Лабораторная работа №1. Бизнес-логика

Выполните следующие задания:
1. Создайте C# решение в среде Visual Studio. Назовите его в соответствии с вашим вариантом задания. В качестве исходного типа проекта выберите проект динамической библиотеки (.dll). Назовите проект также согласно вашему варианту или просто Model. Данный проект будет содержать в себе бизнес-логику вашей будущей программы, т.е. содержать ключевые сущности, структуры данных и способы их взаимодействия.
2. Создайте сущность-интерфейс согласно вашему варианту. Интерфейс в языке C# описывается с помощью ключевого слова interface. Опишите ключевые свойства и методы интерфейса. Не забудьте о правильном именовании элементов в соответствии с RSDN. Подумайте, какие свойства и методы в дальнейшем будут являться общими (т.е. будут в интерфейсе), а какие должны быть реализованы в конкретных классах.
3. Создайте два класса, реализующих данный интерфейс. Классы ОБЯЗАТЕЛЬНО должны иметь различные реализации методов интерфейса. При этом, дочерние классы не должны иметь никаких ссылок друг на друга, также, как и интерфейс не должен ничего знать о классах, его реализующих.
4. Реализуйте валидацию свойств с помощью механизмов обработки исключений – если на вход свойству приходит некорректное значение, выходящее за допустимые пределы, свойство должно выбросить исключение соответствующего типа с описанием возникшей ошибки. Например, если в свойство «Возраст» пытаются присвоить отрицательное значение, необходимо выбросить экземпляр IncorrectArgumentException. Внимательно продумайте все возможные некорректные варианты данных, в том числе и ссылки на null. В случае, если механизмы валидации одинаковы у всех свойств, измените архитектуру бизнес-логики: вместо реализации интерфейса двумя классами, сделайте наследование двух классов от абстрактного класса с приобретением механизмов валидации.
5. Добавьте в решение еще один проект. Это можно сделать с помощью контекстного меню панели «Обозреватель решения». Для этого кликните правой кнопкой по названию решения в панели и выберите пункт «Добавить проект». Создайте исполняемый проект консоли (Win32 Console Application) и назовите его «ConsoleLoader». Данный проект является временным и необходим для начального тестирования бизнес-логики.
Примечание: в последующих лабораторных вы избавитесь от этого проекта и создадите вместо него проект графического интерфейса (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 простыми заданиями в консоли: создайте переменную-ссылку на интерфейс, поочередно присвойте в неё экземпляры реализующих классов и вызовите реализации методов – покажите, что это разные реализации. Вероятно, для демонстрации ваших классов в проекте ConsoleLoader придется добавить методы ввода/вывода классов бизнес-логики через консоль.

Варианты:

1. Геометрические фигуры с собственными реализациями расчета площади фигуры: круг, прямоугольник, разносторонний треугольник.
2. Система скидок с собственными реализациями расчета скидки: процентная скидка, сертификат, накопительная.
3. Работники фирмы: почасовая оплата труда, оплата по окладу и ставке.
4. Товары IT-магазина: процессоры, видео-адаптеры, звуковые карты (или любые другие категории товаров).
5. Транспортные средства: автомобили, мотоциклы, яхты.
6. Предложите свой вариант.

Лабораторная работа№2. Пользовательский интерфейс

Выполните следующие задания:

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 все дочерние классы 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. Отсеивание неправильных значений можно выполнить либо после ввода (вывести на экран сообщение пользователю, что поле введено неправильно), либо отсеивать значения на момент ввода – с помощью элемента MaskedTextBox. MaskedTextBox – аналог обычного TextBox, однако позволяет использовать специальные маски ввода, запрещающие вводить в поле цифры или, наоборот, символы. Реализация правильного отсеивания неправильных значений остается на ваше усмотрение, однако любые сообщения об ошибке пользователю должны четко пояснять, что именно не так сделал пользователь и что ему нужно сделать.

9. Особое внимание обратите на визуальную аккуратность создаваемых вами пользовательских интерфейсов. Старайтесь выравнивать элементы по левому краю относительно друг друга, делать одинаковые отступы 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 длинным, но и не слишком коротким. 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 – 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. Было бы неплохо иметь какой-то механизм, при котором в отладочной версии программы эта кнопка была, а в релизной (от англ. Release) версии этой кнопки не было. Для этого в Visual Studio существует механизм сборок с предустановленными сборками Debug и Release. Переключиться между ними можно на панели инструментов, находящейся вверху в центре экрана.

