Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

Кафедра компьютерных систем в управлении и проектировании (КСУП)

Д. В. Гарайс, А. Е. Горяинов, А. А. Калентьев

Новые технологии в программировании

Методические указания по лабораторным работам

2015

Корректор: Осипова Е. А.

Гарайс Д. В., Горяинов А. Е., Калентьев А. А.
Новые технологии в программировании: методические указания по лабораторным работам. — Томск: Факультет дистанционного обучения, ТУСУР, 2015. — 79 с.

© Гарайс Д. В., Горяинов А. Е.,
Калентьев А. А., 2015
© Факультет дистанционного
обучения, ТУСУР, 2015

Содержание

Введение 5
1 Лабораторная работа № 1. Бизнес-логика 6
1.1 Integrated development environment 6
1.2 Знакомство с Microsoft Visual Studio 7
1.3 Первое приложение на C# 10
1.4 Ввод/вывод на языке C# 12
1.5 Задание на лабораторную работу 15
1.6 Выбор варианта лабораторной работы 16
1.7 Варианты заданий 17
1.8 Рекомендуемая литература 18
2 Лабораторная работа № 2. Пользовательский интерфейс 19
2.1 Создание оконного приложения 19
2.2 Дизайнер форм 20
2.3 Валидация данных 21
2.4 Условная компиляция 24
2.5 Сериализация 25
2.6 Задание на лабораторную работу 28
2.7 Список используемых источников 31
3 Лабораторная работа № 3. Система контроля версий 32
3.1 Об управлением версиями 32
3.1.1 Локальные системы управления версиями 33
3.1.2 Централизованные системы управления версиями 34
3.1.3 Распределённые системы контроля версий 35
3.1.4 Краткий экскурс в историю появления Git 36
3.1.5 Особенности Git 37
3.1.6 Слепки вместо патчей 37
3.1.7 Почти все операции — локальные 38

3.1.8 Git следит за целостностью данных....39
3.1.9 Чаще всего данные в Git только добавляются....40
3.1.10 Три состояния....40
3.2 Сервис Github....41
3.3 Инструменты работы с Git....43
3.3.1 Работа с помощью командной строки....44
3.3.2 Работа с помощью SourceTree (GUI)....47
3.4 Удачная модель ветвления для Git....56
3.5 Задание на лабораторную работу....56
3.6 Список использованных источников....57
4 Лабораторная работа № 4. Юнит-тестирование....58
4.1 Задание на лабораторную работу....59
5 Лабораторная работа № 5. Рефакторинг и сборка установщика....68
5.1 Задание на лабораторную работу....69
5.2 Список использованных источников....73
6 Лабораторная работа № 6. Проектная документация....74
6.1 Задание на лабораторную работу....74
6.2 Список использованных источников....79

Введение

Ввиду ограниченности времени, отведенного на выполнение лабораторных работ, студентам предлагается следующая модель работы: часть лабораторных работ выполняется обязательно, часть лабораторных работ остаётся необязательной для выполнения.

Обязательные для выполнения лабораторные работы:

- Лабораторная работа № 1. Бизнес-логика.
- Лабораторная работа № 2. Пользовательский интерфейс.
- Лабораторная работа № 4. Юнит-тестирование.
- Лабораторная работа № 5. Рефакторинг и сборка установщика.

Необязательные для выполнения лабораторные работы:

- Лабораторная работа № 3. Система контроля версий.
- Лабораторная работа № 6. Проектная документация.

Выполнение необязательных лабораторных работ не оценивается, оно позволит студенту получить некоторые практические навыки, которые помогут в повседневной работе в области разработки программных систем.

1 Лабораторная работа № 1. Бизнес-логика

Целью данной работы является реализация простого проекта в IDE Visual Studio 2013 на языке C#.

1.1 Integrated development environment

Когда программирование только зарождалось и им еще занималась не программисты, а научные работники, не было никаких средств для эффективного написания кода, программы писались в примитивных текстовых редакторах и компилировались из командной строки. Такой подход не вызывает сложностей при написании небольших программ. При разработке крупных программных проектов он становится затруднительным. Поэтому со временем стали появляться специализированные среды разработки, позволяющие по возможности автоматизировать часть задач и упростить написание кода, такие среды называются IDE.

IDE (*Integrated development environment*, интегрированная среда разработки) — система программных средств, используемая программистами для разработки программного обеспечения. В базовом виде IDE включают в себя текстовый редактор, компилятор, средства автоматизации сборки, отладчик. Однако современные IDE включают в себя большое количество дополнительных инструментов, позволяющих значительно облегчить процесс написания кода, например:

1) Подсветка синтаксиса.
2) Статические анализаторы, которые проверяют ошибки в программе прямо во время написания (а не на этапе компиляции).
3) Инструменты для эффективного рефакторинга.
4) Фреймворки для генерации и написания модульных тестов и др.

Существует множество различных IDE, которые поддерживают как несколько языков (Microsoft Visual Studio, IntelliJ IDEA, NetBeans), так и

всего один (PyCharm, Delphi). Одной из лучших IDE для разработки приложений для Windows является Microsoft Visual Studio. Кроме того, что Microsoft Visual Studio позволяет эффективно разрабатывать на популярных языках (C++, C#, VB.NET), значительным преимуществом является возможность ее расширения различными плагинами, начиная от продвинутой подсветки синтаксиса и интеграции систем контроля версий до подключения дополнительных языков программирования. На сайте Microsoft можно скачать бесплатную версию Visual Studio Express 2013, которая понадобится для выполнения лабораторных работ.

1.2 Знакомство с Microsoft Visual Studio

После установки и запуска Microsoft Visual Studio Express 2013 (далее MSVS) запустится стартовое окно программы. Для того чтобы создать проект, необходимо нажать *Create Project* на стартовой странице либо выбрать *FILE->Create->Project*, после чего появится окно создания проекта.

В лабораторных работах будут использоваться 3 вида проектов:

1) *Class library* — библиотека классов — проект такого типа компилируется в файл формата *.dll. Используется для описания бизнес-логики приложения.
2) *Console application* — консольное приложение — компилируется в *.exe. Позволяет создать приложение с которым можно взаимодействовать через консоль.
3) *Windows forms application* — оконное приложение — компилируется в *.exe. Позволяет создавать оконные приложения для Windows, используя технологию WinForms.

После создания проекта MSVS создает ***Solution*** (Решение), в котором находится созданный проект. Решение содержит элементы, необходимые для создания приложения. Решение может включать один или несколько проектов, а также файлы и метаданные, необходимые для определения ре-

шения в целом. Решение необходимо для того, чтобы хранить все проекты, которые относятся к одному приложению, а также для отслеживания их взаимодействий. MSVS хранит определение решения в двух файлах: *.sln и *.suo. Файл решения (*.sln) содержит метаданные, которые определяют решение, в том числе:

1) Проекты, связанные с решением.
2) Элементы, которые не связаны с определенным проектом (текстовые файлы, картинки и т. д.).
3) Конфигурации сборки, определяющие, какие конфигурации проекта применяются в каждом типе сборки.

Для добавления файлов в проект необходимо вызвать контекстное меню решения (нажать правой кнопкой на корневом узле в Обозревателе решения (Solution Explorer)), после чего выбрать пункт Добавить (Add). Таким образом, в решение можно добавить новые проекты или различные файлы (Create element…).

Project (Проект) MSVS служит контейнером для файлов с исходным кодом, подключенным библиотекам и файлам. Управление проектом также осуществляется через контекстное меню, которое можно вызвать через Обозреватель решения. Через контекстное меню можно добавлять файлы с исходным кодом в проект (Add->Create element…).

Важно!!! После установки MSVS файлы с расширением *.cs ассоциируются с ней. То есть такие файлы будут открываться в MSVS. Следует понимать, что открытые таким образом файлы не добавляются в проект или решение и они не могут быть скомпилированы. Чтобы добавить существующий файл в проект, необходимо вызвать контекстное меню проекта и в пункте Добавить выбрать существующий проект (Add->Exiting element…).

Часто один проект должен использовать некоторые типы данных, определенные в другом проекте. Для этого необходимо в основной проект добавить ссылку на зависимый проект. Для этого необходимо вызвать контекстное меню элемента References основного проекта и выбрать пункт Добавить ссылку (Add Reference…). После этого появится окно, изображенное на рис. 1.1. Зависимость можно добавить как на проекты, находящиеся в том же решении (для этого надо выбрать пункт Решение), так и на существующие сборки, входящие в .NET Framework, или дополнительно установленные библиотеки (пункт Сборки). После выбора необходимой библиотеки необходимо установить флаг слева от названия в положение «используется» и нажать ОК. После этого можно использовать типы данных, определенные в выбранной библиотеке в вашем проекте.

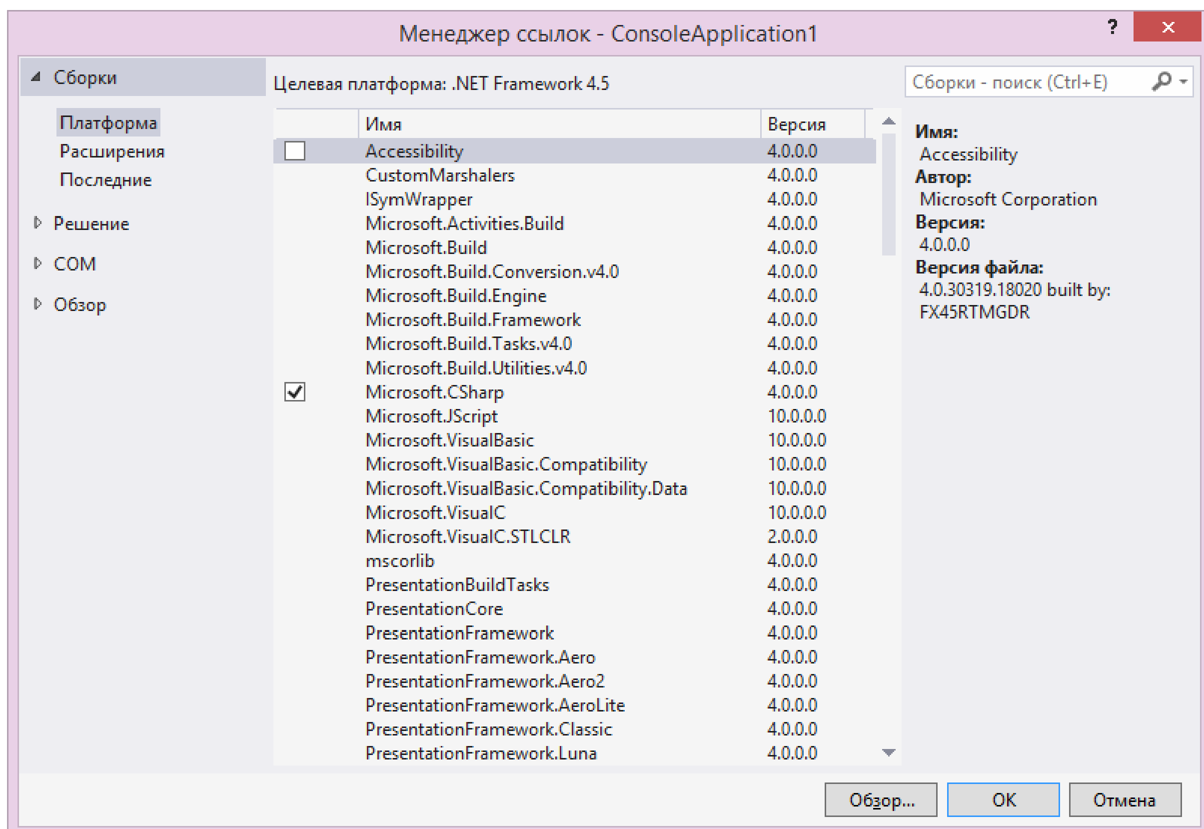


Рисунок 1.1 — Окно добавления зависимостей проекта

1.3 Первое приложение на C#

Во всех учебниках по программированию в качестве первой программы всегда используется «Hello, World». Единственная функция этой программы — выводить в консоль строку «Hello, World!».

В первую очередь необходимо создать новое консольное приложение. Назовем его «HelloWorld». После создания MSVS сгенерирует Решение HelloWorld и проект HelloWorld. Будет сгенерирован единственный файл с кодом Program.cs. Ниже следует его содержимое.

```
namespace HelloWorld
{
	class Program
	{
		static void Main(string[] args)
		{
		}
	}
}
```

Рассмотрим листинг подробнее. На первой строке находится объявление пространства имен, название которого по умолчанию совпадает с именем проекта.

Затем идет объявление класса Program. Ввиду того, что C# — полностью объектно-ориентированный язык, в нем не может быть функций, которые не принадлежат какому-либо классу, в данном случае класс Program необходим для определения метода Main. Класс Program является обычным классам, и с ним можно делать все то же, что и с другими, созданными пользователем, классами. Например, его можно переименовать.

Далее идет объявление функции Main. Любое консольное приложение должно содержать метод Main. Метод Main — точка входа в программу, начало ее выполнения. В случае отсутствия этого метода будет ошибка при компиляции, потому что компилятор не будет знать, где начинается ваша программа. Соответственно, этот метод нельзя переименовывать и перегружать.

Вообще существует несколько вариантов сигнатуры этого метода:

```
static void Main(string[] args)
static int Main(string[] args)
static void Main()
static int Main()
```

Эти функции будут отличаться типом возвращаемого значения и наличием входных параметров. Рассмотрим подробнее: в первом случае из программы не возвращается никаких значений (void) и программа принимает на вход параметры. Строка *string[] args* содержит в себе список параметров, которые пользователь может передать в программу при вызове из командной строки. Во втором случае программа возвращает в консоль целое число, обычно это делается, чтобы после окончания выполнения программы пользователь мог узнать, корректно она завершилась или нет, если корректно, то возвращается 0, в противном случае 1 либо код произошедшей ошибки. В третьем и четвёртом случаях отсутствуют входные аргументы программы. Читатель, знающий C++, может провести прямую аналогию с функцией Main.

Теперь необходимо добавить в функцию Main код, который будет выводить необходимую строку. Сделать это можно следующим образом:

```
System.Console.Write("Hello, World!");
```

или

```
System.Console.WriteLine("Hello, World!");
```

Отличия этих двух способов в том, что в первом случае каретка останется на той же строке, а во втором передвинется на следующую строчку. То есть если у нас будет несколько вызовов System.Console.Write подряд, то аргументы функции будут выводиться на одной строке, а если будет несколько вызовов System.Console.WriteLine, каждый аргумент будет выводиться на новой строчке.

Для того чтобы запустить приложение, необходимо либо на верхней панели нажать кнопку Запуск (Start), либо через главное меню — Отладка (Debug)-> Начать отладку (Start debugging), также можно нажать клавишу F5.

При первом запуске скорее всего вы увидите, как мелькнет окно консоли и тут же пропадет. Это не ошибка, просто программа очень быстро выполнилась и закрылась. Чтобы посмотреть результаты выполнения, необходимо остановить выполнение программы. Для этого, например, можно заставить программу ждать, пока не будет нажата какая-нибудь клавиша клавиатуры. Для этого необходимо в конец программы добавить строку:

```
System.Console.Read();
```

Если запустить программу теперь, то на экране появится окно консоли с текстом “Hello, World!”. Чтобы окно пропало, надо нажать произвольную клавишу.

В итоге ваша программа должна выглядеть следующим образом:

```
namespace HelloWorld
{
	class Program
	{
		static void Main()
		{
			System.Console.WriteLine("Hello, World!");
			System.Console.Read();
		}
	}
}
```

1.4 Ввод/вывод на языке C#

В общем случае, когда говорят о программах, будь то консольное приложение или оконное, подразумевается, что пользователю необходимо с ней взаимодействовать. Программа должна сообщить пользователю о процессе или результатах выполнения, а также работать с данными, кото-

рые пользователь ей подает. Поэтому в данной части речь пойдет о операторах ввода и вывода на C#.

