.

Секция [Files] описывает файлы, которые необходимо поместить в установщик. Для вашей программы вы должны будете указать папку, в которой лежит ваша скомпилированная программа. Файлы можно указывать как по одному, так множество файлов с помощью стандартных масок имен файлов. Например, путь "C:*.*.exe" для параметра Source скопирует в установщик все файлы с диска C:\ пользователя, имеющих расширение *.exe. Путь "C:\install\sample*.dll" скопирует все файлы в папке C:\install\, начинающихся со слова sample и имеющих расширение *.dll. Разумеется, главным ограничением является существование указанных файлов – в противном случае компиляция установщика выдаст ошибку о том, что такой файл или файлы не существуют.

Программа работает как с абсолютными путями, так и с относительными. Например, путь "*.*.exe" скопирует все файлы с расширением *.exe из папки, в которой располагается сам сценарий сборки установщика *.iss.

Если же использовать символы "..\" в относительных путях, то можно подняться на папку выше исходной. Например, команда:

```
Source: "..\MyProg.exe"; DestDir: "{app}"; Flags: ignoreversion
```

Во время компиляции установщика поднимется на одну папку выше текущего сценария сборки установщика, и будет искать файл MyProg.exe. Если же набрать команду:

```
Source: "..\install\MyProg.exe"; DestDir: "{app}"; Flags: ignoreversion
```

то при компиляции установщика InnoSetup поднимется на одну папку выше, зайдет в соседнюю папку install и будет искать файл MyProg.exe в этой папке. С помощью "..\" можно подняться на любое количество папок вверх.

Важное правило при разработке установщиков (как и в программировании в целом) – нельзя использовать абсолютные пути! Абсолютные пути означают, что путь к файлу привязан к вашей текущей структуре папок. Например, если вы укажете путь до файла "C:\install\program.exe", то при запуске вашего сценария на другом компьютере, может возникнуть ошибка – так как на другом компьютере может не существовать папки install, также как и диска C:. Поэтому все пути, которые вы указываете в сценарии, должны быть относительными.

Полное описание секций сценариев InnoSetup можно найти в документации программы или на сайте разработчика. Стоит отметить, что с помощью InnoSetup можно создавать достаточно сложные установочные пакеты благодаря большому числу встроенных функций. Полный их перечень, а также дополнительных встроенных переменных можно найти в документации. Также в папке программы можно найти примеры готовых сценариев сборки установщика.

2.5.2 Автоматизация сборки установочного пакета

InnoSetup позволяет создать сценарий для сборки установщика, однако запуск самого сценария, а также возможная подготовка файлов для установщика, может потребовать дополнительных действий от разработчика, которые придется выполнять вручную. Средства Visual Studio позволяют автоматизировать сборку установщика с помощью автоматического запуска сценария InnoSetup.

Прежде всего, сценарий сборки установщика необходимо поместить в папку решения. Еще лучшим решением будет создать для всех файлов установки отдельную папку в папке решения. Например, в папке решения NoteApp создать подпапку InstallScripts и поместить в неё сценарий *.iss. Это решает две проблемы:

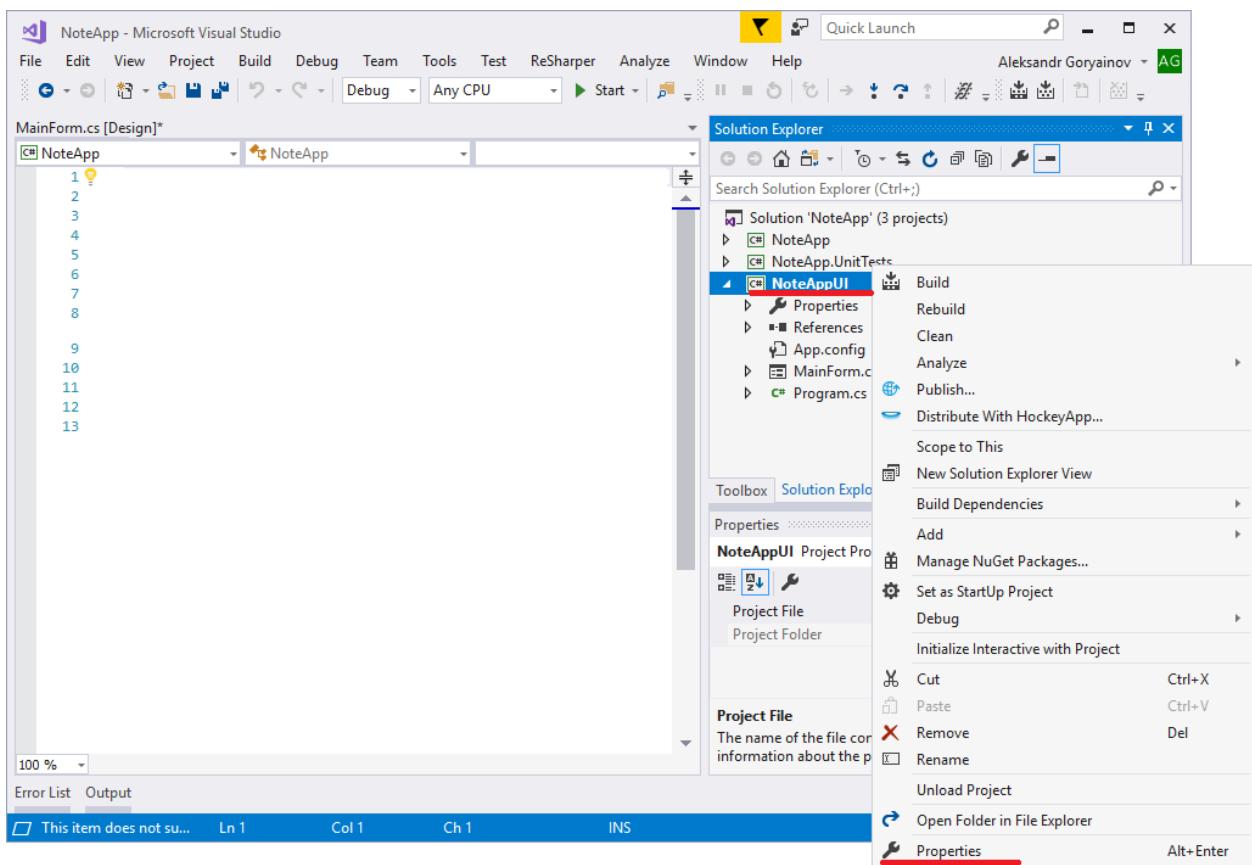
- 1) Так как файл сценария находится в папке решения, относительные пути до файлов компиляции теперь упростятся.
- 2) Файлы сборки установщика теперь будут находиться под версионным контролем, а значит другой разработчик в вашей команде сможет скачать ваш репозиторий и собрать установщик на своём компьютере.

Файлы *.iss такие же текстовые файлы, как и файлы исходного кода, а значит по мере их исправления (по мере развития проекта) можно будет сохранять разные версии сценариев в версионном контроле.

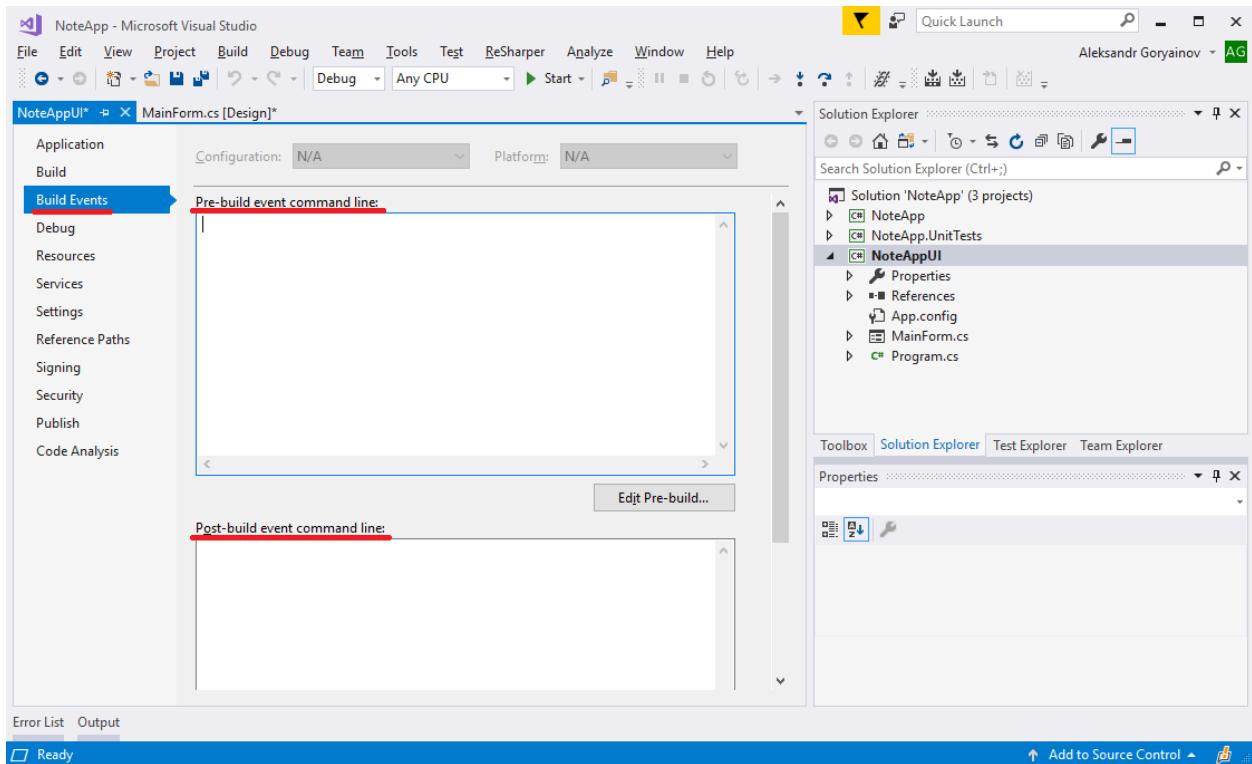
Как уже было сказано выше, после копирования файла сценария в папку решения, необходимо исправить относительные пути в сценарии – относительный путь должен сначала подняться в папку решения, а затем зайти в нужную папку bin\, куда Visual Studio компилирует программу. Этого будет достаточно для программ с нечастыми релизами.

Однако более качественным решением будет запуск сценария сборки установщика прямо в среде Visual Studio. Для этого необходимо сделать дополнительные настройки в вашем решении.

Откройте ваше решение в Visual Studio. Откройте настройки проекта пользовательского интерфейса (например, NoteAppUI). Это можно сделать через контекстное меню по нажатию на имя проекта в дереве решения:



После выбора настроек, перед вами появится следующее окно, где необходимо перейти во вкладку Build Events в меню слева:



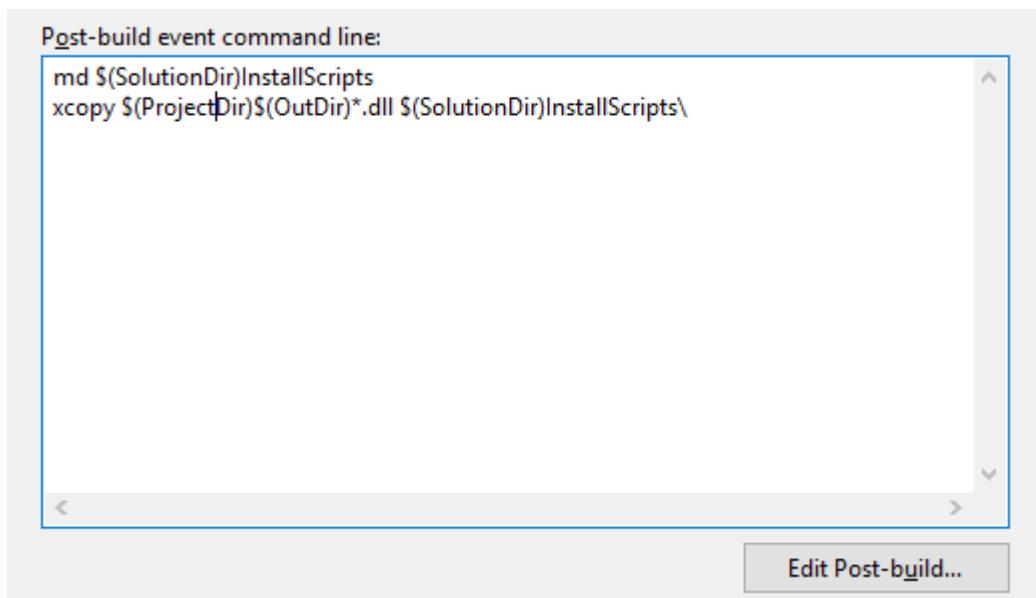
Помимо прочих настроек проекта, в данном окне можно задать команды, которые будут выполнять перед компиляцией (Pre-build events) и после успешной компиляции (Post-build events).

В качестве команд можно использовать любые команды командной строки Windows, а также запускать bat-файлы.

Например, используя команду xcopy можно скопировать файлы из одной папки в другую, с помощью команды del – удалить ненужные файлы по указанному пути (например файлы *.pdb, чтобы они не попали в установщик), команда md – создать подпапку. Подробнее о командной строке и синтаксисе команд можно ознакомиться в [9].

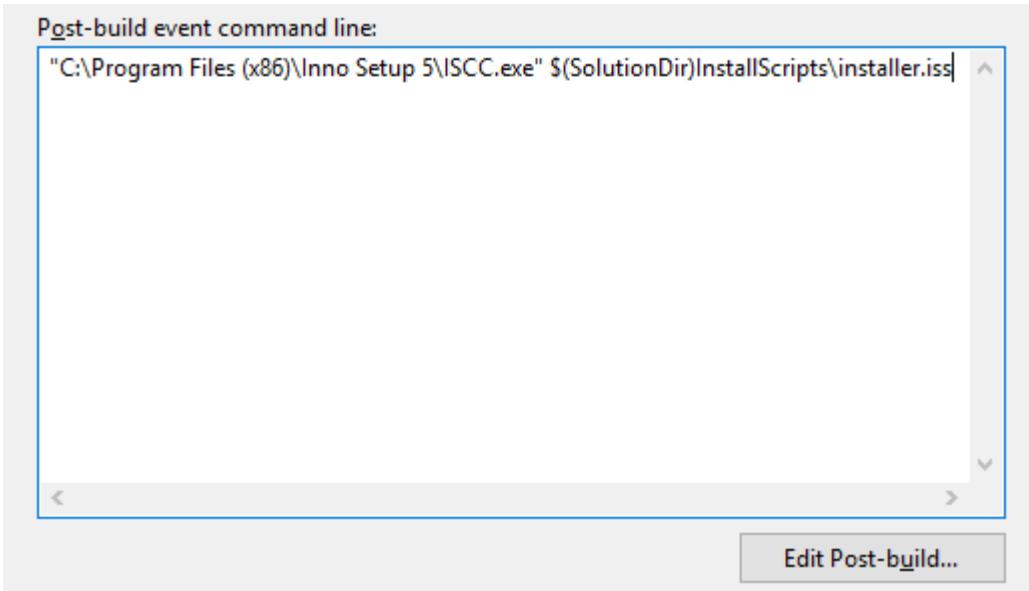
Так как большинство команд командной строки (или оболочки Shell) работают с именами файлов или путями, в Visual Studio есть специальные макросы, которые позволяют получить, например, папку текущего решения, или папку компиляции программы и другие пути. С перечнем макросов можно ознакомиться, нажав кнопку Edit Post-build в настройках проекта, или на сайте Microsoft [8]. Например, макрос \$(OutDir) возвращает путь к выходному каталогу – папку, в которую компилируется данный проект.

Таким образом, прописав в качестве событий успешного построения следующие команды:



мы добьёмся того, что после успешной компиляции решения, сначала команда md создаст каталог InstallScripts в папке решения (путь до папки решения указывает макрос \$(SolutionDir)), а затем команда xcopy скопирует из папки компиляции все файлы с расширением *.dll и переместит их в созданную папку InstallScripts. Путь до папки компиляции указывается с помощью двух макросов: \$(ProjectDir) указывает путь до папки проекта, а макрос \$(OutDir) добавляет к пути выходные папки bin\Debug или bin\Release (в зависимости от конфигурации сборки).

Обратите внимание, что данные команды будут отрабатывать после *каждой* успешной компиляции. Если вы допустите ошибку в команде (например, зададите несуществующий путь для копирования файлов), тогда в результате компиляции на панели «Error List» (Список ошибок) будет показан текст ошибки, что команды оболочки Shell не удалось выполнить.



то после компиляции будет автоматически запускаться компиляция установочного пакета согласно сценарию `installer.iss`. При этом в качестве «`C:\Program Files (x86)\Inno Setup 5\ISCC.exe`» надо задать путь до исполняемого файла программы InnoSetup (в зависимости от папки, куда вы установили программу на своём компьютере), а через пробел надо задать путь до вашего сценария сборки установщика.

Зная команды Shell вы можете создавать события построения любой сложности. Это касается не только сборки установщика, но и других действий на ваше усмотрение. Например, можно написать команды, которые будут отправлять уведомление об успешной компиляции проекта вам на почту или отправлять собранный установщик заказчику или тестировщику (хотя эти задачи обычно решаются другими инструментами).

Таким образом, для автоматизации сборки установщика средствами Visual Studio, необходимо написать следующую последовательность команд:

- 1) Создать подпапку `Release` в папке `InstallScripts`.
- 2) Создать подпапку `Installers` в папке `InstallScripts`.
- 3) Скопировать в созданную подпапку `Release` все файлы скомпилированной программы (`*.exe` и `*.dll`, остальные файлы можно удалить).
- 4) Запустить InnoSetup для сборки установщика.
- 5) Поместить собранный установщик в папку `Installers`
- 6) Удалить подпапку `Release` (на случай, чтобы при следующей компиляции в установщик не попали старые результаты компиляции).

При этом, разумеется, нужно будет подправить сам сценарий сборки установщика, чтобы он брал файлы программы из папки `Release`, а выходной файл установщика помещал в подпапку `Installers`. Кроме того, в файл `gitignore` версионного контроля необходимо добавить исключение для папки `Installers`, чтобы эта папка, а также все созданные в ней установщики не попадали в коммиты под версионный контроль. Дополнительно вы можете подправить сценарий `*.iss` таким образом, чтобы к имени установщика добавлялась текущая дата (год, месяц, день), чтобы установщики разных версий программы можно было отличать друг от друга.

Предложенный подход автоматизации средствами Visual Studio подходит для самостоятельной разработки небольших приложений. В коммерческой разработке автоматизация сборки выполняется средствами так называемого сервера непрерывной интеграции (Continuous Integration). Однако суть автоматизации остается прежней – для автоматизации сборки необходимо будет прописывать сценарии с использованием команд Shell, только не в настройках проекта через Visual Studio, а в отдельном приложении.

2.5.3 Приёмочное тестирование

Приёмочное тестирование – тестирование, выполняемое при представителе заказчика для сдачи готового проекта перед внедрением в эксплуатацию. По завершению приёмочного тестирования подписывается акт приёма-передачи (в случае, если заказчик подтверждает, что программа работает верно без замечаний), или акт замечаний (в случае, если у заказчика есть замечания по программе). На приёмочном тестировании демонстрируется работа всей системы, а не отдельным компонентов – это также демонстрирует, что сами компоненты программы или аппаратного комплекса правильно взаимодействуют между собой.

В случае строго регламентированного процесса разработки (например, для систем с повышенными требованиями к надежности), процедура приёмочного тестирования может быть описана и утверждена в техническом задании. Однако в большинстве случаев к данной процедуре относятся более неформально – на самом тестировании перед заказчиком выполняют ряд простых или ключевых действий в программе, дают заказчику самостоятельно поработать с программой. Если после нажатия на случайные кнопки программа не выдала ошибки – приём проекта состоялся. При этом акт приёма-передачи или акт замечаний также зачастую составляются неформально или передаются устно.

