План практики по ООП

Настройка интернета

В аудиториях кафедры ИВТ доступ в интернет осуществляется через прокси-сервер ОмГТУ и для выхода в интернет необходимо настроить прокси в системе. Для этого нужно зайти в панель управления => Свойства браузера (свойства обозревателя) => Подключения => Настройка сети. В появившемся окне нужно поставить галочку “Использовать прокси-сервер для локальных подключений”, и прописать адрес proxy.omgtu и порт 8080. Кроме того стоит поставить галочку “не использовать прокс-сервер для локальных адресов”. При работе вне сети ОмГТУ чтобы отключить прокси достаточно убрать галочку “Использовать прокси-сервер для локальных подключений”.

Кроме того на кафедре есть две wifi-точки: IVTwifi и IVTwifi2. Пароль от них: test911MegatestOmsk1 (регистр важен). При работе через эти точки также требуется включить прокси.

Установка и настройка системы контроля версий

Используемая система: git.

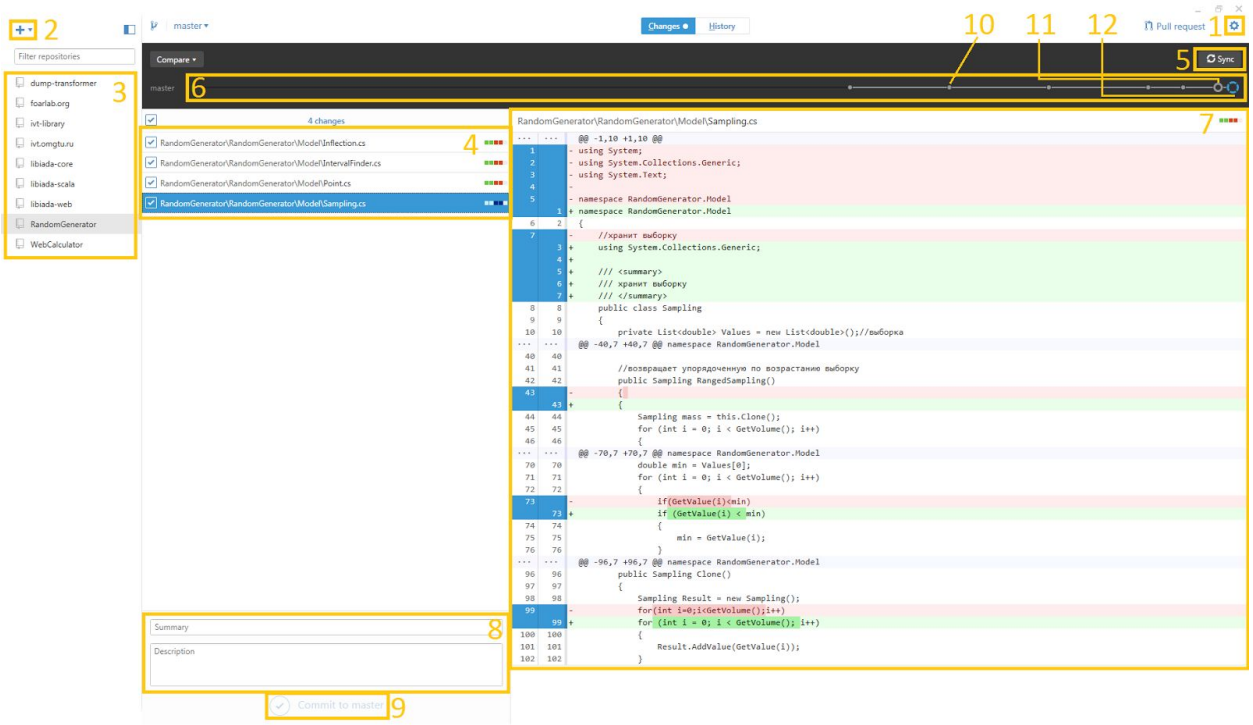
Альтернативы: SVN, Mercurial

Используемый сервис: github.com

Альтернативы: bitbucket.org

Используемое клиентское приложение: github desktop

Альтернативы: git extensions, git for windows (консоль git)