Для получения данных с клавиатуры можно использовать следующий метод:

```
string str = Console.ReadLine();
```

Как видно, этот метод возвращает строку и не существует его перегрузок или других методов, чтобы считывать целочисленные, символьные и другие типы переменных. Соответственно, для того чтобы ввести с клавиатуры, необходимо получить строковое значение, а затем сконвертировать его в необходимый нам тип данных. Для этого можно использовать класс Convert. В нем определено множество статических методов конвертирования строк в иные типы данных. Далее следуют примеры использования класса Convert.

```
int intValue = Convert.ToInt32(Console.ReadLine());
double doubleValue = Convert.ToDouble(Console.ReadLine());
bool boolValue = Convert.ToBoolean(Console.ReadLine());
long longValue = Convert.ToInt64(Console.ReadLine());
```

При этом надо следить, чтобы типы вводимых значений совпадали с типами указанных переменных, потому что в противном случае возникнет исключение. Например, если ввести строку 123.456 и попытаться сконвертировать в целочисленное значение, то сгенерируется исключение.

Для более корректной работы программы, если программа ожидает от пользователя какое-то определенное значение, то лучше ему об этом сказать. Необходимо придерживаться правила: перед тем как считывать что-либо с консоли, необходимо сообщить пользователю, что именно он должен ввести: смысл вводимой информации, тип данных, максимальное и минимальное допустимые значения и т. п. Примером таких запросов могут служить:

«Введите имя пользователя (не больше 20 знаков)»

«Введите возраст, целочисленное значение, от 1 до 100»

«Введите пол, 0 — мужской, 1 — женский»

Для вывода текста на экран можно использовать команды, про которые говорилось в предыдущей главе: Console.Write или Console.WriteLine. Далее идут примеры использования этих команд.

```
Console.WriteLine(s);// переменная
Console.WriteLine(55.3);// константа
Console.WriteLine(y*3+7);// выражение

Console.Write(z);// переменная
Console.Write(-5.3);// константа
Console.Write(i*3+7/j);// выражение
```

Очень часто в процессе работы необходимо выводить осмысленные предложения с результатами выполнения программы. Например, «Через насос X было перекачано Y литров воды, температура насоса Z градусов», где в различные моменты выполнения программы X, Y, Z могут быть различными значениями. Конечно, можно использовать следующий подход:

```
Console.WriteLine(«Через насос » + X + «, было перекачано » + Y +
« литров воды, температура насоса » + Z + « градусов»);
```

Однако такой подход слишком громоздкий, и с ним возникает масса трудностей, например при необходимости добавить дополнительные данные, даже читать такую запись достаточно затруднительно. Поэтому принято использовать подход с использованием форматной строки. Сама строка формата содержит большую часть отображаемого текста, но всякий раз, когда в нее должно быть вставлено значение переменной, в фигурных скобках указывается индекс. В фигурные скобки может быть включена и другая информация, относящаяся к формату данного элемента, например та, что описана ниже:

- Количество символов, которое займет представление элемента, снабженное префиксом-запятой. Отрицательное число указывает, что элемент должен быть выровнен по левой границе, а положительное — по правой. Если элемент на самом деле занимает

больше символов, чем ему отведено форматом, он отображается полностью.

- Спецификатор формата предваряется двоеточием. Это указывает, каким образом необходимо отформатировать элемент. Например, можно указать, должно ли число быть форматировано как денежное значение либо его следует отобразить в научной нотации, в степенном виде, либо шестнадцатиричном.

```
Console.WriteLine(«Через насос {0} было перекачано {1} литров воды,
температура насоса {2} градусов», X, Y, Z);
```

1.5 Задание на лабораторную работу

1. Создайте проект на языке C# в среде Microsoft Visual Studio. Назовите его в соответствии с вашим вариантом задания, в качестве исходного проекта выберите проект динамической библиотеки (*.dll). Назовите его либо согласно вашему варианту, либо просто Model. Данный проект будет содержать в себе бизнес-логику приложения, т. е. ключевые структуры данных и способы их взаимодействия.

2. Создайте сущность-интерфейс согласно вашему варианту. Опишите ключевые свойства и методы интерфейса. Не забудьте о правильном именовании типов данных согласно RSDN. Подумайте, какие свойства и методы будут являться общими (будут в интерфейсе), а какие должны быть реализованы в конкретных классах.

3. Создайте 2 или более класса, реализующих данный интерфейс. Классы обязательно должны иметь различные реализации методов интерфейса. При этом дочерние классы не должны иметь никаких ссылок друг на друга, так же как и интерфейс не должен ничего знать о дочерних классах.

4. Реализуйте проверку правильности передаваемых свойствам данных (валидацию свойств) с помощью механизма обработки исключений — если на вход приходят некорректные данные, выходящие за допустимые

пределы, свойство должно сгенерировать исключение соответствующего типа с описанием ошибки. Например, если свойству Возраст пытаются присвоить отрицательное значение, необходимо сгенерировать экземпляр исключения IncorrectArgumentException. Внимательно продумайте все возможные некорректные варианты входных данных, в том числе ссылки на null. В случае если механизмы валидации у всех свойств одинаковы, измените архитектуру: вместо реализации интерфейса используйте наследование от абстрактного класса, в котором будут реализованы механизмы валидации.

5. Добавьте в решение еще один проект, на этот раз консольное приложение, и назовите его «ConsoleLoader». В этом проекте будет проводиться первичное тестирование бизнес-логики приложения.

ПРИМЕЧАНИЕ: данный проект является временным и впоследствии будет заменён на проект графического интерфейса Windows (WinForms Application). Однако если вы уже можете продемонстрировать работу бизнес-логики на оконном пользовательском интерфейсе, можете сразу создать необходимый проект.

6. Продемонстрируйте корректную работу бизнес-логики. Создайте переменную-ссылку на интерфейс и присваивайте в нее экземпляры реализуемых классов. Продемонстрируйте разную реализацию интерфейсных свойств и методов. Для этого необходимо реализовать ввод с клавиатуры значений, которыми будут инициализированы поля классов-наследников.

1.6 Выбор варианта лабораторной работы

Выбор варианта лабораторных работ осуществляется по общим правилам с использованием следующей формулы:

$$V = (N \times K) \text{ div } 100,$$

где V — искомый номер варианта,

N — общее количество вариантов,

div — целочисленное деление,

при V = 0 выбирается максимальный вариант,

K — код варианта.

Студент имеет возможность выбрать собственный вариант, не представленный в нижеследующем списке.

1.7 Варианты заданий

1. Геометрические фигуры с различными реализациями расчета площади фигуры: круг, прямоугольник, треугольник.

2. Трехмерные фигуры с различными реализациями расчета объема: шар, пирамида, параллелепипед.

3. Работники фирмы с различными способами начисления зарплаты: почасовая оплата, оплата по окладу и ставке.

4. Транспортные средства с различными реализациями расчета затраченного топлива: машина, машина-гибрид, вертолет.

5. Система скидок с различными реализациями расчета скидок: процентная, по сертификату.

6. Система библиотечных карточек для разных изданий: книга, журнал, сборник, диссертация. Каждое издание характеризуется различным набором полей, перегружаемый метод возвращает информацию об издании в виде строки, оформленной по ГОСТу [3].

7. Различные пассивные элементы электрических схем: резистор, конденсатор, индуктивность. Перегружаемый метод — расчет комплексного сопротивления элемента.

8. Расчет координаты для различных видов движения: равномерное, равноускоренное, колебательное.

9. Расчет затраченных калорий в зависимости от вида упражнений: бег (интенсивность, расстояние), плавание (стиль, расстояние), жим штанги (вес, количество повторений).

1.8 Рекомендуемая литература

1. Г. Шилдт. C# 4.0 Полное руководство / Г. Шилдт. — М. : Вильямс, 2011.

2. Microsoft Developer Network (MSDN) [Электронный ресурс]. — URL: msdn.microsoft.com/ru-RU/ (дата обращения: 21.12.2014).

3. Библиографическое описание. Государственный УНПК [Электронный ресурс]. — URL: http://www.ostu.ru/libraries/bibl_opisanie.php (дата обращения 18.01.2015).

2 Лабораторная работа № 2. Пользовательский интерфейс

Целью данной работы является знакомство с разработкой оконных приложений в среде Microsoft Visual Studio.

2.1 Создание оконного приложения

Для создания оконного приложения на основе уже существующей логики (решения) необходимо добавить в него новый проект WinForms (WinForms Application Project). Это можно сделать через контекстное меню решения в Обозревателе решений либо через главное меню (File->Add ->Create Project…). После нажатия на кнопку OK на форме добавления проекта новый проект добавится в ваше решение.

Рассмотрим подробнее содержимое нового проекта. По-прежнему в нем есть узел References и Properties. К этому добавился файл App.config, Program.cs, Form1.cs.

- App.config — данный файл позволяет создавать конфигурации приложения, что позволить менять некоторые параметры приложения без его перекомпиляции, в лабораторных работах тема конфигураций рассматриваться не будет.
- Program.cs — файл, содержащий метод Main. Именно отсюда запускается главная форма приложения с помощью строки

```
Application.Run(new Form1());
```

- Form1.cs — файл в котором хранится код формы. Вообще для описания логики формы используется 2 файла: Form1.cs, где пользователь описывает логику взаимодействия элементов на форме, и Form1.Designer.cs, который генерирует MSVS, когда пользователь изменяет форму через дизайнер.

Для того, чтобы открыть дизайнер формы, необходимо дважды кликнуть на узел формы в Обозревателе решений, чтобы открыть код с пользовательской логикой к форме, необходимо либо выбрать форму в Обозревателе решений и нажать F7, либо в контекстном меню выбрать пункт *Перейти к коду*.

2.2 Дизайнер форм

При запуске дизайнера форм вы увидите окно, изображенное на рис. 2.1.

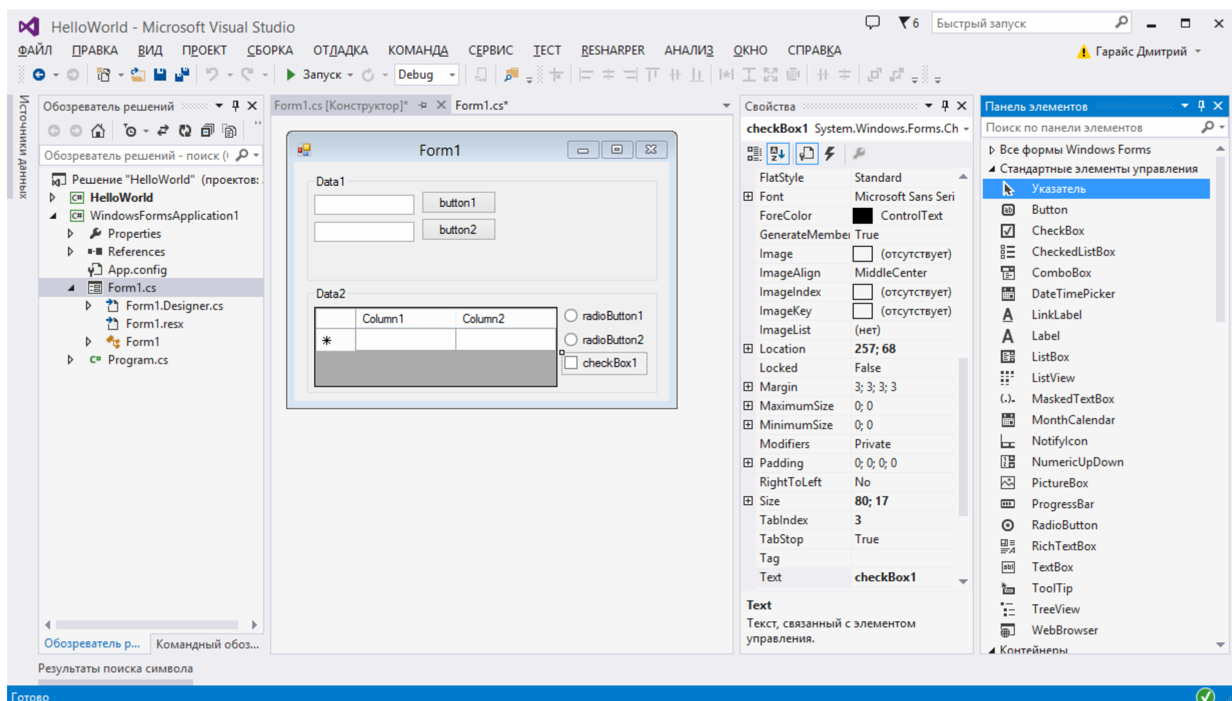


Рисунок 2.1 — Окно дизайнера форм

Для эффективной работы с дизайнером окон, помимо самого дизайнера, необходимы окна *Панель элементов* (Toolbox) и *Свойства* (Properties). В случае если эти окна не появились, можно настроить их отображение, используя Главное меню->Вид->Панель элементов и Главное меню->Вид->Окно свойств.

На *Панели инструментов* находятся все доступные для использования элементы управления. Для добавления того или иного элемента на форму его необходимо перетащить с панели инструментов.

В окне *Свойства* отображаются свойства выбранного элемента управления (чтобы выбрать элемент, необходимо кликнуть на него в дизайнере). Через это окно можно изменять основные свойства элементов, например имя, по которому можно обратиться к этому элементу в коде, название элемента, читаемое для пользователя (например, Главная форма, вместо Form1), размеры элемента, также можно задать его начальное значение. При размещении новых элементов пользовательского интерфейса на форме Visual Studio генерирует их имена автоматически, например gridControl1, button3 и т. д. Однако такие имена не отражают назначения элемента и усложняют понимание кода. Необходимо переименовывать элементы управления согласно нотации RSDN, это будет оцениваться в лабораторных работах. В верхней части окна *Свойства,* можно перейти в режим *События*, здесь отображаются все доступные события элемента и их обработчики, также здесь можно назначить новый обработчик для любого события.

2.3 Валидация данных

При разработке интерфейса программы важнейшей задачей для программиста является обеспечение правильности вводимых пользователем данных. В идеале, программист должен так спроектировать и запрограммировать интерфейс, чтобы пользователь НЕ МОГ совершить ошибку, даже если бы захотел.

Для этого обязательно после ввода данных проводится проверка их корректности. Реализовать проверку можно несколькими способами:

- при нажатии на определенную кнопку (например, Добавить объект) происходит проверка всех данных на форме и в случае обна-

ружения некорректных пользователю выводится сообщение с ошибкой;

- при изменении конкретного поля. В данном случае проверки реализуются за счет событий. У любого элемента управления, который позволяет вводить произвольные данные, существует событие, позволяющее проводить проверку корректности, например это событие Validating у элемента TextBox или событие CellValidating у элемента DataGridView. Данные события срабатывают, когда текущий элемент управления теряет фокус, то есть когда пользователь переключается на любой другой элемент, например другое поле ввода или кнопку. Преимущество использования этого подхода в том, что в случае ввода некорректных данных пользователь СРАЗУ видит, в каком поле ошибка и может оперативно её поправить, в то время как при использовании первого подхода ошибка проявляется после того, как все данные введены, при этом пользователю придется разбираться в чем ошибка и искать необходимое поле;
- третий подход заключается в том, что пользователь не может ввести то, чего быть не должно. Для этого используются более сложные пользовательские элементы, например поля ввода с проверкой на основе регулярных выражений [1,2,3]. Такой подход самый лучший, потому что пользователь просто не может написать неправильно. Например, ясно, что в имени не может быть цифр, тогда полю, которое отвечает за ввод имени, присваивается соответствующее регулярное выражение, и при нажатии цифр элемент управления сверяется со своим регулярным выражением и просто игнорирует некорректные данные, введенные пользователем. Простым случаем такого элемента управления может служить MaskedTextBox. MaskedTextBox — аналог обыч-

ного TextBox, однако позволяет использовать специальные маски ввода, запрещающие вводить в поле цифры или, наоборот, символы. Именно его рекомендуется использовать при реализации заданий лабораторной работы.