Разумеется, перед демонстрацией продукта заказчику, необходимо самостоятельно несколько раз пройти процедуру приёмочного тестирования:

- 1) Выполнить процедуру непосредственно из-под среды разработки – разработчик убеждается, что последние изменения в программе не внесли ошибок в базовую функциональность и приложение функционирует. Собирается защищенный установщик программы (возможно, с обfuscацией кода и внедрением системы лицензирования).
- 2) Защищенный установщик устанавливается на «чистую» виртуальную машину тестировщиком и полностью проверяется процедура приёмочного тестирования всей системы целиком. Если никаких ошибок в установленном из защищенного установщика приложении не обнаружено, можно приглашать представителя заказчика.
- 3) Защищенный установщик устанавливается на «чистую» виртуальную машину перед представителем заказчика и полностью выполняется процедура приёмочного тестирования. В завершении, при желании заказчика, ему дают возможность самостоятельно поработать в приложении.

В данном курсе лабораторных работ процедура приёмочного тестирования описана в техническом задании каждого варианта лабораторных работ. Успешное прохождение приёмочного тестирования перед преподавателем (играющего в этот момент роль заказчика), подводит итог разработке ПО в данном семестре и означает, что лабораторные работы выполнены правильно. После чего остается подготовить проектную документацию.

2.5.4 Задание

1. Реализовать дополнительную функциональность согласно ТЗ:

- 1.1. Ознакомьтесь с пунктом «Дополнительная функциональность» в вашем варианте заданий. В случае непонимания пунктов дополнительной функциональности или наличия вопросов по правильности их реализации, обсудите с руководителем.
- 1.2. Реализуйте пункты.
- 1.3. Проведите приёмочное тестирование программы, собранной в конфигурации «Релиз». Порядок приёмочного тестирования описан в техническом задании.
- 1.4. Убедитесь, что код оформлен согласно требованиям RSDN, а также проверьте наличие xml-комментариев.
- 1.5. Исправьте UML-диаграмму классов бизнес-логики согласно измененному коду.
- 1.6. Сделайте коммит в репозиторий с соответствующим комментарием.

2. Создать сценарий сборки установочного пакета:

- 2.1. Создайте сценарий сборки установочного пакета в программе InnoSetup на основе примера.
- 2.2. Укажите в сценарии название приложения, автора, версию. Сгенерируйте новый уникальный идентификатор приложения. Добавьте ярлык, укажите пути установки. Убедитесь, что сценарий работает – соберите установщик, установите программу, запустите программу из папки установки.
- 2.3. Создайте в локальном репозитории (в папке решения) папку InstallScripts. Поместите в неё созданный сценарий сборки установочного пакета. Таким образом, актуальный сценарий сборки установщика будет всегда рядом с исходным кодом.
- 2.4. Откройте сценарий сборки и исправьте абсолютные пути к файлам на относительные пути. Это необходимо для того, чтобы сценарий сборки работал на любом компьютере, который синхронизируется с вашим репозиторием. В противном случае, сценарий будет работать только на вашем компьютере – такие детали усложняют командную разработку приложений.
- 2.5. Убедитесь, что приложение удаляется с помощью автоматически созданного ярлыка.
- 2.6. Если сценарий исправлен и работает корректно, сделайте коммит с соответствующим комментарием. Синхронизируйте с удаленным репозиторием.

3. Провести приёмочное тестирование:

- 3.1. Проведите приёмочное тестирование программы согласно алгоритму, описанному в техническом задании.
- 3.2. В случае обнаружения ошибок в приложении (функциональных, ошибок верстки и дизайна, грамматических ошибок), исправьте их и сделайте коммит.

4. Проверить, что задание лабораторной работы выполнено верно:

- 4.1. Зайдите в свой аккаунт GitHub и убедитесь, что синхронизация прошла успешно.
- 4.2. Оформите отчет по лабораторной работе.
- 4.3. Если все задания выполнены верно, ответьте на вопросы для самоподготовки и переходите к защите лабораторной работы. На защите продемонстрируйте работу новой функциональности и всего приложения в целом, сценарий сборки установочного пакета, структуру файлов установленного приложения, структуру файлов в удаленном репозитории и историю коммитов.

2.5.5 Вопросы для самоподготовки

1. Что такое приёмочное тестирование? В чем его задача? Когда оно проводится?
2. Что такое уникальный идентификатор приложения?
3. Что происходит в процессе установки приложения на компьютер?
4. Почему в установочном скрипте должны использовать только относительные пути к файлам вместо абсолютных?
5. Почему сценарий установки должен быть под версионным контролем?
6. Что такое итерационная разработка ПО?
7. Что такое рефакторинг? Чем рефакторинг отличается от оптимизации кода?

2.5.6 Содержание отчета

Отчет по лабораторной работе должен содержать:

- Титульный лист, оформленный согласно требованиям ОС ТУСУР [].
- Актуальная UML-диаграмма классов.
- Описание сценария сборки установочного пакета.
- Описание состава установочного пакета – перечень файлов, содержащихся в установочном пакете.
- Текущая история коммитов ветки develop (допустимо в виде снимка экрана соответствующей страницы GitHub или VisualStudio).

Отчёт сдается с указанием даты защиты лабораторной и подписью студента.

2.5.7 Список источников

Язык программирования C#

1. Шилдт Г. C# 4.0: полное руководство.: пер. с анг. Бернштейн И.В. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.: ил.
2. Руководство по программированию в Windows Forms [Электронный ресурс]. / matanit.com: сайт о программировании. — URL: <https://metanit.com/sharp/windowsforms/> (дата обращения 27.08.2018).
3. Соглашение по оформлению кода команды RSDN [Электронный ресурс]. / RSDN (Russian Software Developers Network): сайт о программировании. — URL: <https://rsdn.org/article/mag/200401/codestyle.XML> (дата обращения 27.08.2018).

Сборка установочного пакета с использованием InnoSetup

4. Inno Setup: официальный сайт. [Электронный ресурс]. / Jordan Russell's software. — URL: <http://www.jrsoftware.org/isinfo.php> (дата обращения 6.08.2018).
5. Inno Setup: полная официальная документация к программе. [Электронный ресурс]. / Jordan Russell's software. — URL: <http://www.jrsoftware.org/ishelp/> (дата обращения 6.08.2018).
6. Inno Setup: создание инсталлятора на примере развертывания C# приложения. [Электронный ресурс]. / блог пользователя maisvendoo, сайт habr.com. — URL: <https://habr.com/post/255807/> (дата обращения 6.08.2018).

Автоматизация сборки установочного пакета

7. Свойства проекта: разделы Compile, Build, Build Event в Visual Studio [Электронный ресурс]. / Professorweb.ru: сайт о разработке ПО. — URL:

- https://professorweb.ru/my/programs/visual-studio/level2/2_7.php (дата обращения 27.08.2018).
8. Перечень макросов событий до или после сборки проекта [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/ru-ru/visualstudio/ide/reference/pre-build-event-post-build-event-command-line-dialog-box> (дата обращения 27.08.2018).
 9. Перечень команд оболочки Shell [Электронный ресурс]. / Официальный портал Microsoft. — URL: <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands> (дата обращения 2.11.2018).

Требования к оформлению отчета

10. ОС ТУСУР 01-2013. Работы студенческие по направления подготовки и специальностям технического профиля. Общие требования и правила оформления. – Томск: ТУСУР, 2013. – 57с. – URL: https://storage.tusur.ru/files/40668/rules_tech_01-2013.pdf (дата обращения 27.08.2018)

2.6 Составление проектной документации

Цель работы: изучить виды технической документации и получить умения их составления.

Задачи:

- 1) Изучить виды технической документации, этапы, на которых они разрабатываются, а также ответственных за их разработку исполнителей.
- 2) Разработать пояснительную записку к разработанному приложению.
- 3) Провести ретроспективу по выполненному проекту.
- 4) Составить календарный план и смету для аналогичной программы.

2.6.1 Виды документации на различных этапах разработки

Техническая документация – необходимое зло в разработке, важное для успешного завершения проектов. Если коммуникации внутри команды важны для координации текущей работы в команде, то документация фиксирует результаты проделанной работы, исключая недопонимание с заказчиком, разработчиков между собой, а также избавляет от двойной работы. Так, например, в начале разработки программисты обсуждают будущую архитектуру приложения. И очень важно, чтобы свои архитектурные идеи они зафиксировали в виде документа. Разумеется, правильное оформление диаграмм классов и пакетов может отнять время, которое хотелось бы посвятить написанию кода. Но в будущем эти диаграммы дадут следующие преимущества:

- 1) Если кто-то в команде спрашивает: «А как устроен этот модуль?», остальной команде не надо отвлекаться и объяснять ему архитектуру. Достаточно отправить его читать диаграммы классов.
- 2) Если в команду приходят новые разработчики, особенно junior уровня, опять же не надо надолго отрывать тимлида от работы – краткий инструктаж по основным принципам архитектуры проекта, а детали новый разработчик может изучить уже по диаграммам.
- 3) Несколько месяцев (или лет) назад ваша команда разработала приложение для заказчика. Заказчик вновь обратился к вам, чтобы вы разработали следующую версию программы. По прошествии лет никто в вашей команде не будет помнить, как была устроена программа. И в этом случае готовые диаграммы пакетов и классов позволят быстрее разобраться в старом проекте.

Чем больше людей участвует в разработке, и чем больше проект, тем больше требуется технической документации. Это замедляет время разработки, но без документации команда в какой-то момент завязнет в изучении собственного проекта и внутренних коммуникациях.

Написание и поддержание актуальной документации по проекту может занимать много времени, но потраченное время в разы окупается в долгосрочных проектах.

На разных этапах разработки требуется различная документация. Далее перечислены часто используемые документы:

- 1) Договор на оказание услуг
- 2) Техническое задание

- 3) Календарный план и смета
- 4) Список спецификаций
- 5) Поведенческие диаграммы
- 6) Макеты интерфейса
- 7) Структурные диаграммы
- 8) План тестирования
- 9) Руководство пользователя и пояснительная записка
- 10) Акт приёма-передачи
- 11) Ретроспектива

Договор на оказание услуг – договор между заказчиком и исполнителем (разработчиком и или компанией по разработке), фиксирующий тот факт, что разработчик обязуется разработать некоторый продукт по требованиям заказчика, а заказчик обязуется оплатить услуги по разработке. В договоре указываются права и обязанности сторон. Для защиты собственных прав разработчику стоит указать в договоре сроки приёма и оплаты заказчиком (на случай, если заказчик будет задерживать оплату) и штрафы за нарушение сроков. Также в договоре важно указать, что в случае изменения утвержденного технического задания (а техническое задание часто уточняется и дорабатывается в ходе разработки), сроки выполнения и стоимость разработки также могут измениться и это будет утверждаться дополнением к договору. Заказчик же в свою очередь может настоять на включение в договор пунктов о сроках разработки и сдачи проекта, и штрафах при их нарушении. Такие пункты защищают обе стороны в случае недобросовестной второй стороны.

Приложениями к договору, как правило, являются техническое задание, календарный план и смета проекта.

Техническое задание – документ, описывающий требования к разрабатываемой системе: функциональные, системные, требования эргономики, требования к производительности и т.д. Важными разделами технического задания являются назначение проекта, целевая аудитория / контекст использования – именно данные разделы позволяют разработчику лучше понять конечного пользователя и в итоге сделать качественный продукт. Содержание остальных разделов во многом определяется проектом (автоматизированная система, программное обеспечение). Так, техническое задание может содержать только перечень функциональных требований, а может содержать точные макеты интерфейсов или описание требуемой архитектуры проекта, если приложение будет встраиваться как часть другой инфраструктуры. Подробно с содержанием различных технических заданий можно ознакомиться в [1].

Календарный план и смета – документы, описывающие основные сроки выполнения проекта и стоимость отдельных этапов. Так, в случае разработки продолжительных проектов часто используется итерационный подход к разработке. Итерационный подход означает, что каждый месяц (или иной период времени) исполнитель предоставляет заказчику новую функциональность в проекте, а заказчик выполняет оплату этих работ. Такой подход удобен и исполнителю, и заказчику: заказчик оплачивает работу частями, а не целиком; при этом каждый месяц заказчик видит результаты работы исполнителя. Исполнитель же получает оплату каждый месяц и может не волноваться, что заказчик потом не оплатит проект в конце. В случае же, если заказчик не оплатит следующий месяц работы, то проект можно остановить и не тратить ресурсы на неоплачиваемую работу. Подробнее о календарном плане и смете проекта можно ознакомиться в п. 2.6.2.

Список спецификаций – составляется на основе технического задания, либо уже содержится как часть технического задания. Как документ, создается после подписания договора и начала работ по проекту. Фактически, каждая спецификация - это простое предложение, описывающее программу в формате: «Если с системой сделать действие X, то система должна отреагировать результатом Y». Например:

«SR101. Если нажать кнопку Open, должно появиться окно выбора файла проекта расширением *.proj»

«SR503. Если ввести в текстовое поле числовое значение из 16 цифр, справа от текстового поля должен появиться значок, что поле заполнено правильно»

«SR504. Если ввести в текстовое поле не числовое значение (буквы, символы и т.д.), справа от текстового поля должен появиться значок, что поле заполнено неправильно»

Таким образом, спецификации описывают поведение всей системы простыми легко проверяемыми предложениями. Каждая спецификация в дальнейшем может быть как задачей для разработчика, так и описанием тестового случая для тестировщика – в этом главное преимущество спецификаций. Но для этого спецификации должны быть сформулированы лаконично и однозначно.

Каждую спецификацию следует нумеровать и обозначать специальным идентификатором (например, SR101, как в примере выше, где SR – software requirement). Уникальные номера спецификаций позволяют ссылаться на них в других документах или в задачах. Например, передавая выполненную задачу на тестирование, разработчик может просто сказать тестировщику: «Задачи SR503 и SR504 выполнены, их можно тестировать». После чего тестировщику достаточно открыть список спецификаций и уже детально ознакомиться с тем, что именно нужно проверить.

Поведенческие диаграммы – диаграммы, описывающие процесс работы пользователя в приложении и самого приложения. К таким диаграммам можно отнести блок-схемы, диаграммы вариантов использования, диаграммы нотации IDEF. Поведенческие диаграммы могут составляться на этапе анализа технического задания или его составления, либо после начала работ по проекту перед созданием макетов и проектированием системы. В отличие от ТЗ или списка спецификаций, поведенческие диаграммы иллюстрируют процесс работы, показывая ветвления в действиях пользователя или программы, циклические операции, а также могут описывать роли пользователей. Поведенческие диаграммы помогают добиться единого видения поведения разрабатываемой системы у разработчиков, а также быстрее включать новых разработчиков в процесс разработки. Наглядность поведенческих диаграмм – их главное преимущество.

Макеты интерфейса – документ с утвержденными макетами всех окон/страниц пользовательского интерфейса, а также с указанием связей между окнами. Также макеты могут показывать реакцию, например, на ввод некорректных данных. Схема, показывающая связи переходов между окнами приложения, называется screenflow (англ. «поток экранов»). Как правило, макеты интерфейса утверждаются заказчиком, чтобы убедиться в едином видении разрабатываемого продукта между заказчиком и исполнителем.

Структурные диаграммы – диаграммы, описывающие внутреннюю структуру приложения как набор взаимодействующих подсистем. К таким диаграммам можно отнести диаграммы пакетов (высокоуровневое представление структуры программы), диаграммы классов (среднеуровневое представление), описания отдельных алгоритмов (низкоуровне-

вое представление). Структурные диаграммы составляются на основе технического задания, списка спецификаций и поведенческих диаграмм на этапе проектирования, и модифицируются в ходе разработки. В составлении диаграмм участвуют наиболее опытные разработчики команды, менее опытные разработчики будут участвовать лишь в реализации созданной архитектуры. Структурные диаграммы являются прямым руководством для разработчиков, как именно должна быть реализована система.

Зачастую, на этапе проектирования диаграммы рисуются в обобщенном виде, без уточнения многих деталей. Полные диаграммы пакетов и классов часто составляются уже после реализации, на основе готовой системы.

План тестирования – документ, описывающий порядок и приоритеты тестирования проекта. План тестирования определяет приоритеты тестирования на основе технического задания, описывает тестовые случаи с тестовыми данными для проверки спецификаций, группирует тестовые случаи по категориям – ручные/автоматизированные тесты, ежедневные/еженедельные/приёмочные тесты и т.д. Каждый тестовый случай имеет свой уникальный номер, например, T677, где T – testing (тестирование). Также тестовый случай содержит ссылку на спецификацию, для которой он составлен. Например, T677 может ссылаться на спецификацию SR503 и т.д. Уникальные номера тестовых случаев позволяют быстрее и эффективнее назначать задачи среди тестировщиков, а ссылки на спецификации позволяют: а) отчитываться перед руководителем проекта, какие именно функции/спецификации в системе были протестированы; б) в случае изменения спецификаций (например, если заказчик решил поменять часть технического задания) позволят быстро найти связанные с ними тестовые случаи и поправить их согласно новым спецификациям.

Уникальные номера спецификаций, схем, диаграмм и тестовых случаев, а также кросс-ссылки между собой позволяют быстро ориентироваться в документации и поддерживать её актуальность.

Для малых и средних проектов, когда время выполнения проекта составляет не больше года, и команда проекта не превышает десяти человек, введение полноценной документации с кросс-ссылками может только затруднять разработку. Однако для больших, комплексных проектов, в которых задействованы сотни и тысячи работников это является чуть ли не обязательным требованием.

План тестирования составляется старшим тестировщиком (senior QA, lead QA) и в дальнейшем является внутренним руководством для отдела тестирования. Особенно важно составлять план тестирования для проведения регрессионного тестирования: точное описание тестовых случаев с начальными данными позволяет назначать на регрессионное тестирование даже новых тестировщиков, не знакомых с проектом.

Руководство пользователя и пояснительная записка – два документа, описывающих разработанную систему. Так, предыдущие документы составляются с позиции **разрабатываемого** продукта, руководство пользователя и пояснительная записка составляются для **разработанного** приложения. Оба документа разрабатываются после завершения проекта или незадолго до его завершения на основе всех предыдущих документов.

Руководство пользователя должно содержать полную информацию о том, как взаимодействовать с приложением: от установки до исключительных ситуаций. Если приложение

спланировано сразу для нескольких групп пользователей, имеющих разные функциональные возможности в приложении, то руководство пользователя описывает работу для каждой группы.

В отличии от руководства пользователя, пояснительная записка описывает то, как приложение устроено внутри. Фактически, пояснительная записка является документом, с помощью которого другой разработчик сможет разобраться с логикой работы программы. Например, пояснительная записка может быть отдана в отдел поддержки, где разработчики изучат документацию и смогут исправлять ошибки, возникающие у пользователей.

В случае разработки на заказ, пояснительная записка отдается заказчику. Это важно, так как по прошествии времени этот заказчик может обратиться к вам за разработкой новой версии программы. Так, если заказчик предоставит вам не только исходный код, но и пояснительную записку, это сэкономит вам и вашей команде много нервов в попытках вспомнить как работает приложение. По этой же причине важно не удалять **любую** техническую документацию после завершения проекта, а лучше хранить её в репозитории проекта или специальных хранилищах для восстановления – backup. О составе пояснительной записке подробнее в п. 2.6.3.

Акт приёма-передачи – является юридическим документом, подтверждающим, что исполнитель выполнил техническое задание в полном объёме. Подписывая акт приёма-передачи, заказчик соглашается, что проект выполнен, а, следовательно, вы, как исполнитель, можете требовать с заказчика оплату по проекту (или оставшуюся часть). Актом приёма-передачи может завершаться не только проект, но и каждый отдельный этап, при необходимости. Для исполнителя подписанный акт в будущем может быть юридическим доказательством, что заказчик считает проект выполненным – на случай возможных претензий или отказе об оплате.

Ретроспектива – процедура, выполняемая по завершению проекта для определения недостатков процесса разработки, мер по их устраниению, а также документ, в котором недостатки и меры фиксируются. Ретроспектива проводится тимлидом, где каждый участник команды может высказать своё мнение по завершенному проекту: что было хорошо, что было плохо, и что можно изменить, чтобы в следующий раз разработка была лучше. После высказанных замечаний принимаются решения о том, что поменять в процессе разработки. Подробнее о проведении ретроспективы в п.2.6.4.