На рисунке:

1-Меню настроек(логина и репозитория)
2-Меню создания и клонирования удалённых репозиториев
3-Список локальных репозиториев
4-Список изменённых,добавленных и удалённых

файлов
5 - Кнопка синхронизации изменений с сервером (pull и push)
6 - История изменения
7 - Изменения в выбранном файле
8 - Сообщение коммита (обязательно для заполнения при создании коммита)
9 - Кнопка создания коммита
10 - Зафиксированный глобально коммит
11 - локальный коммит
12 - незафиксированные изменения (текущее состояние)

1. Зарегистрироваться на github'е (и подтвердить почту).
2. Один человек из группы (команды разработчиков) создаёт репозиторий (хранилище исходнков). Для этого необходимо зайти в свой профиль на сайте, перейти во вкладку "Repositories" и нажать "New". На открывшейся странице нужно ввести название репозитория и выбрать "public" в качестве области видимости, после чего нажать "Create repository".
3. Далее необходимо добавить остальных разработчиков в список коллабораторов. Для этого нужно перейти на страницу созданного репозитория "Settings" и внутри вкладку "Collaborators". В появившемся поле необходимо ввести логины остальных разработчиков.
4. Следующий этап - это установка и настройка клиентского приложения для работы с сервером системы контроля версий. После зугрузки и установки приложения по ссылке выше необходмо ввести логин и пароль, а также псевдоним (желательно использовать логин github чтобы избежать путаницы) и почту. Эти настройки также можно изменить после установки (в меню настроек в правом верхнем

углу окна).

5. При работе в университетской сети необходимо отдельно настроить прокси для клинета git. Для этого необходимо открыть в текстовом редакторе файл .gitconfig, лежащий в папке пользователя(не в документах, а именно в папке пользователя) и добавить туда строчки

```
[http]
     proxy = http://proxy.omgtu:8080
[https]
     proxy = https://proxy.omgtu:8080
```

6. При работе вне сети Омгту нужно не обязательно закомментировать эти строчки с помощью символа # в начале строк, иначе клиет не сможет подключиться к интернету.
7. После входа в учётную запись появляется возможность создать локальную копию удалённого репозитория (clone). Для этого необходимо нажать кнопку “+” в левом верхнем углу окна, выбрать вкладку “clone”, в загрузившемся списке выбрать созданый ранее репозиторий и нажать одноимённую кнопку. После этого нужно выбрать где будет расположен репозиторий на локальной машине.
8. После создания локальной копии репозитория в него можно вносить изменения и делать коммиты (отправлять изменения на удалённый сервер), при этом как глобально так и локально будет сохраняться вся история изменений.
9. В первую очередь имеет смысл настроить технические аспекты. Для того чтобы на сервер не отправлялись временные и другие “ненужные” файлы имеет смысл добавить в репозиторий файл .gitignore, в котором

перечисляются исключаемые файлы (т.е. такие файлы, создание, изменение и удаление которых git будет игнорировать, и они не будут отправляться на сервер и отображаться в истории). Кроме того имеет смысл добавить в репозиторий файл .gitattributes в котором описывается каким расширениям какое содержимое соответствует (текстовое, бинарное, и т.д.), чтобы git мог правильно определять изменения в файлах и отображать их там где необходимо. Версии вышеупомянутых файлов с содержимым по умолчанию, подходящим для большинства языков можно сгенерировать в настройках репозитория (Repository settings). После добавления файлов в локальную историю их нужно отправить на сервер. В интерфейсе кнопка для соответствующего действия расположена в правом верхнем углу окна и называется либо “Publish” (если не было создано ещё ни одного коммита) либо “Sync” (в других клиентах git данная операция скорее всего будет называться push).
10. Таким образом процесс отправки изменений на сервер имеет два этапа: коммит (фиксация набора изменений в локальной истории) и push (отправка коммитов на удалённый сервер).

Здесь стоит также отметить что возможности git намного шире и в частности позволяют осуществлять ветвление, слияние, редактирование веток в истории и много другое.