При этом стоит отметить, что нельзя полагаться только на один подход, в любом случае необходимо комбинировать эти подходы для обеспечения большей безопасности ввода данных.

Также следует отметить, что все предупреждения об ошибках должны нести явный смысл, они должны сообщать, где произошла ошибка и в чем конкретно ошибка. То есть сообщение типа «Age error» абсолютно некорректно, потому что непонятно, в чем именно ошибка возраста. Лучше использовать сообщения типа «Age value must be greater than zero.», в этом сообщении ясно видно, в чем именно заключается ошибка.

В случае появления ошибки необходимо сообщить о ней пользователю, для этого можно использовать класс MessageBox. Пример использования приведен ниже.

```
MessageBox.Show(errorText, errorWindowCaption, MessageBoxButtons.OK,
     MessageBoxIcon.Error);
```

В данном случае errorText — текст ошибка, errorWindowCaption — название окна ошибки. MessageBox очень удобный класс для обеспечения взаимодействия с пользователем, потому что легко настраивается за счет параметров, например изменением третьего параметра можно добавить окошку кнопок, а четвертого — изменить картинку на форме таким образом, что окошко будет не сигнализировать об ошибке, а предлагать дополнительную информацию или спрашивать пользователя о некоторых неочевидных действиях (например, следует ли сохранить несохраненный проект при закрытии программы).

2.4 Условная компиляция

Иногда у разработчика возникает необходимость откомпилировать код так, чтобы для разных конфигураций одни участки кода компилировались, а другие — нет. Для этого используются директивы препроцессора и механизм под названием «условная компиляция».

В C# существуют следующие директивы: **#define**, **#if**, **#else**, **#endif**. Их смысл в общем-то ясен. Рассмотрим пример условной компиляции.

```
#define PARAM1
#if PARAM1
Console.Write("Defined PARAM1");
#else
Console.Write("Not defined PARAM1");
#endif
```

В таком виде скомилированный код при вызове программы выведет на экран строку: «Defined PARAM1», если же закомментировать первую строку в листинге и заново скомпилировать приложение и запустить, то выведется строка «("Not defined PARAM1"». То есть на основе одного и того же кода, путем задания констант с помощью директивы #define можно получать программы с различной функциональностью. Чаще всего это используется, когда необходимо использовать в универсальном приложении платформозависимые вещи, например запись на диск в мобильном приложении, для платформ Android и iOS, в этом случае код может выглядеть следующим образом:

```
#define ANDROID
#if ANDROID
//используем API ОС Android
#else
//используем API ОС iOS
#endif
```

Очень часто при написании приложения версия Debug, используемая разработчиками для отладки, отличается от версии Release, которая поставляется конечному пользователю. Переключиться между ними можно

на панели инструментов, находящейся вверху в центре экрана. Эти версии отличаются обычно тем, что в Debug больше разнообразных проверок на корректность данных и, возможно, есть элементы управления, которые нужны исключительно для отладки, но ненужные конечному пользователю. И чтобы не удалять их всякий раз, когда компилируешь программу в Release, используется условная компиляция. Для этого в MSVS существует константа DEBUG, автоматически определяемая в соответствующем режиме. Используя этот механизм, можно, например, спрятать кнопку, которая генерирует отладочную информацию, следующим образом:

```
public class AddObjectForm : Form
{
        …
        public AddObjectForm()
        {
            InitializeComponent();
            …
            #if !DEBUG
            CreateRandomDataButton.Visible = false;
            #endif
        }
        …
}
```

В этом коде мы используем директивы препроцессора #if, указывая тот код, который будет скомпилирован только для сборки Release. В нашем случае, мы устанавливаем для созданной кнопки CreateRandomDataButton поле Visible в состояние false.

2.5 Сериализация

Сериализация представляет собой процесс преобразования объекта в поток байтов для хранения объекта или передачи его в память, базу данных или файл. Ее основное назначение — сохранить состояние объекта для того, чтобы иметь возможность воссоздать его при необходимости. Обратный процесс называется десериализацией. С помощью сериализации разработчик может выполнять такие действия, как отправка объекта

удаленному приложению посредством веб-службы, передача объекта из одного домена в другой, передача объекта через брандмауэр в виде XML-строки и хранение информации о безопасности или конкретном пользователе, используемой несколькими приложениями.

Сериализация бывает нескольких видов:

- *Двоичная сериализация*. При двоичной сериализации используется двоичная кодировка, обеспечивающая компактную сериализацию объекта для хранения или передачи в сетевых потоках на основе сокетов.
- *XML-сериализация*. При XML-сериализации открытые поля и свойства объекта или параметры и возвращаемые значения методов сериализуются в XML-поток. XML-сериализация приводит к образованию строго типизированных классов с открытыми свойствами и полями, которые преобразуются в формат XML. Для управления процессом сериализации или десериализации можно применять атрибуты к классам и членам класса.
- *SOAP-сериализация.* XML-сериализация может также использоваться для сериализации объектов в потоки XML, которые соответствуют спецификации SOAP. SOAP — это протокол, основанный на XML и созданный специально для передачи вызовов процедур с использованием XML. Как и в обычной XML-сериализации, атрибуты можно использовать для управления формой SOAP-сообщений в литеральном стиле, генерируемых веб-службой XML.

Рассмотрим пример сериализации объекта класса Person на примере XML-сериализация. Класс Person описан ниже.

```
public class Person
{
      public string Name { get; set; }
      public string Surname { get; set; }
```

```
        public int Age { get; set; }
    }
```

Для того чтобы сериализовать данный объект, можно воспользоваться следующим кодом:

```
var writer = new
    System.Xml.Serialization.XmlSerializer(typeof(Person));
using (var file = System.IO.File.Create(_filePath)
{
    writer.Serialize(file, _person);
    file.Close();
}
```

В данном примере данные, которые хранятся в объекте _person, сохраняются в файл с именем _filePath. Адрес файл должен служить параметром метода для сериализации, потому что пользователь должен иметь возможность выбирать, куда именно и под каким именем данные должны быть сохранены.

Для десериализации можно воспользоваться следующим кодом:

```
var reader = new
    System.Xml.Serialization.XmlSerializer(typeof(Person));
var file = new System.IO.StreamReader(_filePath);
_person = (Person)reader.Deserialize(file);
```

В данном случае, данные загружаются из файла с именем _filePath и присваиваются объекту _person.

При этом в процессе сериализации в той директории, которую укажет пользователь, появится файл с именем, которое хранилось в переменной _filePath. Этот файл можно открыть с помощью блокнота, и он может выглядеть следующим образом:

```
<?xml version="1.0"?>
<Person>
  <Name>Jonh</Name>
  <Surname>Connor</Surname>
  <Age>16</Age>
</Person>
```

Хорошо видно, что xml-сериализация точно повторяет структуру сохраняемого объекта и представляет его в удобном для чтения виде.

2.6 Задание на лабораторную работу

1. Создайте в решении новый проект WinForms (WinForms Application Project) и задайте ему соответствующее имя. Если проект бизнес-логики назван как Model, для проекта пользовательского интерфейса логично дать название View. Данный подход в проектировании архитектуры приложения называется Model-View: когда бизнес-логика и пользовательский интерфейс разделены на разные сборки. В дальнейшем такой подход облегчает ориентирование в рамках проекта. Обратите внимание, что теперь данный проект должен быть стартовым, для этого установите его запускаемым проектом по умолчанию.

ПРИМЕЧАНИЕ: ранее созданный проект ConsoleLoader теперь можно удалить. Удаление проекта из решения не приводит к его физическому удалению с носителя, в отличие от классов проекта. Помните об этом при удалении каких-либо компонентов проекта.

2. Добавьте в проект View новую форму. Название формы должно отражать назначение формы и оканчиваться словом Form. Как и имена других классов, имя формы оформляется в стиле Pascal.

3. Добавьте на форму элемент GridControl из панели инструментов. Для повышения удобства пользовательского интерфейса лучше сначала разместить на форме элемент GroupBox, в который поместить GridControl. Это позволит поместить в заголовок GroupBox фразу, поясняющую назначение GridControl. Под GridControl разместите две кнопки Button. Назовите кнопки Add Object и Remove Object, где вместо Object подставьте название того объекта, который реализован в вашей бизнес-логике.

4. Создайте внутри формы поле, хранящее список (List) сущностей, соответствующих вашему варианту. Список должен иметь возможность хранения в себе всех дочерних классов вашей сущности (все виды геометрических фигур, все типы работников, все виды скидок и т. д.).

5. Необходимо реализовать следующую логику формы: GridControl должен отображать (без возможности редактирования) все объекты созданного списка. Кнопка Add Object должна добавлять новый объект в GridControl и в список объектов. Кнопка Remove Object должна удалять выбранный в GridControl объект и удалять его из списка объектов.

6. Для добавления новых объектов в программу нужно разработать специальную форму, которая вызывалась бы по нажатию клавиши Add Object. В форме должна присутствовать возможность заполнения полей, общих для всех дочерних классов, выбор в виде ComboBox или RadioButton типа объекта и, в зависимости от типа объекта, должна появляться возможность заполнения полей данного типа объекта. Например, если создается новый работник, то в форме обязательно есть поля ФИО и даты принятия на работу, но в зависимости от RadioButton с типом оплаты должны появляться поля либо почасовой оплаты, либо оплаты по ставке.

7. На форме создания нового объекта должны присутствовать кнопки Ok и Cancel. Если пользователь нажмет кнопку Ok — в главной форме должен быть добавлен созданный объект. Если пользователь нажмет Cancel — должна быть выполнена отмена добавления.

8. Форма создания нового объекта должна учитывать ограничения на значения полей объекта (например, неотрицательный размер стороны геометрической фигуры). Фактически, здесь должна производиться обработка исключений при попытке ввода неправильных значений.

9. Особое внимание обратите на визуальную аккуратность создаваемых вами пользовательских интерфейсов. Старайтесь выравнивать элементы по левому краю относительно друг друга, делать одинаковые отсту-

пы между элементами, правильно подписывать элементы, кнопки и заголовки. Грамотно рассчитывайте размеры элементов — если в TextBox должно вводиться целое число со значением до 100, не имеет смысла делать его длиннее 50 пикселей. Также поля для фамилии должны быть подходящего размера, чтобы корректно отображать обычную фамилию, — не слишком длинные, но и не слишком короткие. Аккуратность и удобство пользовательского интерфейса может стать решающим фактором в выборе именно вашей программы конечным пользователем.

10. При тестировании и отладке программы не очень удобно вручную добавлять новые объекты — необходимость каждый раз вводить данные для 10 объектов может сильно пошатнуть психическое состояние разработчика (или вашего преподавателя). Чтобы облегчить тестирование программы, а значит, и собственную разработку, добавьте на форму создания нового объекта кнопку Create Random Data. По нажатию данной кнопки все поля будут заполняться случайными правильными данными для объекта. Пользователю останется только нажать кнопку Ok для добавления нового объекта на главную форму.

11. Кнопка Create Random Data является отладочной, и в версии, которая будет поставляться конечному пользователю, этой кнопки быть не должно — не будет же бухгалтерия создавать «случайных» работников со «случайными» зарплатами! Удалять же и заново создавать эту кнопку при необходимости нового установщика опять же не очень удобно — вы можете просто забыть это сделать. Используйте механизм условной компиляции.

12. Добавьте форму, на которой можно будет провести поиск объекта по каждому из полей общих для всех дочерних классов. Помните, что результатом поиска может быть не один объект. Добавьте на главную форму кнопку для вызова формы поиска.

13. Добавьте возможность сохранения и загрузки введенных пользователем данных, используя любой механизм сериализации, на ваше усмот-

рение. Сохранять данные необходимо в файл с расширением, которое будет характерно только для вашей программы (не надо использовать известные форматы, например *.doc, *.txt или *.xml).

2.7 Список используемых источников

1. Регулярные выражения. Википедия, свободная энциклопедия. [Электронный ресурс]. — URL: https://ru.wikipedia.org/wiki/Регулярные_выражения (дата обращения 29.12.2014).

2. Элементы языка регулярных выражений — краткий справочник. Microsoft Software Developer Network [Электронный ресурс]. — URL: http://msdn.microsoft.com/ru-ru/library/az24scfc(v=vs.110).aspx (дата обращения 29.12.2014).

3. Регулярные выражения в .NET Framework. Microsoft Software Developer Network [Электронный ресурс]. — URL: http://msdn.microsoft.com/ru-ru/library/hs600312(v=vs.110).aspx (дата обращения 29.12.2014).

3 Лабораторная работа № 3. Система контроля версий

Целью данной работы является знакомство с распределённой системой управления версиями файлов Git и процессом работы с ним, а также с сервисом GitHub.

3.1 Об управлением версиями

Описание этой главы основано на книге [1]. Более подробную информацию о работе с Git и особенности его устройства можно взять напрямую из источника.

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 изменял 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.

3.1.1 Локальные системы управления версиями

Многие люди, чтобы управлять версиями, просто копируют файлы в другой каталог (некоторые 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. 3.1). 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.

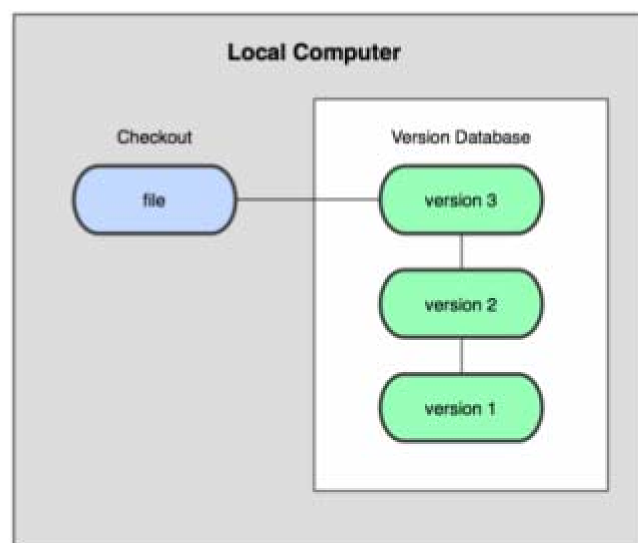


Рисунок 3.1 — Схема хранения версий на локальном компьютере

3.1.2 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 (см. рис. 3.2).

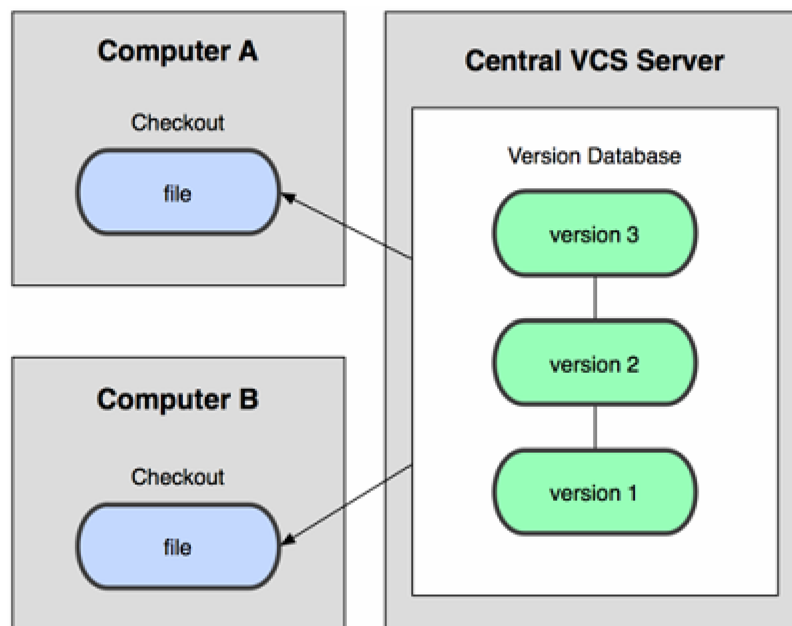


Рисунок 3.2 — Схема централизованной СУВ

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 раз-

работчики 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.