Выше перечислены только ключевые виды документов, используемые для проектов средней сложности. Согласно ЕСПД, видов программной документации гораздо больше, также в каждой компании могут вводиться собственные виды документов.

Отдельно стоит упомянуть такие документы как резерв проекта и резерв итерации. Эти два документа используются как альтернатива техническому заданию в случаях разработки по гибким методологиям, таким как Scrum, Kanban, Extreme Programming. **Резерв проекта (project backlog)** – документ, содержащий список всех потенциально возможных функций в разрабатываемой приложении. В отличии от технического задания, где разработке подлежат все функциональные возможности, резерв проекта описывает функции, которые **могут быть** разработаны, но какие именно будут разработаны, будет решено в процессе разработки. Так, некоторые функции из резерва проекта могут быть не реализованы никогда, в то время как техническое задание должно быть реализовано полностью.

Также отличием резерва проекта от технического задания является то, что в техническом задании все функции системы проверяются на непротиворечивость между собой. В то время как в резерве проекта записаны отдельные функции, и непротиворечивость будет проверяться только в том момент, когда будет принято решение на их разработку. При этом все функции в резерве проекта проранжированы по сложности и приоритету. Приоритет функции оценивает заказчик, а сложность оценивает исполнитель.

На первый взгляд резерв проекта может показаться странной заменой для технического задания – как можно разрабатывать приложение, если в самом начале даже не знаем, какие именно функции мы должны разработать? Но на практике резервы проекта применяются достаточно часто, особенно для проектов с широкой аудиторией.

Например, заказчик хочет разработать аудиоплеер с тремя функциями: 1) возможность делиться своими плейлистами с другими пользователями; 2) возможность создания собственных миксов в режиме реального времени; 3) возможность создания визуализаций для музыки. Каждая из этих функций может занять месяц разработки, но заказчик не уверен, что идея приложения понравится пользователям. Оплачивать же три месяца разработчикам и не знать, захотят ли устанавливать приложение пользователи – заказчик не хочет. В таком случае, вместо составления ТЗ он составляет резерв проекта, в котором описываются все три функции. Вместо приложения со всеми тремя функциями, исполнитель за первый месяц реализует приложение только с одной функцией, например, возможностью делиться своими плейлистами с другими пользователями. При этом в процессе проектирования не учитывается тот факт, что в дальнейшем в приложение могут быть добавлены две другие функции – не надо учитывать их непротиворечивость.

Несмотря на ограниченную функциональность, это приложение уже можно установить и пользоваться. После первых установок и проверке на фокус-группе, заказчик по отзывам пользователей узнает, что им было бы интереснее видеть в следующей версии функцию создания собственных миксов. Теперь заказчик достоверно знает, что пользователи хотят вторую функцию, и следующий месяц оплачивает разработку новой функции. Исполнитель модифицирует архитектуру приложения, чтобы новая функциональность не конфликтовала с предыдущей, и через месяц выпускает новую версию программы. Заказчик получает новых пользователей и новые отзывы. Более того, сами пользователи в отзывах могут предлагать другие функции для программы, которые заказчик может самостоятельно добавить в резерв проекта.

Таким образом, резерв проекта позволяет хранить возможные функции приложения для разработки, но, в отличие от технического задания, позволяет менять приоритеты в разработке после каждой итерации. **Резерв итерации (iteration backlog, spring backlog)** – документ, в котором описаны те функции, которые будут разрабатываться в течение этой итерации. Подробнее о данных документах можно почитать в методологии управления проектами Scrum.

2.6.2 Календарный план и смета проекта

Два ключевых вопроса для заказчика перед подписанием договора это: 1) когда проект будет выполнен? 2) сколько это будет стоить? В зависимости от ответа на эти вопросы, заказчик примет решение, нанимать разработчика или нет. Так, у заказчика заранее могут

быть некоторые ожидания по срокам выполнения и стоимости. Вопрос стоимости для заказчика более важен: если большинство заказчиков готово подождать дополнительное время для завершения проекта, то тратить лишние финансы заказчики не хотят.

И здесь очень важно уметь правильно оценить и обосновать сроки и стоимость перед заказчиком. С одной стороны, если вы оцените проект в полгода, хотя на самом деле он будет выполняться год, заказчик, вероятно, согласится с вами работать, но по прошествии шести месяцев он откажется платить дальше – оставшиеся полгода вы как разработчик будете работать бесплатно. Или, что еще хуже, заказчик уйдет и потребует платить неустойку – тогда вы останетесь в минусе.

С другой стороны, если вы оцените годовой проект в полтора года, вероятно, и стоимость проекта будет завышена – в смету войдут лишние шесть месяцев разработки, которые будут стоить не дешево. В таком случае, вы просто не получите проект.

Разработчик должен уметь правильно оценивать сроки и стоимость работы, чтобы не терять заказчиков.

Здесь есть и другой подводный камень: заказчик должен понимать, за что он платит деньги. Суть в том, что, если вы оцениваете крупный проект на несколько лет, его стоимость в любом случае будет высокой. И в этом случае, вы, как разработчик, должны аргументировать каждый час разработки, за который просите деньги у заказчика. Если заказчик будет видеть, за что он платит, он охотнее согласиться отдать проект вам. А необоснованные сметы вызывают у людей подозрение и недоверие к исполнителю, даже если исполнитель называет честную стоимость.

Как же правильно оценить сроки разработки и её стоимость? На этот случай есть простое правило:

Правильно и максимально точно можно оценить только те задачи, которые вы уже многократно делали.

Как следствие из этого правила: не надо браться за проекты, которые вы не умеете делать – наверняка вы неправильно оцените сроки и стоимость, что приведет к потере заказчика и доверия к вам как разработчику.

Следует браться только за те проекты, которые вы уже делали, или которые незначительно отличаются от того, что вы делали. Если же в проекте много неизвестных вам элементов, то стоит отказаться или предложить начать разработку с более простого приложения. Дело в том, что все неизвестные вам элементы вы будете оценивать либо «вилкой» между наилучшим и наихудшим вариантами, либо будете оценивать наугад. Однако заказчику назвать вилку в цене или сроках нельзя: «проект займет от шести до десяти месяцев». Заказчик на это не согласится. Вам придется назвать конкретную стоимость разработки, а значит всю разницу, что будет стоить итоговая цена работы, вы оплатите из своих сбережений – следовательно, вы будете работать в убыток. Поэтому не следует браться за проекты, которые вы не можете правильно оценить. Далее мы рассмотрим вопрос, как оценить сроки разработки и её стоимость, а также разберем создание календарного плана и сметы проекта.

Как указывалось ранее, разработку лучше разделить на этапы, например, по месяцу (итерационная разработка), или по логическим этапам: макетирование, проектирование, реализация, тестирование и т.д. (каскадная разработка). Для каждого этапа указывается дата

начала, дата завершения, а также что в ходе этапа будет реализовано и что будет предоставлено заказчику. Отдельной графой указывается стоимость этапа.

Сам календарный план и смета оформляются в виде таблицы. Рассмотрим примеры календарных планов для каскадной и итерационной разработки. Например, нам необходимо разработать приложение для ведения списка задач (TODO-листы) с привязкой задач к календарю. Программа должна иметь мобильную и веб-версию, а также позволять нескольким пользователям создавать общие списки задач (для командной работы, например).

Календарные планы для такого проекта будут выглядеть следующим образом (табл. 2.1, 2.2).

Таблица 2.1 Пример календарного плана каскадной разработки

#	Срок проведения, начало – конец	Описание	Стоймость, руб
1	2018.09.01 – 2018.09.20 (20 дней)	Макетирование Создание макетов мобильной версии: - главное окно для создания списков задач - окно просмотра/редактирования списка задач - окно просмотра/редактирования задачи - окно поиска пользователей - окно настроек приложения - окно редактирования аккаунта Создание макетов веб-версии: - окно авторизации - окно регистрации - главное окно для создания списков задач - окно просмотра/редактирования списка задач - окно просмотра/редактирования задачи - окно поиска пользователей - окно настроек приложения - окно редактирования аккаунта	25 000
2	2018.09.21 – 2018.10.10 (20 дней)	Проектирование Структура базы данных Диаграммы пакетов мобильной и веб-версии Диаграммы классов мобильной и веб-версии	40 000
3	2018.10.11 – 2018.12.19 (70 дней)	Реализация Разработка приложения согласно спроектированных диаграммам	120 000
4	2018.12.01 – 2018.12.20 (20 дней)	Тестирование и отладка Составление плана тестирования Тестирование функциональных требований к приложениям Нагрузочное тестирование веб-версии	40 000
5	2018.12.20 – 2018.12.29 (10 дней)	Внедрение и подготовка проектной документации Сборка установщика мобильной версии, размещение веб-версии на сайте, написание пояснительной записки и руководств пользователя (для мобильной и веб-версии)	15 000
Итого: 4 месяца			240 000

Календарный план проекта для каскадной методологии в каждой отдельной строке будет описывать классические этапы разработки: макетирование, проектирование, реализация, тестирование, внедрение. Сроки выполнения каждого этапа будут отличаться, так как, например, макетирование занимает меньше времени, чем реализация. Стоимости также

будут отличаться. Например, макетирование и проектирование занимают равное количество дней, однако стоимость работы проектировщика больше, чем стоимость работы дизайнера. Поэтому этап проектирования будет оцениваться дороже.

Теперь рассмотрим календарный план для итерационной разработки:

Таблица 2.2 Пример календарного плана итерационной разработки

#	Срок проведения, начало – конец	Описание	Стоймость, руб
1	2018.09.01 – 2018.09.30 (30 дней)	Разработка мобильного приложения для создания TODO-листа Приложение позволяет создавать пользователю создавать единый список задач и сохранять его между сессиями приложения	40 000
2	2018.10.01 – 2018.10.31 (31 день)	Добавление функции календаря в приложение Приложение позволяет назначать задачам даты выполнения, уведомлять пользователя о ближайших задачах, устанавливать задачам приоритеты, создавать несколько списков задач	40 000
3	2018.11.01 – 2018.11.30 (30 дней)	Разработка веб-версии приложения Создается веб-версия приложения с аналогичной функциональностью и системой авторизации. Задачи пользователя в мобильной и веб-версии должны синхронизироваться в общей базе данных. Добавление авторизации и синхронизации в мобильную версию.	80 000
4	2018.12.01 – 2018.12.29 (29 дней)	Добавление функции общих списков задач Мобильная и веб-версии позволяют создавать списки задач с общим доступом для нескольких пользователей. Задачи могут назначаться на конкретных пользователей, пользователи могут подписываться на уведомления о выполнении задач.	80 000
Итого: 4 месяца			240 000

В случае календарного плана для итерационной разработки все итерации имеют одинаковый период, а каждая итерация завершается выпуском новой версии приложения. При этом новые функции описываются в календарном плане. Задачи макетирования, проектирования и тестирования равномерно распределяются по всем итерациям, за счет чего в каждой итерации участвуют все специалисты, и стоимость разработки этапов примерно одинаковая.

Заказчик может обратить внимание, что третий и четвертый этапы вдвое дороже первого и второго. Это объясняется тем, что, начиная с третьего этапа необходимо одновременно поддерживать две версии программы – мобильную и веб-версию. А это означает двойной объём работы или даже наём отдельной команды, специализирующейся в веб-проектах. Таким образом, есть понятное для заказчика обоснование увеличения стоимости, и это не будет для него сюрпризом.

Оценка времени выполнения. Как говорилось ранее, правильно оценить время выполнения задачи можно только в том случае, если вы делали эту или схожую задачу. Для примера, попробуйте оценить время выполнения создания и настройки репозитория нового проекта или написания заново бизнес-логики вашего варианта приложения. Возможно, при первом создании репозитория вы потратили порядка четырех часов на изучение пособия и на создание самого репозитория. Однако теперь, когда вы уже знаете, как создавать репозиторий и настраивать его, ваша оценка данной задачи значительно уменьшится, например, до получаса. Аналогично и для разработки бизнес-логики – если в начале данного курса вам

было сложно оценить, сколько времени вам понадобится на разработку, то сейчас, разработав приложение, вы можете оценить время, за которое вы разработаете такую же программу.

Как правило, оценку времени приводят в человекочасах. То есть, оценка задачи в 8 человекочасов означает, что один человек выполнит эту задачу за 8 часов, а два человека выполнят задачу за 4 часа. Для того, чтобы можно было оценить задачу в человекочасах, необходимо декомпозировать эту задачу на подзадачи до тех пор, пока вы не сможете достаточно точно оценить каждую подзадачу. Достаточно точной оценкой принято считать не более 4 часов. То есть, если вы смогли декомпозировать разработку программы на подзадачи, каждая из которых занимает не более 4 часов, значит:

- а) вы хорошо представляете нюансы разработки этого проекта и с высокой степенью вероятности можете его выполнить;
- б) вы можете рассчитать точные сроки и стоимость проекта для заказчика.

Например, вам необходимо разработать приложение-калькулятор. Калькулятор – простое приложение, и оценить время его разработки не сложно. Декомпозируем его разработку на подзадачи, при чём за основу декомпозиции возьмем классические этапы разработки:

- 1) Составление технического задания: 2 часа (описать функции калькулятора не сложно, так как каждый из нас пользовался калькулятором)
- 2) Макетирование: 1 час (опять же, макет приложения в одно окно не займет много времени)
- 3) Проектирование: 4 часа (если реализуется инженерный калькулятор или калькулятор программиста, над архитектурой придется немного подумать)
- 4) Реализация: 8 часов (пока сложно оценить, но грубая оценка – 8 часов)
- 5) Тестирование: 1 час
- 6) Сборка установщика: 2 часа (вместе с написанием сценария для автоматической сборки установщика).

В этом списке задач мы видим, что реализация занимает больше 8 часов, и, вероятно, мы не до конца представляем себе выполнение этой задачи. Следовательно, её следует разделить на более мелкие подзадачи. Например:

- 4.1) Создание и настройка репозитория проекта: 0,5 часа
- 4.2) Верстка формы калькулятора: 2 часа
- 4.3) Создание простых (стандартных) математических операций в приложении: 3 часа
- 4.4) Создание инженерных математических операций: 3 часа

После декомпозиции все задачи занимают меньше 4 часов, а общая сумма выполнения равна 8,5 человекочасам – мы получили более точную оценку.

Итого, общие трудозатраты на разработку калькулятора равны 18,5 человекочасов. Аналогично декомпозиция выполняется для итерационного подхода, где в качестве отдельных задач будут не этапы, а планируемые функции.

Если у вас уже есть техническое задание или описание функций (как в случае лабораторных работ), то проще произвести оценку времени по каждой спецификации или функциональному блоку

Чем сложнее проект, тем более детальную декомпозицию придется проводить при оценке сроков. Однако, чем больше опыта в разработке у вас будет, тем более крупные

оценки вы сможете давать. Например, вы сможете достаточно точно оценивать блоки по 20 человекочасов без необходимости их декомпозиции на задачи по 4 часа. Таким образом, чем больше вы разрабатываете, тем точнее будут ваши оценки.

Оценка сроков выполнения. Заказчика интересует не только количество человеко-часов проекта, так как от них напрямую зависит стоимость. Но заказчика также интересует дата, когда проект будет завершен. А значит, надо как-то перевести рассчитанные человекочасы проекта в календарные даты.

Один сотрудник за рабочий день отрабатывает 8 часов. Однако, если проект оценен в 40 человекочасов, это не означает, что он будет выполнен за 5 рабочих дней. Так, если в команде будет 5 разработчиков, то они могут закончить проект за один день, верно?

Всё не так просто. Есть два аспекта, которые необходимо учитывать:

- 1) Разные виды работ должны выполнять разные специалисты. Так, например, программисту нельзя поручить задачу по созданию макетов интерфейса, а дизайнеру нельзя поручить проектирование архитектуры – каждый должен делать только свои задачи. Поэтому нельзя разделить все задачи проекта поровну на всех участников команды: если в проекте из 40 часов только 2 часа посвящены дизайну интерфейсов, то дизайнер отработает в проекте только 2 часа.
- 2) Не все работы можно выполнять параллельно. Например, нельзя начать задачи по написанию кода, пока не будет написано техническое задание. Поэтому, если из 40 часов проекта 4 часа посвящены написанию технического задания, то все разработчики будут ждать, пока менеджер проекта или тимлид не закончат написание ТЗ. До тех пор вся команда будет простоять.

Для того, чтобы оценить, выполнение каких задач можно распараллелить, используются диаграммы Гантта. Диаграммы Гантта показывают, какие задачи и в какое время выполняет каждый участник команды и от кого зависит начало его работ.

Представим, что у нас есть приложение, для которого была произведена следующая оценка по срокам:

#	Задача	Сроки, ч
1	Составление ТЗ	8
2	Макетирование	6
3	Проектирование архитектуры	8
4	Составление плана тестирования	6
5	Разработка функции #1 (задача 1)	4
6	Разработка функции #2 (задача 2)	6
7	Разработка функции #3 (задача 3)	4
8	Разработка функции #4 (задача 4)	8
9	Разработка функции #5 (задача 5)	4
10	Разработка функции #6 (задача 6)	6
11	Разработка функции #7 (задача 7)	6
12	Разработка функции #8 (задача 8)	4
13	Тестирование функций #1-8 (2 часа на каждую функцию)	16
14	Отладка	8
15	Код ревью	8
16	Сборка установщика	4
		Итого: 106

Предполагается, что в проекте участвует 6 человек:

- менеджер проекта
- старший разработчик
- младших разработчика

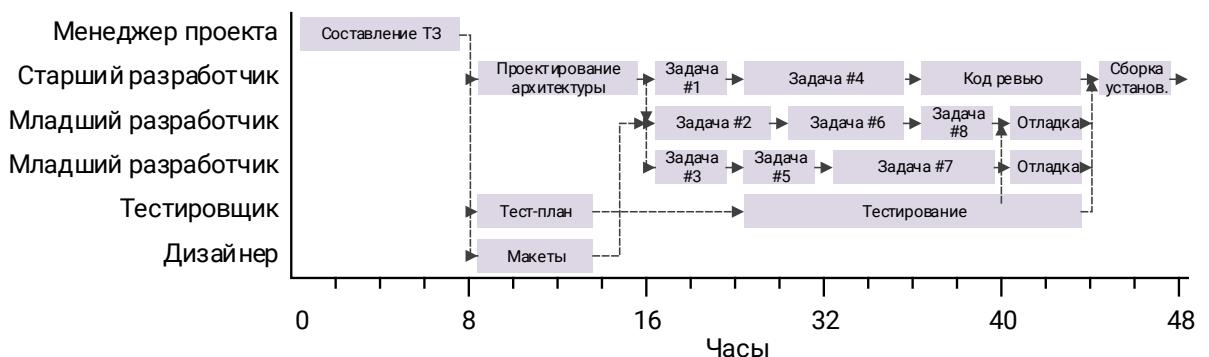
- тестировщик
- дизайнер

Если разделить 106 часов работы на 6 человек, мы получим 17,7 часов работы для каждого. Это значит, что проект можно было бы закончить за 2,5 рабочих дня. Однако данный расчет неверен. Необходимо учесть, что каждую из задач могут выполнять только определенные участники команды:

- менеджер проекта: составление ТЗ;
- дизайнер: макетирование;
- тестировщик: составление плана тестирования, тестирование;
- старший разработчик: проектирование, код ревью.
- все разработчики: разработка функций #1-8, отладка, сборка установщика.

Также помним, что выполнение некоторых задач не может начаться, пока не завершится другая задача. Например, проектирование архитектуры и макетирование можно начать только после завершения составления технического задания. А тестирование можно начать только после того, как будет реализована хотя бы одна функция программы.

Для того, чтобы понять в какой последовательности будут выполняться задачи, и кто их будет делать, разместим задачи на диаграмме Ганнта:



По оси абсцисс на диаграмме Ганнта откладываются рабочие часы, а по оси ординат для каждого сотрудника заводится полоса, на которой будут размещены его задачи в проекте. Каждая задача у работника обозначается полоской, длина которой равна часам, требуемых на её выполнение.