Для того, чтобы кнопка исчезала и появлялась в зависимости от сборки, необходимо в конструктор формы создания дописать примерно следующий код:

```
        #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.

Итоговый вариант кода может выглядеть примерно следующим образом:

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

Теперь попробуйте скомпилировать и запустить программу в режиме Release, а потом в режиме Debug. Вы убедитесь, что при сборке Debug-версии эта кнопка будет присутствовать, а в Release - нет.

Данный механизм разбиения сборок широко используется в разработке ПО. Так в Debug-версии, как правило, могут присутствовать дополнительные проверки корректности данных, или механизмы измерения скорости выполнения алгоритмов, т.е. различные инструменты, необходимые раз-

работчику, но ненужные конечному пользователю. Чтобы не отключать каждый раз все инструменты вручную, разработчики выделяют два варианта сборки - Debug и Release.
12. 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. Переименуйте элементы согласно правилам RSDN.

Критерии приёма лабораторной работы №2:

Критерий	Баллы
Правильность выполнения задания	1
Понимание работы кода	1
Реакция программы на ввод неправильных данных	1
Неизбыточность алгоритмов	1
Оформление кода	1
Своевременность	2
Теоретический вопрос	1
Эргономика пользовательского интерфейса	1
Правильность формулировок сообщений пользователю	1

Вопросы по выполнению задания и критериям оценки можно задать преподавателю во время занятий.

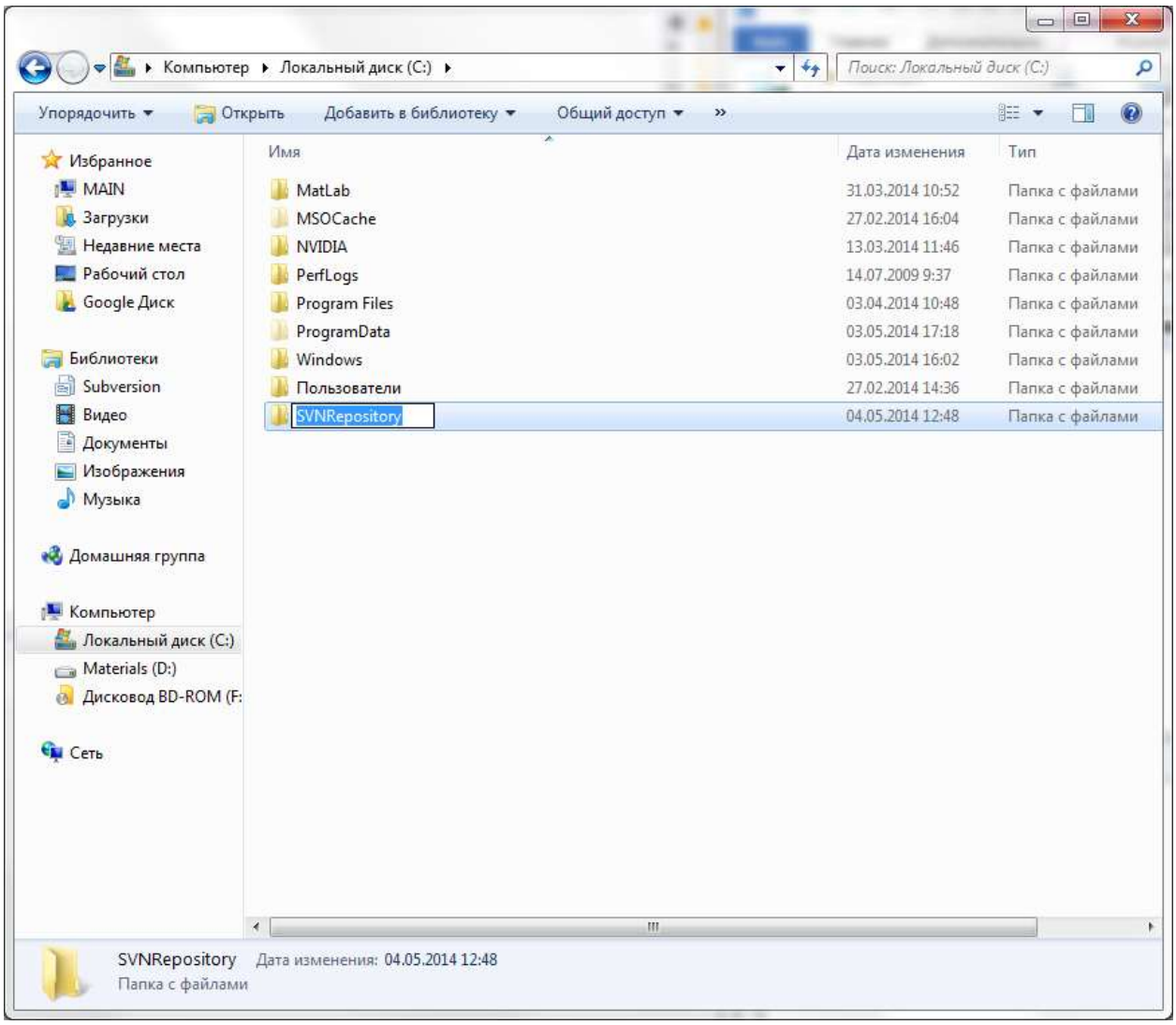
Лабораторная работа №3. Системы версионного контроля

Любой разрабатываемый продукт постоянно модифицируется – появляются новые функции, удаляются старые, при этом программный код переписывается раз за разом. Очень часто возникает необходимость сохранить проект перед началом серьезного рефакторинга проекта или же, наоборот, вернуться к более ранней версии программы, посмотреть, как там была реализована та или иная задача. Самое очевидное решение – копировать весь проект в специальный архив (бэкап – от англ. «backup») – далеко не самое лучшее. Подобный архив через год разработки будет занимать значительное пространство, при этом навигация среди версий будет достаточно сложной. Для решения обозначенной задачи существуют так называемые системы версионного контроля.
Системы версионнного контроля предназначены для решения следующих задач:

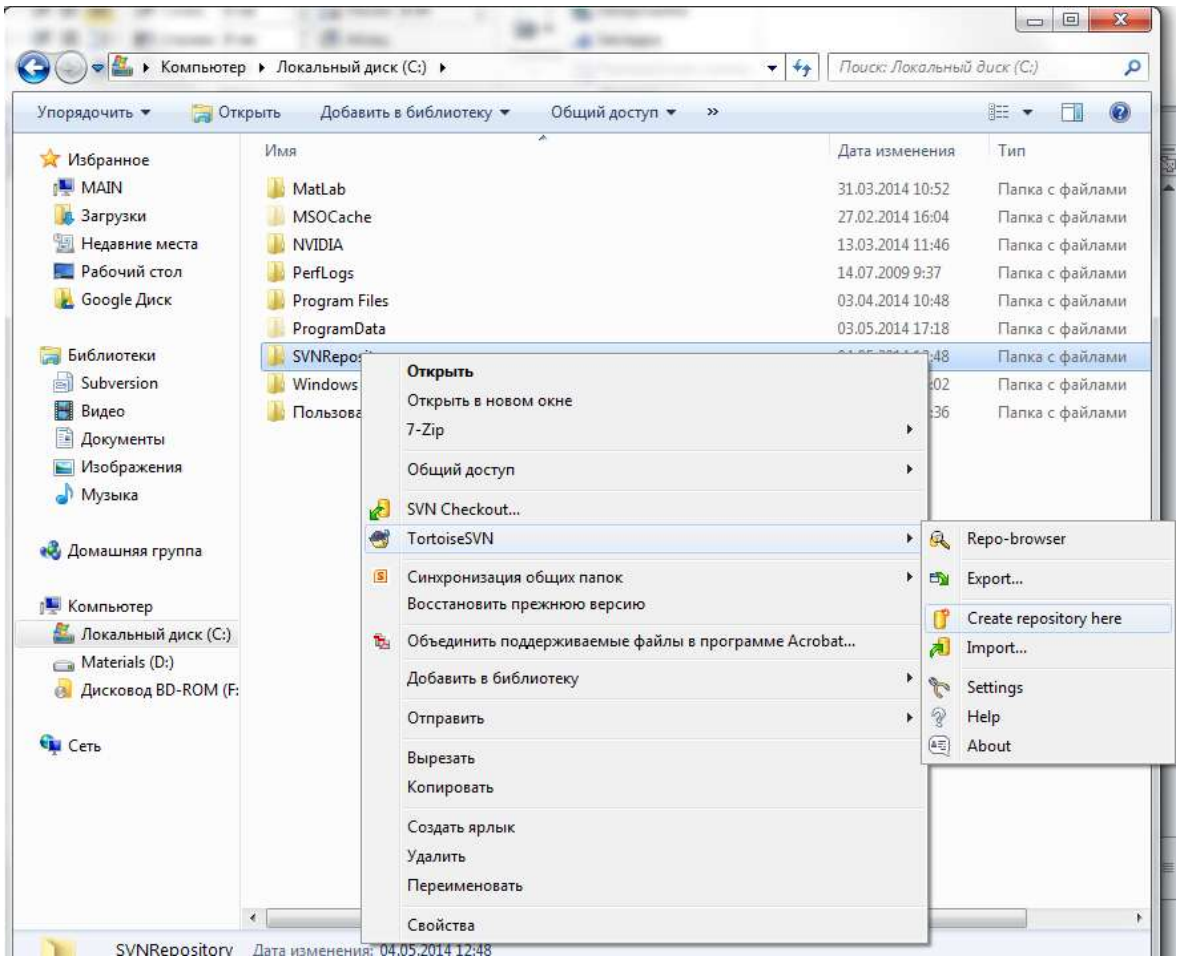
1) Хранение промежуточных версий исходного кода программы для возможности быстрого восстановления предыдущих версий проекта.
2) Синхронизация исходного кода в команде разработчиков.

В рамках данной лабораторной мы воспользуемся системой версионного контроля SVN, как наиболее простой в изучении.

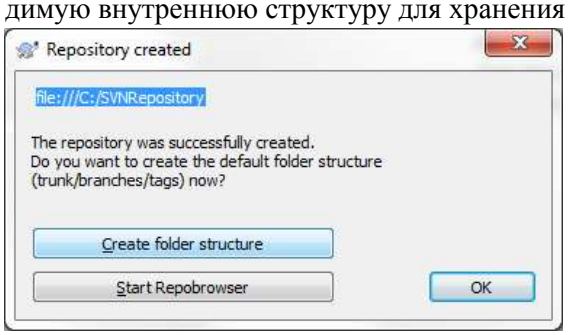