3.1.3 Распределённые 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 (см. рис. 3.3).

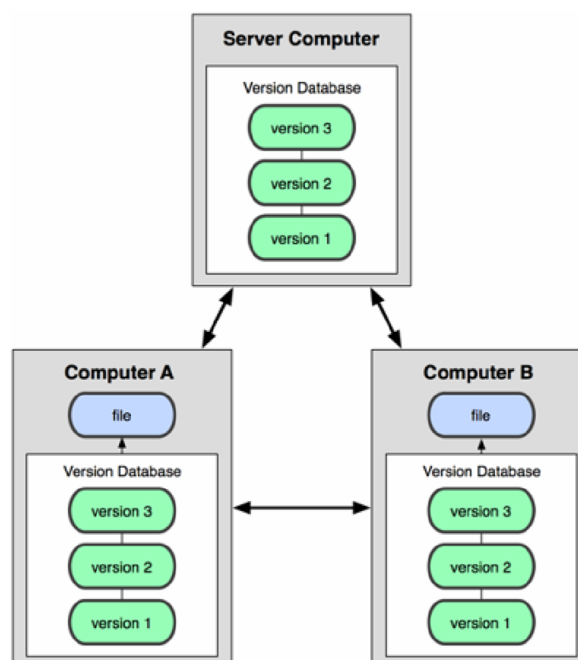


Рисунок 3.3 — Схема распределённой СУВ

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.

3.1.4 Краткий экскурс в историю появления Git

История Git тесно связана с историей 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.

3.1.5 Особенности Git

Так что же такое Git в двух словах? Эту часть важно усвоить, поскольку если вы поймете, что такое Git и каковы принципы его работы, вам будет гораздо проще пользоваться им эффективно. Изучая Git, постарайтесь освободиться от всего, что вы знали о других СУВ, таких как Subversion или Perforce. В Git совсем не такие понятия об информации и работе с ней, как в других системах, хотя пользовательский интерфейс очень похож. Знание этих различий защитит вас от путаницы при использовании Git.

3.1.6 Слепки вместо патчей

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, как показано на рис. 3.4.

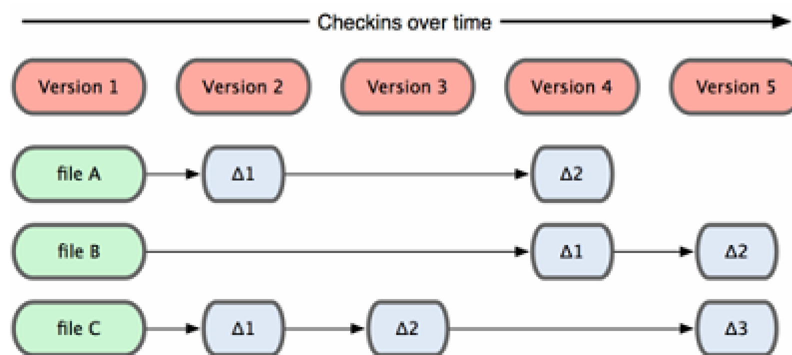


Рисунок 3.4 — Схема представления данных в других СУВ

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, со-

хранят 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. То, как Git подходит к хранению данных, похоже на рис. 3.5.

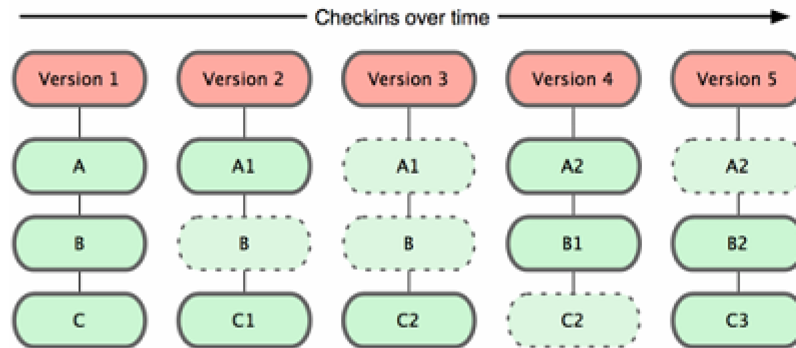


Рисунок 3.5 — Схема представления данных в Git

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.

3.1.7 Почти все операции — 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 СУВ 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.

3.1.8 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.

3.1.9 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 СУВ, 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.

3.1.10 Три состояния

Теперь внимание. Это самое важное, что нужно помнить про Git, если вы хотите, чтобы дальше изучение шло гладко. 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.

3.2 Сервис Github

GitHub — это один из крупнейших веб-сервисов для хостинга IT-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 [2]. После регистрации — заведите репозиторий (зайдите во вкладку Repositories и нажмите на New). При заведении 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 14.12.2014). 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, коммен-

тировать 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 СУВ 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 расмотрена 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.

3.3 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). В данном разделе будут рассмотрены инструменты для работы с Git под Windows.

3.3.1 Работа с помощью командной строки

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*, расположенного 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».

Другая полезная команда, 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. 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 кобинаций, 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 скобочках 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.

3.3.2 Работа с помощью SourceTree (GUI)

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, как на рис. 3.6.

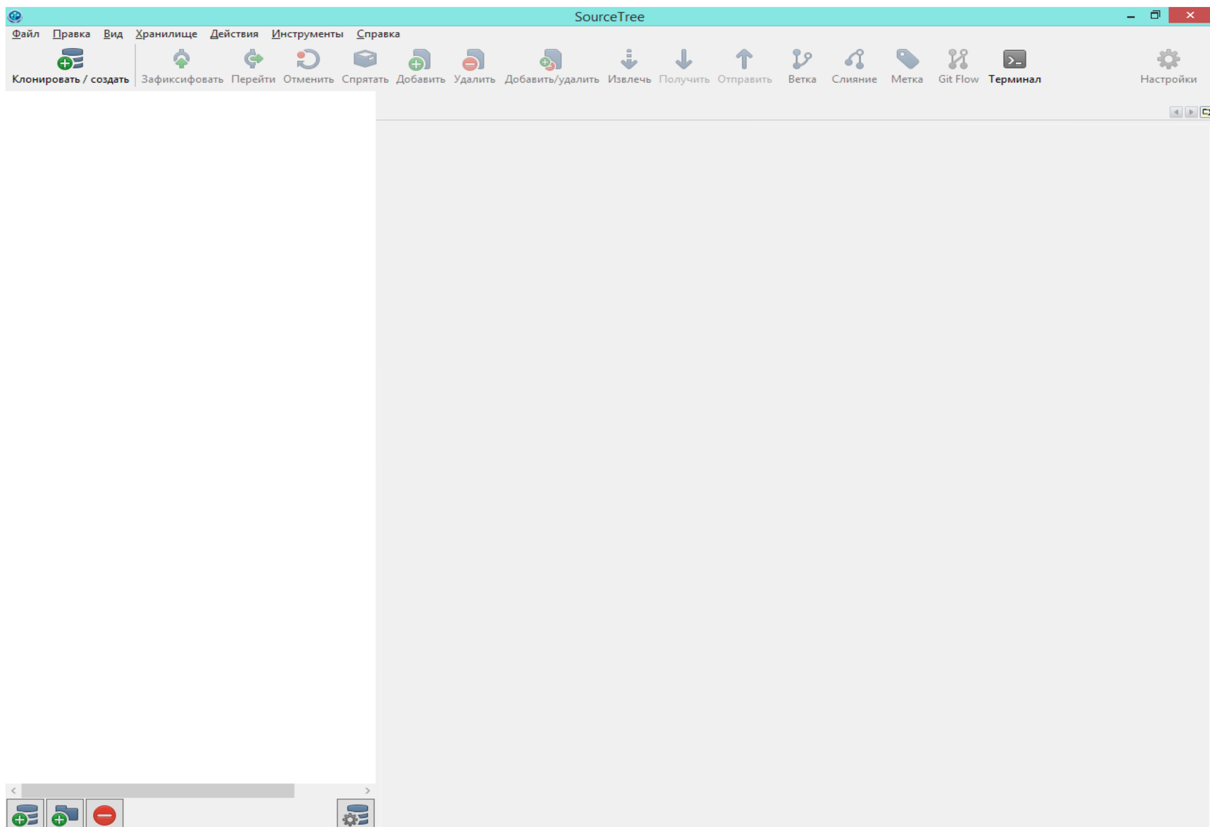


Рисунок 3.6 — Пользовательский интерфейс SourceTree

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 как на рис. 3.7.

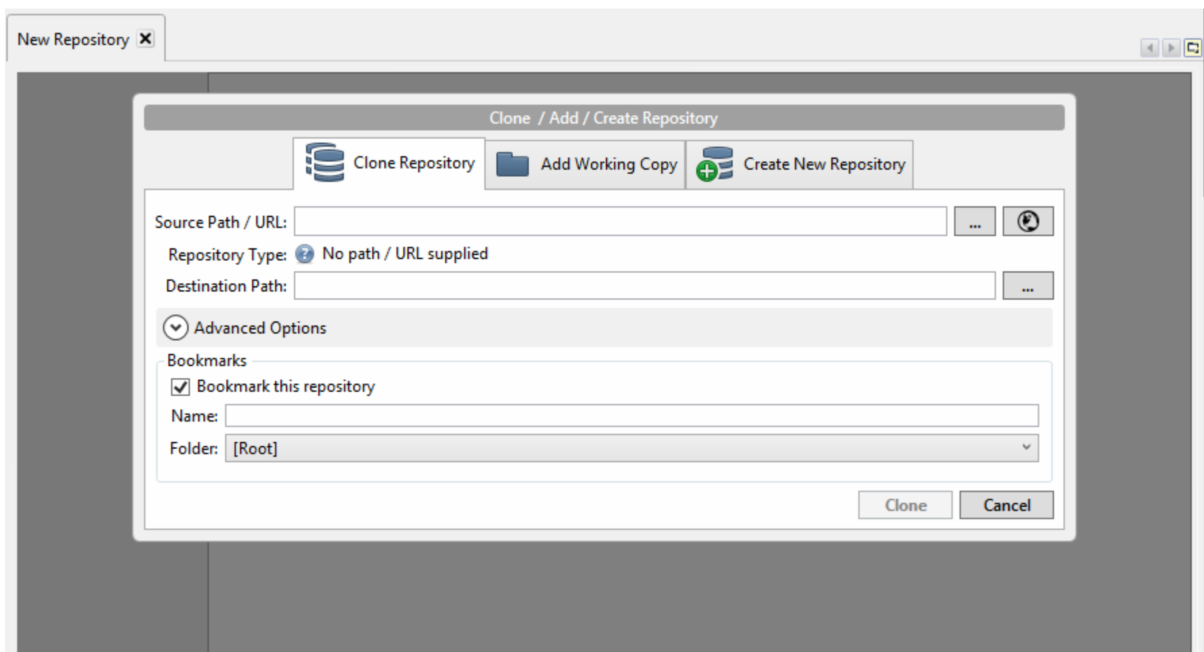


Рисунок 3.7 — Форма клонирования Git-репозитория

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 (см. рис. 3.8).

Рисунок 3.8 — Ссылка на репозиторий в пользовательском интерфейсе *SourceTree*

Помимо этого, GitHub поддерживает обращение к репозиторию не только как к *.git, но и по URL (см. рис. 3.9).

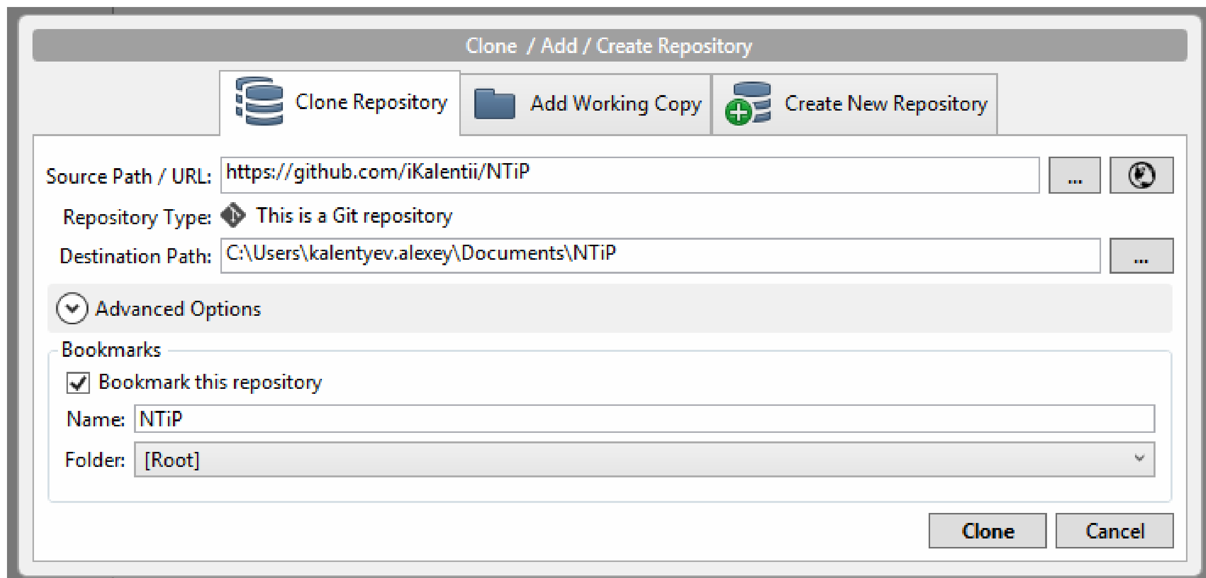


Рисунок 3.9 — Форма клонирования Git-репозитория со ссылкой

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* (см. рис. 3.10).

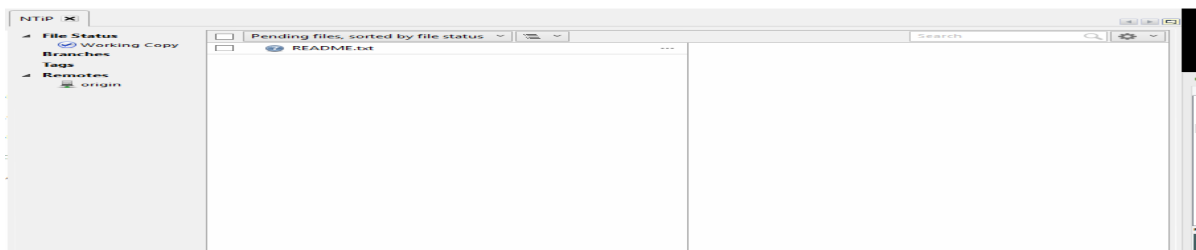


Рисунок 3.10 — Форма с файлом, не находящимся под версионным контролем

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* (см. рис. 3.11).