Стрелочки между задачами обозначают, какие задачи заинтересованы в результатах предыдущей. Например, для того, чтобы начать проектирование, макетирование и составление плана тестирования, необходимо техническое задание.

Некоторые задачи, такие как отладка, могут выполняться одновременно несколькими сотрудниками. Другие же задачи, такие как сборка установщика, могут выполняться любым разработчиком, но только одним, и только после завершения всех остальных задач. Тестирование также может выполняться несколькими тестировщиками, однако в данном проекте участвует только один тестировщик, следовательно, задача не может быть распределена.

Диаграмма Ганнта должна составляться таким образом, чтобы полоски каждого сотрудника были максимально заполнены, и выполнялось как можно больше параллельных задач – это обеспечит наиболее быстрое завершение проекта.

После того, как диаграмма Ганнта составлена, видно, что весь проект будет завершен за 48 рабочих часов календарного времени. То есть, проект может быть выполнен за 6 рабочих дней. Именно этот срок озвучивается заказчику или указывается в календарном плане.

Диаграмма Ганнта применяется как для планирования небольших проектов/итераций, так и для управления персоналом в проекте. Планирование в рамках одной диаграммы Ганнта проектов в тысячи человеко-часов и десятками сотрудников затруднительно.

Также диаграмма Ганнта может иметь другую развертку – по этапам разработки, а не сотрудникам. В таком случае, для каждого этапа согласно техническому заданию и оценке задач выписываются их задачи и размещаются на диаграмме Ганнта с указанием их зависимостей друг от друга.

Расчет стоимости проекта. После декомпозиции проекта на отдельные задачи и оценки времени выполнения, достаточно легко рассчитать стоимость проекта. В стоимость проекта входят следующие статьи расходов:

- 1) Оплата времени разработчиков.
- 2) Стоимость расходных материалов; арендной платы; оборудования, закупаемого специально для проекта.
- 3) Норма прибыли.

Оплата времени разработчиков – если проект выполняется одним разработчиком, то для определения оплаты времени разработчика достаточно умножить количество полученных человеко-часов по проекту на стоимость часа. Так, если время разработки калькулятора занимает 18,5 человеко-часов, а стоимость одного часа middle разработчика составляет 350 руб, то оплата времени разработчиков составит $18,5 \text{ ч} \times 350 \text{ руб/ч} = 6475 \text{ руб.}$

В случае, если над проектом работает команда разработчиков, то необходимо оценивать стоимость часа работы каждого отдельного специалиста. Для примера предположим, что над калькулятором работает три человека: старший разработчик (тимлид), младший разработчик, дизайнер (совмещающий с тестированием). Тогда необходимо разделить выполняемые задачи среди участников:

Таблица 2.3 Разделение задач проекта среди участников

Разработчик	Работы	Время, ч	Итого, ч
Старший разработчик	- Составление ТЗ	2	8,5
	- Проектирование	4	
	- Создание и настройка репозитория	0,5	
	- Сборка установщика	2	
Младший разработчик	- Верстка формы по макету	2	8
	- Реализация простых мат. операций	3	
	- Реализация инженерных мат. операций	3	
Дизайнер-тестировщик	- Макетирование	1	2
	- Тестирование	1	
Итого:			18,5ч

Не стоит удивляться тому, что участник совмещает работы дизайнера и тестировщика: для малых команд совмещение нескольких обязанностей - это нормальная ситуация.

Теперь, исходя из стоимости часа каждого участника, можно рассчитать его оплату, а также общую стоимость рабочих часов.

Основываясь на данных сайтов вакансий, можно выяснить средний уровень зарплаты для этого разработчика в месяц, а затем рассчитать стоимость его часа. Для этого необходимо разделить зарплату разработчика на количество рабочих часов в месяц. Принято считать, что разработчик работает 21 день в месяц (9 из 30 дней это выходные). В один день разработчик отрабатывает 8 часов. Таким образом, за один месяц человек отрабатывает $21 \text{ день} \times 8 \text{ ч/день} = 168 \text{ ч}$. Поделив средний уровень зарплаты разработчика на количество часов в месяц, мы получим стоимость часа разработчика.

Должность	Средняя зарплата, руб/мес	Средняя стоимость часа, руб/час
Старший разработчик	90 000	$90\ 000 / 168 =$ ~ 550 руб
Младший разработчик	35 000	$35\ 000 / 168 =$ ~ 200 руб
Дизайнер	35 000	$35\ 000 / 168 =$ ~ 200 руб

Для аккуратности расчетов результат деления будем округлять до пятидесяти. Таким образом оценивается стоимость часа для нанимаемых под проект сотрудников. Для текущих сотрудников вашей компании вы можете не узнавать среднюю зарплату по рынку, а делить их текущую зарплату.

Теперь, зная стоимость часа, можно рассчитать стоимость работы каждого участника проекта:

Участник	Количество часов в проекте, ч	Стоимость
Старший разработчик	8,5ч	$8,5 \times 550 = 4\ 675 \text{ руб}$
Младший разработчик	8ч	$8 \times 200 = 1\ 600 \text{ руб}$
Дизайнер	2ч	$2 \times 200 = 400 \text{ руб}$
Итого: 6 675 руб		

Расчет накладных расходов – в данном разделе сметы рассчитываются все второстепенные расходы: аренда офиса, электроэнергия, печать документации (покупка бумаги для принтера и т.п.), канцелярия, чай для сотрудников и т.п. Если для выполнения проектов необходимо купить оборудование – например, компьютеры для новых разработчиков или дополнительный сервер для нового веб-сервиса – их стоимость также закладывается в смету. Аренда помещения рассчитывается исходя из стоимости аренды офиса в день и количества календарных дней разработки.

Оплата времени разработчиков и стоимость расходов составляют **себестоимость проекта**, т.е. выполнение проекта по этой стоимости оплатит все расходы, но не принесет никакой прибыли. **Норма прибыли** – это тот процент, который добавляется к себестоимости проекта, чтобы получить прибыль. Норма прибыли варьируется в зависимости от политики компании, а также области разработки. В своих расчетах используйте норму прибыли в 15%. Также в стоимости проектов учитываются выплачиваемые налоги (на доход, социальные отчисления и т.д.), но учет данных статей в настоящем пособии рассматриваться не будет.

На основе проведенных расчетов можно составить смету выполнения проекта:

#	Статья расходов	Затраты, руб.
1	Оплата времени разработки: - Старший разработчик - Младший разработчик - Дизайнер	6 675 4 675 1 600 400
2	Накладные расходы: - Аренда помещения (3 дня по 1500 руб/день) - Печать документации - Канцелярия - Другие расходы	4 600 4 500 100 - -
3	Себестоимость (сумма пунктов 1 и 2)	11 275
4	Норма прибыли (15% от себестоимости)	1 725
		Итого: 13 000

Это пример **внутренней сметы** проекта, так как конечному заказчику не стоит показывать смету, разглашающую норму прибыли. В случае сметы для заказчика норма прибыли распределяется в другие статьи расходов, например, прибавляется к оплате времени разработки.

Таким образом, заказчику предоставляется две таблицы – календарный план и смета проекта. Диаграмма Гантта, разбивка по часам и другие таблицы могут предоставляться на ваше усмотрение для **дополнительного обоснования** стоимости и сроков проекта.

2.6.3 Пояснительная записка

Пояснительная записка – документ, описывающий внутреннюю структуру и логику работы разработанного приложения. Документ составляется разработчиками и для разработчиков: в дальнейшем, если потребуется поддержка разработанного приложения или добавление новой функциональности, пояснительная записка должна помочь разработчикам разобраться в логике приложения и сэкономить время на изучении кодовой базы.

Типовая структура пояснительной записи:

- 1) Назначение приложения.
- 2) Группы пользователей и их функциональные возможности в приложении.
- 3) Стек технологий разработки. Системные требования.
- 4) Поведенческие диаграммы.
- 5) Пользовательский интерфейс (на примере реальных данных).
- 6) Диаграммы пакетов приложения.
- 7) Диаграммы классов приложения.
- 8) Описание тестирования приложения.
- 9) Описание сборки установщика.
- 10) Описание модели ветвлений в репозитории проекта.
- 11) Приложения к пояснительной записке.

Разделы 1 и 2 заимствуются из технического задания. Если раздел 3 указан в техническом задании, он также заимствуется. Если техническое задание не указывает требуемый стек технологий, в разделе описывается выбранный вами стек технологий. Если выбор стека технологий обоснован какими-то неочевидными требованиями, их необходимо прописать в данной разделе – это убережет других разработчиков от ошибок при смене/обновлении стека технологий. Пример неочевидных требований:

«Для проекта был выбран язык Си++, так как Java и C# не обеспечивают требуемой производительности при работе с библиотекой OpenGL»

«Для тестирования приложения была выбрана библиотека NUnit версии 2.6.4 вместо версии 3.6.1, так как в версии 3.6.1 нет поддержки свойства ExpectedException в атрибуте TestCase».

«Выбрана версия .NET Framework 4.0 вместо более новой для поддержки приложения в операционных системах Windows XP».

Именно такие нюансы могут быть неочевидны для сторонних разработчиков. Предупреждение о причинах выбора конкретных версий продуктов сэкономит время другим разработчикам. Если же конкретных причин выбора той или иной версии или технологии нет, придумывать лишнего не надо.

Поведенческие диаграммы. В данном разделе приводятся блок-схемы работы пользователя в приложении, диаграммы вариантов использования или просто текстовое описание сценариев работы с приложением. О составлении блок-схем и диаграмм вариантов использования можно прочитать в [1].

В рамках предлагаемых лабораторных работ сложность приложений не высока, по этой причине данный пункт в пояснительной записке к лабораторным работам можно пропустить.

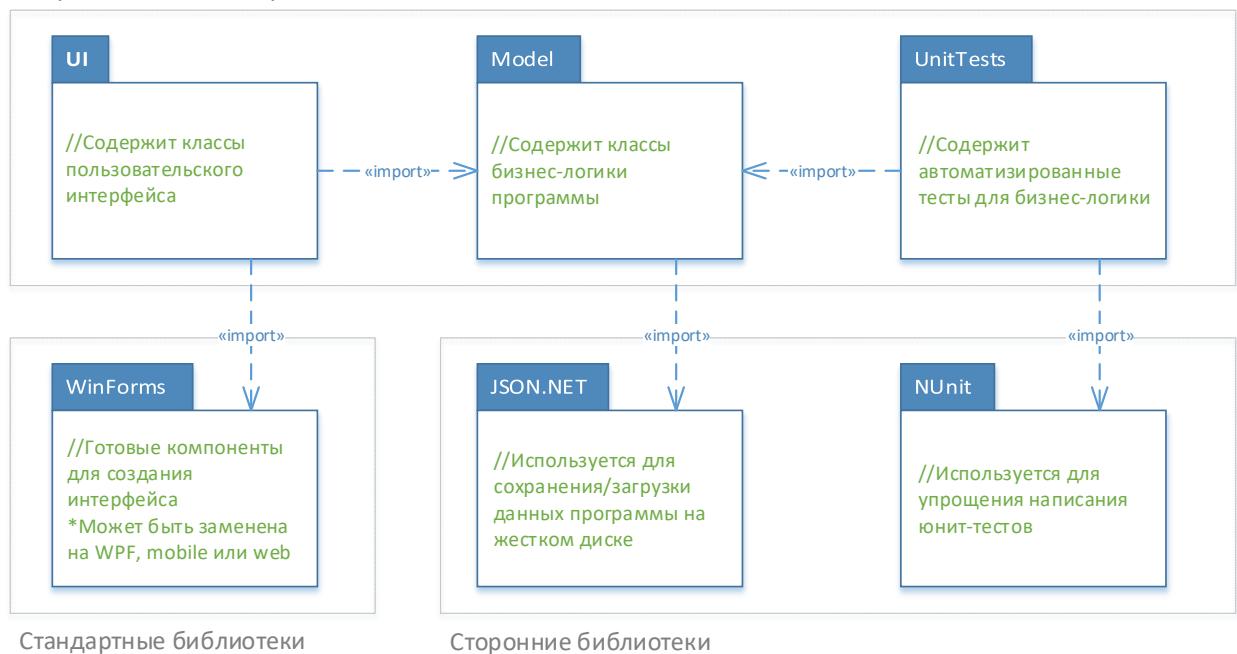
Пользовательский интерфейс. Если в техническом задании описана работа с пользовательским интерфейсом приложения, то данный раздел копируется из технического задания. Если описания в техническом задании нет, и разработка пользовательского интерфейса являлась задачей проекта, необходимо привести примеры всех страниц приложения с текстовым пояснением о назначении полей интерфейса. Также в виде текста необходимо описать, как именно пользователь будет решать задачи, описанные в разделе 1.

В раздел вставляются примеры разработанного интерфейса, но ни в коем случае не макетов. Это объясняется тем, что макеты интерфейса визуально могут отличаться от реального интерфейса и это разница может ввести в заблуждение других разработчиков.

Примеры разработанного интерфейса стоит приводить с реальными данными в окнах, так как это улучшает понимание работы с пользовательским интерфейсом.

Диаграмма пакетов. Диаграмма пакетов отображает архитектуру приложения, разделенную на отдельные пакеты – библиотеки. Внутри библиотек описываются доступные извне классы, между пакетами рисуются направленные линии, обозначающие связи между библиотеками. Пример краткой диаграммы пакетов:

Разрабатываемые сборки



Это пример краткой диаграммы пакетов – она описывает только связи между пакетами, но не описывают их содержимое. В пояснительной записке к лабораторным работам необходимо привести **полную диаграмму пакетов**, где внутри каждого пакета будут перечислены используемые классы. Обратите внимание, что на диаграмме пакетов также рисуются сторонние библиотеки, для которых также необходимо указать те классы, которые вы использовали в своём приложении.

Составление диаграмм пакетов описано в [1].

Диаграммы классов. Для проектов бизнес-логики и проектов пользовательского интерфейса составляются диаграммы классов. Важным правилом при составлении диаграмм классов является то, что не нужно их «перегружать» - отображайте не более 10-12 классов на одной диаграмме, если это возможно. Так, если в проекте бизнес-логики используется порядка 20 классов, то стоит подумать о разделении диаграммы классов на две – по двум отдельным диаграммам будет проще изучить работу приложения, чем по одной сложной диаграмме.

Помните, цель диаграмм не просто показать архитектуру приложения в виде схемы, а объяснить разработчику принцип работы приложения в понятной иллюстративной форме. Если ваша диаграмма не понятна другим разработчикам – эта диаграмма бесполезна

Для лабораторных работ UML-диаграмма классов составлялась во второй лабораторной работе, посвященной бизнес-логике. При составлении диаграммы классов для пояснительной записи необходимо только актуализировать диаграмму из лабораторной работы №2, указав на ней дополнительные функции, реализованные в лабораторной работе №5, а также указав классы из сторонних библиотек, которые вы используете. Диаграмму классов для проекта пользовательского интерфейса в пояснительной записи представлять не надо.

Составление диаграмм классов описано в п. 2.2.1 и [1].

Описание тестирования приложения. Данный раздел описывает, как производилось тестирование приложения: какие функции подлежат тестированию, какие функции тестируются вручную, какая часть кода тестируется автоматически, степень покрытия кода

тестами. В данном разделе также можно привести описание приёмочного тестирования как общего системного теста, проверяющего работоспособность всего приложения.

Описание сборки установщика. Описываются инструменты и последовательность сборки установщика. Если установщик собирается автоматически – описать, с помощью каких инструментов это обеспечивается. Если установщик собирается вручную – описать процедуру, которую нужно выполнить разработчику, чтобы собрать установщик. Важно уточнить, используется ли обfuscация кода в процессе сборки установщика, внедряется ли в приложение функции защиты и лицензирования (программные ключи, аппаратные ключи и т.д.).

Описание модели ветвления. Так как современная разработка ПО требует хранения кода под версионным контролем, в данном разделе необходимо описать используемую систему версионного контроля, дать ссылку на репозиторий. Также описываются текущие открытые ветки (при сдаче проекта ветки feature и hotfix должны быть закрыты), и правила, использовавшиеся при создании веток (так называемая модель ветвления). Если в отношении репозитория есть дополнительные правила – разграничение доступа к веткам репозитория, доступ к репозиторию со стороны тестировщиков или дизайнеров, правила составления комментариев к коммитам – всё это описывается в данной разделе.

Приложения в пояснительной записке создаются на усмотрение самих разработчиков. Например, в качестве приложения к пояснительной записке может быть прикреплен план тестирования, руководство пользователя, дополнительные диаграммы классов, не вошедшие в основную часть пояснительной записи.

2.6.4 Проведение ретроспективы

В ходе разработки любого проекта, коммерческого и бесплатного, технического или гуманитарно-социального, командного или индивидуального, всегда возникают трудности и проблемы. Каждая из проблем в разной степени влияет на процесс разработки и имеет разную цену решения. При этом проблемы далеко не всегда относятся непосредственно к разрабатываемой программе. Вот пример проблем, которые могут возникнуть в процессе разработки:

- За неделю до релиза программы заказчик попросил добавить новую функциональность в программу. Цена решения – увеличение количества часов на проект, увеличение стоимости пропорционально часам на перепроектирование, разработку и тестирование. Команда будет не очень довольна, но эта проблема не критична.

- Один из ключевых сотрудников ушёл в отпуск: цена решения – увеличение сроков выполнения проекта или увеличение нагрузки на других сотрудников. Цена решения не велика, так как отпуска сотрудников обговариваются заранее, и влияние отпуска на проект можно спрогнозировать заранее. Проблема не критична

- Один из ключевых сотрудников заболел: проблема аналогична предыдущей с той лишь разницей, что сотрудник заболевает неожиданно. Предсказать такое заранее сложно, а потому проблема более критична, чем отпуск.

- Недостаточная квалификация сотрудников: например, команда, занимавшаяся разработкой мобильных приложений под Android, взялась за разработку приложения под iOS. Здесь увеличиваются риски неправильного проектирования, неправильного планирования и оценки сроков проекта. Последствия такой проблемы предсказать сложно, так как любой «подводный камень» может остановить всю разработку на неделю.

- Плохое взаимодействие сотрудников в команде – несмотря на высокую квалификацию разработчиков в команде, они плохо общаются между собой. В результате, разработка проекта сильно замедляется.

- В офисе разработчиков сломался кондиционер. Проблема может показаться несерьезной, однако исследования показывают, что в летнюю жару без кондиционера эффективность каждого из сотрудников может упасть на 30%. А, следовательно, на 30% увеличиваются и сроки проекта. Незначительная легко устранимая проблема приводит к замедлению всей команды.

- В офисе над серверной прорвало трубу и залило серверы. Серьезная и сложно предсказуемая проблема. Если внутренняя инфраструктура разработки заранее не предусматривала создание частых копий (backup), то последствия такой проблемы могут быть фатальными для всей компании.

В любом случае, руководитель проекта и руководитель команды заинтересованы в том, чтобы минимизировать последствия любых потенциальных и возникших проблем. Для этого руководитель команды должен своевременно отслеживать проблемы и проводить специальные мероприятия. Например, аттестацию сотрудников для подтверждения их квалификации. Или профилактику оборудования в серверной. Или так называемые тим-билдинги (от англ. team-building – строительство команды).

Однако далеко не все проблемы могут быть очевидны для руководителя команды. Например, руководитель команды не сможет узнать, что вашему компьютеру не хватает мощности для работы над последним проектом, если вы сами ему об этом не скажите. Руководитель команды может не заметить, что кому-то из сотрудников не хватает квалификации по каким-либо технологиям, или то, что в команде есть проблемы с общением между собой. Для этого руководитель команды проводит так называемые ретроспективы.