Налало разработки приложения

Используемый язык: C#

Альтернативы: Java, JavaScript, Python и любые другие объектно-ориентированные языки

Используемая IDE: Visual Studio 2013

Альтернативы: VS 2015, monoDevelop, VS code

1. Для создания проекта в первую очередь необходимо запустить visual studio (далее vs) и выбрать “New project”.
2. В появившемся окне слева выбрать язык C# (Visual C#) и тип приложения Windows Forms Application.
3. Далее необходимо задать название и расположение проекта. Можно сразу указать расположение соответствующее локальной копии репозитория (либо перенести проект сразу после создания).
4. После создания пустой версии приложения его необходимо скомпилировтаь (убедившись таким образом, что оно работает) и открыть клиент системы контролля версий, если проект был перенесён в папку репозитория, то в списке текущих изменений должны повиться новые файлы, причём можно проконтроллировать что файлы .gitignore и .gitattributes также были созданы правильно, если в списке файлов нет файлов с расширением exe, dll и тому подобных, а для всех файлов с исходными кодами можно просмотреть изменения построчно.
5. Если всё в порядке то следует создать коммит (при создании коммита обязательно написать сообщение (Summary), описывающее сделанные изменения) и отправить его на сервер командой sync.
6. После этого в приложение можно начать добавлять функционал. В первую очередь

стоит “собрать” интерфейс, это можно сделать открыв файл Form1.cs в дереве решения (справа) и перетаскивая элементы интерфейса из меню “toolbox” (слева) (если оно закрыто, его (и остальные элементы интерфейса vs) всегда можно открыть через меню View (Вид)).

7. Интерфейс должен включать два поля ввода (TextBox) одно поле для вывода результата (Label либо TextBox) и 4 кнопки для операций (сложение, вычитание, умножение и деление).
8. Для каждого элемента можно задать его расположение, название, текст и прочее через меню свойств properties (справа внизу), открывается выбором соотвутствующего пункта в контекстном меню, либо нажатием клавиши F4 на выбранном элементе интерфейса. Кроме того в этом меню можно просматривать, добавлять и удалять обработчики событий (таких как нажатия кнопок) нажав на значок молнии.
9. Чтобы добавить событие нажатия на кнопку достаточно дважды кликнуть по ней.
10. В созданном методе перед тем как собственно выполнять соответствующую операцию, необходимо извлечь и преобразовать в числа введённые пользователем значения. Чтобы прочитать значение в виде строки необходимо обратиться к свойству Text объекта TextBox (если объект не был переименован, то обращение будет выглядеть как textBox1.Text и textBox2.Text для первого и второго аргументов соответственно, если же объекты были переименованы, то вместо textBox1 и textBox2 необходимо подставить новые имена).
11. Для того чтобы преобразовать полученные значения в числовой вид существует много подходов, один из самых простых заключается

в использовании статического класса Convert. При использовании промежуточных переменных код, использующий этот класс будет выглядеть следующим образом:

```
double имяВещественнойПеременной = Convert.ToDouble(имяСтроковойПеременной);
```

12. Наконец после выполнения самой математической операции необходимо вывести результат на форму чтобы пользователь мог его увидеть. Для это также можно использовать свойство Text третьего поля или метки, в виде textBox3.Text (в случае текстового поля) или label1.Text (в случае метки). Имена также могут отличаться если объекты были переименованы. Кроме того необходимо преобразовать численное значение в строковое, для этого у всех объектов имеется метод ToString(). Итоговый код будет выглядеть следующим образом:

```
label1.Text = вещественнаяПеременнаяСРезультатом.ToString();
```

13. Аналогичным образом реализуются функции для трёх оставшихся математических операций.
14. Если всё сделано правильно, приложение будет компилировать и с его помощью можно будет выполнять простейшие математические операции. В это случае стоит сделать ещё один коммит и отправить его на сервер (в дальнейшем упоминание коммита будет подразумевать также и синхронизацию изменений с сервером).

Рефакторинг

Используемая библиотека для рефакторинга (на самом деле не только для рефакторинга): ReSharper

Альтернативы: множество других плагинов для vs, а также её встроенный функционал

Рефакторинг - это изменение (улучшение) структуры существующего кода без изменения его функциональности.