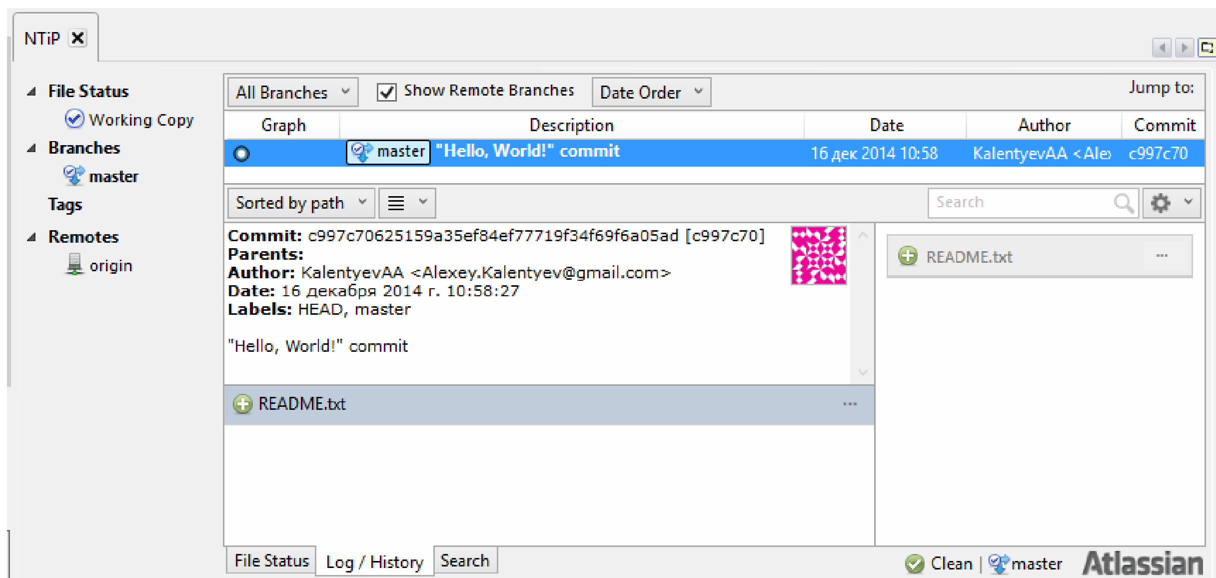


Рисунок 3.11 — Первый коммит в *SourceTree*

При необходимости получить/залить изменения из/на GitHub нажмите кнопку *Pull/Push* (см. рис. 3.12).

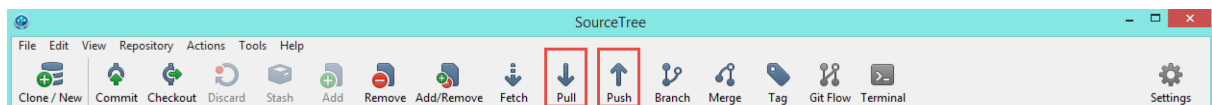


Рисунок 3.12 — Рабочая панель *SourceTree*

Измените файл README.txt. После этого в SourceTree появится предупреждение, что у вас есть не добавленные в коммит изменения (см. рис. 3.13).

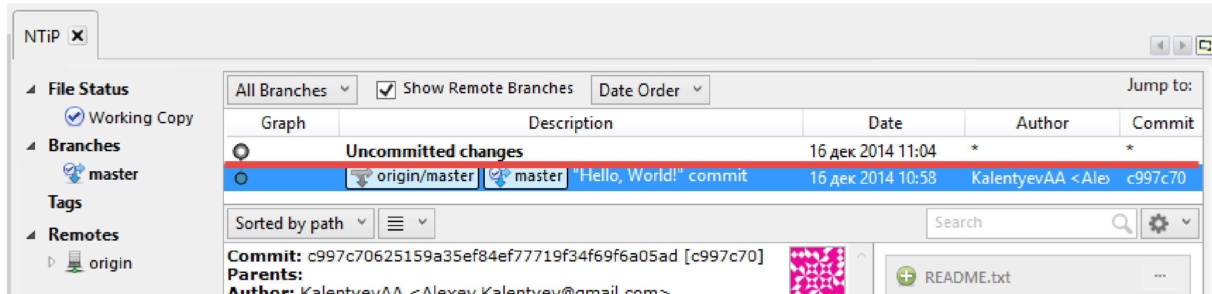


Рисунок 3.13 — Показ не добавленных в коммит изменений в *SourceTree*

Перейдя на выделенный пункт, вы увидите, какие именно файлы были изменены, а в правом окне вы увидите, что именно было изменено в файле (см. рис. 3.14).

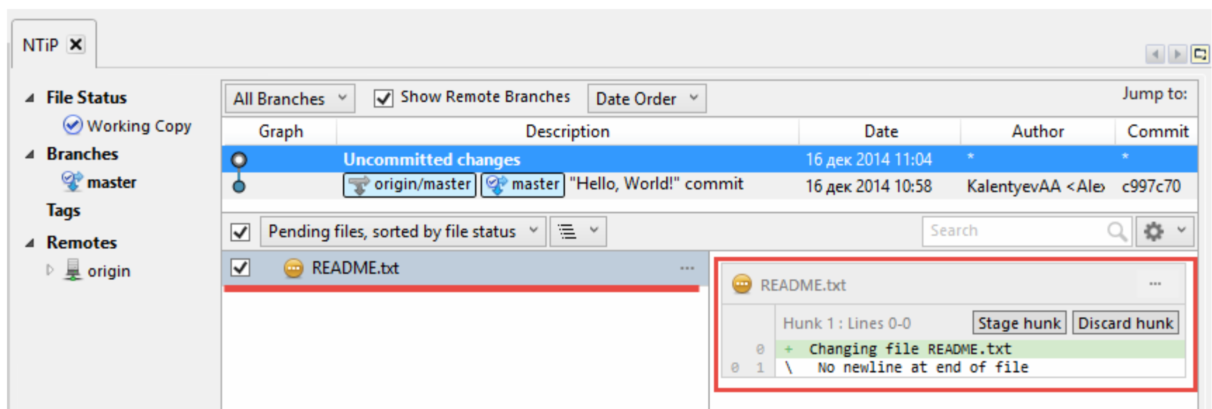


Рисунок 3.14 — Отображение изменённых файлов в SourceTree

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* (см. рис. 3.15).

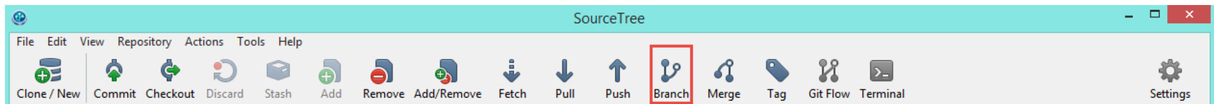


Рисунок 3.15 — Рабочая панель *SourceTree*

Введите имя ветки (см. рис. 3.16).

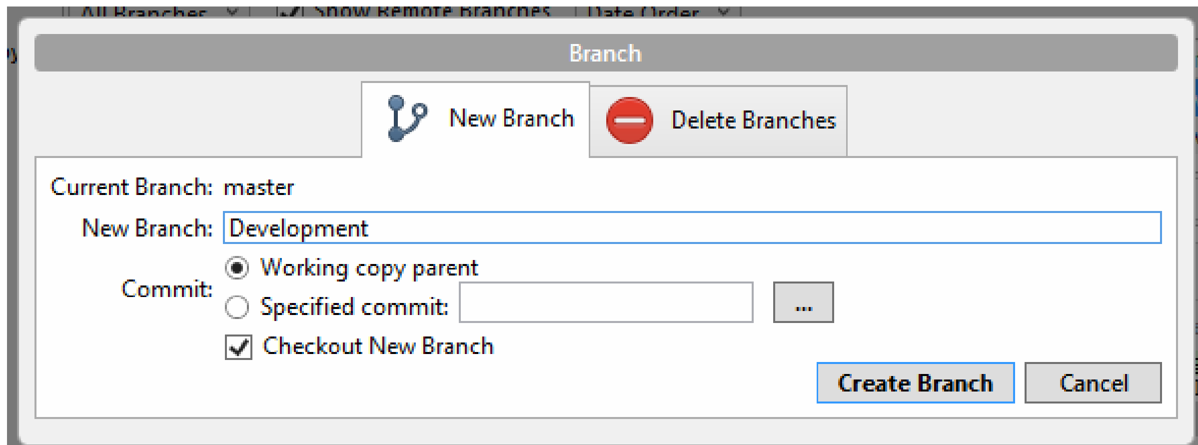


Рисунок 3.16 — Диалог создания ветки в *SourceTree*

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 (см. рис. 3.17).

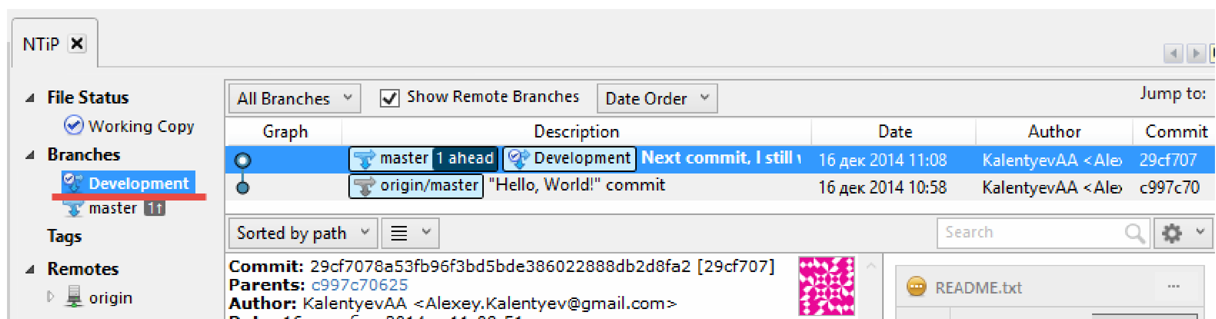


Рисунок 3.17 — Созданная ветка в *SourceTree*

Поменяйте файл README.txt несколько раз и выполните коммит после каждого изменения (см. рис. 3.18).

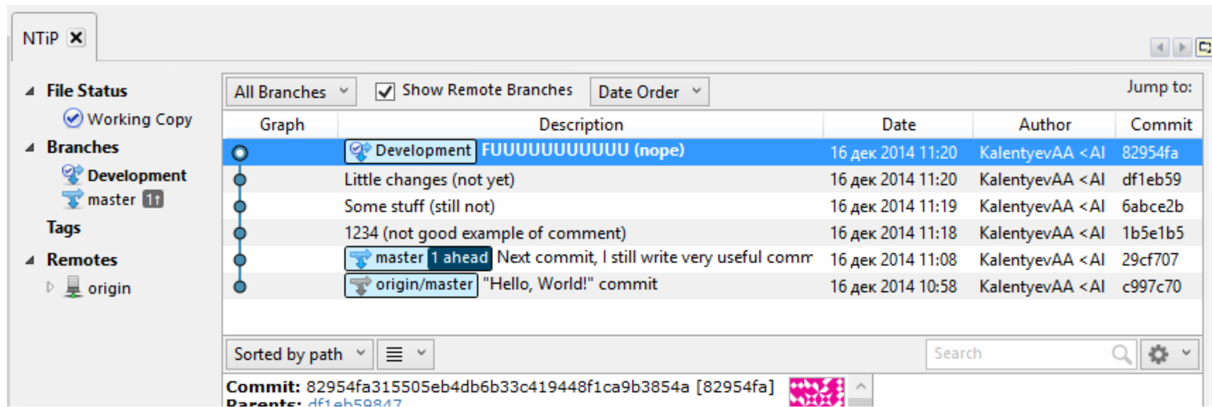


Рисунок 3.18 — *SourceTree* после нескольких коммитов

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* (см. рис. 3.19).

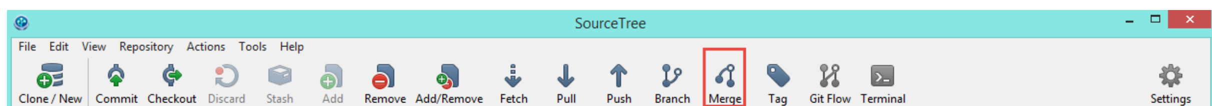


Рисунок 3.19 — Рабочая панель *SourceTree*

В окне Merge выбираем последний коммит в ветке *Development*. Всё, вы выполнили слияние двух веток (см. рис. 3.20).

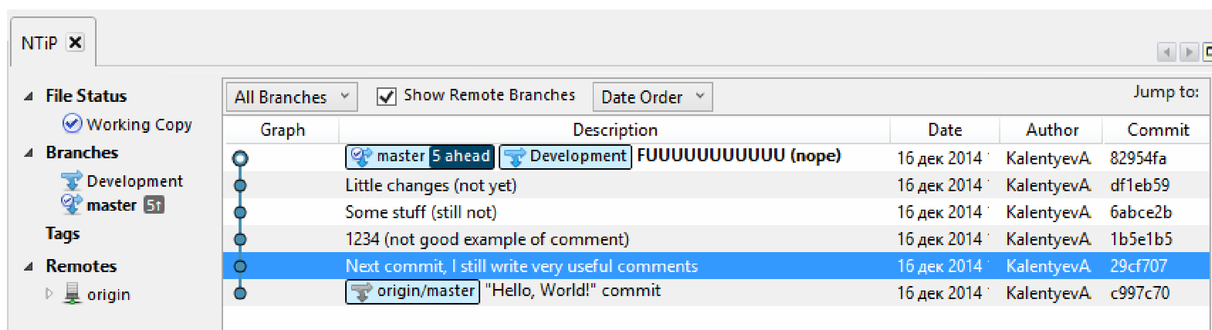


Рисунок 3.20 — Выполнение слияния с веткой *Development*

Для того чтобы увидеть систему ветвления вашего проекта в *Source-Tree* (см. рис. 3.21), необходимо до выполнения слияния с основной веткой выставить настройки (*Tools->Options->Git*), показанные на рис. 3.22.

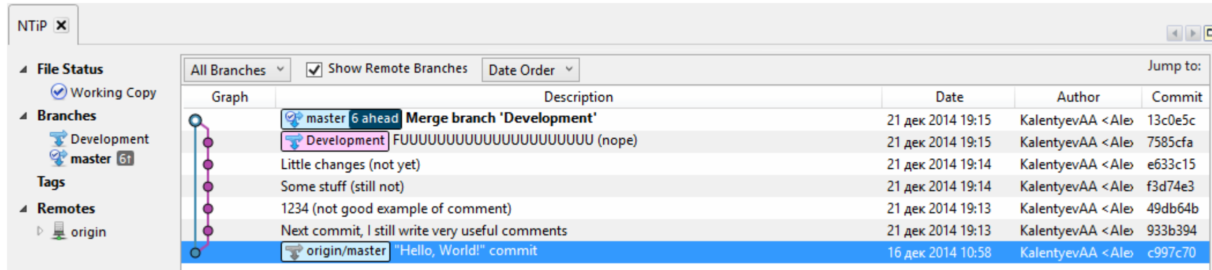


Рисунок 3.21 — Отображение системы ветвления, после слияния с веткой *Development*

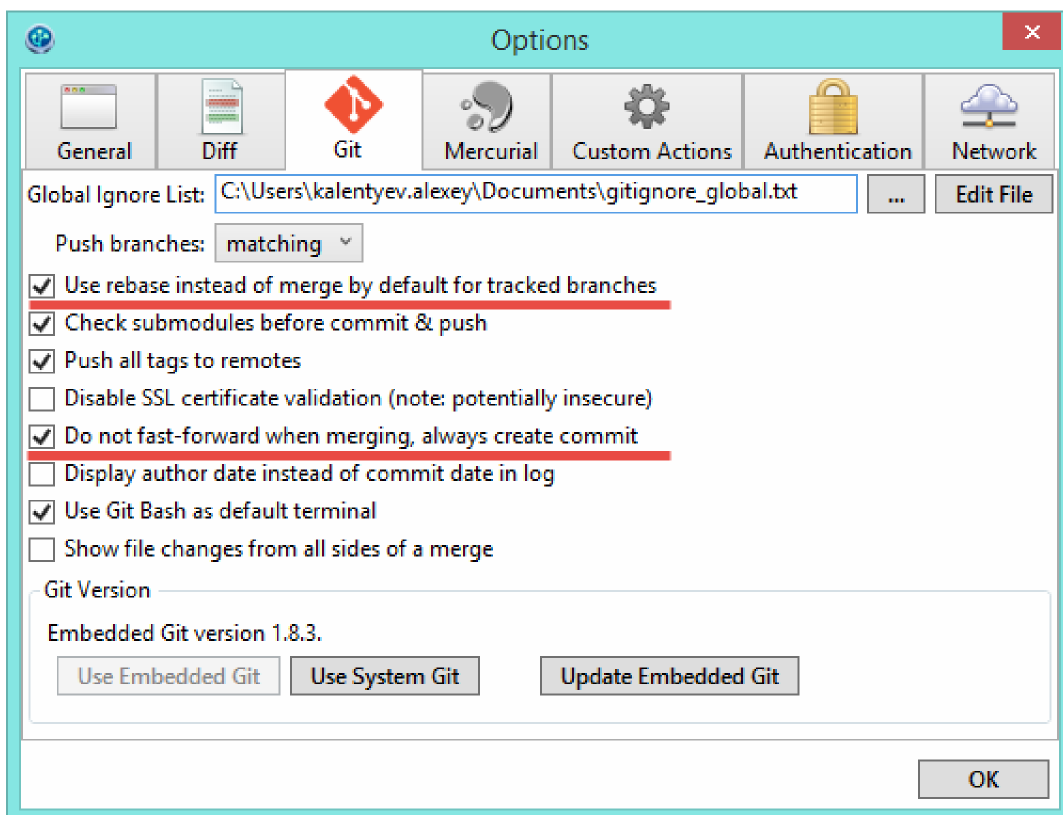


Рисунок 3.22 — Настройки *SourceTree* для показа системы ветвления проекта

После этого можно удалить ветку *Development*. Это можно сделать, например, наведя мышкой на ветку и выбрав *Delete Development* в контекстном меню (см. рис. 3.23).