Ретроспектива – мероприятие, которое проводится по окончанию проекта или его этапа, где каждый участник команды высказывает своё мнение о процессе разработки. Каждый участник по очереди должен сказать, что с его точки зрения было успешным в последнем проекте, какие проблемы есть в процессе разработки, и что стоит поменять. После того, как все участники команды, включая руководителя, выскажут своё мнение, руководитель запишет все возникшие проблемы и поручит их устранение участникам команды. Например, для повышения квалификации младших сотрудников поручит одному из старших провести учебные семинары. Или отправит сотрудников на конференцию за счет компании. Или потребует у вышестоящего руководства обновить оборудование. Решения будут зависеть от сложности проблем и доступных ресурсов.

Таким образом, ретроспективы являются важным инструментом для улучшения процесса разработки. На самом деле, многие проблемы решаются непосредственно на ретроспективе – например, проблемы с коммуникацией в команде. Когда человек может открыто высказывать свои проблемы другим членам команды, а также слышит конструктивную критику в свой адрес, вся команда сплочается и старается устранить возникшие разногласия.

Ретроспективы имеет смысл проводить даже в том случае, если вы работаете над самостоятельным проектом. Своеобразное подведение итогов позволит вам оценить собственные недостатки, найти проблемы, а также выработать план их устранения.

По этой причине, вам предлагается провести свою первую ретроспективу по завершению данного курса лабораторных работ. В своей ретроспективе по курсу лабораторных работ ответьте на следующие вопросы:

- Выполнена ли задача в полном объёме согласно ТЗ? Если задача выполнена не в полном объёме, то почему? Если в ходе разработки ТЗ частично или полностью пришлось изменить, то почему?

- Выполнена ли задача в срок? Сколько потребовалось человеко-часов на выполнение всех лабораторных, включая написание пояснительной записи? Какие задачи заняли больше всего времени и создали больше всего трудностей? Были ли это форс-мажорные трудности или этих трудностей можно было избежать? Если программа разработана не в срок, то почему? Если какие-либо этапы разработки были завершены не в срок, то почему? Сколько времени заняло написание документации (составление отчетов, рисование диаграмм) от общего времени разработки?

- Было ли ТЗ написано в достаточном для реализации объёме? Были ли найдены в ТЗ ошибки или противоречия? Достаточно ли понятна работа приложения из предоставленных макетов интерфейса?

- Были ли найдены ошибки в примечаниях от руководителя?

- Возникали какие-либо затруднения при работе со средой разработки, системой версионного контроля, редактором диаграмм? Удобна ли в использовании система версионного контроля? Удалось ли следовать индивидуальной модели ветвления при разработке?

- Возникли ли какие-либо затруднения при проведении приёмочного тестирования? Сколько раз пришлось проводить приёмочное тестирование и исправлять замечания, прежде чем заказчик принял проект? Если более одного раза, то почему? Что можно исправить или изменить, чтобы в будущем сдача проекта проходила быстрее?

- Общее заключение. Чему удалось научиться в ходе выполнения лабораторных работ? Можно ли считать проект завершенным успешно?

Данный перечень вопросов может использоваться вами при проведении ретроспектив других проектов, а также при написании заключений лабораторных работ, курсовых работ, дипломных и выпускных квалификационных работ.

2.6.5 Задание

1. Составить пояснительную записку к разработанному приложению:

1.1. Помимо разделов, описанных в п. 2.6.3, пояснительная записка должна иметь титульный лист и содержание.

2. Составить ретроспективу по результатам завершения проекта:

2.1. Документ оформляется в свободной форме, без титульного листа, с указанием ФИО студента, номера группы и даты составления, и должен содержать ответы на описанные в п. 2.6.4 вопросы.

3. Составить календарный план и смету проекта аналогичной программы:

3.1. Календарный план и смета проекта составляется с позиции разработки аналогичного проекта. Представьте, что вас попросили разработать программу, аналогичную вашему варианту, без использования ранее написанного кода. Так как вы уже знаете, как делать каждый из этапов проекта, создание аналогичной программы займет

для вас гораздо меньше времени. А, главное, вы теперь можете спланировать вашу работу в календарном плане и рассчитать её стоимость.

- 3.2. Оцените время выполнения каждого этапа (каждой лабораторной).
 - 3.3. Подсчитайте требуемое на весь проект количество часов.
 - 3.4. Составьте календарный план выполнения проекта и диаграмму Ганнта с разбивкой на этапы/лабораторные работы.
 - 3.5. Рассчитайте смету проекта.
- 4. Проверить, что задание лабораторной работы выполнено верно:**
- 4.1. Проверьте документы на полноту описания. Если вы пропустили какие-либо пункты документов, или считаете нужным дописать пункты, сделайте это.
 - 4.2. Перечитайте все три документа, которые вы составили. Проверьте наличие грамматических, орфографических, лексических ошибок.
 - 4.3. Проверьте документы на соответствие требованиям ОС ТУСУР.
 - 4.4. Если есть возможность, дайте прочитать ваши документы другому человеку, разбирающемуся в тематике разработки. Сторонний взгляд часто находит ошибки, которые не замечаете вы сами.
 - 4.5. Если все документы составлены верно, выполните печать документов, подпишите и укажите дату защиты, подготовьте вопросы для самопроверки и переходите к защите лабораторной работы.

2.6.6 Вопросы для самоподготовки

1. Какие виды технической документации существуют в процессе разработки ПО?
2. Каково назначение технического задания? Какие пункты в нём должны быть отражены? Кто ответственен за написание технического задания?
3. Как рассчитать трудозатраты на разработку ПО по техническому заданию? Как рассчитать стоимость разработки ПО? Как составить календарный план разработки?
4. Что такое ретроспектива? Как и когда она проводится? Кто в ней участвует? Кто ответственен за её проведение?
5. Каково назначение пояснительной записки? В чем её отличие от руководства пользователя? Какие пункты должны быть отражены в пояснительной записке к ПО? Как описывается программная часть?
6. В чём отличие диаграммы пакетов и диаграммы классов?

2.6.7 Содержание отчета

В качестве отчета по лабораторной работе сдаётся пояснительная записка по проекту. Также сдаче в печатной форме подлежат ретроспектива и календарный план аналогичного проекта.

Отчёт сдается с указанием даты защиты лабораторной работы и подписью студента.

2.6.8 Список источников

Проектная документация

1. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд. / Г. Буч, Д. Рамбо, И. Якобсон; пер. с англ. Мухин Н. – М.: ДМК Пресс, 2006. – 496 с.: ил.

2. Фаулер М. UML. Основы, 3-е издание. – пер. с англ. Петухов А. – СПб: Символ-Плюс, 2004. – 192 с.: ил.
3. Диаграммы пакетов в Sparx Enterprise Architect [Электронный ресурс]. / Sparx Systems. — URL: http://www.sparxsystems.com/resources/uml2_tutorial/uml2_packagediagram.html (дата обращения 27.08.2018).
4. Диаграммы классов в Sparx Enterprise Architect [Электронный ресурс]. / Sparx Systems. — URL: http://www.sparxsystems.com/resources/uml2_tutorial/uml2_classdiagram.html (дата обращения 27.08.2018).
5. [Видео] Диаграммы классов в Enterprise Architect – пошаговое руководство (Class diagram in Enterprise Architect – Step by step guide) / Sparx Systems India: Youtube-канал. — URL: <https://www.youtube.com/watch?v=SFgSHpqJq1U> (дата обращения 27.08.2018).
6. Диаграмма Ганнта [Электронный ресурс]. / Википедия: онлайн-энциклопедия. — URL: https://ru.wikipedia.org/wiki/Диаграмма_Ганнта (дата обращения 27.08.2018).
7. Что такое диаграмма Ганнта и как её составить? [Электронный ресурс]. / fb.ru: онлайн-журнал. — URL: <http://fb.ru/article/142650/diagramma-ganta---vash-pomoschnik-v-planirovaniich-toe-takoe-diagramma-ganta-i-kak-ee-sostavit> (дата обращения 27.08.2018).
8. Как диаграммы Ганта упрощают работу с проектами [Электронный ресурс]. / блог пользователя Hygger, Habr.com. — URL: <https://habr.com/company/hygger/blog/415271/> (дата обращения 27.08.2018).

Требования к оформлению отчета

9. ОС ТУСУР 01-2013. Работы студенческие по направления подготовки и специальностям технического профиля. Общие требования и правила оформления. – Томск: ТУСУР, 2013. – 57с. – URL: https://storage.tusur.ru/files/40668/rules_tech_01-2013.pdf (дата обращения 27.08.2018)

3 Варианты заданий

3.1 Записная книжка NoteApp

Назначение приложения

Пользовательское приложение NoteApp, предназначено для ведения персональных записей и заметок. Приложение должно:

- 1) Обеспечивать стабильную работу приложения при порядке 200 заметок.
- 2) Обеспечивать категоризацию заметок, навигацию по созданным заметкам.
- 3) Предоставить инструменты для просмотра и редактирования заметок.
- 4) Сохранять и восстанавливать заметки между сессиями приложения.
- 5) Выполнять промежуточные сохранения заметок на машине пользователя на случай аварийного завершения программы, отключения компьютера и т.д. – для защиты от потери данных.

Приложение-референс: десктоп-версия программы Evernote

Пользовательский интерфейс

(лабораторная работа «Разработка пользовательского интерфейса»)

После запуска приложения перед пользователем появляется главное окно (рис. 71). Двухколоочная верстка главного окна содержит список всех созданных заметок в левой панели и отображает текущую выбранную заметку в правой панели. В списке заметок показаны названия заметок, в один момент времени может быть выбрана только одна заметка (далее – текущая заметка).

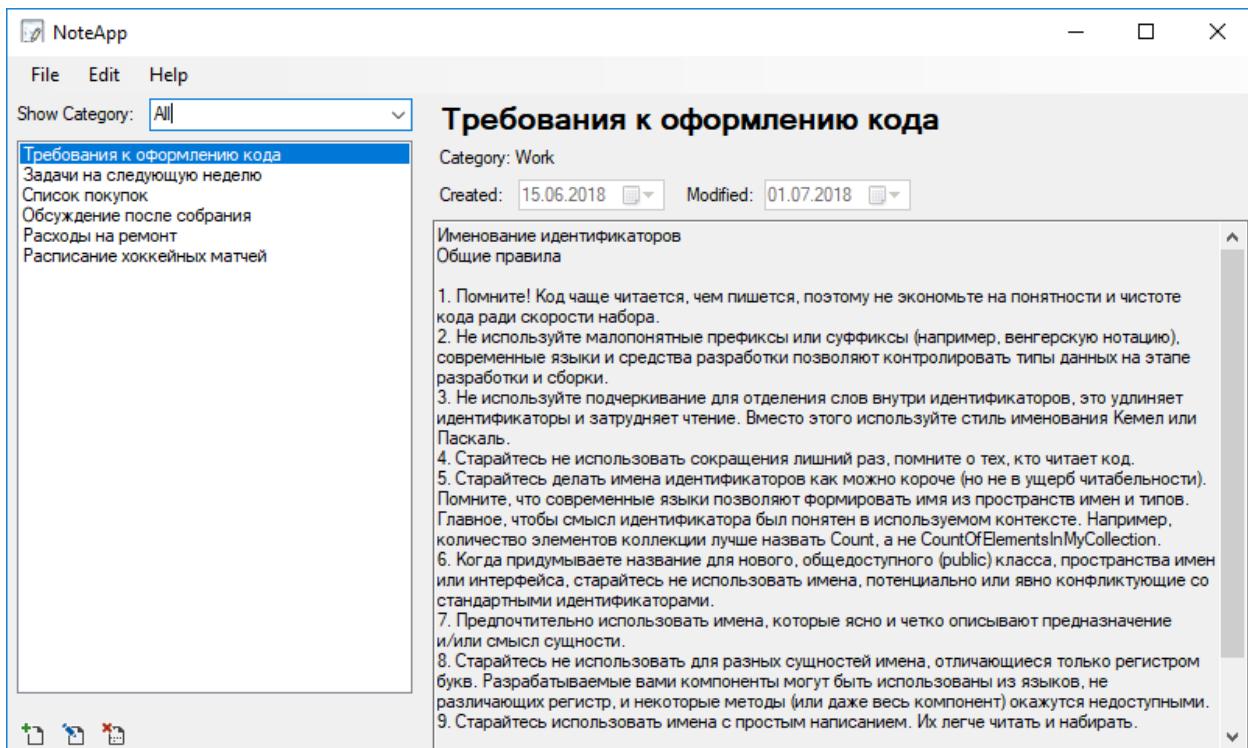


Рисунок 71 – Макет главного окна приложения NoteApp

На панели со списком заметок внизу располагаются три кнопки в виде пиктограмм: Add Note («Создать новую заметку»), Edit Note («Редактировать текущую заметку»), Remove Note («Удалить текущую заметку»).

При выборе заметки в списке, выбранная заметка отображается в правой панели. Главное окно не позволяет редактировать содержимое заметки – только просмотр.

При нажатии на кнопку Add Note и Edit Note появляется окно создания/редактирования заметки в диалоговом режиме (рис. 72). Для новой заметки окно изначально не заполнено (Установлены лишь название заметки по умолчанию, дата создания и дата редактирования). Для редактирования уже существующей заметки все поля должны быть предзаполнены данными текущей заметки.

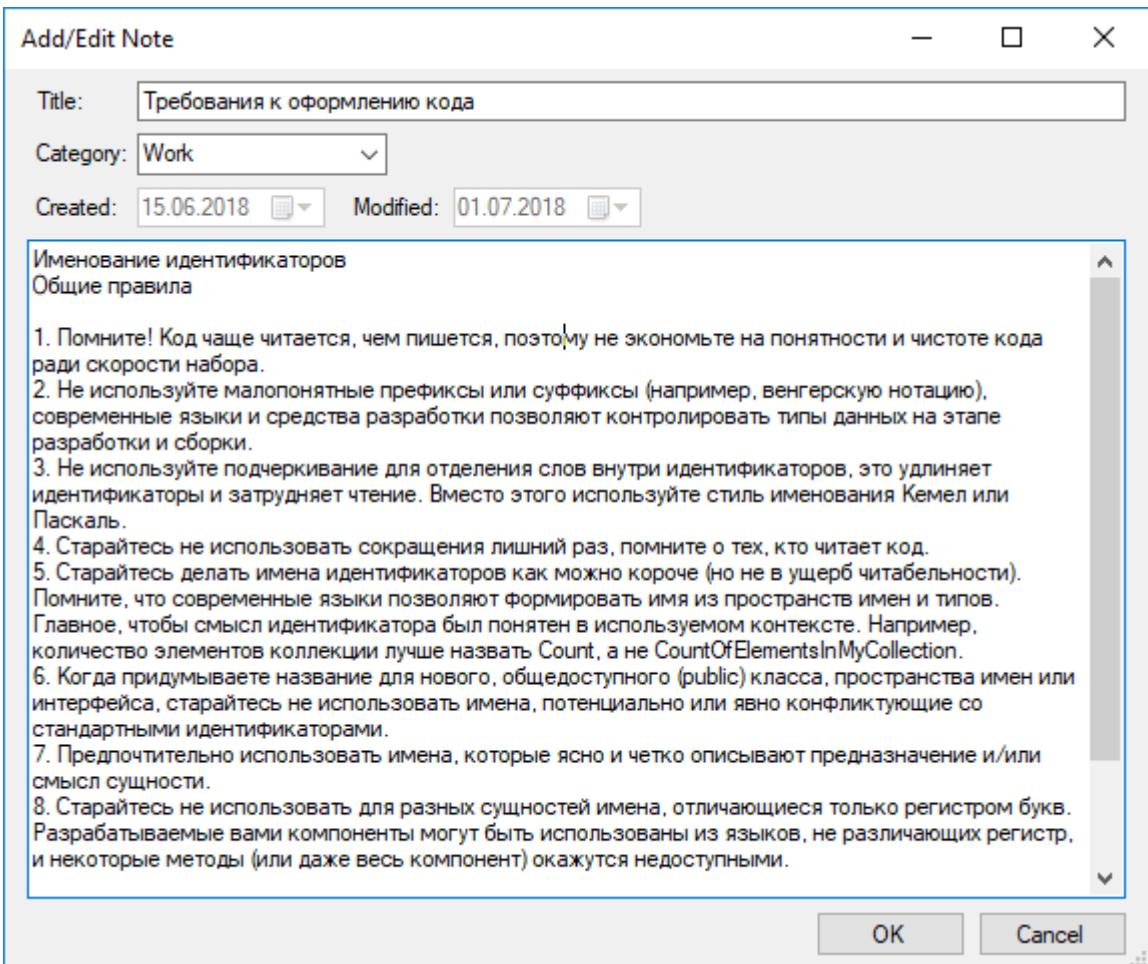


Рисунок 72 – Макет окна редактирования заметки в приложении NoteApp

При нажатии на кнопку OK окно создания заметки закрывается, в список заметок главного окна добавляется новая заметка. При редактировании текущей заметки, нажатие на кнопку OK должно обновить название заметки в списке заметок (если название текущей заметки было изменено), и обновить отображаемую заметку в правой панели приложения. При нажатии кнопки Cancel создание/редактирование заметки отменяется (новая заметка не добавляется, исходная заметка остается без изменений). Реализация передачи данных между двумя окнами см. п. 2.3.5 «Передача данных между формами».

В случае ввода пользователем некорректных данных (нарушение допустимой длины названия заметки), данная ситуация должна быть обработана соответствующим образом (см. п. 2.3.2 «Получение данных со стандартных элементов управления»).

При нажатии на кнопку Remove Note главного окна текущая запись удаляется. Перед удалением должно появиться окно с запросом на разрешение записи: «Do you really want to remove this note: <Название текущей записи>». При нажатии на кнопку OK происходит удаление, при нажатии на кнопку Cancel удаление отменяется.

Меню главного окна содержит следующие пункты:

- *File*:
 - *Exit* (Выйти из приложения – Alt+F4)
- *Edit*:
 - *Add Note* (Создать новую заметку)
 - *Edit Note* (Редактировать текущую заметку)
 - *Remove Note* (Удалить текущую заметку)
- *Help*
 - *About* (Вызвать окно «О программе» - F1) (см. рис. 73)

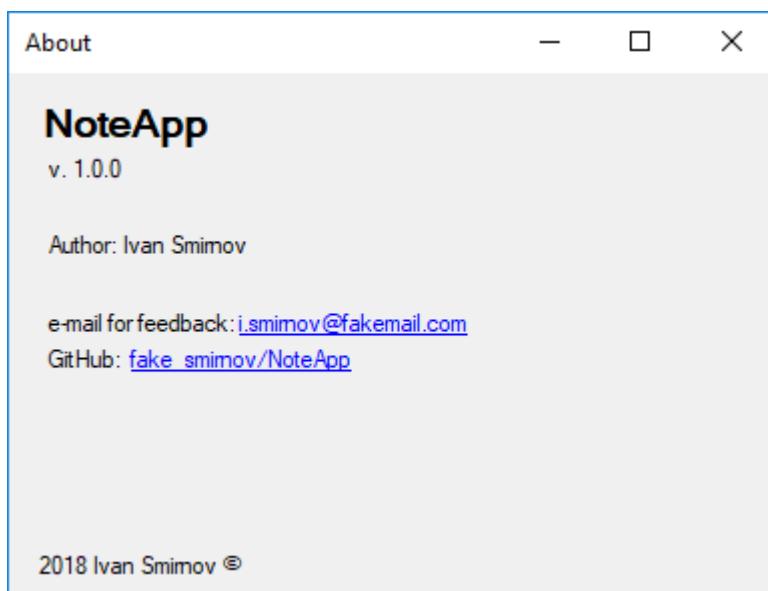


Рисунок 73 – Макет окна «About» приложения NoteApp

Таким образом, в программе должно быть реализовано три окна:

- Главное окно.
- Окно создания/редактирования заметки.
- Окно «About».

Верстка главного окна и окна создания/редактирования заметки должна быть адаптивной. Окно «About» имеет фиксированный размер.

Создание дополнительных элементов управления для уменьшения дублирования кода остается на усмотрение разработчика или руководителя.

Загрузка заметок осуществляется *при запуске программы* до вывода главного окна пользователю, сохранение заметок в файл должно выполняться в случаях: а) создания новой заметки; б) удаления заметки; в) закрытии приложения.