ПРИМЕЧАНИЕ: перед выполнением данной лабораторной работы сделайте копию вашего решения VisualStudio. Неправильное выполнение данной лабораторной работы может привести к потере исходного кода предыдущих лабораторных работ.
1. Скачайте и установите программу версионного контроля TortoiseSVN. Обратите внимание на разрядность вашей операционной системы.
2. Создайте пустую папку, в которой будет хранится ваш репозиторий – зашифрованное хранилище всех версий вашей программы. В дальнейшем вы не будете работать непосредственно с данной папкой, так как она играет роль сервера, с которым вы будете синхронизировать свою программу.



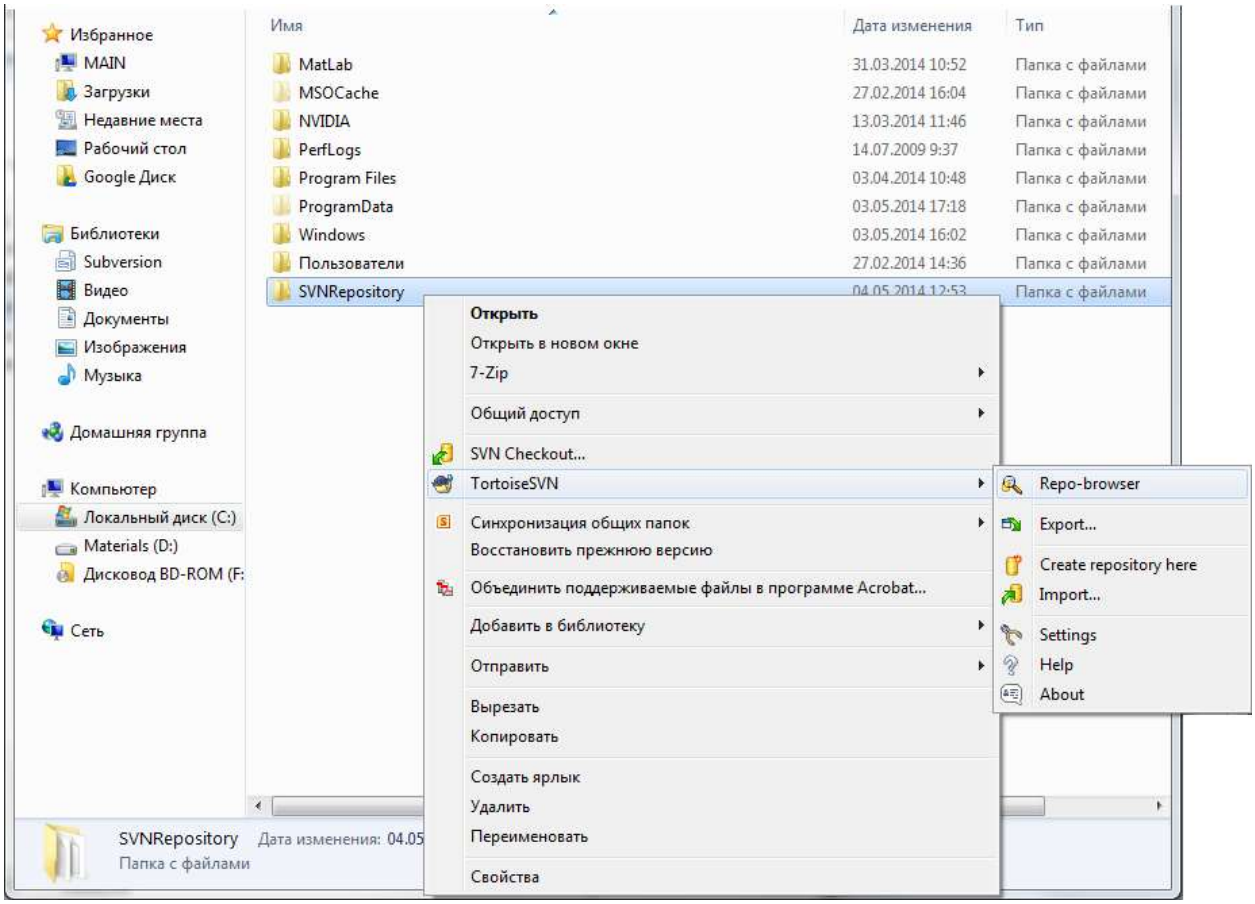
3. Через контекстное меню создайте в этой папке репозиторий:



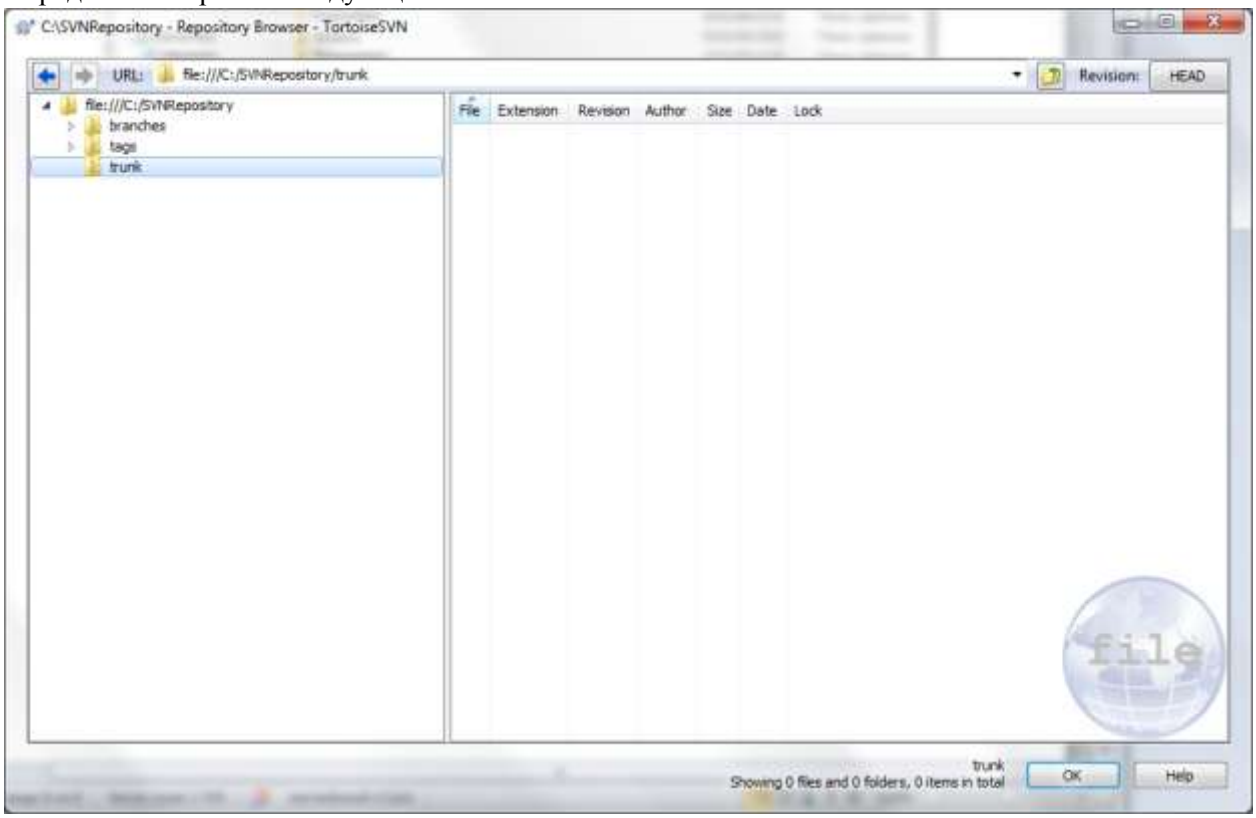
В появившемся окне нажмите кнопку Create folder structure – она автоматически создаст необходимую внутреннюю структуру для хранения версий программы.



4. Откройте репозиторий через браузер репозитория Repo-browser:



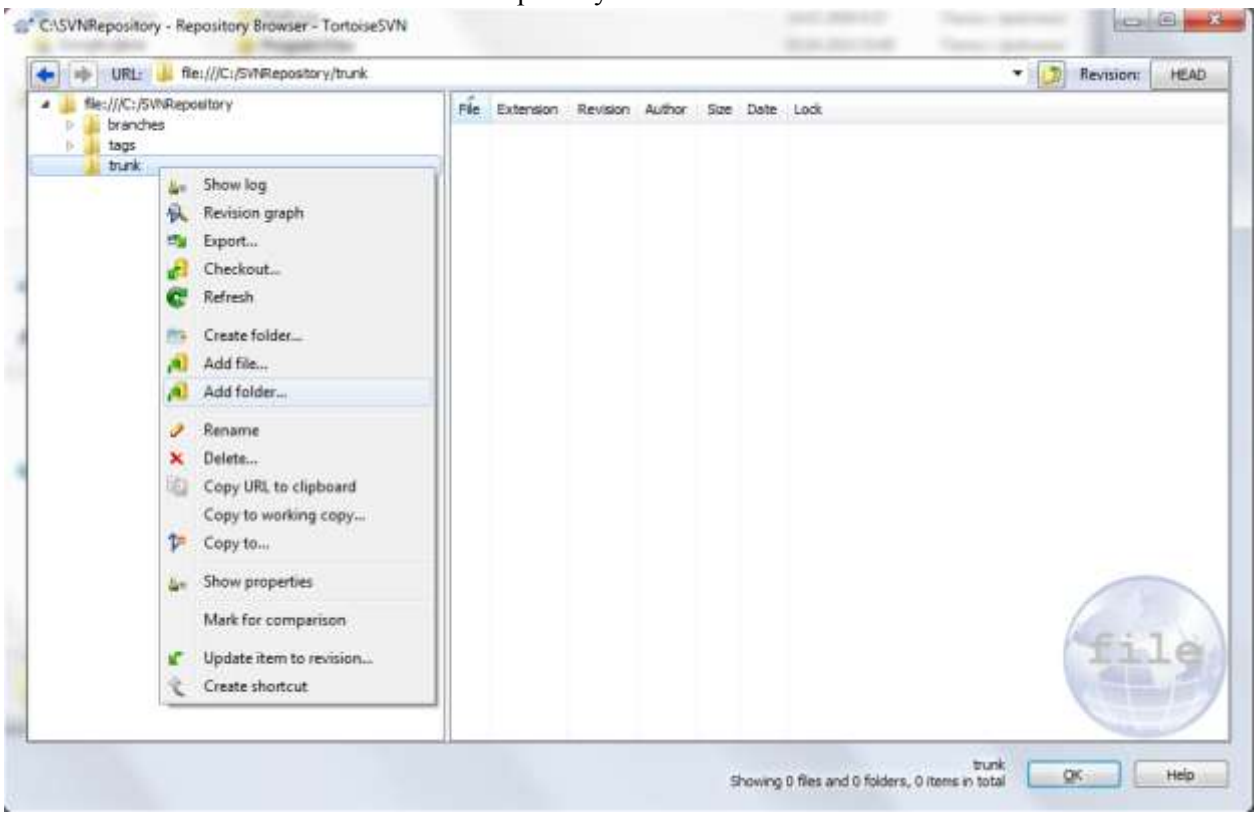
Перед вами откроется следующее окно:



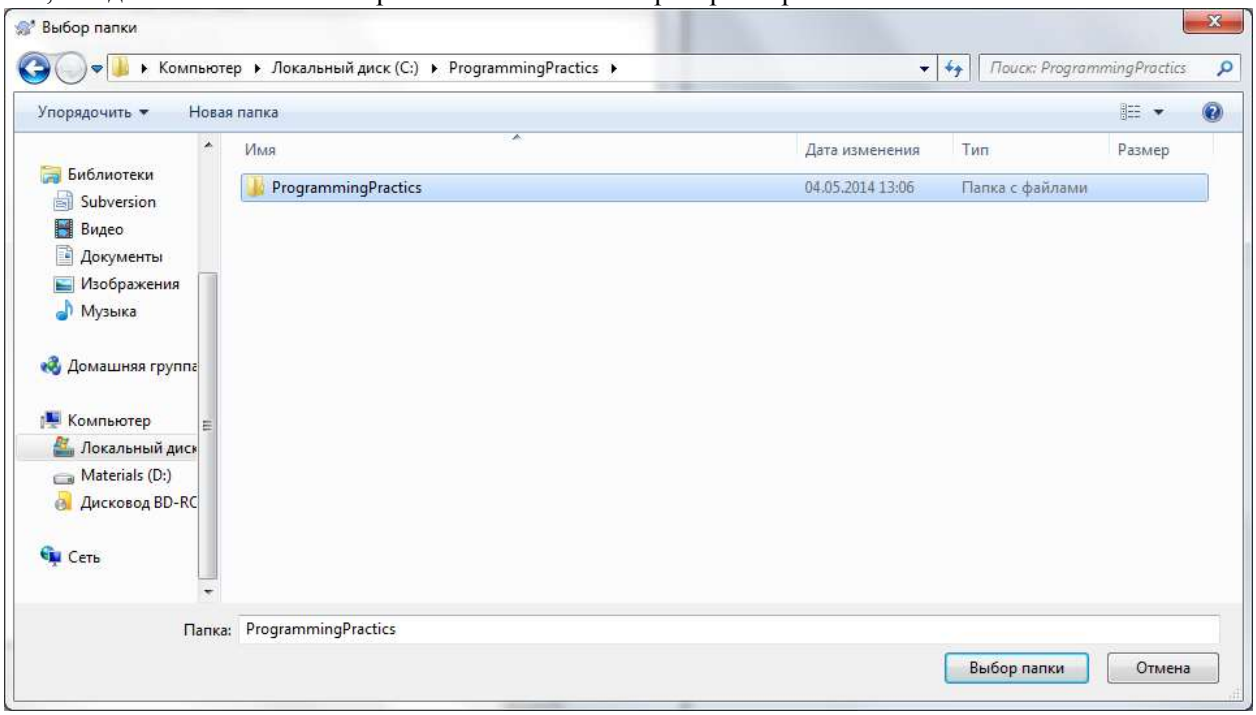
Данное окно отражает внутреннюю организацию репозитория. Здесь же созданы 3 папки – trunk, branches, tags. Как правило, основной папкой работы является trunk. Изучение назначения остальных папок остается на самостоятельную работу. При этом, данное деление на папки весьма условно – вы можете создать собственную организацию репозитория согласно вашим требованиям.

В настоящий момент папка trunk пуста, в неё необходимо добавить тот проект, который необходимо подчинить версионному контролю.

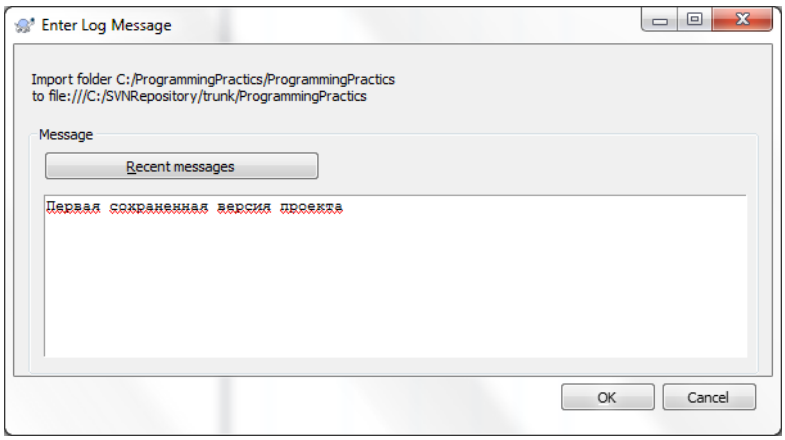
5. В контекстном меню папки trunk выберете пункт Add Folder



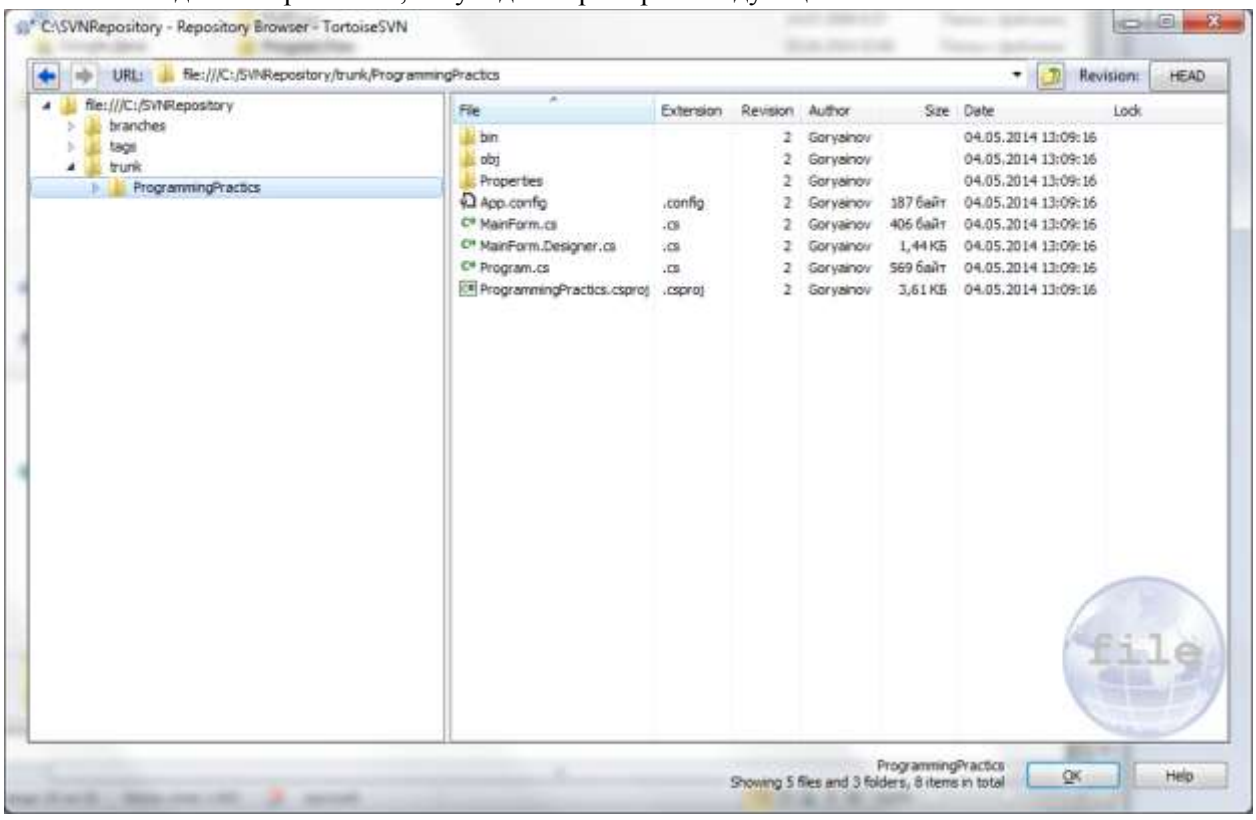
И выберете тот проект, который вы хотите сохранить под версионным контролем. В данном случае, это должна быть папка с решением ваших лабораторных работ:



Фактически, это действие создает первую ревизию вашего проекта. Ревизия – определенная созраненная версия проекта с уникальным номером. Для ревизии вы можете оставить собственный комментарий:

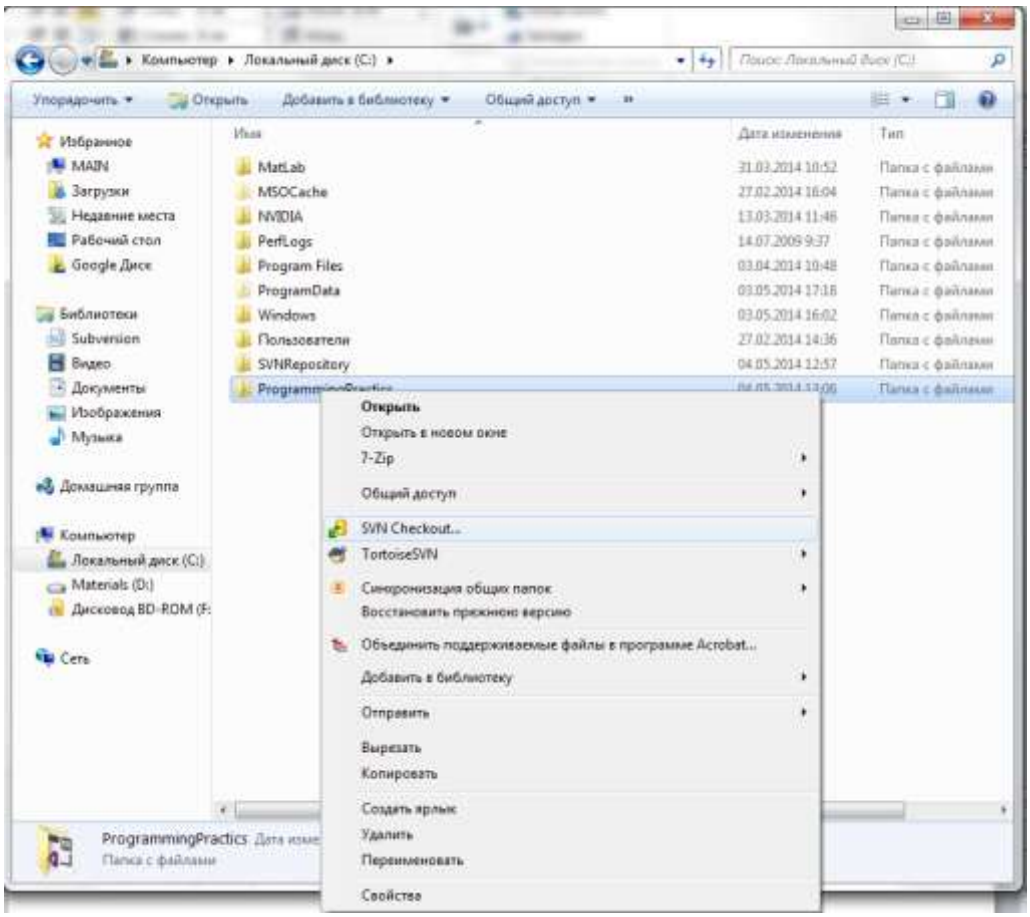


Если вы всё сделали правильно, вы увидите примерно следующее окно:

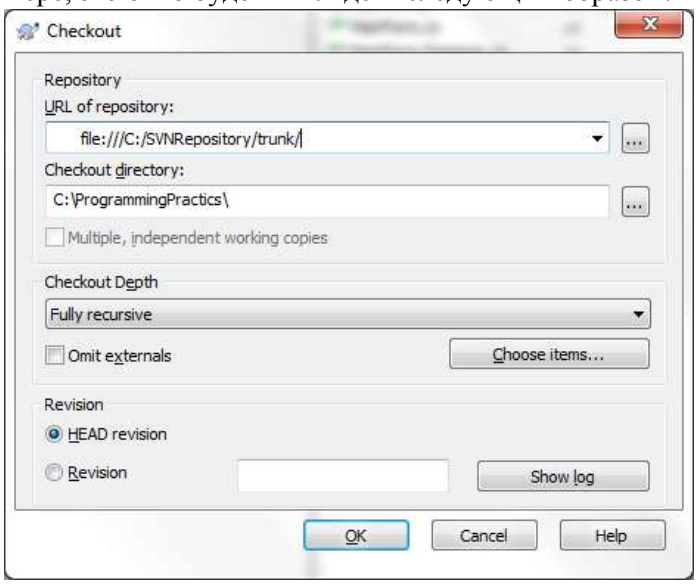


Теперь ваш проект находится под версионным контролем. Однако этот проект хранится в зашифрованном виде. Для того, чтобы получить доступ к репозиторию необходимо создать так называемую рабочую копию – папку, в которой вы будете работать с данным проектом, и которую вы будете синхронизировать с репозиторием.

7. Для этого выберете папку с вашим проектом (либо создайте новую) и в контекстном меню нажмите SVN Checkout:

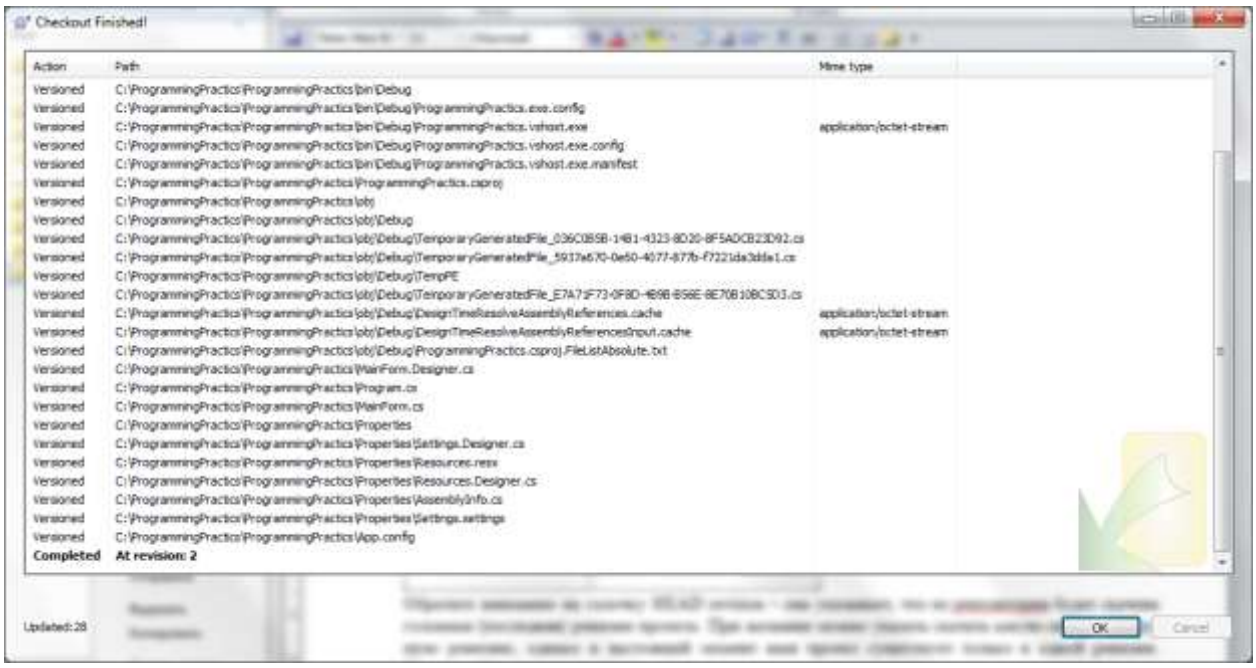


В появившемся окне необходимо указать путь репозитория и папку рабочей копии. В нашем примере, это окно будет выглядеть следующим образом.

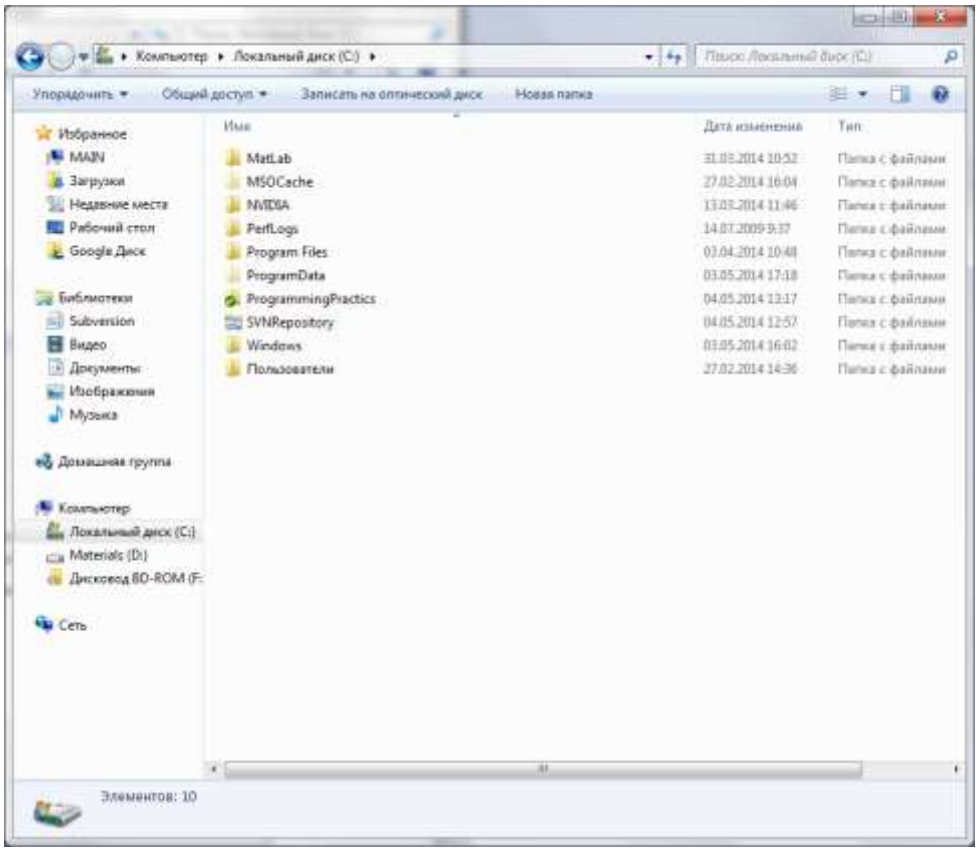