1. Написанные ранее четыре функции для выполнения базовых математических операций очевидно в значительной степени дублируют друг друга. Можно избавиться от значительной части этого дублирования, при этом возможны два подхода: написание отдельного метода для обработки всех четырёх событий нажатия на кнопки, с выяснением внутри метода какая именно кнопка была нажата, либо написание отдельного метода, вызываемого отдельными обработчиками нажатия на кнопки, который принимает код операции и делает все необходимые преобразования, вычисления и вывод результатов.
2. Здесь будет описан только первый вариант, однако второй вариант также несложно написать исходя из данного. Для его реализации в первую очередь нужно привязать события нажатия на все кнопки к одному методу, это можно сделать (как описывалось выше) в окне свойств кнопок, зайдя на вкладку событий (кнопка с молнией), и вписав вместо обработчика “Click” один и тот же метод у всех кнопок.
3. Внутри метода необходимо выяснить какая именно кнопка была нажата. Проще всего это сделать по имени, которое хранится в самой кнопке, передаваемой в качестве первого

аргумента метода (object sender). Однако сначала нужно преобразовать этот объект в кнопку (изначально она приведена к объему предку всех объектов - классу object). В целом код доступа к имени кнопки будет выглядеть следующим образом:

```
((Button)sender).Name
```

4. Чтобы выбрать какую именно операцию выполнять в зависимости от имени кнопки удобно использовтаь оператор switch

```
switch(((Button)sender).Name)
{
     case “имя первой кнопки”:
          //выполнение операции
          break;
     case “имя второй кнопки”:
          //выполнение операции
          break;
     default:
          throw new Exception(“Неизвестная операция”);
}
```

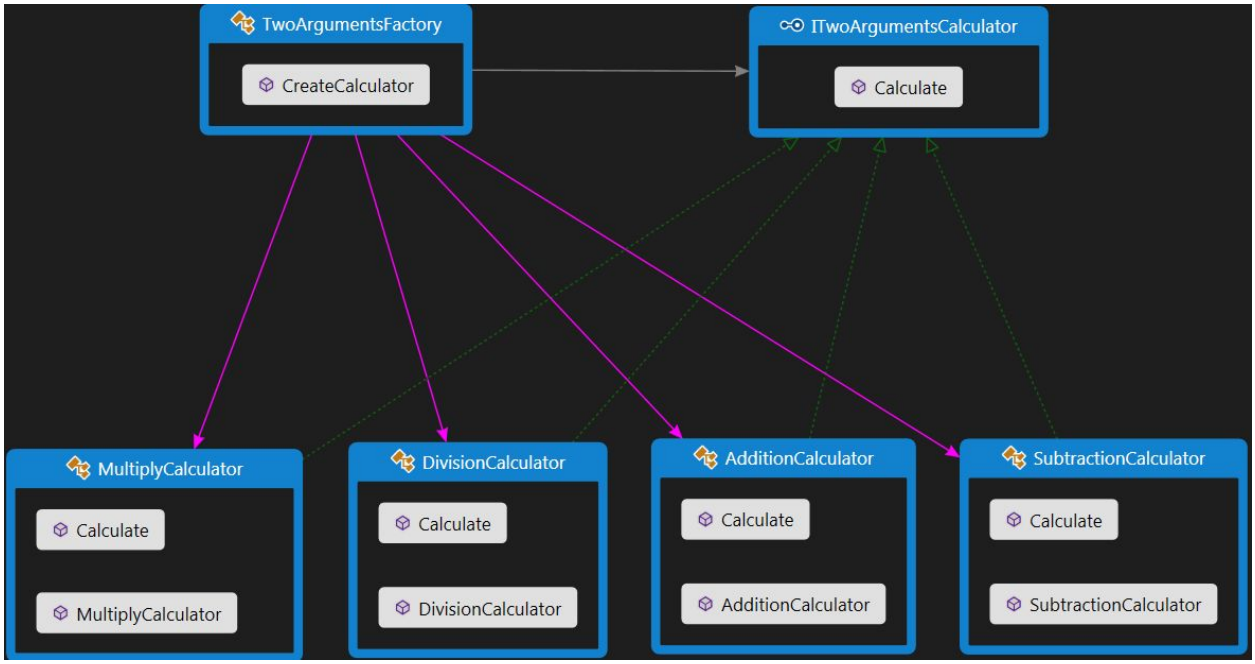
5. В примере выше если метод вызван неизвестной кнопкой и непонятно какую операцию нужно выполнить, “выбрасывается” исключение (Exception). Исключения являются механизмом работы с нештатными ситуациями в объектно-ориентированных языках программирования. Кроме базового исключения существует множество его наследников, отвечающих за конкретные ошибочные ситуации и проблемы (например NullReferenceException, ArgumentException и т.д.). Программист также может создавать своих наследников класса Exception для определения специфичных ошибочных ситуаций. Исключение пробрасывается по стеку вызовов вверх до тех пор пока не будет поймано с помощью команды catch либо не выйдет на верхний