Допущения первой версии программы:

**В первой версии программы заметки в списке отображаются в порядке их создания или ином произвольном порядке.*

**В первой версии программы при запуске приложения текущей записью может быть первая запись в списке.*

**В первой версии программы необязательно реализовывать логику выпадающего списка с выбором категории заметок.*

**Все описанные допущения будут устранены в лабораторной работе «Расширение функциональности и релиз программы» (см. п. «Дополнительная функциональность»).*

Требования к бизнес-логике

(лабораторная работа «Разработка бизнес-логики приложения»)

Необходимо реализовать следующие типы данных:

- Перечисление «Категория заметки», содержащее значения «Работа», «Дом», «Здоровье и Спорт», «Люди», «Документы», «Финансы», «Разное».

- Класс «Заметка» с полями «Название», «Категория заметки», «Текст заметки», «Время создания», «Время последнего изменения». Название, категория и текст заметки доступны для изменений. Время создания инициализируется один раз при создании объекта «Заметка» и больше не модифицируется, доступна для чтения. Время последнего изменения меняется автоматически при изменении названия, категории или текста заметки. Название ограничено 50 символами. Название по умолчанию «Без названия». Допустимы заметки с одинаковыми названиями. Реализует интерфейс ICloneable.

- Класс «Проект». Содержит список(или словарь) всех заметок, созданных в приложении.

- Класс «Менеджер проекта». Реализует метод для сохранения объекта «Проект» в файл и метод загрузки проекта из файла. Сохранение и загрузка осуществляются в один и тот же файл «...\\My Documents\\NoteApp.notes», имя которого задано закрытой константой внутри класса. Формат данных – json, библиотека сериализации (преобразования данных в json-формат) – Newtonsoft JSON.NET (подробнее см. п. 2.2).

Дополнительная функциональность

(лабораторная работа «Расширение функциональности и релиз программы»)

- В класс «Проект» добавить метод, который возвращает список заметок, отсортированный по дате изменения. Добавить перегруженный метод сортировки заметок, принимающий на вход категорию заметок – метод возвращает отсортированный по дате изменения список заметок, принадлежащих только этой категории. В случае отсутствия заметок данной категории, метод возвращает пустой список.

- В левой панели главного окна обеспечить сортировку списка заметок по дате изменения. Таким образом, последние измененные записи должны находиться вверху списка, а записи с более ранними датами изменения – внизу.

- В левой панели главного окна над списком заметок реализовать выпадающий список категорий заметок. При выборе пользователем определенной категории, в списке заметок должны отображаться только заметки выбранной категории. При выборе пункта «All» отобразить все заметки.

- В класс «Проект» добавить свойство «Текущая заметка». Свойство текущей заметки меняется в случае просмотра пользователем какой-либо заметки в программе. Значение свойства должно также сохраняться в файл. При запуске программы значение свойства должно загружаться из файла, и пользователю в главном окне должна отобразиться последняя просмотренная им заметка.

- Реализовать удаление текущей заметки по нажатию на клавишу Delete. Перед удалением должно появиться окно с запросом на разрешение записи.

Приёмочное тестирование

Для приёмочного тестирования выполните следующую последовательность действий:

1. Установите приложение на компьютер с помощью собранного установочного пакета.
2. Запустите приложение. Окно программы должно быть пустым – в приложении не должно быть заметок.
3. Создайте три заметки в приложении разных категорий.
4. Переключитесь между заметками, показав, что смена текущей заметки происходит корректно.
5. Переключите отображаемую категорию заметок – в списке заметок должны остаться только заметки целевой категории. Снова отобразите все категории заметок – список заметок должен восстановиться.
6. Выберите вторую заметку и нажмите кнопку редактирования. Должно открыться окно редактирования заметки.
7. Введите название заметки более 50 символов. Элемент управления с названием заметки должен указать на некорректное значение.
8. Введите название заметки менее 50 символов. Элемент управления с названием должен стать корректным.
9. Поменяйте текст заметки. Нажмите «OK». Отредактированная заметка должна подняться в списке заметок на первую позицию, время изменения заметки должно поменяться, отображаемый текст заметки также должен измениться.
10. Выберите вторую заметку и нажмите кнопку редактирования. Должно открыться окно редактирования. Измените название заметки, её текст и категорию. Нажмите «Cancel». Исходная заметка должна остаться без изменений.
11. Удалите третью заметку.
12. Закройте приложение. Должно произойти сохранение заметок в целевой файл.
13. Запустите приложение. В программе должны восстановиться заметки, созданные в предыдущую сессию.
14. Дайте руководителю провести исследовательское тестирование программы.

Варианты развития программы (по желанию студента, вне учебного курса дисциплины или для повышения оценки за пройденный курс)

* Создать элемент управления NoteControl для просмотра и редактирования заметок. Созданный элемент управления должен использоваться в правой панели главного окна и в окне создания/редактирования заметки. У элемента управления должно быть булево свойство ReadOnly, с помощью которого задается режим работы элемента управления (режим просмотра или режим редактирования заметки). Для удобства работы с элементом управления, элемент должен иметь свойство типа Note (класс «Заметка»), через которое

* Переделать текст записи на формат RichText вместо обычного текста string. Преимущество RichText по сравнению с обычным текстом – он может хранить рисунки, таблицы, разные шрифты и т.д. Для работы с RichText уже существует готовый элемент управления

RichTextBox. Разберитесь с данным элементом управления и переделайте записи под формат RichText.

* Добавьте в программу функцию экспорта текущей записи в doc-файл – найти библиотеку для работы с doc-файлами предлагается самостоятельно.

* Добавьте в программу функцию печати текущей записи – для работы с принтером на платформе .NET существуют готовые классы.

3.2 Контакты ContactsApp

Назначение приложения

Пользовательское приложение ContactsApp, предназначено для ведения и хранения контактов. Приложение должно:

- 1) Обеспечивать стабильную работу приложения при порядке 200 контактов.
- 2) Обеспечивать поиск, навигацию по созданным контактам по фамилии и имени.
- 3) Предоставить инструменты для просмотра и редактирования контактов.
- 4) Сохранять и восстанавливать контакты между сессиями приложения.
- 5) Выполнять промежуточные сохранения контактов на машине пользователя на случай аварийного завершения программы, отключения компьютера и т.д. – для защиты от потери данных.

Приложение-референс: контакты мобильного телефона

Пользовательский интерфейс

(лабораторная работа «Разработка пользовательского интерфейса»)

После запуска приложения перед пользователем появляется главное окно (рис. 74). Двухколоночная верстка главного окна содержит список всех контактов в левой панели и отображает текущий выбранный контакт в правой панели. В списке контактов показаны фамилии контактов, в один момент времени может быть выбран только один контакт (далее – текущий контакт).

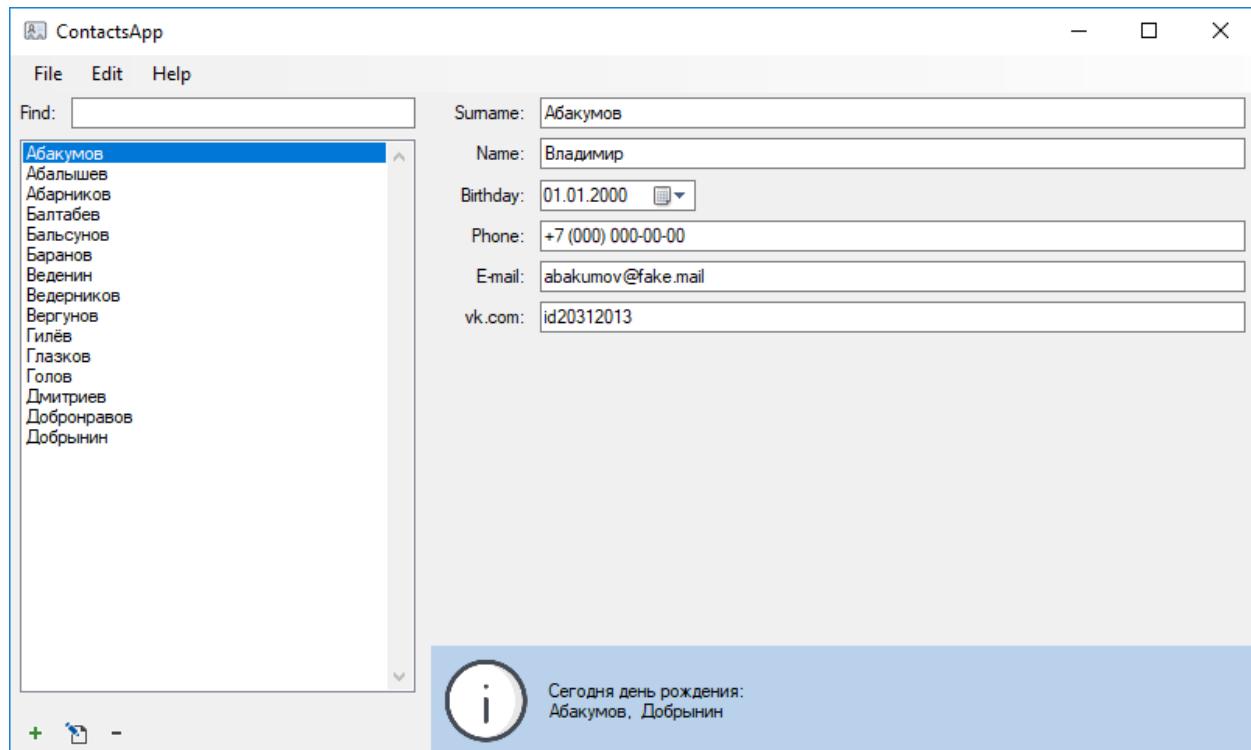


Рисунок 74 – Макет главного окна приложения ContactsApp

На панели со списком контактов внизу располагаются три кнопки в виде пиктограмм: Add Contact («Создать новый контакт»), Edit Contact («Редактировать текущий контакт»), Remove Contact («Удалить текущий контакт»).

При выборе контакта в списке, выбранный контакт отображается в правой панели. Главное окно не позволяет редактировать содержимое контакта – только просмотр. Также в правой панели под текущим контактом отображается информационная панель с сегодняшними именинниками (см. рис. 74).

При нажатии на кнопку Add Contact и Edit Contact появляется окно создания/редактирования контакта в диалоговом режиме (рис. 75). Для нового контакта окно изначально не заполнено (установлена лишь дата рождения по умолчанию). Для редактирования уже существующего контакта все поля должны быть предзаполнены данными текущего контакта.

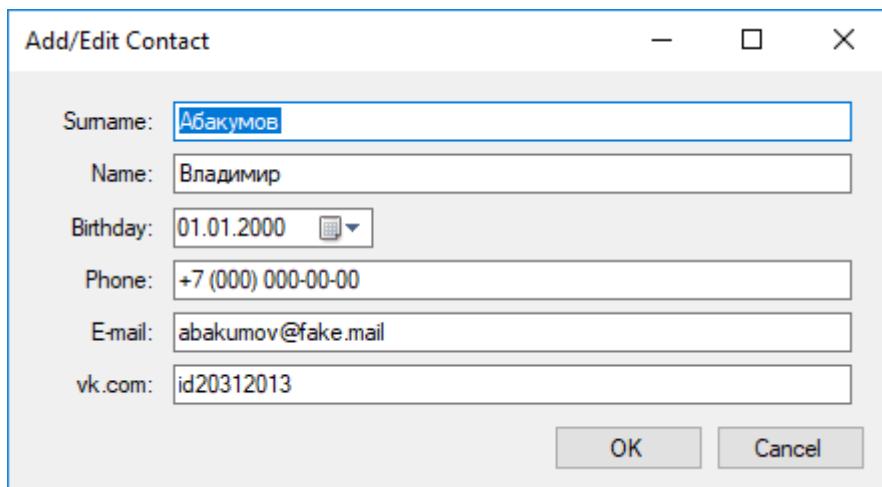


Рисунок 75 – Макет окна редактирования контакта в приложении ContactsApp

При нажатии на кнопку OK окно создания контакта закрывается, в список контактов главного окна добавляется новый контакт. При редактировании текущей контакта, нажатие на кнопку OK должно обновить фамилию контакта в списке контактов (если фамилия текущего контакта была изменена/исправлена), и обновить отображаемый контакт в правой панели приложения. При нажатии кнопки Cancel создание/редактирование контакта отменяется (новый контакт не добавляется, исходный контакт остается без изменений). Реализация передачи данных между двумя окнами см. п. п. 2.3.5 «Передача данных между формами».

В случае ввода пользователем некорректных данных (нарушение допустимой длины фамилии, имени, указание невозможной даты рождения или неправильного номера телефона), данная ситуация должна быть обработана соответствующим образом (см. п. 2.3.2 «Получение данных со стандартных элементов управления»).

При нажатии на кнопку Remove Contact главного окна текущий контакт удаляется. Перед удалением должно появиться окно с запросом на разрешение записи: «Do you really want to remove this contact: <Фамилия текущего контакта>». При нажатии на кнопку OK происходит удаление, при нажатии на кнопку Cancel удаление отменяется.

Меню главного окна содержит следующие пункты:

- *File:*
 - *Exit* (Выйти из приложения – Alt+F4)
- *Edit:*
 - *Add Contact* (Создать новый контакт)
 - *Edit Contact* (Редактировать текущий контакт)
 - *Remove Contact* (Удалить текущий контакт)

- *Help*
 - *About* (Вызвать окно «О программе» - F1) (см. рис. 76)

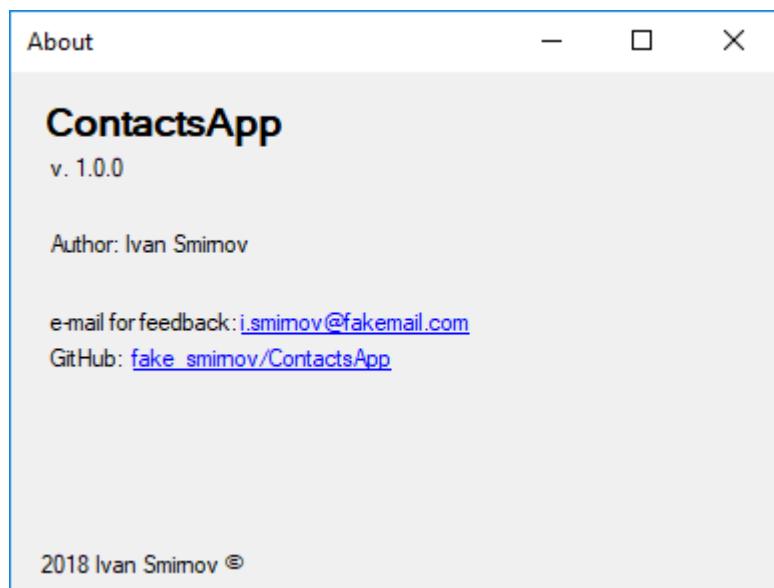


Рисунок 76 – Макет окна «About» приложения ContactsApp

Таким образом, в программе должно быть реализовано три окна:

- Главное окно.
- Окно создания/редактирования контакта.
- Окно «About».

Верстка главного окна и окна создания/редактирования контакта должна быть адаптивной. Окно «About» имеет фиксированный размер.

Создание дополнительных элементов управления для уменьшения дублирования кода остается на усмотрение разработчика или руководителя.

Загрузка контактов осуществляется *при запуске программы* до вывода главного окна пользователю, сохранение контактов в файл должно выполняться в случаях: а) создания нового контакта; б) удаления контакта; в) закрытии приложения.

Допущения первой версии программы:

**В первой версии программы контакты в списке отображаются в порядке их создания или ином произвольном порядке (сортировка по алфавиту в первой версии программы необязательна)*

**В первой версии программы необязательно реализовывать логику выпадающего списка с выбором категории заметок (поиск по имени/фамилии реализуется в дополнительной функциональности).*

**В первой версии программы может не реализовываться информационная панель с фамилиями именинников.*

**Все описанные допущения будут устраниены в лабораторной работе «Расширение функциональности и релиз программы» (см. п. «Дополнительная функциональность»).*

Требования к бизнес-логике

(лабораторная работа «Разработка бизнес-логики приложения»)

Необходимо реализовать следующие типы данных:

- Класс «Номер телефона» с полем «Номер». Поле должно быть числовым и содержать ровно 11 цифр. Первая цифра должна быть ‘7’ (только Российские телефонные номера)

- Класс «Контакт» с полями «Фамилия», «Имя», «Номер телефона», «Дата рождения», «e-mail», «ID ВКонтакте». Все поля доступны для изменений. Фамилия, имя и e-mail ограничены 50 символами каждое, ID Вконтакте ограничен 15 символами. Первая буква в фамилии и имени должна преобразовываться к верхнему регистру. Дата рождения не может быть более текущей даты и не может быть менее 1900 года. Допустимы контакты с одинаковыми фамилиями и именами. Реализует интерфейс ICloneable.

- Класс «Проект». Содержит список(или словарь) всех контактов, созданных в приложении.

- Класс «Менеджер проекта». Реализует метод для сохранения объекта «Проект» в файл и метод загрузки проекта из файла. Сохранение и загрузка осуществляются в один и тот же файл «...\\My Documents\\ContactsApp.notes», имя которого задано закрытой константой внутри класса. Формат данных – json, библиотека сериализации (преобразования данных в json-формат) – Newtonsoft JSON.NET (подробнее см. п. 2.2).

*Возможно, вы обратили внимание, что класс «Номер телефона» содержит только одно поле. Может возникнуть сомнения в целесообразности выделения класса с единственным полем. Однако очевидным расширением данного класса является разделение номера телефона на такие составляющие как «Код страны», «Код города», «Локальная часть номера», а также перечисления «Тип номера» («Мобильный», «Рабочий», «Домашний», «Основной» и т.д.). Таким образом, предполагая появление новых полей в классе, для простоты дальнейшего расширения программы разумно заранее вынести номер телефона в отдельный класс, даже если в первой версии программы предполагается только одно поле в классе.

Дополнительная функциональность

(лабораторная работа «Расширение функциональности и релиз программы»)

- В класс «Проект» добавить метод, который возвращает список контактов, отсортированных по фамилии. Добавить перегруженный метод сортировки контактов, принимающий на вход подстроку – метод возвращает отсортированный по алфавиту список контактов, фамилия или имя которых содержит указанную подстроку. В случае отсутствия контактов с данной подстрокой, метод возвращает пустой список.

- В левой панели главного окна обеспечить сортировку списка контактов по алфавиту.

- В левой панели главного окна над списком контактов реализовать текстовое поле для поиска контактов по подстроке. При вводе пользователем подстроки, в списке контактов должны отображаться только контакты, фамилия и имя которых содержат данную подстроку. Если из текстового поля стереть подстроку, то в панели должны отображаться все существующие контакты.

- В правой панели добавить информационную панель с именинниками. Для этого в классе «Проект» добавить метод, который принимает в качестве аргумента дату (экземпляр стандартного класса DateTime), и возвращает список всех контактов, в которых дата рождения (день и месяц, но не год!) совпадает с датой во входном аргументе. Главное окно должно вызывать метод проекта с текущей датой, и отображать на панели список фамилий

найденных именинников. Для упрощения реализации панель с именинниками должна инициализироваться только один раз в момент запуска приложения. Обновлять панель с именинниками при создании/удалении/редактировании контактов не требуется.

- Реализовать удаление текущего контакта по нажатию на клавишу Delete. Перед удалением должно появиться окно с запросом на разрешение удаление контакта.