Обратите внимание на галочку HEAD revision – она указывает, что из репозитория будет скачена головная (последняя) ревизия проекта. При желании можно указать скачать какую-либо предыдущую ревизию, однако в настоящий момент наш проект существует только в одной ревизии. Нажмите Ok.

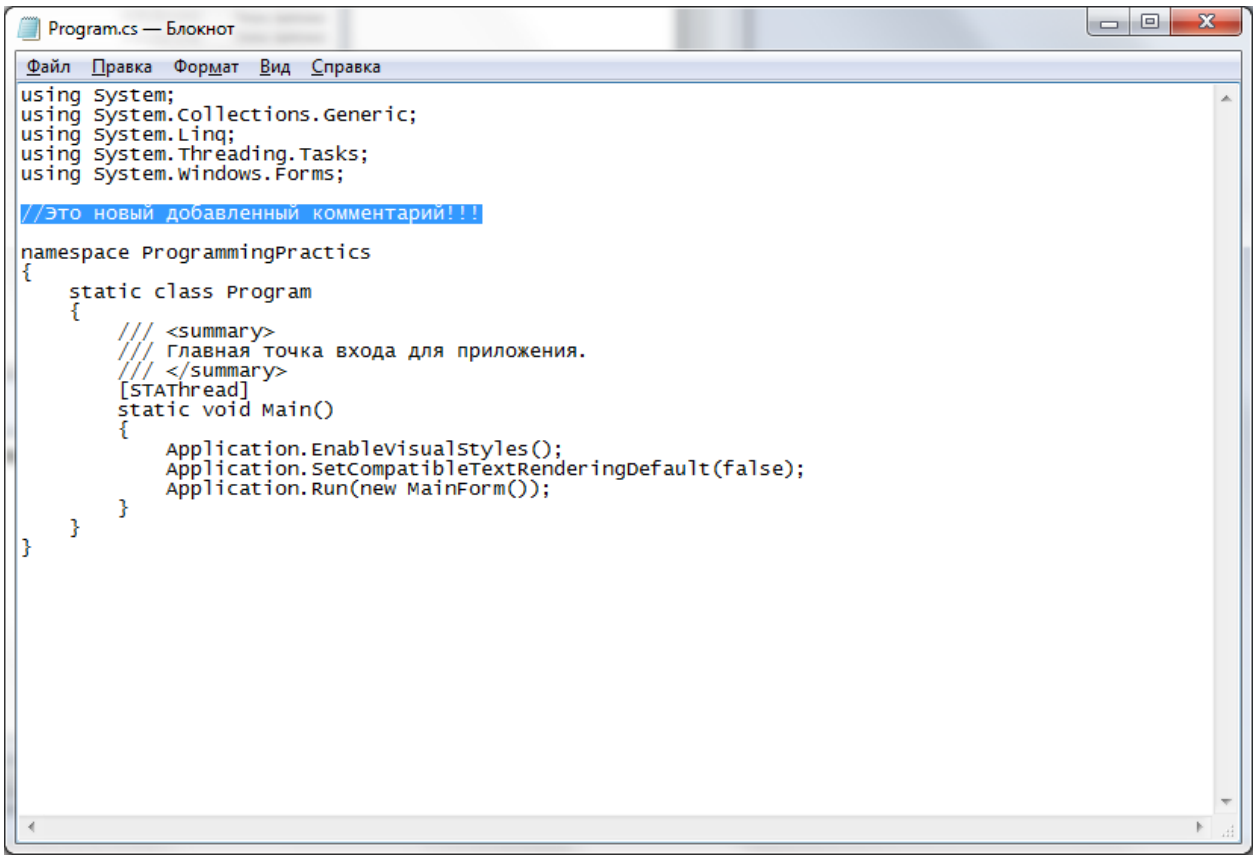
В следующем окне появится отчет, о том что все файлы в папке теперь синхронизированы с репозиторием.



А сама папка с проектом (также, как и все файлы внутри неё) теперь будет отмечена зеленой галочкой:

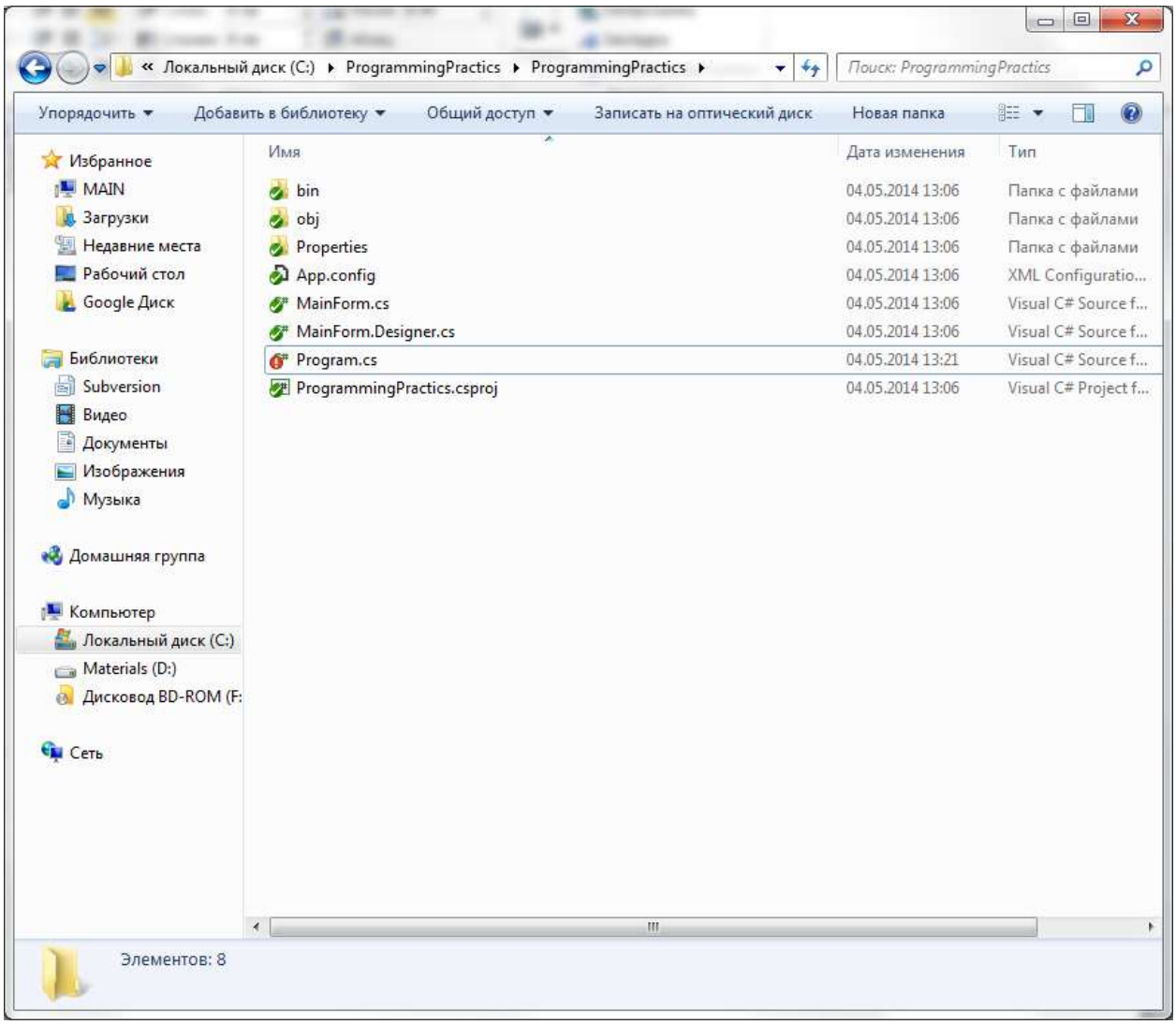


Добавим в какой-нибудь файл проекта небольшие изменения. Например, добавим комментарий:



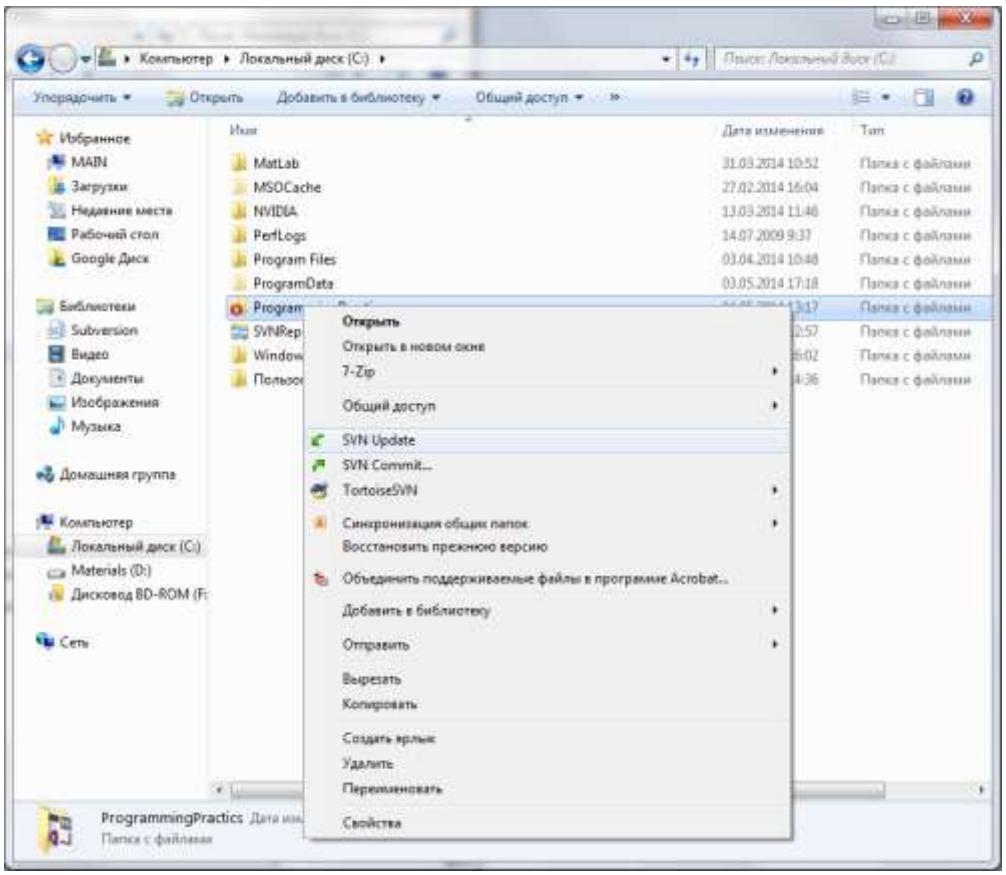
И сохраним его.

Теперь этот файл в папке будет отмечен красным кружочком с восклицательным знаком. Это означает, что ваша версия файла (или папки) отличается от версии, которая хранится в репозитории.

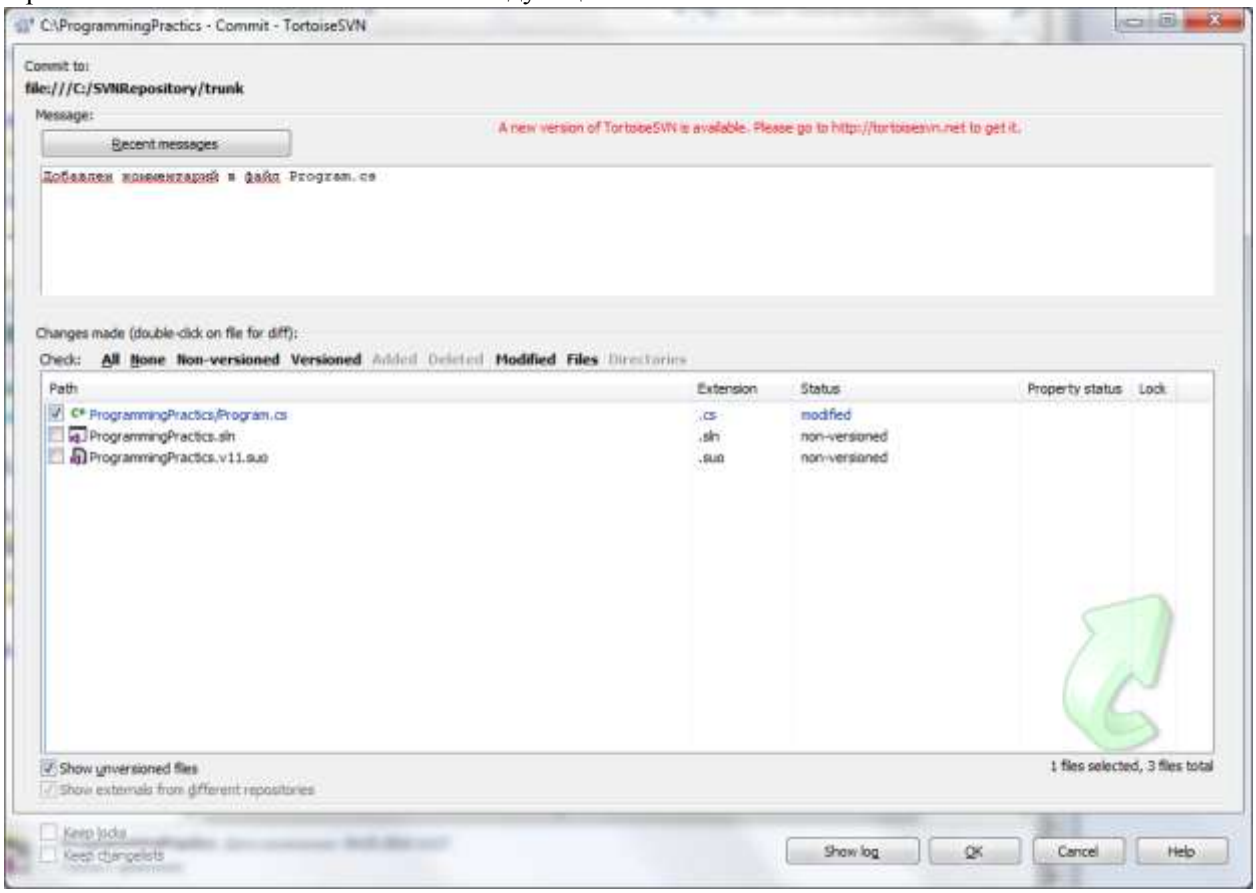


Мы добавили изменения в проект, но пока что это изменения только в нашей локальной рабочей копии. Теперь нужно «залить» изменения проекта на сервер.

Для этого выберете папку проекта и последовательно нажмите SVN Update и SVN Commit. Первое обновит вашу рабочую копию из репозитория, второе зальет изменения вашей рабочей копии в репозиторий, тем самым создав новую ревизию. Перед тем, как залить какие-либо изменения на сервер, обязательно нужно обновить рабочую копию. Это особенно важно при командной разработке.

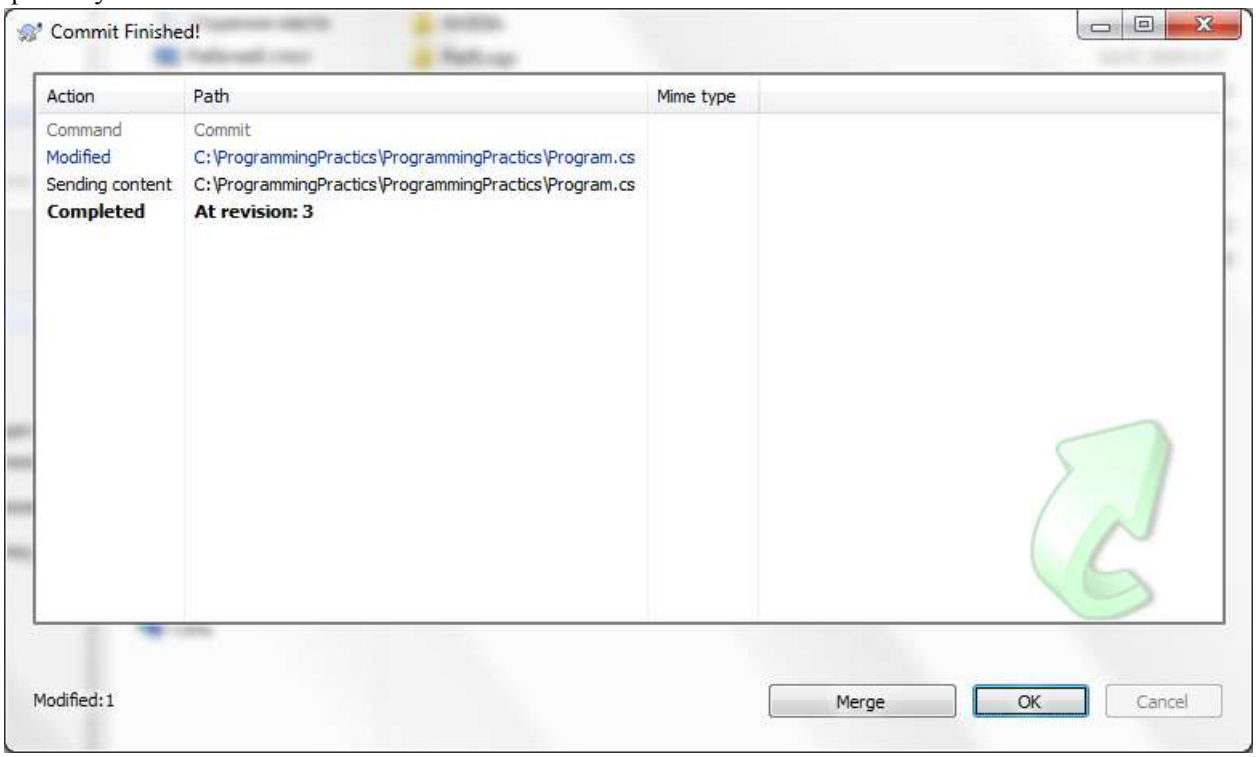


При нажатии SVN Commit появится следующее окно:

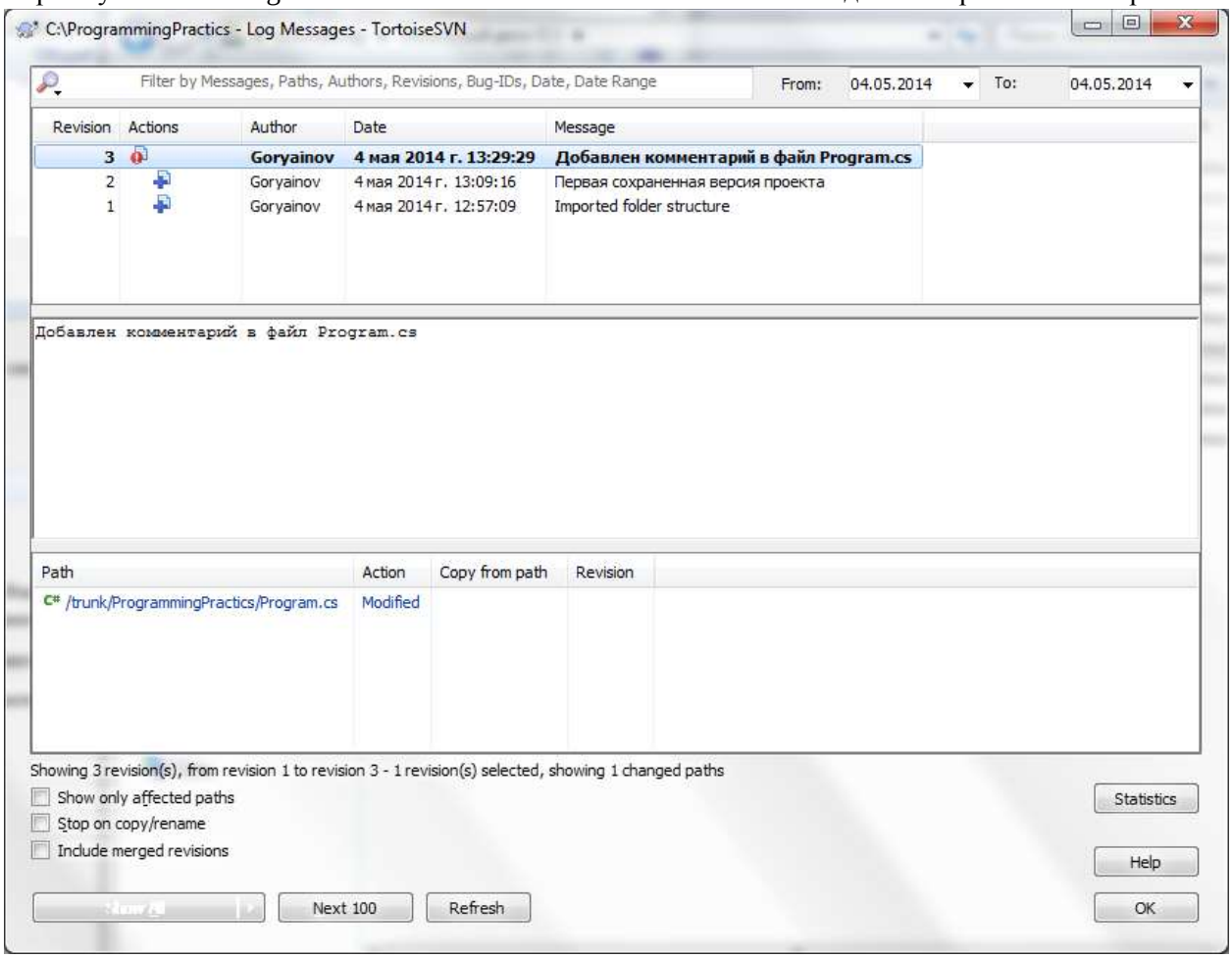


Здесь отображается список всех файлов, которые вы собираетесь залить в репозиторий, также вы можете оставить комментарий к вашей новой ревизии. Комментарий нужно оставлять обязательно, так как это значительно облегчает работу с репозиторием и возвращение к предыдущим версиям программы.

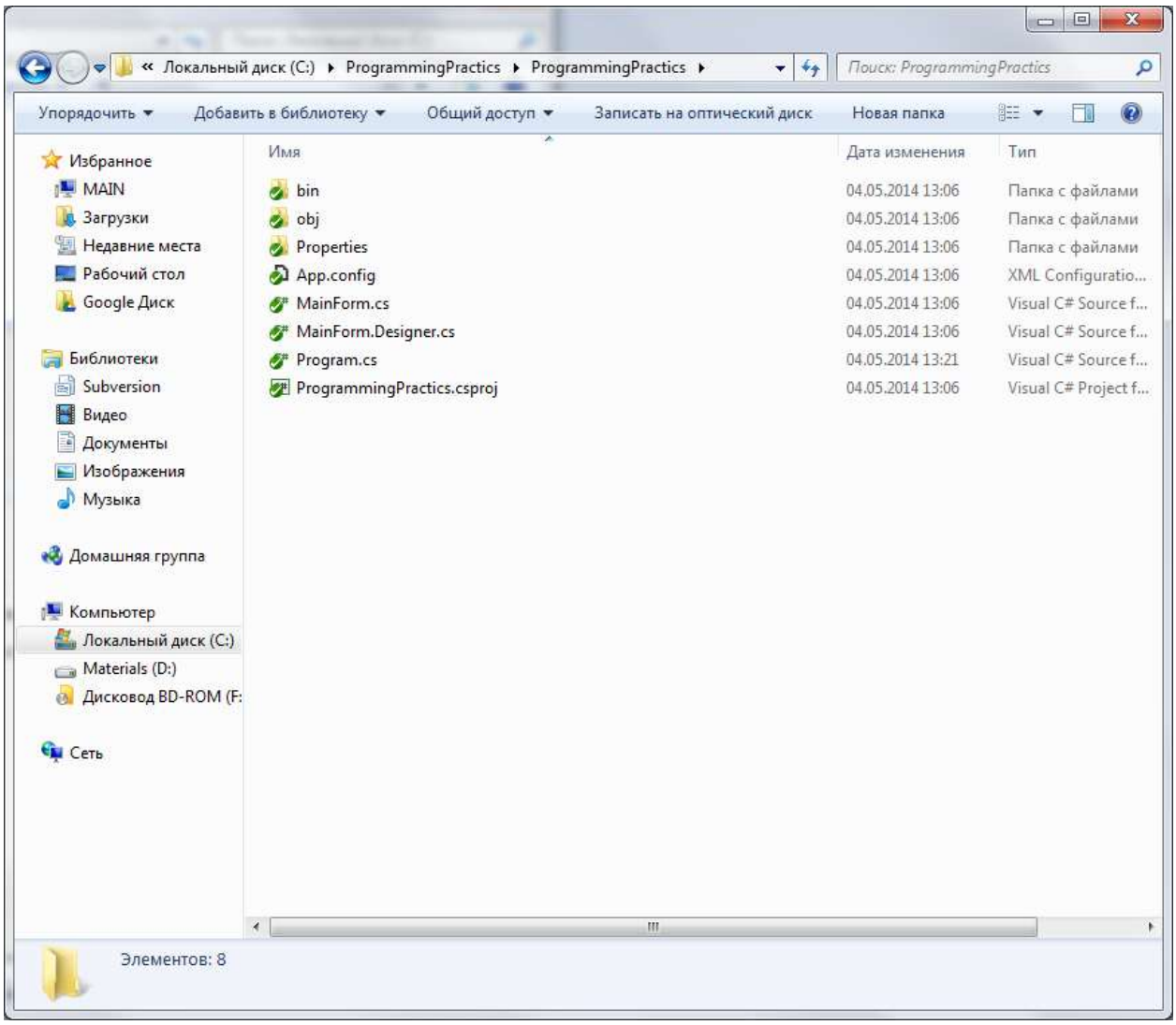
Следующее окно отобразит процесс синхронизации с репозиторием и подтверждение, что все файлы успешно залиты:



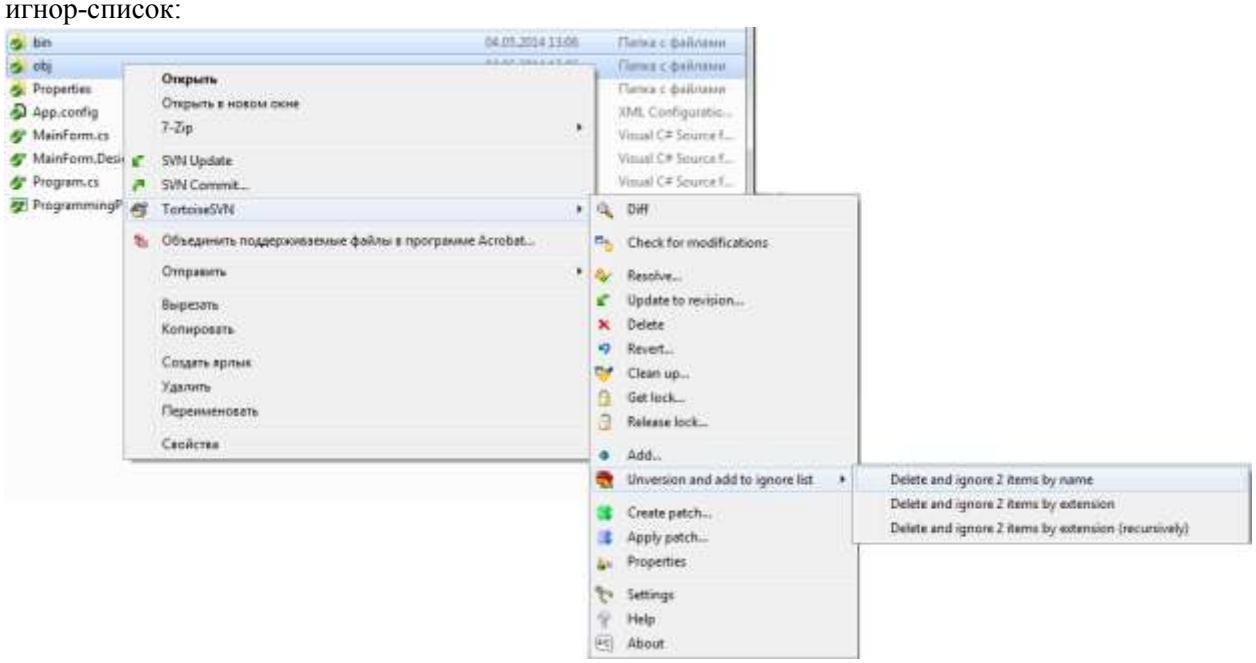
Через пункт Show Log в контекстном меню папки вы можете наблюдать историю ваших версий:



Зайдите в папку проекта. Вы увидите, что папки bin и obj также находятся под версионным контролем.



Однако данные папки представляют лишь промежуточные файлы компиляции и хранить их в репозитории нет никакого смысла. Исключите данные папки из версионного контроля добавив их игнор-список:



После чего снова выполните коммит через операцию SVN Commit. Данные файлы будут удалены из версионного контроля в новой ревизии.

