Unit-Тестирование.

Написание модульных тестов.

[Немного теории о тестировании](#_gbu3tngzxhjb)

[К каким участкам кода нужно писать юнит-тесты](#_8jtia893uqyx)

[Практика юнит-тестирования](#_447rohgcid9m)

[Расположение юнит-тестов](#_7io7gddydh5e)

[Именование тестов](#_gt451sw603t8)

[Наименование классов](#_7hqb2ydsnd6o)

[Наименование методов](#_bqmcqwcfgi5x)

[Фреймворки тестирования](#_k65xrx7j4aal)

[Подход AAA](#_t4710b82yzp9)

[Создаем юнит-тест](#_b5jt5k3fcbq1)

[Атрибут TestInitialize и ClassInitialize](#_412gl2610f48)

[Домашнее задание](#_iv7bbhonfa26)

[Используемая литература](#_uvp6qax5r1ok)

## Немного теории о тестировании

До этого мы никогда не говорили о тестировании, но на самом деле мы им занимались. Например, когда мы создавали наше приложение «Рассыльщик». Мы нажимаем на кнопку, запускаем приложение и проверяем, как работает «Рассыльщик», отправляются ли письма и т.д. И, как правило, то, чем занимаются программисты, проверяя свои программы на соответствие требованиям, называется интеграционным тестированием. Как правило, в программах используется большое число зависимостей и при проведении интеграционного тестирования смотрят на соответствие требованиям работы всех узлов системы.

Наша задача проверить корректность работы каждого отдельного узла. Для этой цели как раз и нужно юнит-тестирование или модульное тестирование.

Модульное тестирование (unit testing) – это процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. (С) Википедия

В свою очередь, UnitTest – это отдельная часть программы, которая вызывает тестируемый модуль и проверяет корректность его работы.

Результат такого теста считается не пройденным, когда не совпадает с ожидаемым результатом.

Таким образом выделим три типа тестирования:

1. Модульное тестирование – это написание отдельного теста для каждого метода программы с применением объектов искусственно смоделированной среды
2. Интеграционное тестирование – это тестирование, когда на соответствие требований проверяется интеграция модулей, их взаимодействие между собой, а также интеграция подсистем в одну общую систему.
3. Системное тестирование. Это тестирование программного обеспечения, выполняемое на полной интегрированной системе, с целью проверки соответствия системы исходным требованиям, как функциональным, так и не функциональным. (С) Википедия

Таким образом наша задача сводится именно к модульному тестированию. То есть написанию тестов для тех методов, изменение кода в которых может привести к ошибкам, в том числе в тех местах, которые мы уже протестировали.

Не все проекты нужно покрывать тестами.

Например, если ваша программа очень простая и короткая, если это проект-прототип, тестовый проект, проект, где практически нет никакой логики, проект для выставки или презентации на 1-2 дня. Тогда покрывать тестами его не обязательно.

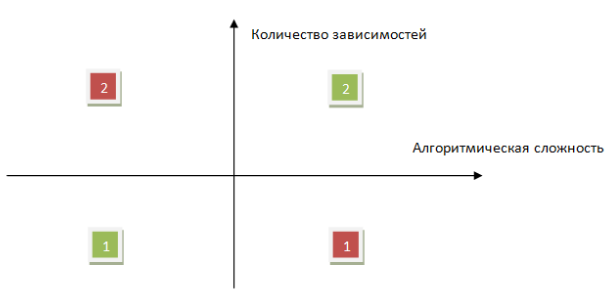
Во всех остальных случаях юнит-тесты значительно облегчат жизнь и даже помогут сделать документацию на ваш проект.

## К каким участкам кода нужно писать юнит-тесты

Надеюсь, интуитивно понятно, что тесты не надо писать абсолютно ко всем функциям приложения.

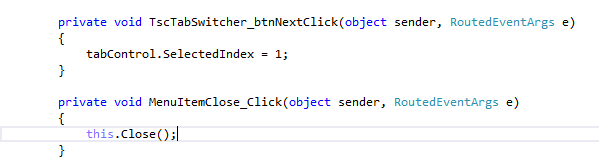
Некоторые считают, что покрыть надо 100% кода, некоторые нет.