Приёмочное тестирование

Для приёмочного тестирования выполните следующую последовательность действий:

1. Установите приложение на компьютер с помощью собранного установочного пакета.
2. Запустите приложение. Окно программы должно быть пустым – в приложении не должно быть контактов.
3. Создайте три контакта в приложении с разными фамилиями.
4. Переключитесь между контактами, показав, что смена текущего контакта в правой панели происходит корректно.
5. Введите в поиск подстроку для поиска контактов – в списке контактов должны оставаться только контакты, содержащие подстроку.
6. Введите в поиск подстроку, которой нет в фамилиях и именах контактов – список контактов должен быть пустым. Сотрите подстроку поиска – список контактов должен восстановиться.
7. Выберите любой контакт и нажмите кнопку редактирования. Должно открыться окно редактирования контакта.
8. Введите фамилию более 50 символов. Элемент управления с фамилией должен указать на некорректное значение.
9. Введите фамилию менее 50 символов. Элемент управления с фамилией должен стать корректным.
10. Покажите, что защита от некорректных значений также работает и для других полей.
11. Поменяйте фамилию контакта, отличную от исходной. Поменяйте номер телефона. Нажмите «OK». Отредактированный контакт должен переместиться в списке контактов согласно алфавиту, отображаемые данные текущего контакта в правой панели также должны измениться.
12. Выберите любой контакт и нажмите кнопку редактирования. Должно открыться окно редактирования. Измените фамилию контакта, номер телефона и e-mail. Нажмите «Cancel». Исходный контакт должен остаться без изменений.
13. Удалите третий контакт.
14. Закройте приложение. Должно произойти сохранение контактов в целевой файл.
15. Запустите приложение. В программе должны восстановиться контакты, созданные в предыдущую сессию.
16. Дайте руководителю провести исследовательское тестирование программы.

Варианты развития программы (по желанию студента, вне учебного курса дисциплины или для повышения оценки за пройденный курс)

* Создать элемент управления ContactControl для просмотра и редактирования заметок. Созданный элемент управления должен использоваться в правой панели главного окна и в окне создания/редактирования заметки. У элемента управления должно быть булево свойство ReadOnly, с помощью которого задается режим работы элемента управления (режим просмотра или режим редактирования контакта). Для удобства работы с элементом управления, элемент должен иметь свойство типа Contact (класс «Заметка»), через которое осуществляется передача данных.

* Добавить поле «Аватар» в контакт, для которого пользователь сможет подгрузить фотографию пользователя в формате jpg и png. Загруженные фото контактов должны копироваться в локальную папку приложения (чтобы обеспечить сохранность фото в случае удаления фото из исходной папки), контакт хранит относительный путь к фото в локальной папке. Файл с фото перезаписывается в случае обновления фотографии.

* Реализовать возможность добавления нефиксированного количества номеров телефонов с пометками «рабочий», «мобильный», «домашний», «основной» и т.д.. Реализовать возможность добавления нефиксированного количества социальных сетей.

* Добавьте в программу функцию экспорта текущего контакта в формат vCard (*.vcf – описание формата можно найти в Интернете) – найти библиотеку для работы с vcf-файлами предлагается самостоятельно. При отсутствии библиотеки реализовать собственный класс, занимающийся конвертированием контакта в требуемый формат.

4 Техники разработки программного обеспечения

4.1 Системы версионного контроля

4.1.1 Работа с репозиторием через консоль

Для работы с помощью командной строки необходимо установить Git клиент, предварительно скачав его с [3]. После установки появится возможность использовать команды для настройки Git с помощью командной строки.

Для запуска командной строки запустите Git Bash из меню Пуск или sh.exe расположенный по пути C:\Program Files (x86)\Git\bin\. Для создания Git репозитория перейдите в папку, в которой вы хотите создать репозиторий и введите следующую команду:

```
$ git init
```

Создав репозиторий в папке – добавьте в неё некоторые файлы (например, файл README.txt). Для добавления этого файла под версионный контроль – наберите команду:

```
$ git add README.txt
```

или

```
$ git add *.txt
```

Для выполнения своего первого коммита, наберите команду:

```
$ git commit -m 'Первый коммит проекта'
```

Флаг -m описывает сообщение, которое будет содержать коммит.

Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда \$ git clone. Каждая такая команда забирает с сервера копию практически всех данных, что есть на сервере. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования. Для клонирования репозитория нужно набрать команду:

```
$ git clone [url]
```

Сейчас измените файл README.txt. После этого можно проверить статус имеющихся файлов командой

```
$ git status
```

После выполнения команды вы увидите, что файл README.txt изменился. Для того, чтобы выполнить коммит, снова выполните команду коммита, только с флагом -a:

```
$ git commit -a -m 'Второй коммит проекта'
```

Флаг -a позволит выполнить коммит модифицированных файлов без необходимости добавления их командой \$ git add.

Очень часто необходимо иметь возможность автоматического игнорирования одного или группы файлов. При этом эти файлы хотелось бы видеть в списке отслеживаемых. Для этого вы можете создать файл .gitignore с перечислением шаблонов соответствующих таким файлам. Такие файлы можно написать вручную либо можно найти в интернете для интересующего вас типа проекта. Для C# .NET проекта файл можно скачать тут [4].

Для того чтобы удалить файл из Git, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда \$ git rm, которая также удаляет файл из вашего рабочего каталога, так что вы в следующий раз не увидите его как «не отслеживаемый».

Другой полезной командой, которую можно выполнить – это удалить файл из индекса, оставив его при этом в вашем рабочем каталоге. Другими словами, вы можете захотеть оставить файл на жёстком диске, и убрать его из-под бдительного ока Git-а. Для этого выполните команду:

```
$ git rm --cached README.txt
```

Если вам будет необходимо просмотреть историю коммитов – наберите:

```
$ git log
```

После выполнения команды будет выведен список коммитов, созданных в данном репозитории в обратном хронологическом порядке. Эта команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем и электронной почтой автора, датой создания и комментарием.

Существует великое множество параметров команды `git log` и их комбинаций, для того чтобы показать вам именно то, что вы ищите. Со параметрами команды предлагается разобраться самостоятельно.

`-p -2` – показ дельты между коммитами для последних двух записей;

`--stat` – показ краткой статистики по каждому коммиту;

`--pretty=<oneline, short, full, fuller>` – команда для изменения параметров лога (в скобках перечислены основные опции вывода)

`--pretty=format: "%h - %an, %ar : %s"` – команда для создания собственного формата вывода лога, которая может быть полезна, когда вы создаёте отчёты для автоматического разбора (парсинга).

`--since` и `--until =2.weeks` – команда для ограничения вывода по времени, может быть за последние несколько дней, недель, месяцев, а также часов, минут, секунд и пр.

На любой стадии может возникнуть необходимость что-либо отменить. Будьте осторожны, ибо не всегда можно отменить сами отмены. Это одно из немногих мест в Git, где вы можете потерять свою работу если сделаете что-то неправильно.

Изучите работу с командой `$ git commit --amend` самостоятельно.

Git поддерживает большое количество команд, поэтому при возникновении вопросов – обязательно обращайтесь к рабочей документации, набрав команду `$ git help`

или для открытия html файла с помощью

```
$ git help git
```

Помимо этого, можно получить помощь на конкретную команду набрав:

```
$ git help <command>
```

Работа с ветками будет рассмотрена в следующем разделе, т.к. её наиболее удобно показать на программе, имеющей пользовательский интерфейс. Желающие разобраться с процессом работы с ветками с помощью командной строки могут воспользоваться информацией из Интернета.

4.1.2 Работа с репозиторием сторонними приложениями

Для удобства работы с СУВ создаются специальные программы. Данный раздел указаний по лабораторным работам посвящён утилите SourceTree. Выбор пал именно на эту программу, т.к. она бесплатна и, с точки зрения авторов, наиболее удобна из исследованных. Скачать её можно по ссылке [5]. Перечень других GUI клиентов для Git можно посмотреть по ссылке [6].

После установки SourceTree и запуска программы будет показано окно:

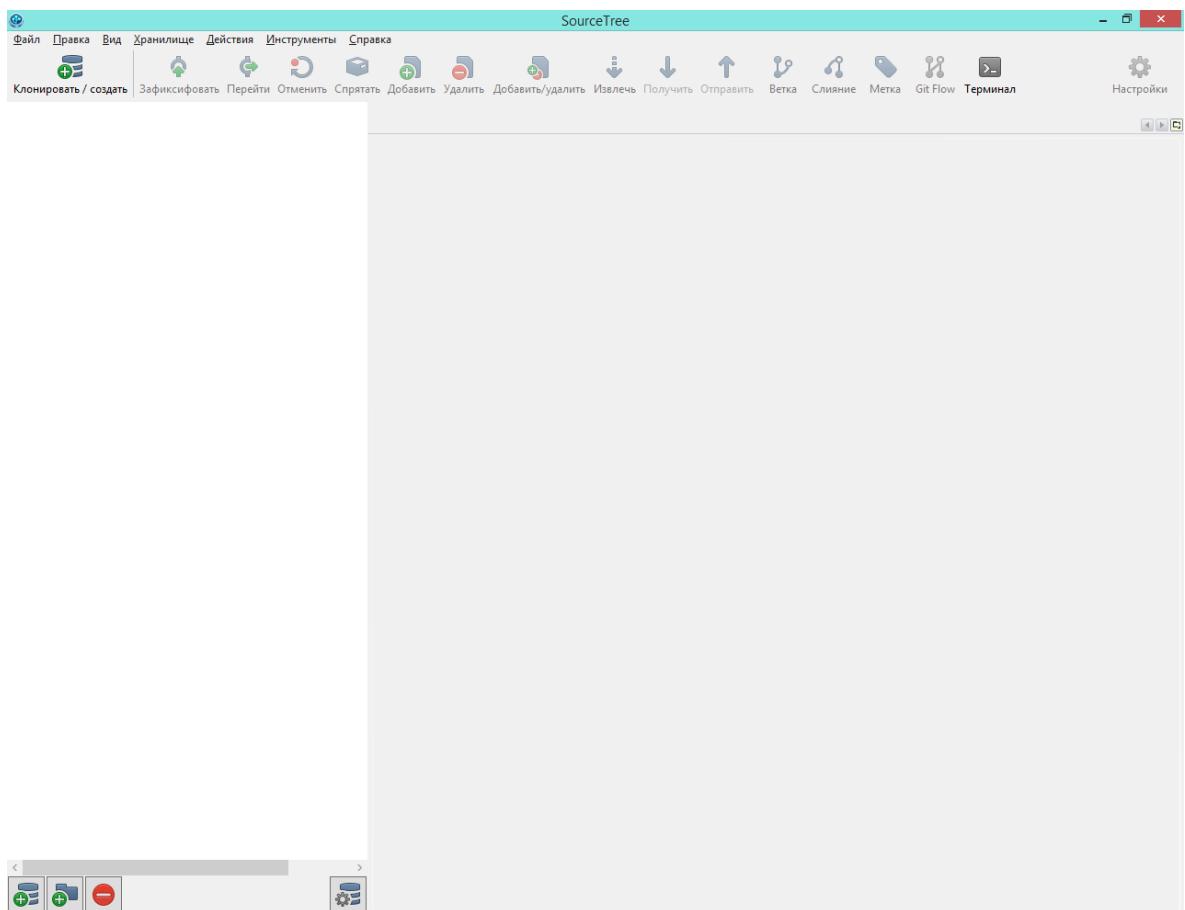


Рисунок 77 – Пользовательский интерфейс SourceTree

Если ОС использует русский язык по умолчанию, то и программа будет на русском. Следует отметить некачественный перевод программы, поэтому для соблюдения идентичности кнопок на пользовательском интерфейсе с командами консоли, работа в дальнейшем будет рассматриваться на англоязычной версии интерфейса.

Для клонирования, имеющегося репозитория с GitHub необходимо нажать на кнопку Clone.

После этого появится форма:

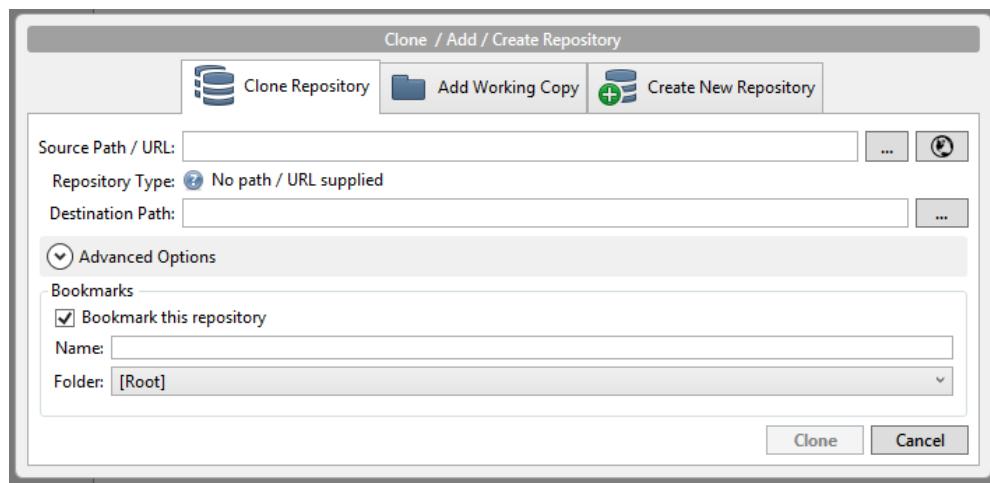


Рисунок 78 – Форма клонирования Git-репозитория

Для клонирования репозитория необходимо будет указать путь до него в поле Source Path/ URL: и локальное расположение репозитория на ПК в поле Destination Path.

Путь репозитория можно взять из GitHub в настройках созданного репозитория:



Рисунок 79 – Ссылка на репозиторий в пользовательском интерфейсе SourceTree

Помимо этого, GitHub поддерживает обращение к репозиторию не только как к *.git, но и по URL:

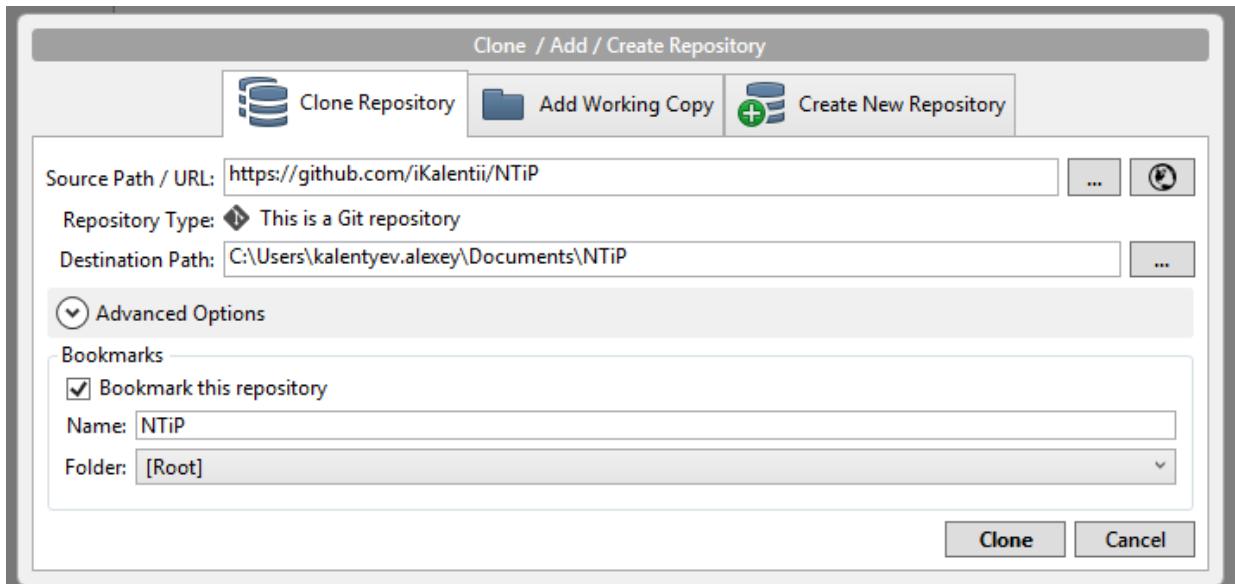


Рисунок 80 – Форма клонирования Git репозитория со ссылкой

Нажав на кнопку Clone, вы создадите локальную копию всех файлов, хранящихся у вас в репозитории.

Добавьте туда файл README.txt. Перейдя в SourceTree вы увидите, что файл появился в окне программы как не добавленный под версионный контроль:

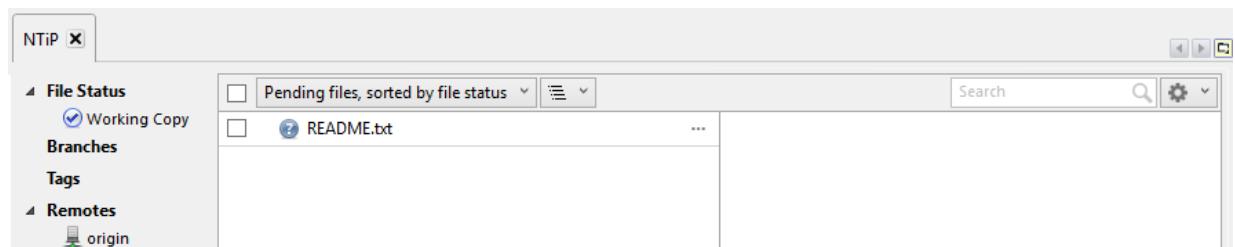


Рисунок 81 – Форма с файлом, не находящимся под версионным контролем

Для добавления файла необходимо выделить его галочкой и нажать на кнопку Commit. После этого появится окно для ввода сообщения о коммите. Отнеситесь к этому шагу ответственно, так как очень часто разработчики не любят писать много комментариев к коммиту, что приводит к неразберихе при необходимости отката к определённой версии.

Это происходит, потому что из тысяч коммитов сложно идентифицировать нужный из-за отсутствия необходимой информации.