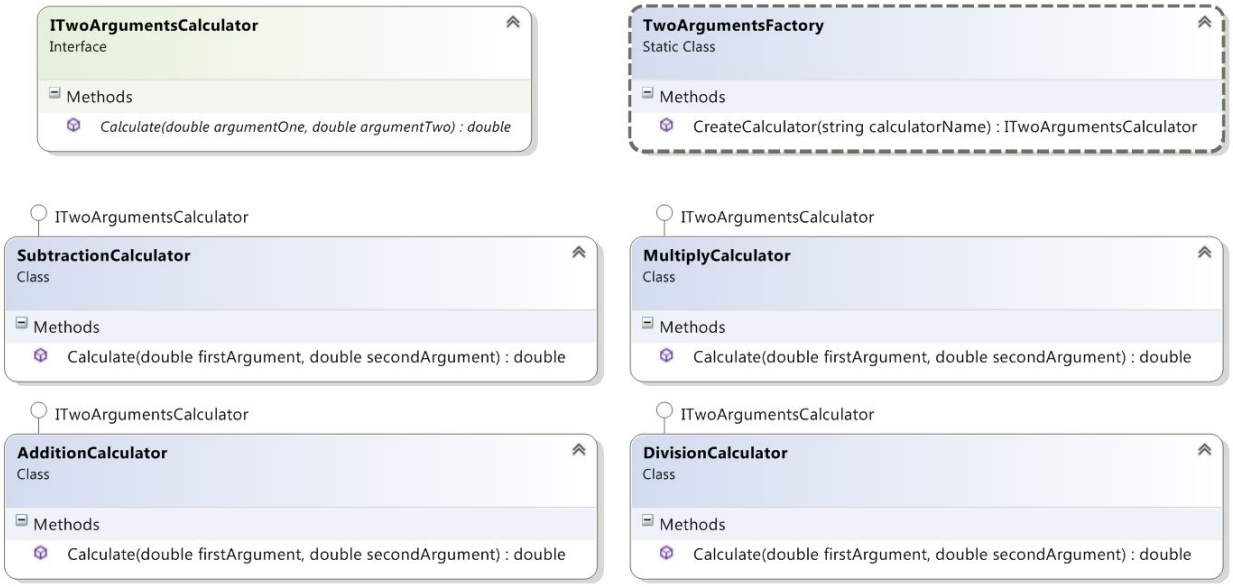
уровень и будет представлено пользователю как ошибка.

6. Если приложение после вышеуказанных изменений компилируется и работает, то следует сделать коммит.

Фабрики, интерфейсы и статические классы

Для облегчения добавления нового функционала имеет смысл отделить код, отвечающий за выполнение самих математических операций от кода отвечающего за обработку входных и выходных данных.