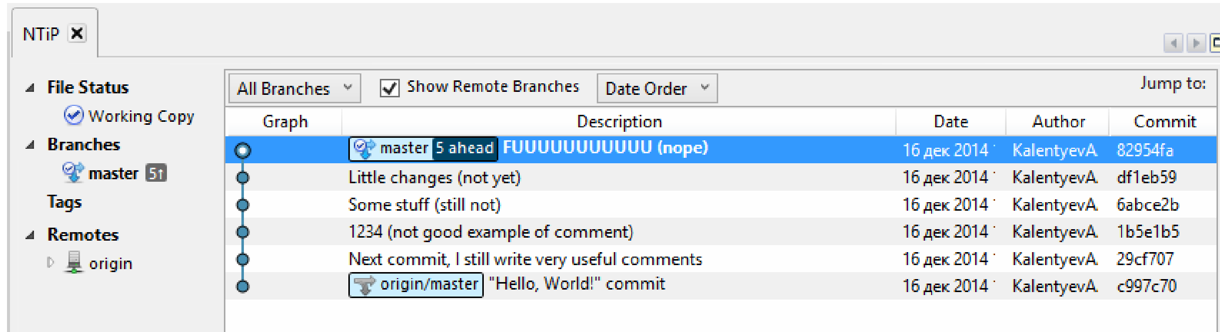


Рисунок 3.23 — Удаление ветки *Development*

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 (см. рис. 3.24).

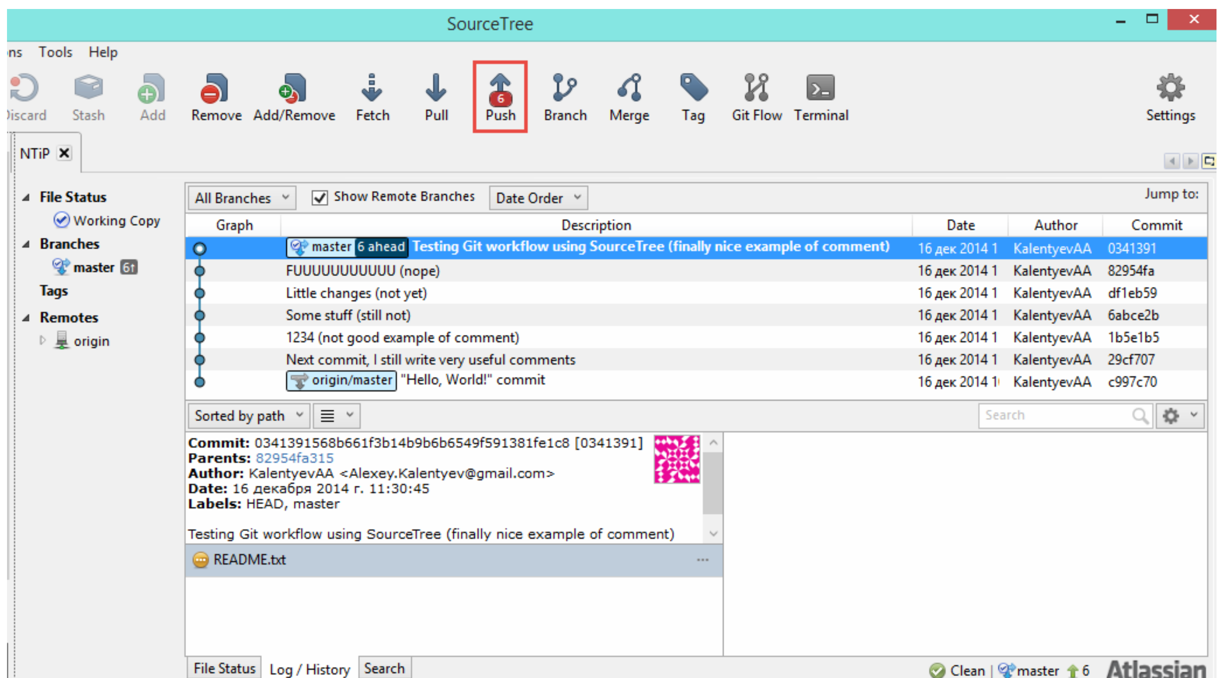


Рисунок 3.24 — Отображение изменений для отправки в изначальный репозиторий

3.4 Удачная модель ветвления для Git

Для того чтобы не увеличивать без необходимости пособие, студенту предлагается ознакомиться с удачной моделью ветвления для Git самостоятельно.

Оригинальная статья на английском находится по адресу [7], хороший русский перевод находится тут [8].

Понимание этой модели ветвления должно сформировать у студента понимание организации достаточно удобного процесса командной разработки с помощью Git. Такая модель наиболее хорошо описывает работу над реальным продуктом в динамике. Показанные концепции стройно ложатся на модель ветвления Git, позволяя наиболее удобно вносить изменения в существующую функциональность продукта, отрабатывать возникающие в процессе работы ошибки, а также параллельно работать над новыми функциями.

3.5 Задание на лабораторную работу

1. Зарегистрируйтесь на GitHub.com.
2. Создайте публичный репозиторий для своего проекта.
3. Установите на своей машине Git клиент.
4. Склонируйте репозиторий на свою машину.
5. Добавьте в склонированный репозиторий ваш проект, над которым вы работали в лабораторных 1 и 2.
6. Сделайте ветку Development и перейдите в неё.
7. Выполните несколько изменений файлов (такими изменениями может быть улучшение читаемости кода, добавление комментариев и пр.). После каждого атомарного изменения выполняйте коммит в репозиторий. К каждому коммиту добавьте осмысленный комментарий, чтобы при необходимости можно было опознать, какие были выполнены изменения.
8. После выполнения некоторых изменений слейте ветку Development и главную ветку master.

9. Отправьте изменения на GitHub.com.

10. Всю дальнейшую работу с лабораторными заданиями выполняйте при параллельном использовании версионного контроля. Любое изменение логически должно быть зафиксировано — делайте коммиты в репозиторий. При необходимости проведения определённых улучшений программы — создавайте дополнительные ветки, над которыми в дальнейшем выполняйте слияние с главной веткой и фиксируйте промежуточные варианты в GitHub.com.

3.6 Список использованных источников

1. Git — Book. [Электронный ресурс]. — URL:http://git-scm.com/book/ru/v1 (дата обращения 18.12.2014).

2. GitHub. [Электронный ресурс]. — URL:https://github.com (дата обращения 18.12.2014).

3. Git — Downloading Package. [Электронный ресурс]. — URL:http://git-scm.com/download/win (дата обращения 18.12.2014).

4. gitignore/VisualStudio.gitignore. [Электронный ресурс]. — URL: https://github.com/github/gitignore/blob/master/VisualStudio.gitignore (дата обращения 18.12.2014).

5. Free Mercurial and Git Client for Windows and Mac | Atlassian SourceTree[Электронный ресурс]. — URL: http://www.sourcetreeapp.com/ (дата обращения 18.12.2014).

6. Git — GUI Clients [Электронный ресурс]. — URL: http://git-scm.com/downloads/guis (дата обращения 18.12.2014).

7. A successful Git branching model// nvie.com. [Электронный ресурс]. — URL: http://nvie.com/posts/a-successful-git-branching-model/ (дата обращения 18.12.2014).

8. Удачная модель ветвления для Git // Хабрахабр [Электронный ресурс]. — URL: http://habrahabr.ru/post/106912/ (дата обращения 18.12.2014).

4 Лабораторная работа № 4. Юнит-тестирование

Юнит-тестирование (англ. «*unit-testing*», или блочное тестирование) — тестирование отдельного элемента изолированно от остальной системы. Относительно парадигмы объектно-ориентированного программирования системой является вся программа, а отдельным элементом — класс или его метод. Юнит-тестирование предназначено для проверки правильности работы отдельно взятого класса. Чтобы исключить из результатов тестирования влияние потенциальных ошибок других классов, тестируемый класс должен быть максимально изолирован, т. е. не использовать объекты и методы других классов. Данное требование в итоге позволяет иначе взглянуть на взаимодействие классов и выполнить рефакторинг на уменьшение связности классов.

Фактически, юнит-тестирование заключается в написании некоторого класса-обёртки, который бы создавал экземпляр тестируемого класса. В классе-обёртке создаются методы-тесты, выполняющие следующий алгоритм:

1. Создаются входные параметры — тестовые данные.
2. Подают тестовые данные на вход общедоступного (с модификатором доступа public в C#) метода тестируемого класса.
3. Сравнивают возвращаемое тестируемым методом значение с некоторым эталоном — заранее известным результатом, который должен получиться при правильной работе данного метода. Данный эталон определяется в ТЗ, спецификациях или другой проектной документации.
4. Если результат работы метода совпадает с эталоном — тест пройден. В любом другом случае тест считается провальным.

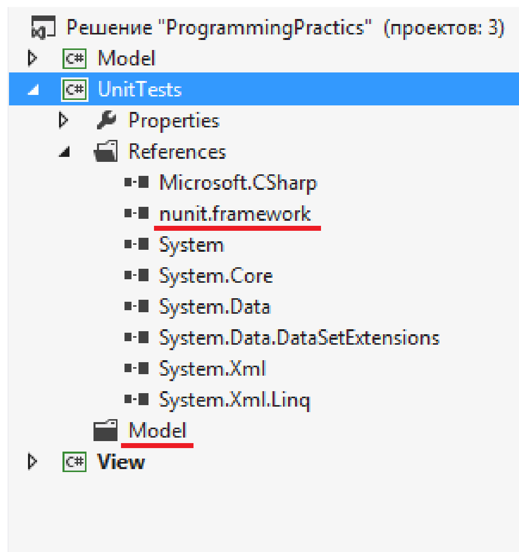
По данному алгоритму проверяется каждый общедоступный метод тестируемого класса. Защищенные или закрытые члены класса тестированию не подвергаются, чтобы не нарушать инкапсуляцию класса.

Таким образом, юнит-тесты представляют программный код, который также требует постоянной поддержки и следования определенным правилам оформления. В отличие от других видов тестирования проведение юнит-тестирования — обязанность разработчиков, а не тестировщиков.

4.1 Задание на лабораторную работу

1. Скачать библиотеку *Nunit* с сайта *nunit.org* и установить.

2. Запустите ваше решение в *Visual Studio* и создайте в нём новый проект — библиотеку классов *UnitTests*:



Новый проект *UnitTests* должен хранить в себе ссылку на библиотеку *nUnit* и на тестируемые проекты решения, например на *Model* (на рисунке не отображено). Также внутри самого проекта создается папка *Model*, в которой будут храниться все тесты для проекта *Model*

Рисунок 4.1 — Дерево решения с добавленным проектом для юнит-тестирования

Поскольку в данном проекте обычно хранятся блочные и интеграционные тесты всего решения, его внутренняя организация должна повторять организацию решения. В нашем случае, внутри проекта необходимо создать отдельную папку для хранения тестов проекта *Model*.

3. Добавьте в проект ссылку на библиотеку *Nunit* для возможности написания тестов в проекте. Данную ссылку можно подключить через вкладку «Расширения» в окне «Менеджер ссылок»:

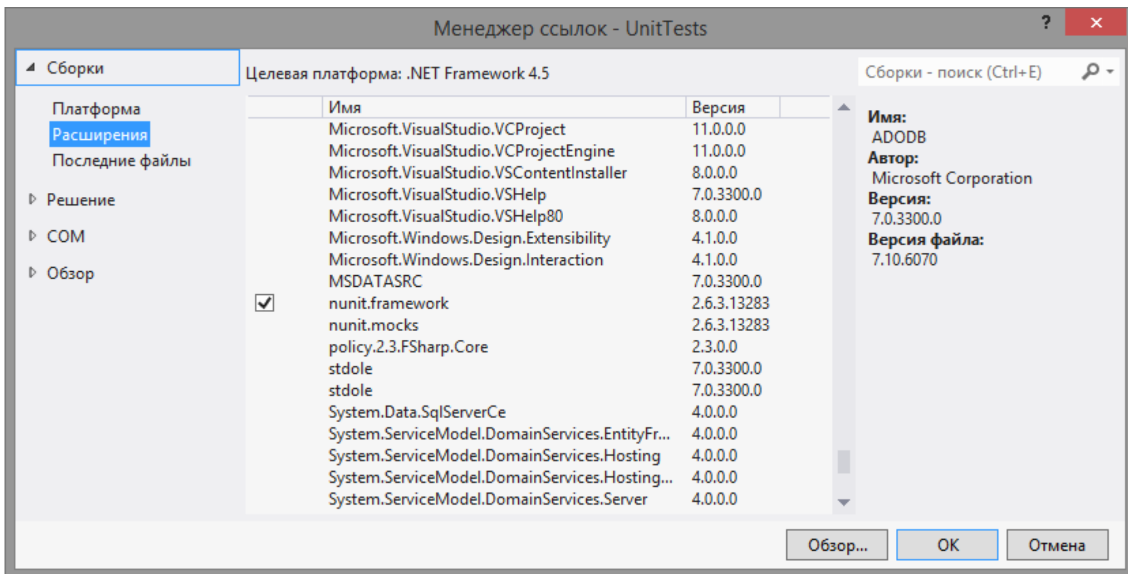


Рисунок 4.2 — Менеджер ссылок проекта в *Visual Studio*

Вам нужно подключить только «*nunit.framework*». Возможная вторая ссылка на «*nunit.mocks*» в данной лабораторной не рассматривается, она необходима для создания более сложных интеграционных тестов (см. «Мок-тестирование», «Мок-объекты»).

4. Создайте внутри папки *Model* проекта *UnitTests* класс с именем <ИмяТестируемогоКласса>*Test.cs*. В нашем примере для тестируемого класса *MyClass* класс с тестами будет называться *MyClassTest*.