Я считаю, что не весь код нужно покрывать тестами. Чтобы определить, какую часть кода надо покрыть, давайте начертим такую схему. По одной оси определим сложность, по другой количество зависимостей

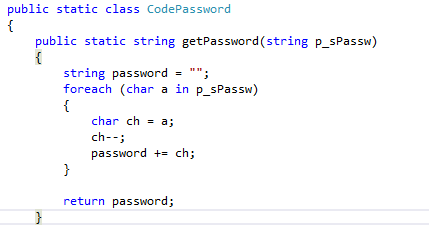


Получилось 4 варианта.

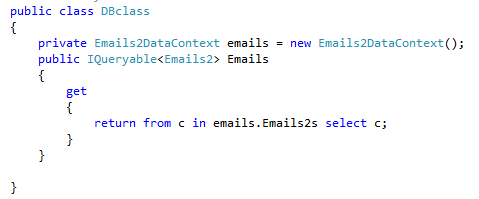
1. Несложный код, где нет зависимостей. Если и так всё ясно, то тестировать нечего. Например, если мы посмотрим в наш класс WPFMailSender, есть ряд обработчиков, которые делают очень простые вещи по нажатию на некоторые пункты меню или кнопки.



1. Запутанный код, где присутствует множество зависимостей. Надеюсь, это не про наше приложение «Рассыльщик». Но если вы столкнетесь с таким кодом, где его проще переписать, чем разобраться, что в нём происходит, то нет смысла покрывать такой код тестами. Проще действительно переписать или провести рефакторинг.
2. Сложный код, где нет зависимостей. То есть методы, которые что-то вычисляют, что-то делают и имеет смысл проверить, делают ли они это правильно. Например, у нас есть в библиотеке CodePasswordDLL класс CodePassword, где зашифровывается пароль. Имеет смысл проверить, всё ли делается правильно.



1. Простой код, где присутствуют зависимости. Когда связываются разные модули системы, разные классы, DLL’ки и так далее. Например, в нашем классе DBClass есть свойство Emails, которое получает данные о email’ах адресатов из базы данных, через dbml объект.



Как видим, варианты 3 и 4 нужно покрыть тестами.

## Практика юнит-тестирования

Юнит тест должен быть:

1. Автоматизированным и повторяемым.
2. Простым в реализации.
3. После написания он должен остаться для последующего использования.
4. Кто угодно в команде должен иметь возможность запустить Unit-тест.
5. Должен запускаться одним нажатием кнопки.
6. Должен выполняться быстро.

### Расположение юнит-тестов

Стандартная рекомендация. Тесты должны располагаться в отдельном проекте. Если в нашем решении несколько проектов, то каждому проекту соответствует проект с тестами. Для нашего приложения «Рассыльщик» используется несколько проектов, отдельный проект для DLL, отдельный проект для UserControl.

### Именование тестов

Рекомендуется проект с тестами именовать, так же как и основной проект, но добавлять слово через точку Test в конце наименования. Например, наш проект MailSender и для него будет создан проект MailSender.Test

### Наименование классов

Если у нас есть класс WPFMailSender, то класс, который будет заниматься тестированием, будет называться WPFMailSenderTest.

### Наименование методов

Наименование методов нужно делать так, что было понятно, какой кусок кода данный метод тестирует, какой сценарий тестирования, входящие переменные, результат или ожидаемое поведение.

Схема наименования может быть такой:

[Тестирующийся метод]\_[Сценарий]\_[Ожидаемое поведение]

Например, у нас в библиотеке CodePasswordDLL есть метод getCodePassword, где мы на вход подаём пароль, а на выходе получаем зашифрованный пароль.

То наименование тестового метода будет таким:

|  |
| --- |
| getCodePassword\_abc\_bcd |

### Фреймворки тестирования

Для того чтобы создавать юнит-тесты, можно использовать различные фреймворки.

Unit Test Framworks:

1. Nunit.
2. MS Unit.
3. xUnit.Net.

Все эти фреймворки похожи между собой. В visual studio практически для всех версий доступен MS Unit. Разобравшись с ним, вы сможете разобраться и с другими фреймворками для тестирования.

### Подход AAA

Если вы поизучаете различные источники по юнит-тестированию, то натолкнётесь на такой вариант составления юнит-теста, как подход AAA.