1. Для этого необходимо организовать иерархию состоящую из нескольких классов, и реализующую шаблон проектироования (паттерн)“фабрика”.
2. Данная иерархия будет включать отдельные классы для каждой математической операции. В каждом таком классе будет метод принимающий два вещественных аргумента, выполняющий операцию и возвращающий вещественный результат.
3. Все вычислительные классы будут реализовывать общий интерфейс (не графический, а интерфейс в терминологии ООП), в котором будет один единственный абстрактный метод:

```
double Calculate(double firstArgument, double secondArgument);
```

4. И наконец в иерархии будет присутствовать статический класс-фабрика, отвечающий за создание экземпляров калькуляторов по их имени (вообще фабрика может принимать любой тип параметра однозначно идентифицирующий

тип создаваемого объекта, в частности числа, другие объекты и т.д., но в данном случае наиболее простым и понятным подходом является использование названий операций).

5. Внутри фабрики будет содержаться один статический метод, внутри которого будет содержаться оператор switch, однако в отличии от обобщённого метода внутри этого оператора не будут выполняться сами математические вычисления, а будут создаваться и возвращаться экземпляры калькуляторов:

```
return new ИмяКлассаКалькулятора();
```

6. Использование фабрики в обобщённом методе из графического интерфейса будет выглядеть следующим образом:

```
ITwoArgumentsCalculator calculator = TwoArgumentsFactory.CreateCalculator(имяКалькулятора);
double result = calculator.Calculate(firstArgument, secondArgument);
```

7. При этом нет необходимости заранее создавать экземпляр класса TwoArgumentsFactory, т.к. данный класс является статическим.
8. Если после всех преобразований приложение работает, следует сделать коммит.

Добавление функционала

1. Идентичным образом с иерархией классов для операций с двумя аргументами создаётся иерархия классов для операций с одним аргументом (например Sin, Ln, 2^x, mod, x^2, и т.д.).
2. Такая иерархия будет включать свои интерфейс калькуляторов, фабрику и собственно классы-калькуляторы.
3. После создания второй иерархии классов также следует сделать коммит.

4. Далее становится возможным добавлять калькуляторы для множества математических операций с одним и двумя аргументами. Здесь каждый участник команды должен выбрать несколько функций с один и двумя аргументами, которые он будет добавлять.
5. Важно отметить, что если работа над функциями будет вестись параллельно несколькими участниками, могут возникнуть проблемы синхронизации. Несмотря на то что git хорошо разрешает проблемы параллельного изменения разных файлов и разных строк одного файла, при добавлении новых калькуляторов будут также вноситься измнения в графический интерфейс и файл проекта (он содержит список файлов, входящих в проект), и именно их параллельные изменения придётся на одни и те же строки, в результате придётся не просто вручную разбираться с конфликтом, но и изменять файл который не предназначен для ручного редактирования. Простейший способ согласовать такие измнения это либо заранее редактировать интерфейс, добавив туда кнопки всех операций. Либо просто распределить по времени написание функций между участниками так чтобы оно не пересекалось. Кроме того на данном этапе стоит делать коммиты после написания каждого класса-калькулятора.

Рефакторинг

После добавления всех функций имеет смысл реорганизовать расположение классов в проекте.

1. Чтобы не запутаться в большом количестве

классов в проекте в него можно добавить папки и переместить часть классов в них. В случае калькулятора возможны два способа организации классов по папкам: первый способ предполагает группировку классов в папкх соответствующих их типу (фабрики, интерфейсы, калькуляторы), но такой подход оказывается непрактичным в дальнейшем использовании.

2. Второй подход предполагет группировку по близости выполняемого функционала (опреации с одниим аргументом и операции с двумя аргументами). Такой подход оказывается более удобным в дальнейшем использовании.
3. После перемещания классов в соответствующие папки стоит произвести рефакторинг их пространств имён (пространство имён отдельнго класса должно включать его путь в проекте). Данный процесс можно автоматизировать с помощью resharper'а.
4. Также можно удалить из классов неиспользуемые using'и. И здесь также можно выполнить удаление увтоматически с помощью resharper'а.

Модульные тесты

Используемая библиотека модульного тестирования: NUnit

Альтернативы: mstest, xUnit и т.д.

С целью автоматизации проверки работоспособности приложений было

разработано множество разных подходов и типов тестирования. Самый простой и популярный из них - модульные тесты (Unit тесты). Идея их заключается в тестировании отдельных методов и классов как чёрных ящиков. Т.е. для конкретной ситуации задаются только входные значения и ожидаемый результат, если после выполнения теста они совпадают, тест считается пройденным, если нет - проваленным.