В дальнейшем, если вам понадобится добавить под версионный контроль новые классы или удалить ненужные файлы, используйте команды Add и Delete в контекстном меню SVN. Учтите, что файлы, копирование файла в папку проекта еще не означает его добавление в версионный контроль – каждый файл необходимо добавлять вручную. Также, как удаление файла обычным способом не гарантирует его удаление из версионного контроля, и файл может восстановится при следующем обновлении рабочей копии. Это особенно проблематично, когда каждый новый созданный класс в решении необходимо также дополнительно добавить в версионный контроль через файловую систему.

Для устранения подобных проблем установите программу VisualSVN. Это плагин к VisualStudio, автоматически добавляющий и удаляющий файлы проекта в версионный контроль при их добавлении/удалении в проекте VisualStudio. Также он делает доступным обновление рабочей копии и синхронизацию с репозиторием непосредственно из-под VisualStudio.

Лабораторная работа №4. Блочное тестирование

Блочное тестирование (юнит-тестирование, “unit-testing”) – тестирование отдельного элемента изолированно от остальной системы. Относительно парадигмы объектно-ориентированного программирования системой является вся программа, а отдельным элементом – класс или его метод. Блочное 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, блочное 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 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.

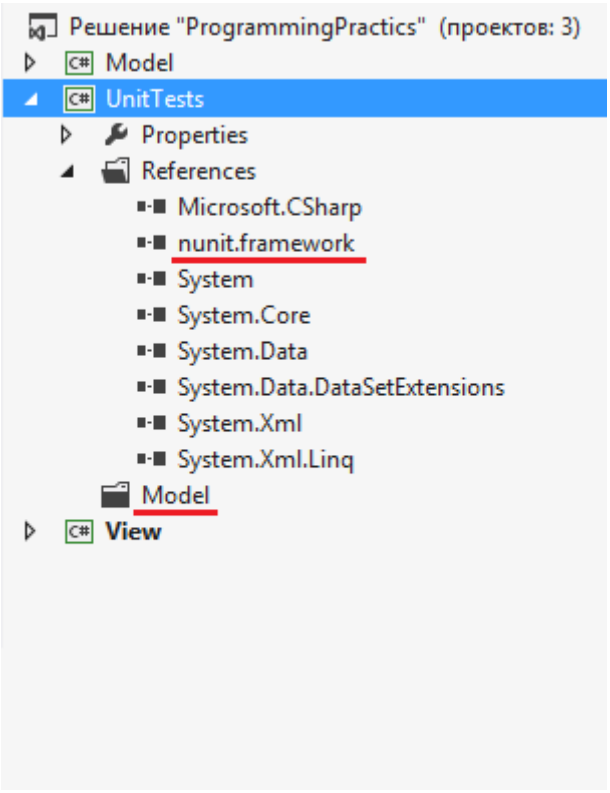
По данному алгоритму проверяется каждый общедоступный метод тестируемого класса. Защищенные или закрытые члены класса тестированию не подвергаются, чтобы не нарушать инкапсуляцию класса. Каждый открытый метод

Таким образом, блочные тесты представляют программный код, который также требует постоянной поддержки и следованию определенным правилам оформления. Требование поддержки при

В отличие от других видов тестирования, проведение блочного тестирования – обязанность разработчиков, а не тестировщиков.

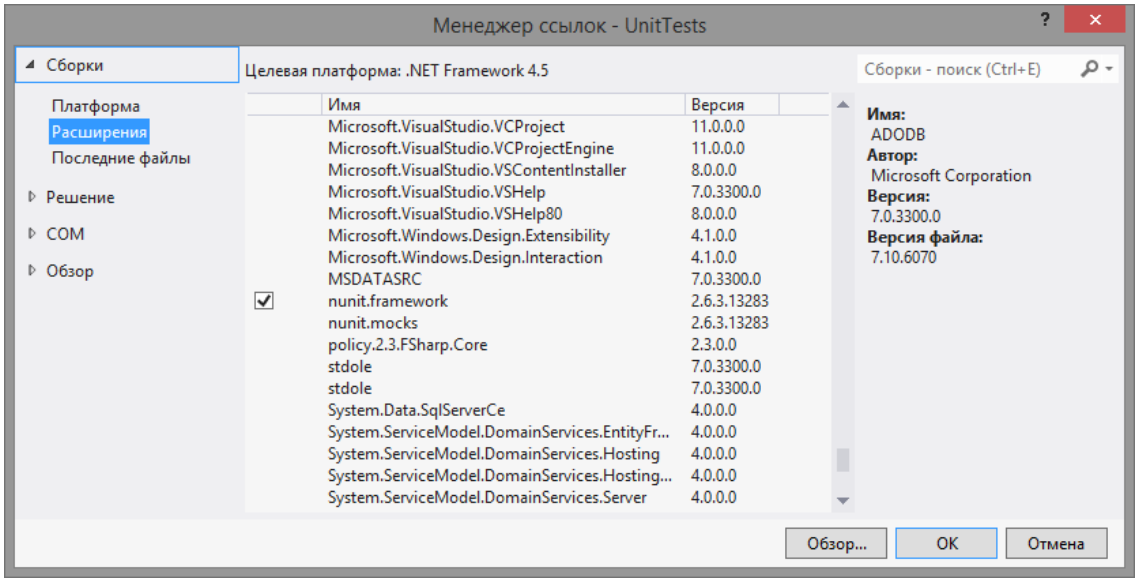
Задание:

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:



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»:



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>
        /// Количество некоторых элементов внутри экземпляра MyClass.
        /// </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 блочным тестированием.

6. В первую очередь, обозначим наш класс блочных 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 UnitTests.Model
{
    /// <summary>
    /// Набор тестов для класса MyClass.
    /// </summary>
    [TestFixture]
    public class MyClassTest
    {
        /// <summary>
        /// Тестирование свойства Count.
        /// </summary>
        /// <param name="count">Значение свойства Count.</param>
        [Test]
        public void 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.

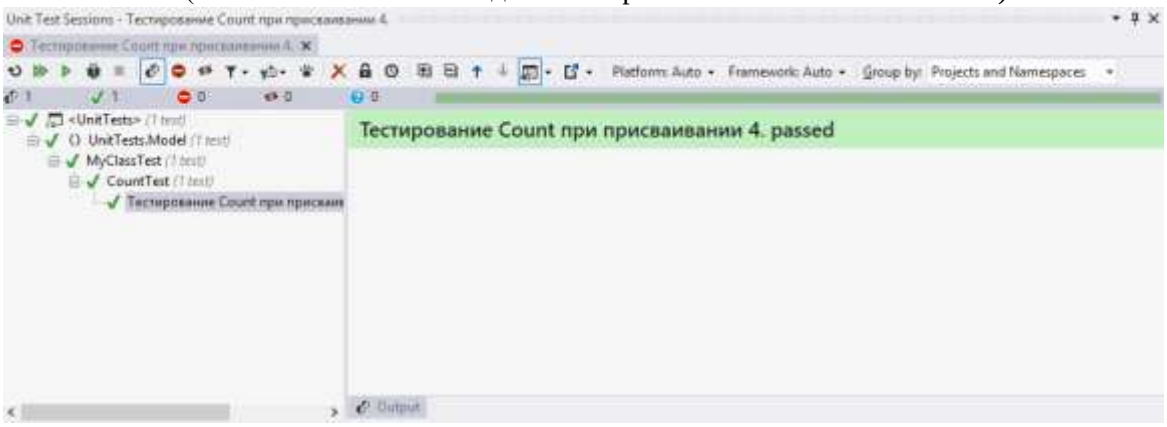
8. Добавим некоторый тестовый случай, проверяющий работу свойства Count при обычном значении, например, 4. Для этого добавим после атрибута [Test] еще один атрибут [TestCase] с описанием входных данных и названием тестового случая:

```
[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:

```
namespace UnitTests.Model
{
    /// <summary>
    /// Набор тестов для класса MyClass.
    /// </summary>
    [TestFixture]
    public class MyClassTest
    {
        /// <summary>
        /// Тестирование свойства Count.
        /// </summary>
        /// <param name="count">Значение свойства Count.</param>
        [Test]
        [TestCase(4, TestName = "Тестирование Count при присваивании 4.")]
        public void CountTest(int count)
        {
            var myClass = new MyClass();
            myClass.Count = count;
        }
    }
}
```

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):



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.

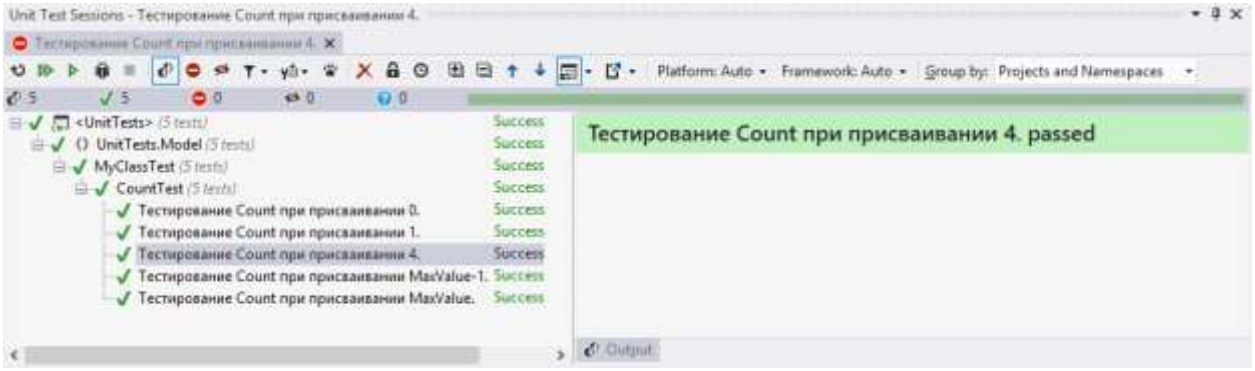
При запуске тестов выполняется компилирование тестируемого проекта и тестирующего проекта. Тесты не будут запущены, если при компилировании какого-либо из проектов возникла ошибка. Для запуска тестов также необязательно делать проект UnitTests запускаемым проектом.