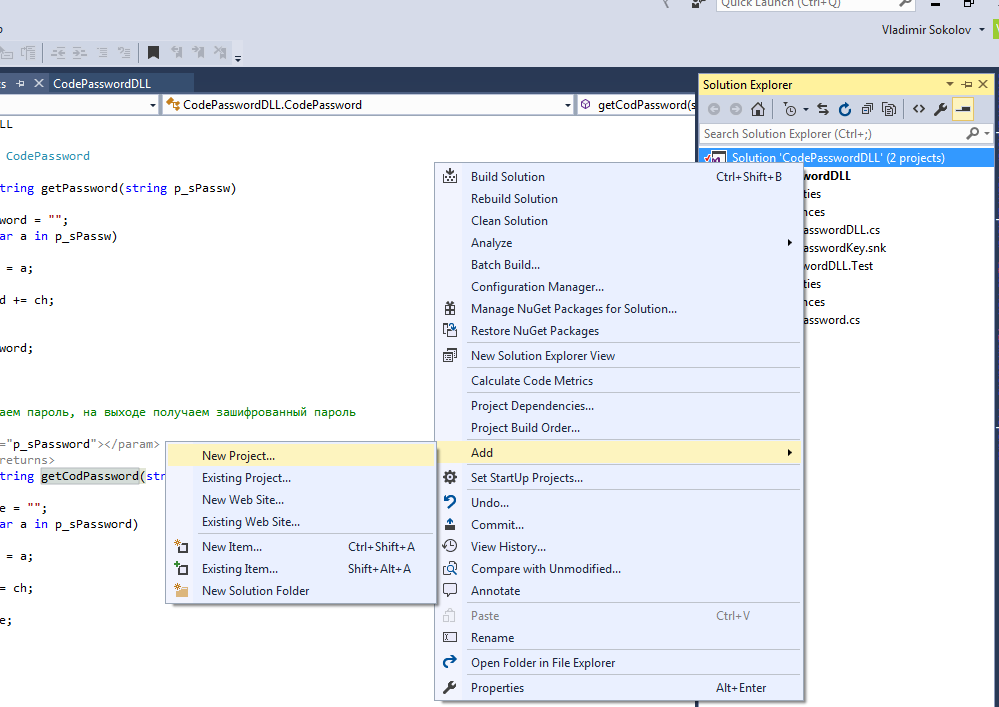
Заключается он в следующем:

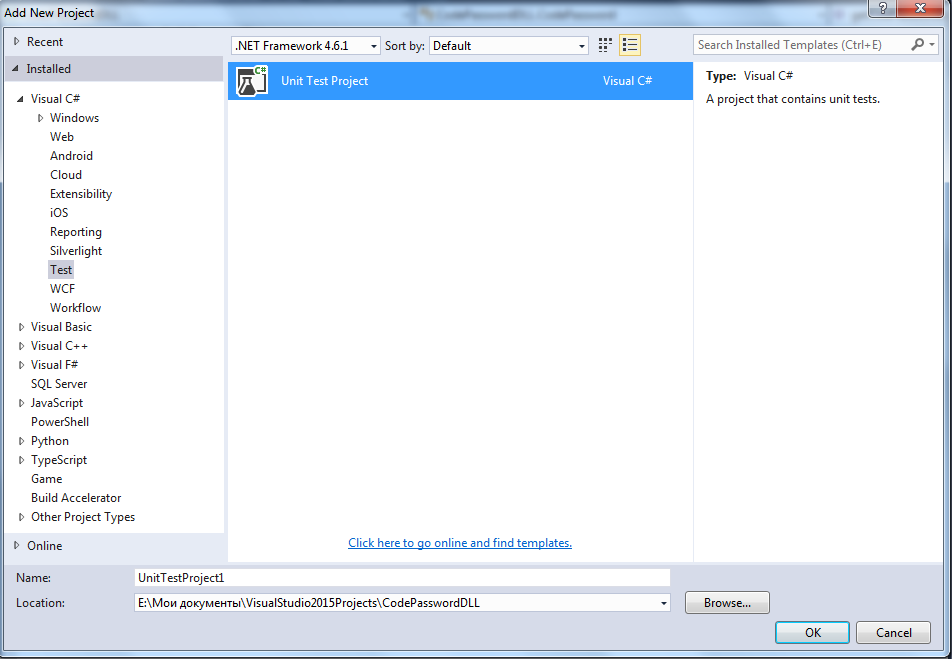
1. Arrange, переменные для того, чтобы выполнить тестирования.
2. Act, определенные действия над системой, которую тестируем.
3. Assert, проверяем операции, которые выполняются над системой завершились успешно.