1. Модульные тесты выделяют в отдельный проект в той же сборке (верхний уровень иерархии приложения), что и тестируемый проект. Соответственно в сборку нужно добавить ещё один проект типа “Библиотека классов” (Class library). Проект с тестами обычно называют по шаблону ИмяТестируемогоПроека.Tests. Внутри такой проект воспроизводит структуру каталогов основного проекта, а классы называются по шаблону ИмяТестируемогоКлассаTests.
2. Проект с модульными тестами должен ссылаться на основной проект (использовать его) и подключать библиотеку для модульного тестирования. Чтобы выполнить первое действие необходимо в дереве проекта нажать правой кнопкой на References (Ссылки) и выбрать пункт Add reference. В появившемся окне нужно поставить галочку у имени основного проекта и нажать ok.
3. Библиотеку для модульного тестирования проще всего подключить через пакетный менеджер nuget, который позволит не хранить саму библиотеку в репозитории, а будет скачивать её из интернета при компиляции, и к тому же через nuget сторонние библиотеки можно обновлять. Чтобы добавить библиотеку таким образом нужно опять же нажать на References

правой кнопкой мыши и выбрать Manage nuget packages (Управление пакетами nuget). В появившемся меню нужно слева выбрать Online и написать в поиск Nunit, после чего нажать Install у найденного пакета с одноимённым названием.

4. Чтобы протестировать конкретный класс-калькулятор необходимо создать в тестовом проекте соответствующий ему тестовый класс. В тестовом классе необходимо создать тестовый метод по шаблону

```
[TestFixture]
public class ИмяТестируемогоКлассаTests
{
	[Test]
	public void ИмяТестируемогоМетодаИлиСитуацииTest()
	{
		// подготовительные действия
		Assert.AreEqual(предполагаемыйРезультат, фактическийРезультат);
	}
	// ...
}
```

5. В вышеприведённом примере присутствуют два атрибута [TestFixture] и [Test], они служат для обозначения соответственно класса с тестами и тестового метода (в различных библиотеках для тестирования эти атрибуты могут называться по другому, а часть из них может вообще отсутствовать). Тестовый метод обязательно должен быть публичным (иначе тестовая библиотека не сможет его увидеть), и (в общем случае) не должен иметь никаких аргументов и возвращаемого значения (иначе тестовая библиотека не будет знать с какими именно значениями вызывать этот метод).
6. Для выполнения различных проверок в тесте существует статический класс Assert, в котором в свою очередь представлено множество методов для осуществления тех или иных проверок. В частности метод AreEqual проверяет

равенство двух значений. При этом в одном методе может быть сколько угодно проверок (вызов Assert).

7. Из присутствующих в проекте классов возможно протестировать все калькуляторы и фабрики.
8. Желательно чтобы тесты конкретных калькуляторов были написаны не теми же кто писал сами калькуляторы.
9. После создания тестов стоит сделать коммит.

Продвинутое тестирование

1. Кроме простых тестов NUnit позволяет создавать также продвинутые тесты.
2. Первый тип таких тестов это тесты, основанные на наборах данных. Пример такого тестового метода:

```
[TestCase(0, 0, 0)]
[TestCase(3, 4, 7)]
[TestCase(-7, -2, -9)]
public void CalculateTest(double firstValue,double secondValue, double expected)
{
      var calculator = new AdditionCalculator();
      var actualResult = calculator.Calculate(firstValue, secondValue);
      Assert.AreEqual(expected, actualResult );
}
```

3. Каждому атрибуту TestCase соответствует свой набор данных с которым будет вызван тестовый метод (в качестве аргументов для этого атрибута могут служить только примитивные типы).