Автоматически ваш коммит будет помещён в ветку master:

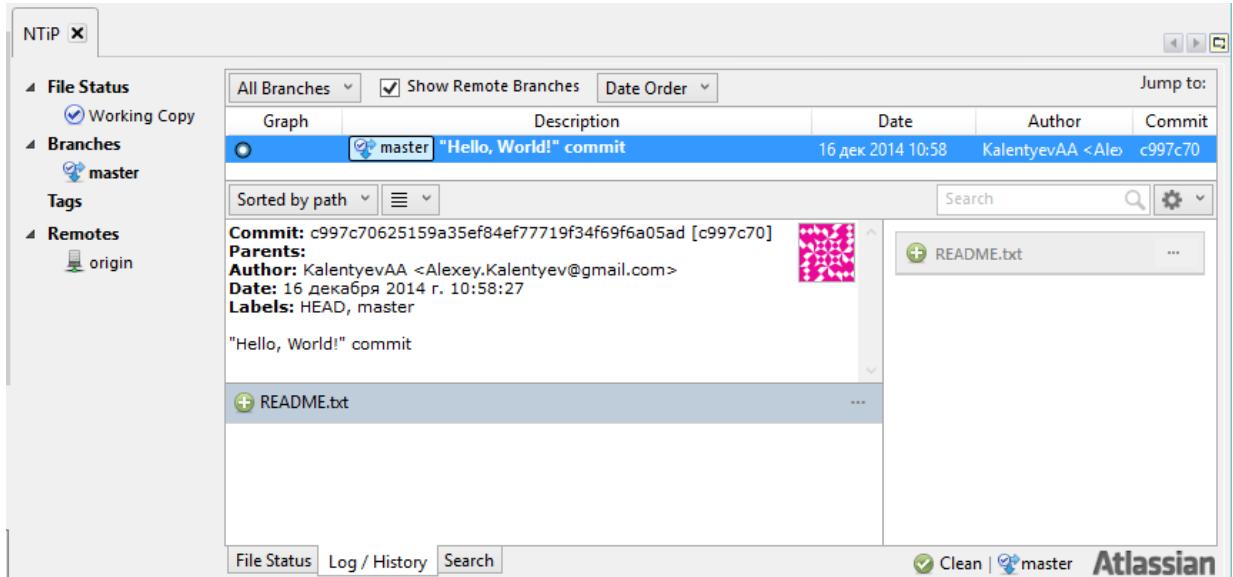


Рисунок 82 – Первый коммит в SourceTree

При необходимости получить/залить изменения из/на GitHub нажмите кнопку Pull/Push:

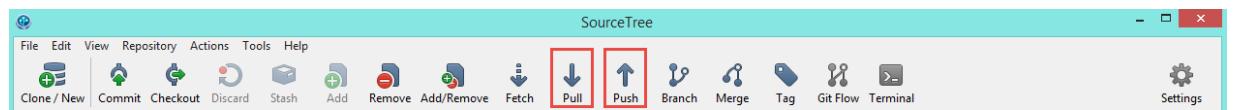


Рисунок 83 – Рабочая панель SourceTree

Измените файл README.txt. После этого в SourceTree появится предупреждение, что у вас есть незакоммиченные изменения:

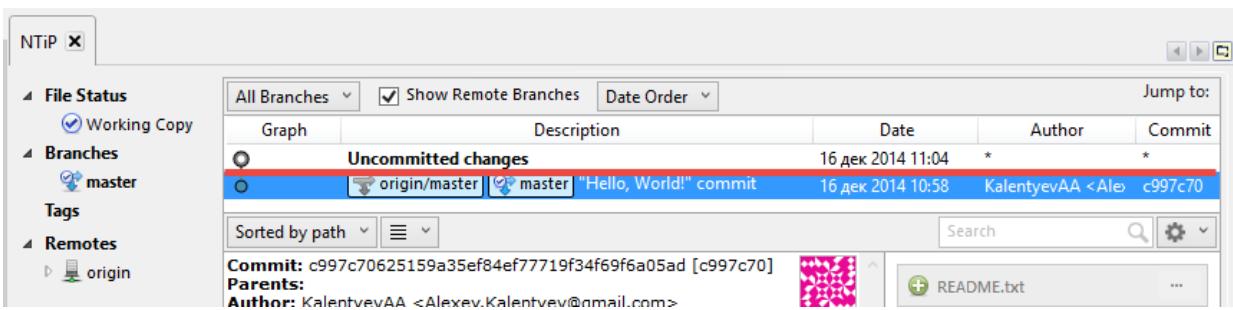


Рисунок 84 – Показ незакоммиченных изменений в SourceTree

Перейдя на выделенный пункт, вы увидите, какие именно файлы были изменены, а в правом окне вы увидите, что именно было изменено в файле:

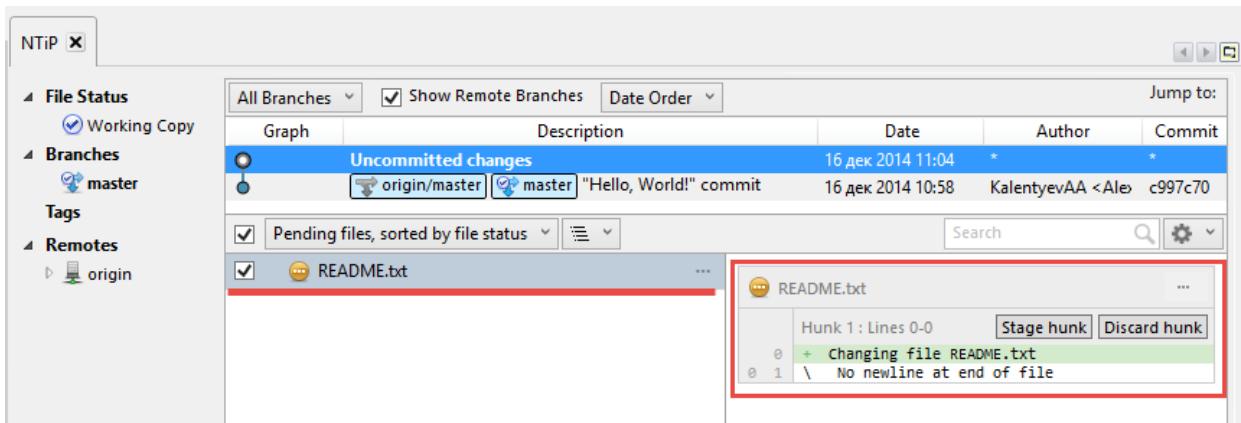


Рисунок 85 – Отображение изменённых файлов в SourceTree

Сделайте коммит изменения. Далее рассмотрим работу с ветками. Наиболее удачная модель работы с ветками представлена в следующей главе, в этой же будет рассмотрен процесс создания веток и их слияния. Для создания ветки нажмите кнопку Branch:



Рисунок 86 – Рабочая панель SourceTree

Введите имя ветки:

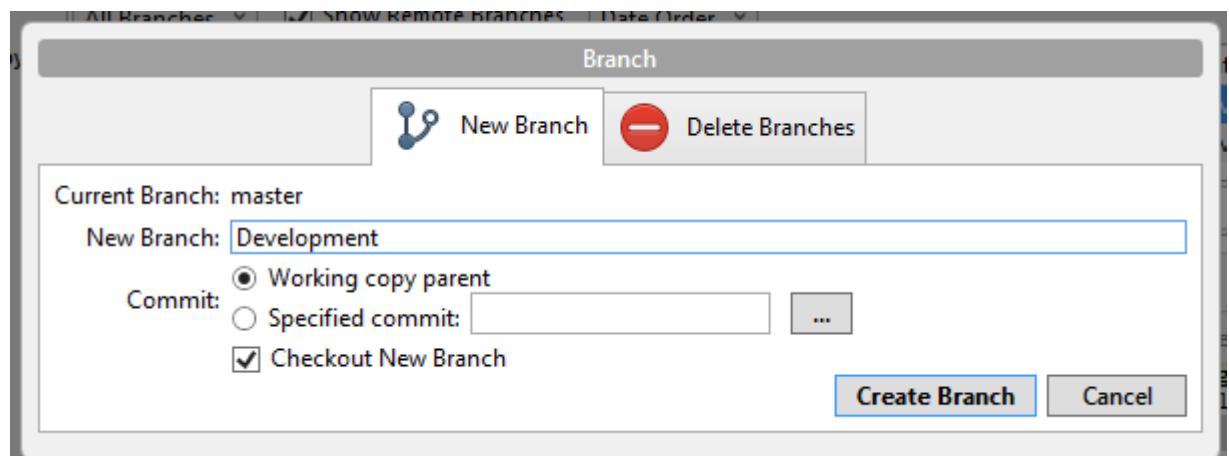


Рисунок 87 – Диалог создания ветки в SourceTree

Следует заметить, что ветки можно создавать как от последнего коммита, так и от любого другого. После создания ветки она будет отмечена галочкой как текущая, что значит, что изменения файлов в локальном репозитории будут отражаться именно на текущей ветке:

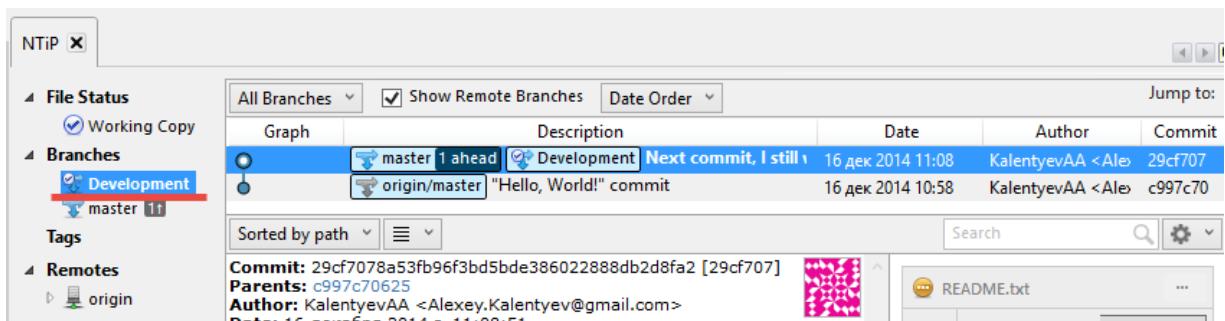


Рисунок 88 – Созданная ветка в SourceTree

Поменяйте файл README.txt несколько раз и выполните коммит после каждого изменения:

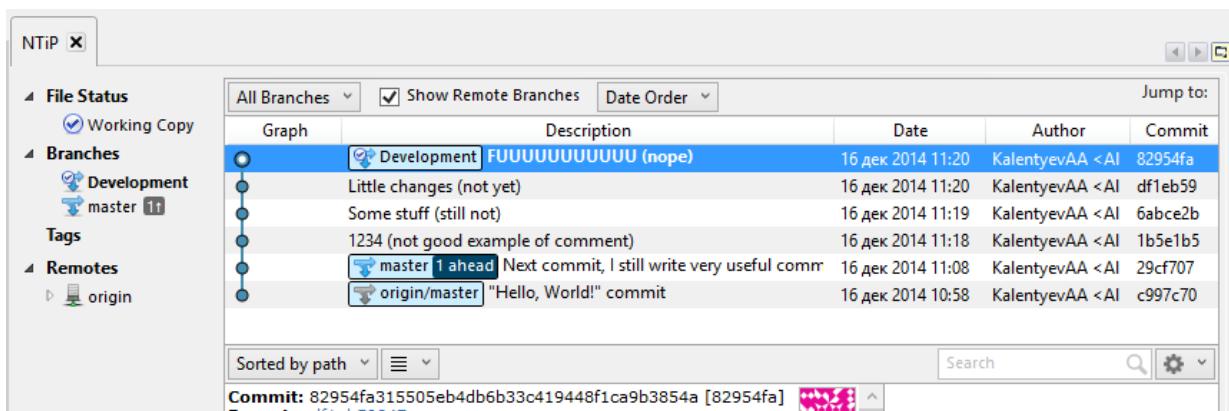


Рисунок 89 – SourceTree после нескольких коммитов

Теперь попробуйте перейти на ветку master. Вы увидите, что Git восстановил версию с содержимым, которое было последним добавлено в ветку. После того, как вы вернулись в master, необходимо выполнить слияние. Запомните, слияние выполняется из ветки, в которую вы хотите выполнить слияние! Для слияния выбираем кнопку Merge:



Рисунок 90 – Рабочая панель SourceTree

В окне Merge выбираем последний коммит в ветке Development. Всё, вы выполнили слияние двух веток:

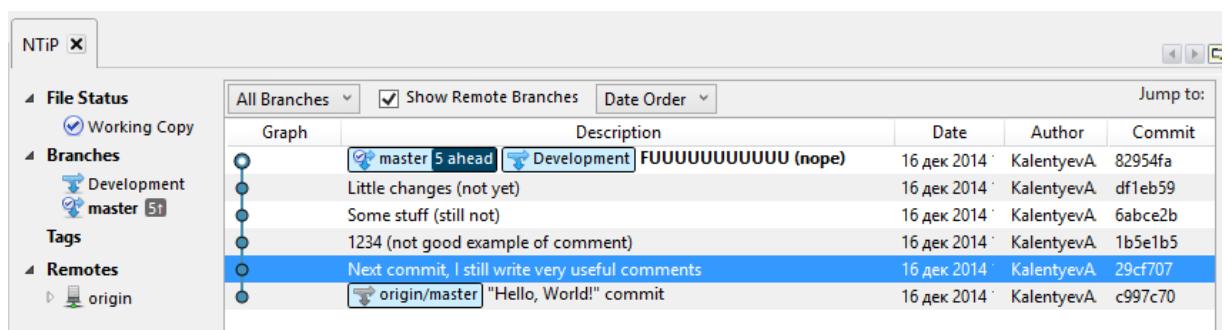


Рисунок 91 – Выполнение слияния с веткой Development

Для того чтобы увидеть систему ветвления вашего проекта в SourceTree, необходимо до выполнения слияния с основной веткой выставить настройки (Tools->Options->Git), показанные на рисунке далее:

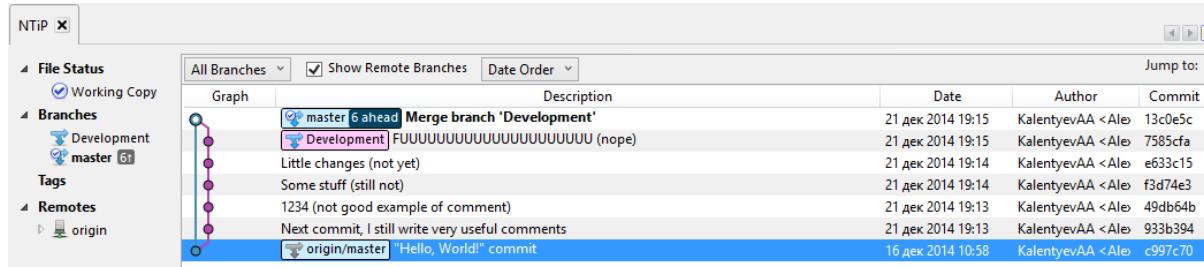


Рисунок 92 – Отображение системы ветвления, после слияния с веткой Development

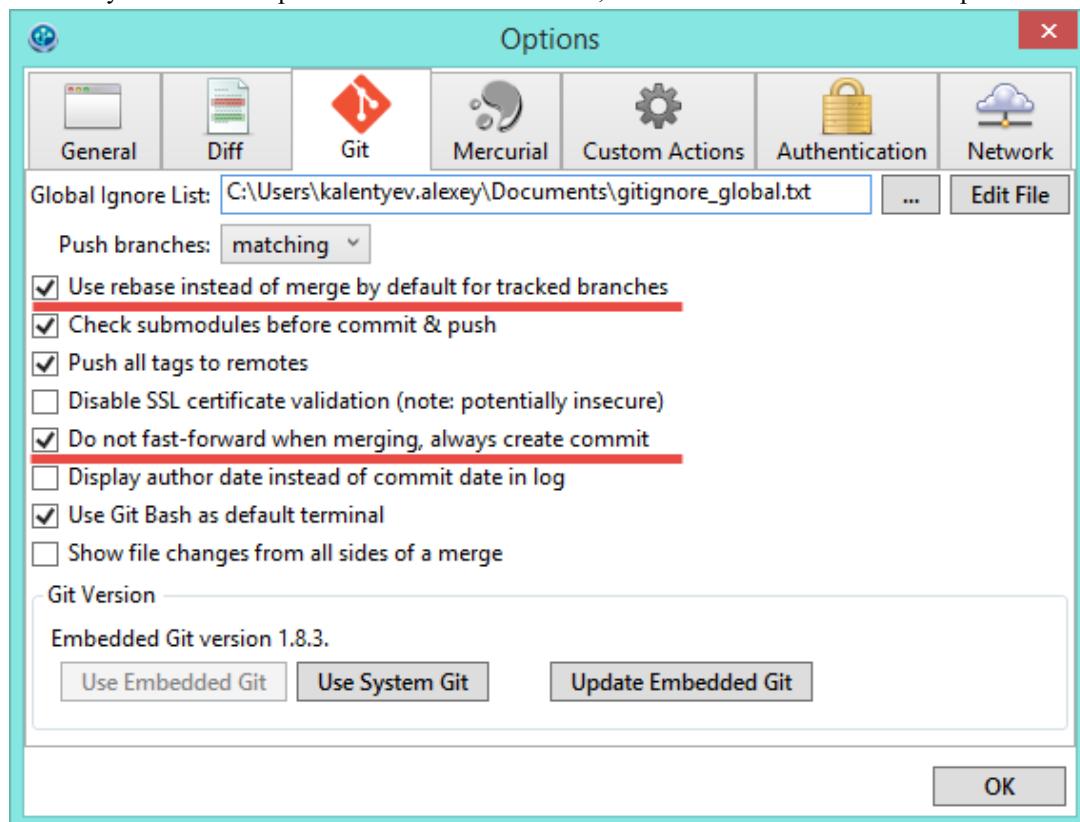


Рисунок 93 – Настройки SourceTree для показа системы ветвления проекта

После этого можно удалить ветку Development. Это можно сделать, например, наведя мышкой на ветку и выбрав Delete Development в контекстном меню:

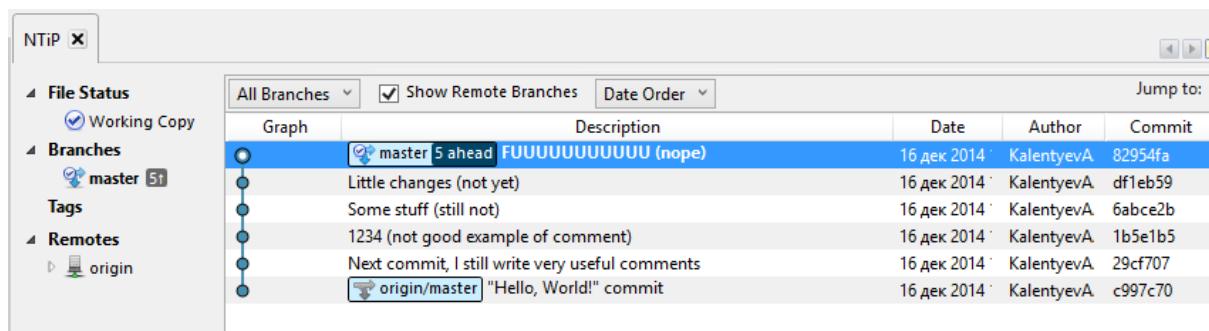


Рисунок 94 – Удаление ветки Development

Можно увидеть, что ветка Development была успешно удалена. Сделав ещё один коммит можно увидеть, что кнопка Push показывает количество произведённых изменений для отправки в репозиторий, от которого вы создали свой в самом начале:

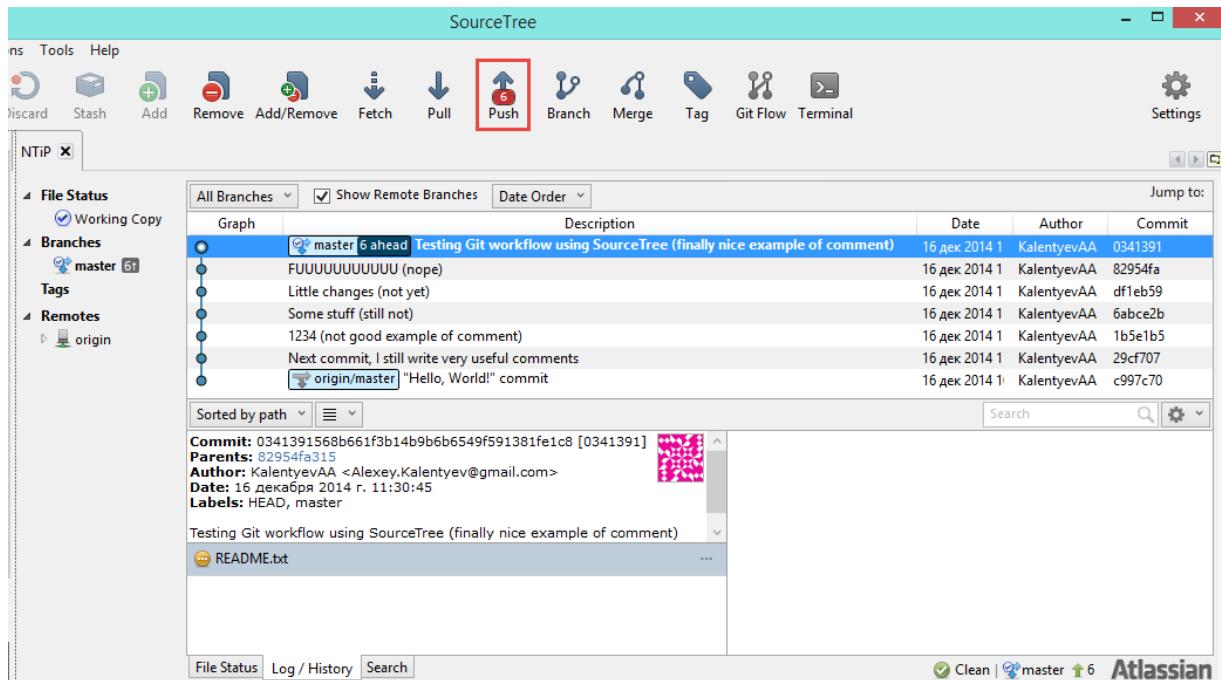


Рисунок 95 – Отображение изменений для отправки в изначальный репозиторий

5 Методические указания к самостоятельной работе

Методические указания к самостоятельной работе в течение курса.

Методические указания к самостоятельной работе по завершению курса.

6 Глоссарий

7 Список литературы

Иконки Visual Studio для использования в своих программах:
<https://www.microsoft.com/en-my/download/details.aspx?id=35825>