## Создаем юнит-тест

Начнём создавать юнит-тесты с того, что протестируем библиотеку CodePasswordDLL, в которой содержиться два метода getPassword и getCodPassword.

Заходим в решение CodePasswordDLL и кликаем правой кнопкой мыши.



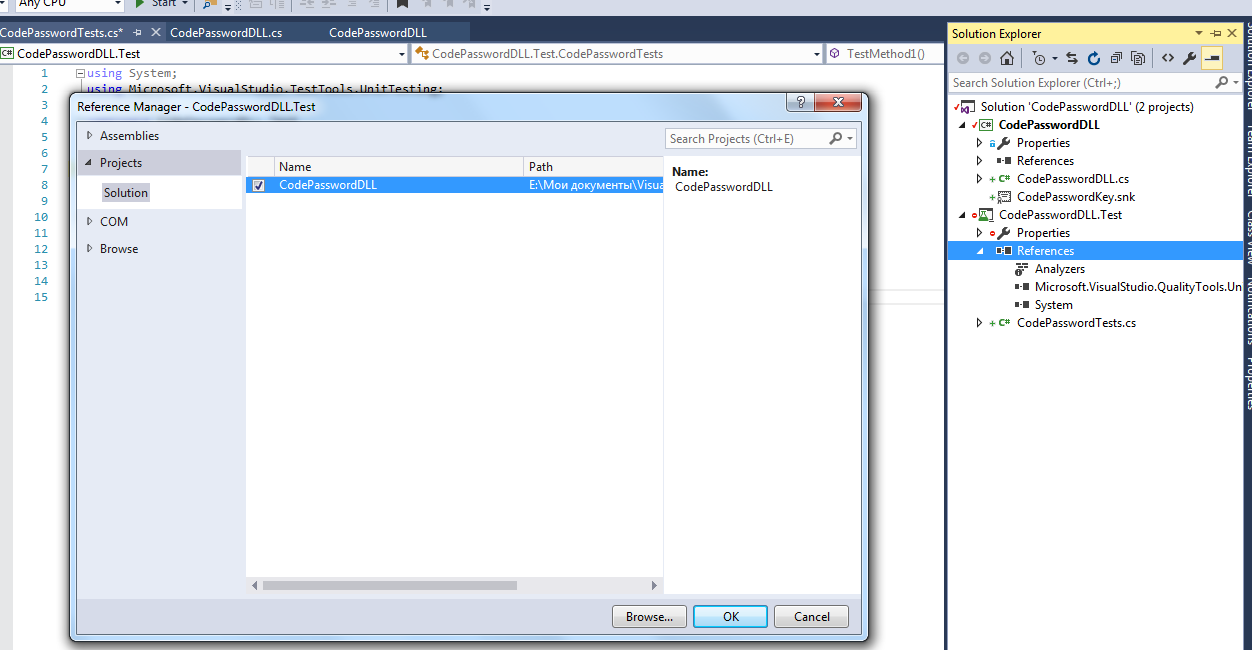


И из списка проектов выбираем Test->Unit Test Project.

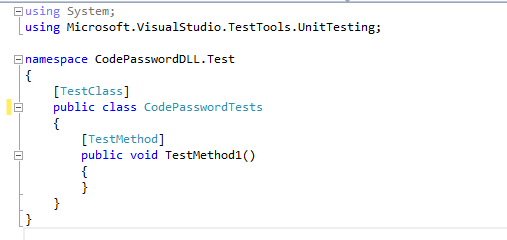
И присваиваем ему название CodePasswordDLL.Tests.

Сразу переименуем класс, который по умолчанию создался UnitTest1.cs на CodePasswordTests.cs.

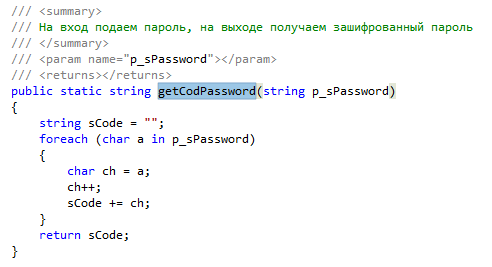
Нам нужно подключить сборку, которую мы тестируем.