Тестирование фабрик

1. Тестирование фабрик в целом схоже с продвинутым тестированием, с той разницей,

что проверяется не равенство значений, а принадлежность созданного фабрикой объекта определённому классу (то что объект является экземпляром класса). Для этого используется специальный метод IsInstanceOf:

```
[TestCase("Название калкулятора", typeof(КлассКалькулятора))]
[TestCase("Название калкулятора", typeof(КлассКалькулятора))]
public void CalculateTest(string name, Type type)
{
	var calculator = TwoArgumentsFactory.CreateCalculator(name);

	Assert.IsInstanceOf(type, calculator);
}
```

2. Для получения типа калькулятора здесь используется специальный оператор typeof, хотя существуют и другие способы извлечь тип из класса.

Тестирование ошибочных ситуаций

1. Также с помощью модульных тестов можно проверить работу приложения в ошибочных ситуациях. Для этого используется следующая конструкция:

```
Assert.Throws<ТипОжидаемогоИсключения>(() => ВызовФункцииВыбрасывающейИсключение);
```

2. В данной строчке кода используются сразу две новых конструкции языка. Первая из них Дженерики (или генерики или generics). Это так называемые средства обобщённого программирования. В классах с конструкциями <> после имени можно подставлять в эти угловые скобки имя класса и поведение дженерика поменяется соответствующим образом. Простейшим примером является класс-коллекция List<T>, например при указании в качестве T типа double

будет хранить вещественные значения, а приуказании типа string будет хранить строки. Соответственно метод Assert.Throws<ТипОжидаемогоИсключения> будет обрабатывать (ожидать) только определённый тип исключения (однако если указать в качестве типа Exception сработает и с любым его потомком).

3. Вторая конструкция это лямбда-выражения (lambda expressions). В большинтство случаев лямбда-выражения используются как анонимные функции состояшие из единственной строки кода (одного утверждения), и возвращающие результат выполнения это единственной строки кода. Например выражение () => 1 + 2 вернёт 3. А выражение (x,y) => x-y вернёт разность между x и y (при ‘том тип аргументов автоматически выводтся из окружающего контекста). В вышеприведённом теста лямба-выражение служит простым способом передать функцию(а не результат её выполнения) в качестве аргумента (такие функции ещё называются анонимными или делегатами).
4. При запуске такой тест скорее всего провалится. Это связано с тем что в C# математически опреации не выбрасывают исключения при возникновении “ошибки”, а возвращают одно из выделенных значений. Например, +∞ при делении положительного числа на 0, -∞ при делении отрицательного числа на 0 и NaN(not a number / не число) в случае других ошибок при выполнении математических опреаций.
5. Чтобы тес проходил, необходимодобавить в калькуляторы проверки выбрасывающие исключение в случае невалидных входных

данных. Например для деления на 0 фрагмент код будет выглядеть следующим образом:

```
if(secondArgument == 0)
{
     throw new Exception("Деление на 0");
}
```

6. При этом на форме необходимо добавить обработку таких исключений. В большинстве объектно-ориентированных языков для этих целей используется конструкция вида:

```
try
{
     // код потенциально выбрасывающий исключения
}
catch(Exception exc)
{
     // обработка исключения
}
finally
{
     // действия которые выполнятся независимо от того,
     // произошло исключение или нет
}
```

7. Таких блоков даже в одно методе можент быть любое число. Если в блоке catch объявлено какое-то конкрентное исключение, то остальные типы исключений не будут отлавливаться, однако будут пойманы все потомки указанного исключения. Чтобы отлавливать несколько типов исключений после одного блока try можно объявить несколько блоков catch. Блок finally является необязательным и обычно служит для закрытия внешних ресурсов (файлов, сетевых соединений, подключений к БД).

Добавление валидации входных данных

todo.

XML-комментарии

Xml-комментарии это особый тип комментариев. Которые служат документацией и всплывающими подсказками в коде. При ниличии resharper'а создание такие комментариев можно частично автоматизировать. Для этого достаточно в соответствующем месте кода поставить /// и структура комментария будет сгенерирована автоматически. Аналогично атрибутам xml-комментарии добавляются перед классами, методами, полями и свойствами. Про особенности синтаксиса и дополнительные возможности xml-комментариев можно прочитать по ссылке.

Добавление нетипичного функционала (сортировка массивов)

todo.

Web-интерфейс и архитектура MVC

todo