5. Рассмотрим написание блочных тестов на примере следующего класса:

```
namespace Model
{
	/// <summary>
	/// Пример класса.
	/// </summary>
	public class MyClass
	{
		/// <summary>
		/// Свойство, выражающее количество чего-нибудь.
		/// </summary>
	public int Count {get; set;}

		/// <summary>
		/// Выполнить деление двух вещественных чисел.
		/// </summary>
		/// <param name=”a”>Числитель.</param>
```

```
        /// <param name="b">Знаменатель.</param>
        /// <returns> Результат деления.</returns>
        public double Divide(double a, double b)
        {
        …
        }
    }
}
```

Реализация класса опущена специально, в настоящий момент она не важна. Важно, что в классе находятся два общедоступных члена с описанием их поведения. На основе этого мы можем придумать тестовые случаи, которые и будут проверяться юнит-тестированием.

6. В первую очередь, обозначим наш класс юнит-тестов как тестовый. Для этого перед объявлением класса добавим атрибут [*TestFixture*]:

```
using NUnit.Framework;

namespace UnitTests.Model
{
    /// <summary>
    /// Набор тестов для класса MyClass.
    /// </summary>
    [TestFixture]
    public class MyClassTest
    {
    }
}
```

7. Далее сформируем метод, который будет выполнять тестирование свойства *Count*. Тест будет заключаться в том, что мы создадим экземпляр тестируемого класса и попытаемся в его свойство *Count* поместить как корректные, так и некорректные значения. Для начала напишем сам метод:

```
using Model;
using NUnit.Framework;
namespace Model
{
    /// <summary>
    /// Набор тестов для класса MyClass.
    /// </summary>
    [TextFixture]
    public class MyClassTest
    {
```

```
        /// <summary>
        /// Тестирование свойства Count.
        /// </summary>
        /// <param name=”count”>Значение свойства Count.</param>
        [Test]
        public double CountTest(int count)
        {
        var myClass = new MyClass();
        myClass.Count = count;
        }
    }
}
```

На вход метод принимает некоторое целочисленное значение, которое мы будем присваивать в свойство *Count*. Входные параметры тестового метода нужны нам для возможности создания нескольких тестовых случаев, проверяемых одним тестом.

8. Добавим некоторый тестовый случай, проверяющий работу свойства Count при обычном значении, например 4. Для этого добавим после атрибута [*Test*] еще один атрибут [*TestCase*] с описанием входных данных и названием тестового случая:

```
[Test]
[TestCase(4, TestName = "Тестирование Count при присваивании 4.")]
public void CountTest(int count)
{
    var myClass = new MyClass();
    myClass.Count = count;
}
```

9. Запустим тест и проверим, выполняется ли он. Если у вас установлен *Resharper*, для запуска тестов достаточно нажать на специальный значок на панели слева:

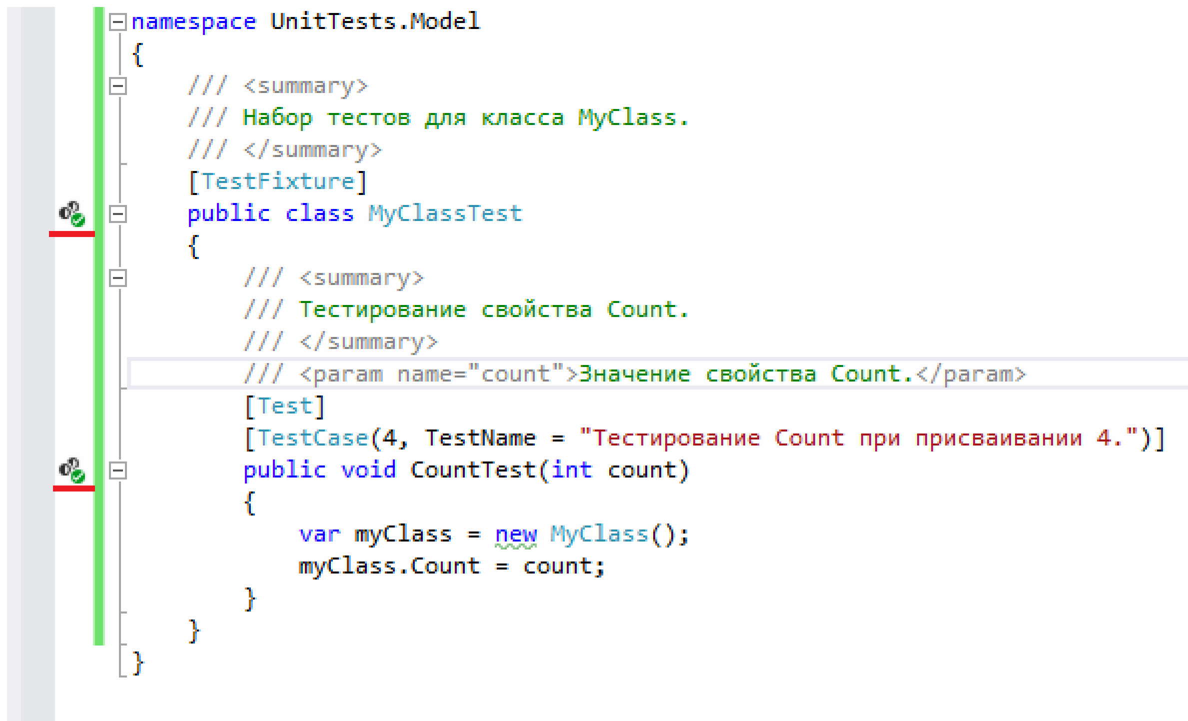


Рисунок 4.3 — Пиктограммы для запуска юнит-тестов

Первый значок предназначен для запуска всех тестов, описанных в данном классе. Второй значок предназначен для запуска конкретного тестового метода. При этом в контекстном меню можно указать, запустить все тестовые случаи или только один конкретный. У нас пока только один тест с одним тестовым случаем, его и запустим. При запуске тестов появится следующая панель «*Unit Test Sessions*» (её можно найти во вкладке *Resharper->Tools->Unit Test Sessions*):

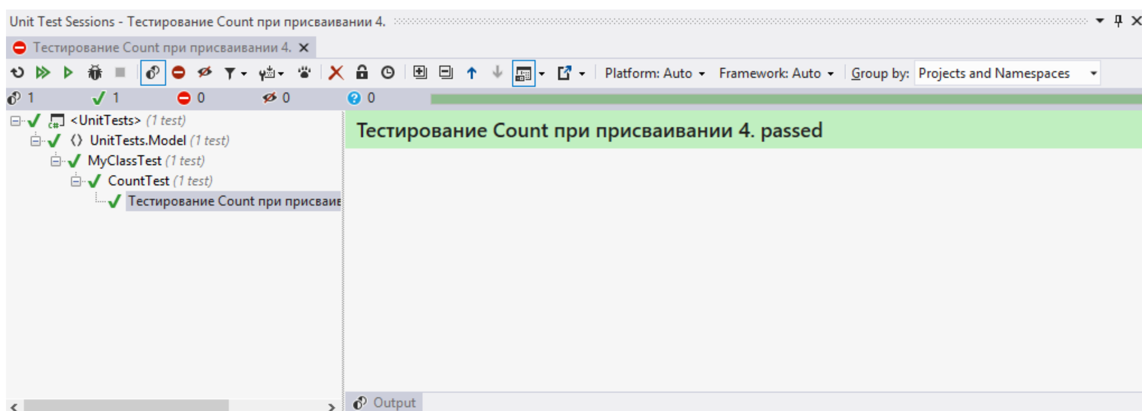


Рисунок 4.4 — Панель сессий тестирования

Данная панель показывает новую созданную сессию тестирования. Каждая сессия может содержать свой набор проверяемых тестов:

- Иерархию тестов относительно проекта и класса.
- Успешное завершение нашего теста (отмечен зеленой галочкой и статусом *Passed*).

У тестов внутри сессии возможны следующие статусы:

- Тест пройден.
- Тест провален — в результате будут показаны исходные данные, ожидаемый и фактический результаты выполнения теста.
- Результат тестирования устарел — если были совершены изменения в тестируемом или тестирующем классах.

При запуске тестов выполняется компилирование тестируемого проекта и тестирующего проекта. Тесты не будут запущены, если при компилировании какого-либо из проектов возникла ошибка. Для запуска тестов также необязательно делать проект *UnitTests* запускаемым проектом.

10. Добавим еще тестовых случаев для свойства Count. Одним из способов формирования позитивных тестовых случаев является определение краевых условий. Например, мы знаем, что свойство *Count* хранит в себе количество некоторых элементов. Это число может быть достаточно большим, но обязательно положительным или 0. Соответственно, мы определили интервал от 0 до максимального значения *int*. На основе этого предположения формируется 4 тестовых случая: минимально допустимое значение, минимально допустимое значение плюс 1, максимально допустимое значение, максимально допустимое значение –1:

```
[TestCase(4, TestName = "Тестирование Count при присваивании 4.")]
[TestCase(0, TestName = "Тестирование Count при присваивании 0.")]
[TestCase(1, TestName = "Тестирование Count при присваивании 1.")]
[TestCase(int.MaxValue, TestName = "Тестирование Count при присваи-
вании MaxValue.")]
[TestCase(int.MaxValue-1, TestName = "Тестирование Count при при-
сваивании MaxValue-1.")]
```

После запуска тестов мы убеждаемся, что свойство действительно может принимать подобные значения.

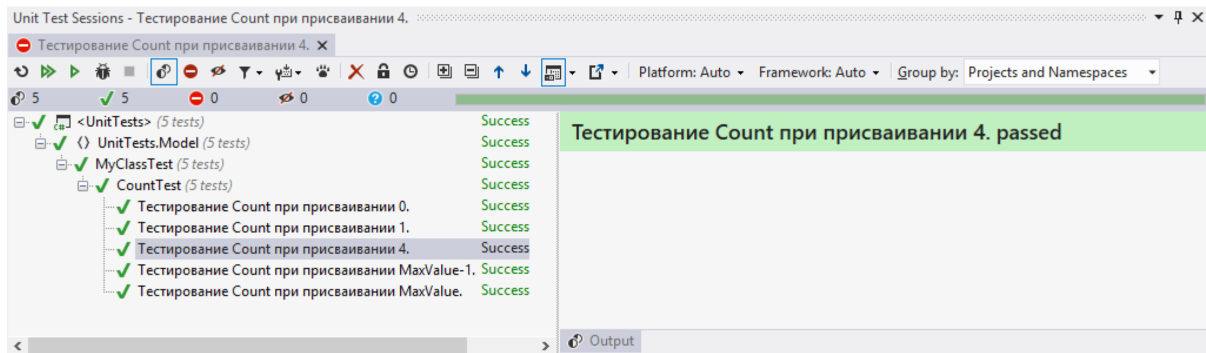


Рисунок 4.5 — Запуск тест-кейсов для одного теста

11. Мы описали *позитивные* сценарии использования свойства *Count*, т. е. мы присваивали такие значения, которые являются корректными для свойства *Count*. Однако куда более важно протестировать *негативные* сценарии — подача заведомо некорректных данных с ожиданием возникновения ошибки или исключения. Если программа не среагировала на некорректные данные исключением — это ошибка, так как неправильные входные данные для метода или свойства в итоге могут привести к неправильной работе других методов и классов, а отлаживать подобные *отложенные* ошибки достаточно трудоёмкий процесс. Поэтому:

- Программа, класс или метод должны реагировать на подачу некорректных данных исключением или любым другим способом сообщения об ошибке.
- Юнит-тестирование должно проверять негативные сценарии для классов и методов для избегания возникновения отложенных ошибок.

Для свойства *Count* подобным негативным сценарием может быть попытка присвоения отрицательного значения. Проверку выполним для двух отрицательных чисел — для –1 и самого большого допустимого отрицательного числа типа *int*. В данных тестовых случаях мы будем ждать

возникновения исключения *ArgumentException*, соответственно форма записи негативного тестового случая несколько изменится:

```
[TestCase(-1, ExpectedException = typeof(ArgumentException), Test-
Name = “Тестирование Count при присваивании -1.”)]
[TestCase(int.MinValue, ExpectedException =
typeof(ArgumentException), TestName = “Тестирование Count при
присваивании минимально допустимого целого числа.”)]
```

После запуска негативных тестов, они получили статус *Failed*.

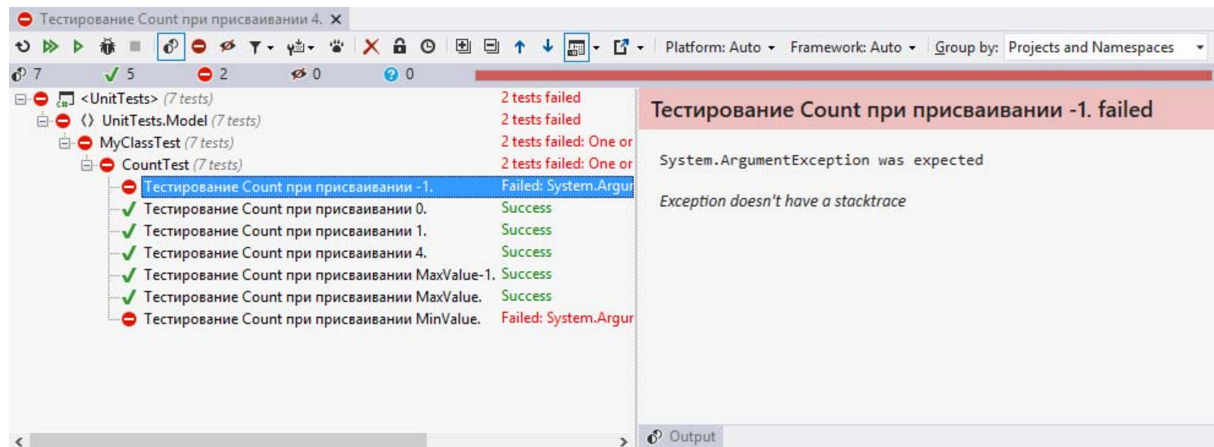


Рисунок 4.6 — Негативный юнит-тест провален

Данный статус означает, что при подаче некорректных данных исключение в свойстве *Count* не возникло, что показывает неправильную реализацию свойства *Count*. Следовательно, в сеттер свойства *Count* необходимо добавить проверку на положительность входного значения.

12. Аналогичным образом выполняется тестирование метода *Divide*. Однако здесь возможны более хитрые тестовые случаи — деление на 0, равенство одного из входных аргументов *double.NaN* или *double.Infinity*. При передаче в метод объекта ссылочного типа также стоит выполнять проверки на равенство объекта *null*.

13. Выполните тестирование ваших классов проекта *Model*. Для каждого класса создайте отдельный тестирующий класс. Для каждого общедоступного метода класса напишите тест с позитивными и негативными тестовыми случаями. Позитивные сценарии составьте на основе анализа

граничных условий, негативные — исходя из назначения метода. По каждому методу ожидается не менее 5 проверяемых тестовых случая. Исправьте реализацию методов, если в результате выполнения тестов вы обнаружите ошибку. Учтите, что ошибки возможны и в самих тестах. Будьте внимательны!

5 Лабораторная работа № 5. Рефакторинг и сборка установщика

Согласно результатам исследований 90 % рабочего времени программист занимается чтением программного кода и только 10 % уходит на его написание. Логично сделать вывод, что такое распределение времени весьма неэффективно с точки зрения разработки программного обеспечения, так как всего лишь десятая доля затрат связана с написанием новой функциональности. Следовательно, необходимо менять соотношение в пользу разработки. Одним из таких инструментов является рефакторинг.

Рефакторинг — процесс улучшения программного кода с целью улучшения его читаемости разработчиками [1]. Именно цель отличает рефакторинг от оптимизации кода, где целью является повышение производительности алгоритмов. Повышение производительности, приводит к использованию нетривиальных численных методов и алгоритмов, что в большинстве случаев, наоборот, приводит к еще более запутанному коду. Рефакторинг, в свою очередь, не рассматривает понятия производительности кода, а стремится реорганизовать программный код для упрощения его понимания.