В результате получился вот такой код:



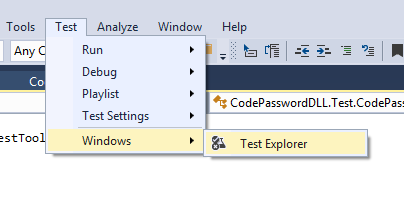
Теперь напишем тест, который будет проверять правильность работы метода getCodPassword, который зашифровывает пароль.

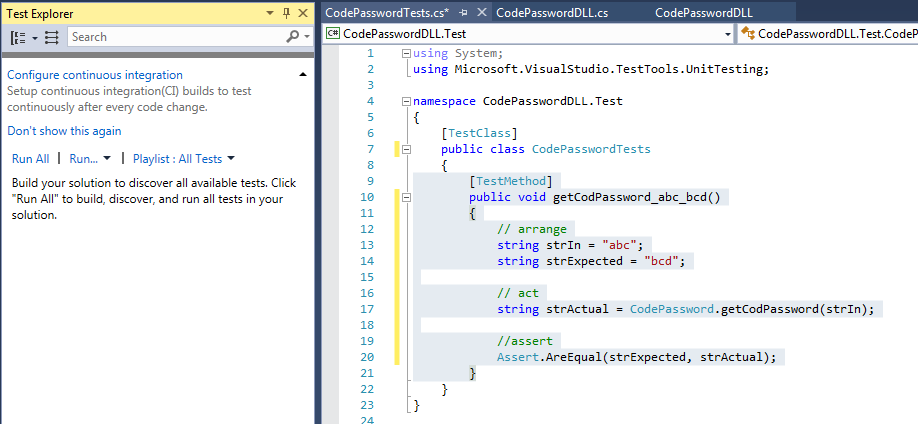


Переименуем сигнатуру метода TestMethod1 в getCodPassword\_abc\_bcd, что означает, что мы тестируем метод getCodPassword, подаём на вход строку “abc” и на выходе получаем строку “bcd”.

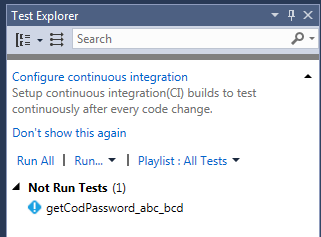
|  |
| --- |
| [TestMethod]  public void getCodPassword\_abc\_bcd()  {  // arrange  string strIn = "abc";  string strExpected = "bcd";  // act  string strActual = CodePassword.getCodPassword(strIn);  //assert  Assert.AreEqual(strExpected, strActual);  } |

Для того чтобы посмотреть, какие юнит-тесты у нас сейчас присутствуют, мы можем воспользоваться окном TestExplorer.

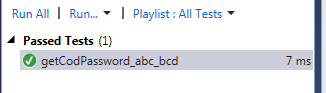




Если мы перестроим решение, то увидим, что в списке тестов в TestExplorer увидим наш тест.



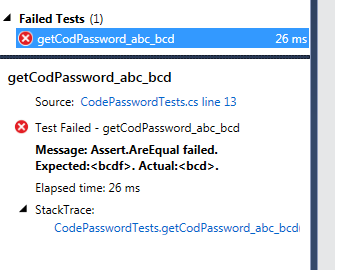
Мы можем запустить все тесты, которые есть в списке и кликнуть по Run All, а можем кликнуть правой кнопкой мыши по тесту и запустить конкретно его. И увидим, что тест прошел без ошибок.



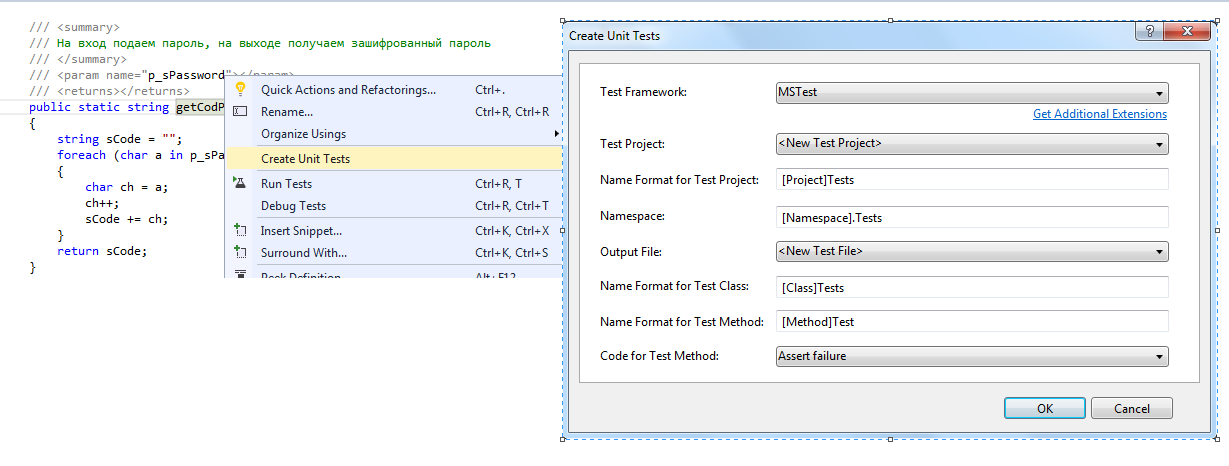
Если, например, мы изменим код теста, переменную strExpected:



То тест будет проходить с ошибкой:



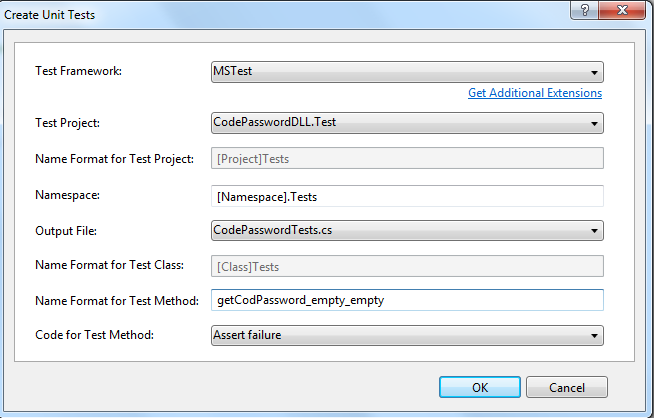
Юнит-тест можно создать другим способом. Прямо из тестируемого приложения, кликнув мышью по тестируемому методу.



Мы можем выбрать Test Framework, по умолчанию выбираем MSTest

Project и OutputFile выбираем те, которые у нас уже есть. И протестируем вариант, если мы подаём на вход пустую строку. Поэтому назовем тест метод вот так getCodPassword\_empty\_empty.

И получаем новый метод в нашем тестовом проекте.



Так как мы выбрали в конце Assert failure, то заканчиваться он будет всегда ошибкой. Поэтому исправим наш код и проверим, что будет, если мы подадим пустую строку в качестве пароля.

|  |
| --- |
| [TestMethod()]  public void getCodPassword\_empty\_empty()  {  // arrange  string strIn = "";  string strExpected = "";  // act  string strActual = CodePassword.getCodPassword(strIn);  //assert  Assert.AreEqual(strExpected, strActual);  } |

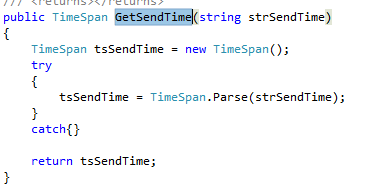
Такой тест тоже проходит отлично.

## Атрибут TestInitialize и ClassInitialize

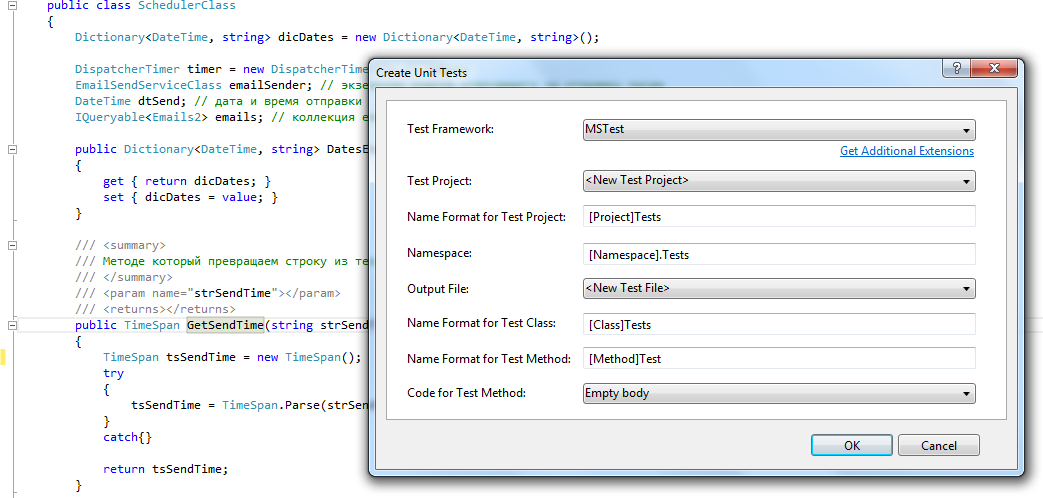
Перед запуском юнит-теста или нескольких ютин-тестов иногда нужно запустить какой-либо метод, и он помечается атрибутом [TestTinitialize].