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 присваивании MaxValue.")]
[TestCase(int.MaxValue-1, TestName = "Тестирование Count при присваивании MaxValue-1.")]
```

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



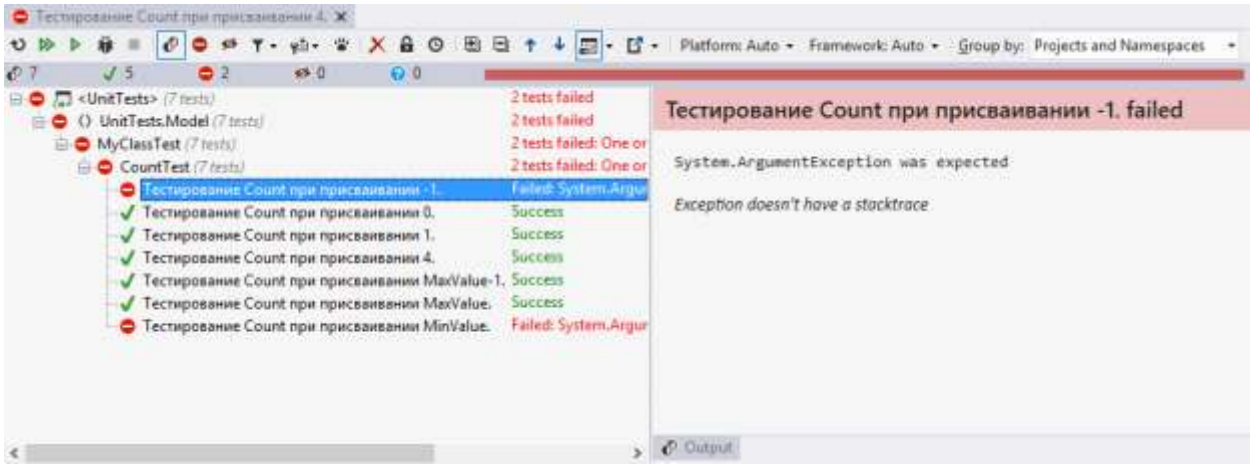
11. Мы описали *позитивные* сценарии использования свойства Count, т.е. мы присваивали такие значения, которые являются корректными для свойства Count. Однако куда более важно протестировать *негативные* сценарии – подача заведомо некорректных данных с ожиданием возникновения ошибки или исключения. Если 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. По этому:

- Программа, класс или метод должны реагировать на подачу некорректных данных исключением или любым другим способом сообщения об ошибке.
- Блочное 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:

```
/// <summary>
/// Тестирование свойства Count.
/// </summary>
/// <param name="count">Значение свойства Count.</param>
[Test]
[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 присваивании MaxValue.")]
[TestCase(int.MaxValue-1, TestName = "Тестирование Count при присваивании MaxValue-1.")]
[TestCase(-1, ExpectedException = typeof(ArgumentException), TestName = "Тестирование Count при присваивании -1.")]
[TestCase(int.MinValue, ExpectedException = typeof(ArgumentException), TestName = "Тестирование Count при присваивании -1.")]
public void CountTest(int count)
{
    var myClass = new MyClass();
    myClass.Count = count;
}
```

После запуска негативных тестов, они получили статус Failed:



Данный статус означает, что при подаче некорректных данных исключение в свойстве Count не возникло, что показывает неправильную реализацию свойство Count. Следовательно, в сеттер свойства Count необходимо добавить проверку на неотрицательность 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 передачи 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!

Теоретические вопросы:

1. Что такое тест? Что такое тестовый случай?
2. Какие виды тестирования существуют?
3. Что такое блочное тестирование?
4. Что такое библиотека Nunit?
5. Как организуется внутренняя структура блочного тестирования внутри решения?
6. Как написать блочный тест на основе Nunit?
7. Что такое атрибут TestCase?
8. Какие параметры существуют у атрибута TestCase? (в лабораторной работе рассмотрены только три параметра, однако при ответе на вопрос следует ознакомиться с документацией)
9. Поясните назначение следующих атрибутов в библиотеке Nunit:
    - Combinatorial
    - Ignore
    - Maxtime
    - Pairwise
    - Random
    - Repeat
    - Setup
    - SetupFixture
    - Teardown
10. Для чего нужен класс Assert? Как его использовать при тестировании?

Критерии оценки работы:

1. Правильность выполнения работы
2. Покрытие исходного кода тестами

3. Правильность определения тестовых случаев
4. Правильность описания тестовых случаев
5. Правильность оформления кода
6. Своевременность (2 балла)
7. Теоретический вопрос (2 балла)
8. Понимание работы кода

Лабораторная работа №5. Сериализация

Одной из наиважнейших функциональностей современного программного обеспечения является возможность сохранения и загрузки пользовательских данных в процессе работы. Это очень удобно, особенно для тех приложений, работу в которых нельзя выполнить за один сеанс и приходится делить её на несколько сеансов. Примером могут послужить Microsoft Office, где в любой момент вы можете сохранить документ, закрыть программу и вернуться к нему через несколько дней. Или Visual Studio – невозможность сохранить исходный код программы значительно усложнила бы разработку хоть сколько-нибудь полезных программ.

В первую очередь необходимо определить все данные, которые необходимо сохранять пользователю. При этом стоит различать те данные, над которыми работает пользователь с помощью программы, и те данные, которые являются настройками самой программы. Несмотря на то, что обе эти модели данных в той или иной мере присутствуют в любой программе, фактически, это две совершенно различные сущности, которые должны храниться раздельно.

Следующим шагом является определение формата файла – спецификации его внутреннего представления. Одним из основных подводных камней, подстерегающих разработчиков на этом этапе, является возможность масштабируемости формата. Суть этой проблемы в том, что при последующем развитии и модификации вашей программы может изменяться модель данных. Следовательно, изменения будет претерпевать и внутреннее представление файла, что может создать проблему совместимости данных одной версии программы с другой. Подобная проблема может омрачить переход ваших пользователей на более новые версии программы – зачем переходить на новую версию, если придется переделывать всю работу, сделанную в предыдущей версии.

Как правило, разработчикам нет необходимости задумываться о поддержке старыми версиями программы более новых версий формата, однако новые версии программы должны поддерживать старые форматы файлов. Отсюда вытекает две рекомендации к разрабатываемым форматам:

1) Старайтесь предугадать дальнейшее развитие проекта и наиболее вероятные изменения в модели данных.
2) Делайте формат данных наиболее гибким к масштабируемости.
3) По возможности сохраняйте в начале файла версию формата.

Если понимание первых двух рекомендаций к разработчику приходит с личным опытом и сильно зависит от разрабатываемого продукта, то выполнение третьей рекомендации не составляет большого труда. В дальнейшем это может значительно облегчить идентификацию формата файла, так как в противном случае придется анализировать структуру всего файла, что является весьма нетривиальной задачей.

Платформа .NET значительно облегчила задачу сохранения структур данных в файл и чтения данных из файла, создав в себе стандартный и гибкий механизм сериализации. Сериализация - процесс перевода какой-либо структуры данных в последовательность битов. Обратной операцией является операция десериализации – восстановление начального состояния структуры данных из битовой последовательности. В общем случае, перевод может осуществляться в любое удобное для хранения представление данных, на практике же чаще всего выполняют хранение в файлах разметки xml.

Сериализация средствами .NET осуществляется с помощью классов ***Serializer*** (***BinarySerializer***, ***SoapSerializer***, ***XmlSerializer***). В первую очередь необходимо отметить необходимый к сериализации класс как сериализуемый. Для этого у класса необходимо поставить атрибут [Serializable]:

```
[Serializable]   // атрибут, указывающий, что этот класс может быть сериализован
public class MySerializableClass
{
    public int Value1;
    public double Value2;
    public bool Value3;

    [NonSerialized] // атрибут у поля, которое не должно сериализоваться
    public string Value4;
}
```

Атрибуты – элементы языка программирования, выполняющие связывание метаданных или декларативной информации с кодом (сборками, типами данных, методами, полями и т.д.). В языке *C#* атрибуты указываются перед необходимым элементом кода в квадратных скобках. В нашем примере атрибут [***Serializable***] указывает, что данный класс может быть сериализован.
Возможны случаи, когда сериализация отдельных полей класса не требуется, например, если эти объекты константные или автоинициализируемые. Чтобы исключить их из процедуры сериализации, необходимо отметить эти поля атрибутом [***NonSerialized***] (см. пример выше). Так как сериализация представляет сохранение данных, в сериализации участвуют только поля класса, методы (процедуры по обработке данных) сериализации не подлежат.
В следующем примере представлены два метода, которые выполняют чтение и запись созданного нами класса с помощью экземпляра ***SoapFormatter***:

```
public static class MySerializationManager
    {
        public const string OutputFilename = "data.xml";

        public static MySerializableClass LoadConfigData()
        {
            Stream stream = File.Open(OutputFilename, FileMode.Open);
            var formatter = new SoapFormatter();
            var data = (MySerializableClass) formatter.Deserialize(stream);
            stream.Close();
            return data;
        }

        public static void SaveConfigData(MySerializableClass data)
        {
            Stream stream = File.Open(OutputFilename, FileMode.Create);
            var formatter = new SoapFormatter();
            formatter.Serialize(stream, data);
            stream.Close();
        }
    }
```

Это лишь один из возможных примеров. В частности, в данном примере чтение и запись осуществляется всегда в один и тот же файл ***data.xml***, а методы чтения и записи инкапсулированы в одном статическом классе ***MySerializationManager***. Суть процедуры открытие файла с помощью стандартного статического класса ***File*** (класс для работы с файлами), помещение всего его содержимого в поток ***Stream***, а затем перевод потока данных в нашу структуру с помощью специального класса ***SoapFormatter***.
Помимо ***SoapFormatter*** есть и другие способы сериализации, например с помощью классов ***XmlSerializer*** и ***BinarySerializer***, не требующих работы с атрибутами. С преимуществами и недостатками тех или сериализаторов, также как и с примерами работы с ними, вы можете ознакомиться на сайте ***msdn***. Если ни один из стандартных механизмов сериализации вас не устраивает, вы можете создать свой собственный инструмент чтения/записи вашего формата файлов. Для этого вам необходимо использовать классы ***File*** и ***Path***.
При сериализации необходимо помнить несколько моментов:

1) Если объект агрегирует в себе экземпляр другого класса, другой класс также должен быть сериализуемым для возможности сериализации. Данное требование распространяется на все агрегируемые объекты любой вложенности.
2) Если вам необходимо сериализовать несколько объектов в один файл, данные объекты должны быть агрегированы в некоторый класс-контейнер, который и будет сериализоваться.
3) Некоторые стандартные типы данных не имеют сериализацию по умолчанию. Например, структура данных словарь (***Dictionary***). Сериализация объектов, содержащих словари, требует отдельной реализации, данная проблема имеет множество решений, некоторые из которых представлены на ***msdn.com***.

1. Добавьте в программу возможность загрузки/сохранения проекта. Добавьте в верхнюю часть главной формы меню программы (элемент *MenuStrip* в панели инструментов *WinForms*). Добавьте в неё такие подменю как *File* и *Help*. В меню File добавьте пункты:
- *New* – создать новый проект. Если ранее были выполнены какие-либо изменения в предыдущем проекте, программа должна предложить сохранить текущий перед созданием нового;
- *Save* – сохранить текущий проект по текущему имени и пути. Если проект не имеет имени и пути, предложить их задать;
- *Save As* – сохранить текущий проект по новому имени и пути;
- *Open* – открыть проект. Если перед открытием проект не был сохранен, предложить сохранить проект;
- *Close* – закрыть программу.

Для задания имени и пути файла при сохранении и загрузке можно воспользоваться стандартными диалогами *SaveFileDialog* и *OpenFileDialog* соответственно.
Меню *Help* должно содержать пункты *User`s Guide* и *About*, работа которых остается на усмотрение студента.
Другой вариант оформления меню – сделать его в виде панели (*Toolbar*), где каждое перечисленное действие будет представлено собственной кнопкой-пиктограммой.
2. Сделайте отображение имени проекта в шапке перед названием программы. По умолчанию имя проекта ***New Project***, папка по умолчанию на усмотрение студента.
3. Сделайте отображение звездочки возле имени проекта в случае, если в нем есть несохраненные изменения**. Если проект был только что сохранен, звездочка не должна отображаться. Такая незначительная на первый взгляд функциональность достаточно удобна для конечного пользователя, хотя может стать нетривиальной задачей при разработке. Одним из возможных решений является создание глобального статического булева свойства ***IsProjectChanged, которое при изменении своего значения производит зажигание события ***ProjectChanged***. Событие ***ProjectChanged*** также нужно реализовать. Суть такова, что на данное событие подписывается главная форма, в обработчике которого происходит смена заголовка в зависимости от имени проекта и наличия изменений в проекте. Событие загорается только при смене значения свойства ***IsProjectChanged***. Так как данное свойство статическое, его может изменить любая часть программы, а истинность значения свойства определяет наличие звездочки в заголовке.
4. Сделайте доступность кнопки Save в зависимости от наличия изменений в проекте.** Если проект был только что сохранен, и в нём еще не было каких-либо изменений, кнопка ***Save должна быть недоступна. Данную функциональность достаточно просто сделать при наличии события *ProjectChanged*. Реализацию предлагается придумать самостоятельно.
5. Добавьте кнопку *Modify*. Программа должна предоставлять возможность не только создания новых объектов в вашей базе данных и удаления их из базы, но возможность редактирования уже существующих объектов. Необходимо добавить кнопку *Modify*, которая открывает окно, аналогичное окну *AddNewObject*, сделанному в лабораторной работе №2, однако при запуске оно должно инициализироваться значениями выделенного в таблице элемента. Пользовательский вариант использования:
- Пользователь выбирает один из объектов в базе данных на главной форме;
- Пользователь нажимает кнопку *Modify*;
- Появляется окно *ModifyObject*, инициализированное значениями выбранного элемента;
- Пользователь может либо изменить значения полей и применить их, нажав кнопку *Ok*, либо отменить изменения, нажав кнопку *Cancel*.

6. Проверить работу программы при различных повреждениях файлов проекта. В качестве повреждения файлов рассмотрите случаи:
- Удаление тега xml из структуры;
- Удаление закрывающего тега из структуры;
- Добавление несуществующего тега в файл;
- Изменение названия тега в файле;
- Перестановка нескольких тегов в файле местами;

- Изменение данных на некорректные – символы вместо чисел, отрицательные значения вместо положительных и т.п.

Во всех данных случаях программа не должна аварийно завершаться. Программа должна сообщить пользователю об ошибке чтения файла и продолжить работу.

Критерии приёма лабораторной работы №4:

Критерий	**Баллы**
Правильность выполнения задания	1
Понимание работы кода	1
Реакция программы на ввод неправильных данных	1
Неизбыточность алгоритмов	1
Оформление кода	1
Своевременность	2
Теоретический вопрос	1
Эргономика пользовательского интерфейса	1
Правильность формулировок сообщений пользователю	1

Вопросы по выполнению задания и критериям оценки можно задать преподавателю во время занятий.

Лабораторная работа №6. 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. 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 блочные тесты. Фактически рефакторинг состоит из следующих этапов:

1) Покрытие целевого исходного кода блочными 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» и Стивена МакКоннелла «Совершенный код».

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. По этому, 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 (метод *get*), то все поля ввода на пользовательском элементе управления должны инициализироваться соответствующими значениями из присвоенного объекта. Таким образом, 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 (метод *set*), 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 присвоении в свойство значения (метод get), все поля ввода на пользовательском элементе должны стать ReadOnly, т.е. только отображать данные, но не позволять их ввод. При получении значения свойства (метод *set*) 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.

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 будет 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 расширение сохраняемых файлов с xml в иной формат.** Программа должна открывать не все xml-файлы, а только xml-файлы собственной базы данных. Для этого придумайте своё собственное расширение для файлов программы.
2) **Изменить установочный скрипт InnoSetup**. При установке установщик должен добавить в реестр Windows записи о новом 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.
3) **Изменить конструктор главной формы и метод Main для возможности принятия переменных из командной строки.**