Единственная оговорка, которую следует помнить, что в результате выполнения рефакторинга программа может потерять в производительности, но функционально программа не должна изменяться. То есть при заданных исходных данных программа должна выдавать такой же результат, что и до рефакторинга, если, конечно, в ходе рефакторинга не было обнаружено, что исходный код содержал ошибки.

Для того чтобы убедиться, что в ходе рефакторинга мы не изменили правильность выполнения программы или её части, можно (и нужно) использовать юнит-тестирование. Фактически рефакторинг состоит из следующих этапов:

1. Покрытие целевого исходного кода юнит-тестами.

2. Проверка правильности выполнения тестов для исходного кода.

3. Изменение исходного кода для повышения его читаемости.

4. Проверка правильности выполнения тестов измененного кода.

5. Если тесты не пройдены, откатить изменения исходного кода или устранить ошибку.

Где этапы 3—5 могут многократно повторяться до тех пор, пока качество кода не станет удовлетворительным. На практике, большинство программистов пренебрегают написанием блочных тестов при рефакторинге, что довольно часто приводит к возникновению ошибок.

В результате проведения рефакторинга повышается читаемость кода, уменьшается его сложность, архитектурно код становится более гибким, его легче модифицировать под расширение новой функциональности. Таким образом, рефакторинг решает огромное количество проблем при разработке ПО и становится полезной практикой. Подход рефакторинга также широко применяется программистами при разборе чужого исходного кода, так как в процессе рефакторинга программист максимально глубоко погружается в принципы работы кода. Более подробно о рефакторинге и качестве программного кода можно почитать в книгах Мартина Фаулера «Рефакторинг»[2] и Стивена МакКоннелла «Совершенный код» [3].

5.1 Задание на лабораторную работу

Перед проведением рефакторинга в данной лабораторной работе выполним следующее задание.

1. Добавьте на главную форму панель, аналогичную окну *AddObject* или *ModifyObject*, которая показывает поля выбранного в таблице объекта. В настоящий момент таблица показывает только основные, общие для всех типов объектов поля, однако для просмотра уникальных полей каждого объекта приходится заходить в окно *ModifyObject*. Такой подход может

привести к случайному изменению данных. Отображение же всех возможных полей в таблице приводит к её избыточности, так как большинство уникальных полей для разных типов дочерних объектов будут всегда пустыми. Для решения данной проблемы добавим на главную форму панель, которая показывает все возможные поля выбранного объекта. Таким образом, в таблице отображаются лишь общие поля, однако если пользователь захочет просмотреть более подробную информацию, ему достаточно выбрать нужный объект в таблице, и его данные появятся на панели. При этом панель не должна предоставлять возможности изменения данных — для этого есть специальное окно *ModifyObject*.

2. Избавьтесь от дублирования кода, создав пользовательский элемент управления *ObjectControl*. Если вы проанализируете свой программный код, то заметите, что в нём присутствует изрядная доля дублирования. Окна *AddObject* и *ModifyObject*, а также ранее добавленная панель имеют одну и ту же верстку и выполняют одну и ту же функциональность. Разница между ними заключается лишь в том, что *AddObject* создает новый объект, *ModifyObject* инициализируется ранее созданным объектом, а панель инициализируется ранее созданным объектом, но не позволяет его менять. Такое дублирование избыточно, что в дальнейшем может привести к проблеме поддержки. Поэтому лучше вынести данную функциональность в отдельный элемент управления, который можно было бы использовать во всех трёх случаях.

Пользовательские элементы управления создаются аналогично формам. Для этого нажмите правой кнопкой по проекту в Обозревателе решения, выберите «Создать новый элемент», затем в появившемся окне выберите *WinForms UserControl* (пользовательский элемент управления *WinForms*).

Как уже упоминалось, пользовательские элементы управления аналогичны формам, они также разрабатываются из уже готовых элементов

управления, например *TextBox*, *Button*, *ListBox*, или элементов, которые вы создали ранее. Однако отличие заключается в том, что формы являются самостоятельными элементами и не могут быть в составе других форм. Для них мы можем вызвать метод *Show*, и форма отобразится на экране. Пользовательский элемент управления не может быть отображен самостоятельно, он обязательно должен располагаться на форме для его отображения.

Разработайте пользовательский элемент управления *ObjectControl*. Целью данного элемента является создание и отображение объектов любого дочернего типа. Таким образом, элемент должен инкапсулировать проверки правильности ввода значений соответствующих полей.

3. Добавьте элементу *ObjectControl* открытое свойство *Object*. Реализуйте свойство *Object* типа данных *Object* (где под *Object* подразумевается ваш базовый класс или интерфейс в бизнес-логике). Логика работы данного свойства следующая: если в данное свойство присвоить некоторый объект (аксессор *set*), то все поля ввода на пользовательском элементе управления должны инициализироваться соответствующими значениями из присвоенного объекта. Таким образом, если в вашем варианте объектом является *Person*, то при присваивании объекта *Person* данному свойству все поля *ObjectControl* должны проинициализироваться данными из присвоенного *Person*. Если из данного свойства запросить значение (аксессор *get*), то элемент должен создать экземпляр *Person* и проинициализировать его значениями из полей ввода.

Таким образом, данное свойство значительно упростит работу с данным элементом управления, позволяя либо инициализировать его значениями, либо получать готовый сформированный объект согласно введенным значениям.

4. Добавьте элементу *ObjectControl* открытое булево свойство *ReadOnly*. Свойство должно работать следующим образом: при присвое-

нии в свойство значения *true* (аксессор *set*) все поля ввода на пользовательском элементе должны стать *ReadOnly*, т. е. только отображать данные, но не позволять их ввод. При присвоении значения *false* пользовательский элемент вновь позволит редактировать данные полей. При получении значения свойства (аксессор *get*) свойство просто возвращает текущее состояние *ReadOnly*. Данное свойство в итоге позволит использовать данный элемент управления для отображения данных, без возможности их изменения.

5. Убедитесь, что у созданного элемента управления кроме двух описанных свойств больше нет открытых членов класса. Два описанных свойства должны быть единственными способами взаимодействия с данным элементом управления: они позволяют инициализировать поля значениями объекта, получить созданный объект из элемента, а также управлять возможностью редактирования.

6. Переделайте пользовательский интерфейс программы с использованием созданного элемента *ObjectControl*. Перепишите главную форму, формы *AddObjectForm* и *ModifyObjectForm* с использованием элемента управления. Учтите, что формы *AddObjectForm* и *ModifyObjectForm* также можно объединить в одну форму, и тем самым сократив количество кода.

7. Соберите установщик программы с помощью программы *InnoSetup*. *InnoSetup* — программа, позволяющая быстро и удобно собирать установщики приложений с помощью скриптов. Изучение написания скриптов для *InnoSetup* остается на самостоятельное изучение. Программа содержит обширную документацию с реальными примерами скриптов.

ПРИМЕЧАНИЕ: для сборки установщика используйте программу, скомпилированную под версией *Release*, а не *Debug*, так как *Debug*-версия содержит лишний промежуточный код, необходимый при отладке. Такая версия будет менее производительной по сравнению с *Release*-версией. После компиляции проекта все необходимые файлы программы можно бу-

дет взять в папке *bin\release* компилирующегося проекта. Не забудьте удалить лишние файлы и оставить только *dll* и *exe*.

*8. Реализуйте возможность запуска файлов проекта по двойному клику в файловой системе. Ваша программа может сохранять базу данных в отдельный файл. Для пользователя было бы гораздо удобнее, если программу можно было бы запускать по двойному клику файла базы данных непосредственно из файловой системы.

Чтобы сделать данную функциональность необходимо:

1. Изменить установочный скрипт *InnoSetup*. При установке установщик должен добавить в реестр *Windows* записи о вашем собственном формате файлов и о том, какая программа по умолчанию должна открывать данный тип файлов. В вашем случае, программой по умолчанию является ваша программа. Реализация подобного скрипта представлена в обучающих примерах программы *InnoSetup*.

2. Изменить конструктор главной формы и метод *Main* для возможности принятия переменных из командной строки.

5.2 Список использованных источников

1. Калентьев А. А. Новые технологии в программировании : учеб. пособие / А. А. Калентьев, Д. В. Гарайс, А. Е. Горяинов. — Томск : Эль Контент, 2014. — 176 с.

2. Фаулер М. Рефакторинг. Улучшение существующего кода [Текст] / М. Фаулер. — М. : Символ, 2008. — 432 с.

3. Макконнелл С. Совершенный код. Мастер-класс / С. Макконнелл. — М. : Русская редакция, 2013. — 896 с.

6 Лабораторная работа № 6. Проектная документация

Лабораторная работа направлена на обучение разработки проектной документации на созданный программный продукт. Имеющиеся типы проектной документации, которые необходимо включить в конечный отчёт, более подробно описаны в [1]. Отчёт необходимо оформить согласно Образовательному стандарту ТУСУРа 01-2013 [2]. Оформление отчёта, согласно стандарту, является обязательным этапом подготовки технического специалиста, т.к. очень часто рабочая документация на предприятии или фирме регламентируется либо внутренними документами, либо ГОСТами. Оформление отчёта можно выполнять в доступном текстовом процессоре (Microsoft Word, OpenOffice, LaTeX, Words и пр.).

6.1 Задание на лабораторную работу

Ниже представлены основные пункты отчёта, которые необходимо включить в конечный документ:

1. Титульный лист.

2. Содержание. При составлении содержания ознакомьтесь с инструментами автоматической генерации содержания по имеющемуся документу (с автоматической вставкой названий глав и номеров страниц). Изучение этой возможности текстового процессора поможет в дальнейшем сэкономить большое количество времени при добавлении/удалении глав из отчёта.

3. Составьте ТЗ на разработанную программу согласно информации из главы 2 учебного пособия [1].

4. Введение, в котором необходимо описать назначение программной документации, разрабатываемой в лабораторной работе.

5. Основная часть, в которой приводится описание программной системы:

a. Составьте UML диаграмму вариантов использования для разработанной программы. Подробнее о том, что такое диаграммы вариантов использования и как их составлять, можно прочитать в главе 7 учебного пособия [1].
b. Составьте UML диаграмму классов. Подробнее о том, что такое диаграммы классов и как их составлять, можно прочитать в главе 7 учебного пособия [1].
c. Для классов, образующих связь типа «общее-частное» (наследование, реализация), приведите описание. В описание включите имеющиеся поля, свойства и методы класса, их типы и входные параметры в случае методов класса. Пример оформления описания класса Person и Student из главы 7 учебного пособия [1] приведён в табл. 6.1 и табл. 6.2.

Таблица 6.1 — Описание класса Person

Название	Тип	Описание
Описание класса		
Класс *Person* — сущность для описания абстрактного человека в программе		
Свойства		
+ DateOfBirth	Data	Дата рождения человека
+ Name	string	Имя человека
+ Surname	string	Фамилия человека
Методы		
+ DoSomeWork()	void	Виртуальный метод, описывающий некоторую работу, совершаемую человеком. Перегружается в производных классах

Окончание табл. 6.1

Название	Тип	Описание
+ GetAge()	int	Метод для расчёта возраста человека. Возвращает возраст человека
# Person (Person, Data)		Конструктор для создания нового человека с помощью фамилии и имени другого человека и даты рождения. *Person* — человек, на основе имени и фамилии которого, планируется создать новый экземпляр класса. *Data* — дата рождения человека
# Person (string, string, Date)		Конструктор для создания нового человека с помощью фамилии, имени и даты рождения. *String* — имя человека. *String* — фамилия человека. *Data* — дата рождения человека
+ TalkAboutOthers (Person)	string	Метод, позволяющий одному человеку рассказать информацию о другом человеке

Таблица 6.2 — Описание класса Student

Название	Тип	Описание
Описание класса		
Класс *Student* — сущность для описания студента университета в программе		
Методы		
+ DoSomeWork()	void	Перегруженный метод базового класса, в котором описывается процесс обучения студентом

Окончание табл. 6.2

Название	Тип	Описание
# Student (Person, Data)		Конструктор для создания нового студента с помощью фамилии и имени другого человека и даты рождения. Вызывает конструктор базового класса
# Student (string, string, Date)		Конструктор для создания нового студента с помощью фамилии, имени и даты рождения. Вызывает конструктор базового класса

При описании свойств и методов класса допускается для краткости использовать принятые в UML обозначения модификаторов доступа. Помимо этого, подобное описание в MSDN-подобном [3] формате можно сгенерировать автоматически при наличии качественных XML-комментариев в коде. Использование подобных инструментов и подробный разбор XML-комментариев в пособии по лабораторному практикуму не предусмотрен, однако для общего развития и повышения профессиональных компетенций подробное изучение этого материала вполне оправдано.

Приведите дерево ветвлений Git, полученное по окончанию работы с проектом. Пример дерева ветвления Git можно найти в данном пособии в главе 3 на рис. 3.21.

Добавьте в отчёт главу, описывающую процесс тестирования вашей программы. В главе необходимо отразить два типа тестирования: функциональное и модульное.

При описании функционального тестирования необходимо привести набор тестовых случаев, выполняемых пользователем с помощью GUI, и результат работы программы на эти тестовые случаи. Помимо позитивных тестовых случаев (ввод правильных данных, корректная работа) необхо-

димо привести примеры негативных тестовых случаев (ввод некорректных данных, возможно, некорректная работа программы). При обнаружении некорректной работы программы необходимо включить результаты тестирования в документ и описать, что было изменено в программе для недопущения подобных ошибок в дальнейшем. Если ошибка будет находиться не на уровне GUI, а на уровне работы класса — необходимо написать модульный тест для покрытия подобного случая.

При описании модульного тестирования — необходимо привести набор тестовых случаев, проверяющих работу классов. Помимо этого, необходимо привести структуру тестового проекта в Visual Studio. Это можно сделать скриншотом как на рис. 6.1.

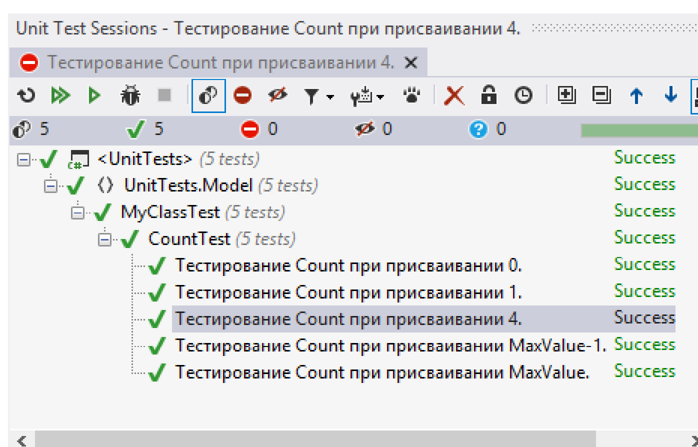


Рисунок 6.1 — Структура тестового проекта в Visual Studio

6. Заключение, глава, в которой необходимо сделать выводы по выполненной работе.

7. Список использованной литературы. Следует отметить, что на большинство источников литературы должны быть ссылки в тексте отчёта. Правильное оформление источников приведено в [2].

8. При необходимости некоторые части можно вынести в приложения. Описание оформления приложений также приведено в [2].

6.2 Список использованных источников

1. Калентьев А. А. Новые технологии в программировании : учеб. пособие / А. А. Калентьев, Д. В. Гарайс, А. Е. Горяинов. — Томск : Эль Контент, 2014. — 176 с.

2. Работы студенческие по направлениям подготовки и специальностям технического профиля. Общие требования и правила оформления. ОС ТУСУР 01-2013 53 с.

3. Microsoft Software Developer Network [Электронный ресурс]. — URL: http://msdn.microsoft.com/ru-ru/default.aspx (дата обращения 18.12.2014).