Для того чтобы выполнить настройку каких-то объектов, которые будут использоваться всеми юнит-тестами в одном тестовом классе, мы используем атрибут [ClassInitialize].

Давайте протестируем метод GetSendTime из класса SchedulerClass.



Кликнем по нему правой кнопкой мыши и выберем “Create Unit Test”, настройки сделаем как на рисунке ниже.

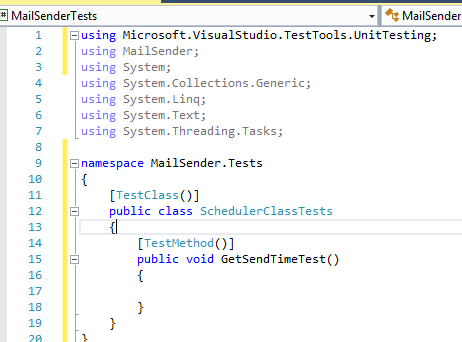


Сделаем серию тестов с целью протестировать метод GetSendTime, который на вход получает строку, а на выходе выдаёт объект TimeSpan. По логике нашей программы метод должен выдавать корректный объект, если мы подаём на вход строку вида “hh:mm”, где hh – это часы ( то есть числа от 0 до 23), mm – это минуты (то есть числа от 0 до 59).

Либо соответственно строки вида “d.hh:mm:ss.ff”

Во всех остальных случаях метод должен возвращать пустой объект TimaSpan, который и будет сигнализировать, что данные на вход поданы ошибочные.

Если мы зайдём в созданный проект для тестирования метода MailSenderTest в класс SchedulerClassTest, то увидим там вот что:



Немного изменим данный код.

|  |
| --- |
| [TestClass()]  public class SchedulerClassTests  {  SchedulerClass sc;  TimeSpan ts;  // Запускается перед стартом каждого тестирующего метода.  [TestInitialize]  public void TestInitialize()  {  sc = new SchedulerClass();  ts = new TimeSpan(); // возвращаем в случае ошибочно введенного времени  }  [TestMethod()]  public void GetSendTime\_empty\_ts()  {  string strTimeTest = "";  TimeSpan tsTest = sc.GetSendTime(strTimeTest);  Assert.AreEqual(ts, tsTest);  }  [TestMethod()]  public void GetSendTime\_sdf\_ts()  {  string strTimeTest = "sdf";  TimeSpan tsTest = sc.GetSendTime(strTimeTest);  Assert.AreEqual(ts, tsTest);  }  [TestMethod()]  public void GetSendTime\_correctTime\_Equal()  {  string strTimeTest = "12:12";  TimeSpan tsCorrect = new TimeSpan(12, 12, 0);  TimeSpan tsTest = sc.GetSendTime(strTimeTest);  Assert.AreEqual(tsCorrect, tsTest);  }  [TestMethod()]  public void GetSendTime\_inCorrectHour\_ts()  {  string strTimeTest = "25:12";  TimeSpan tsTest = sc.GetSendTime(strTimeTest);  Assert.AreEqual(ts, tsTest);  }  [TestMethod()]  public void GetSendTime\_inCorrectMin\_ts()  {  string strTimeTest = "12:65";  TimeSpan tsTest = sc.GetSendTime(strTimeTest);  Assert.AreEqual(ts, tsTest);  }  } |

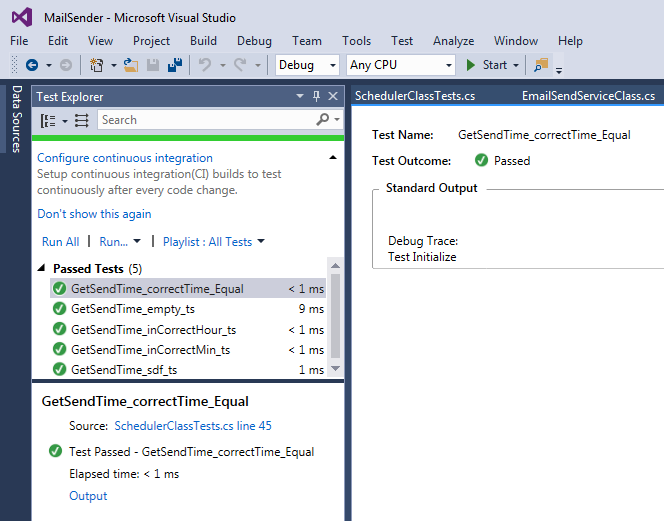
Все тесты проходят отлично.

Добавим ещё одну небольшую фичу. Добавим к объявлениям using ещё одну сборку using System.Diagnostics;

В метод TestInitialize() добавим ещё одну строку. Debug.WriteLine("Test Initialize");

|  |
| --- |
| // Запускается перед стартом каждого тестирующего метода.  [TestInitialize]  public void TestInitialize()  {  Debug.WriteLine("Test Initialize");  sc = new SchedulerClass();  ts = new TimeSpan(); // возвращаем в случае ошибочно введенного времени  } |

При помощи класса Debug из библиотеки System.Diagnostic мы можем отправлять в Output различные сообщения.



Кроме метода TestInitialize с атрибутом [TestInitialize], мы можем создать метод TestCleanup с атрибутом [TestCleanup]. Предположим, мы бы реализовали для тестируемого класса интерфейс IDisposable. И после выполнения каждого TestMethod, нам нужно было бы уничтожать экземпляр класса или возможно удалять что-нибудь ещё, то мы бы воспользовались этим методом.

Теперь рассмотрим вариант с атрибутом [ClassInitialize].

Усложним наш класс SchedulerClass. Сделаем так, чтобы он мог работать если мы для отправки передаём не конкретную дату и время и конкретный текст письма, а несколько писем, которые нужно отправить в разное время. Для простоты пока рассмотрим вариант, что дата отправки одна, а время разное.

Добавим в класс SchedulerClass поле dicDates и свойство, которое ему соответствует DatesEmailTexts

|  |
| --- |
| Dictionary<DateTime, string> dicDates = new Dictionary<DateTime, string>();  public Dictionary<DateTime, string> DatesEmailTexts  {  get { return dicDates; }  set  {  dicDates = value;  dicDates = dicDates.OrderBy(pair => pair.Key).ToDictionary(pair => pair.Key, pair => pair.Value);  }  } |

Теперь видоизменим метод, который вызывается в таймере Timer\_Tick

|  |
| --- |
| private void Timer\_Tick(object sender, EventArgs e)  {  if (dicDates.Count == 0)  {  timer.Stop();  MessageBox.Show("Письма отправлены.");  }  else if (dicDates.Keys.First<DateTime>().ToShortTimeString() == DateTime.Now.ToShortTimeString())  {  emailSender.Body = dicDates[dicDates.Keys.First<DateTime>()];  emailSender.Subject = $"Рассылка от {dicDates.Keys.First<DateTime>().ToShortTimeString()} ";  emailSender.SendMails(emails);  dicDates.Remove(dicDates.Keys.First<DateTime>());  }  }//private void Timer\_Tick(object sender, EventArgs e) |

Теперь атрибут TestInitialize превратим в ClassInitiaze:

|  |
| --- |
| private static SchedulerClass sc;  private static TimeSpan ts;  // Запускается перед стартом каждого тестирующего метода.  [ClassInitialize]  public static void TestInitialize(TestContext context)  {  Debug.WriteLine("Test Initialize");  sc = new SchedulerClass();  ts = new TimeSpan(); // возвращаем в случае ошибочно введенного времени  sc.DatesEmailTexts = new Dictionary<DateTime, string>()  {  { new DateTime(2016, 12, 24, 22, 0, 0), "text1" },  { new DateTime(2016, 12, 24, 22, 30, 0), "text2" },  { new DateTime(2016, 12, 24, 23, 0, 0), "text3" }  };  } |

А добавим метод TimeTick\_Dictionare\_correct при помощи которого будем тестировать метод TimeTick

|  |
| --- |
| [TestMethod()]  public void TimeTick\_Dictionare\_correct()  {  DateTime dt1 = new DateTime(2016, 12, 24, 22, 0, 0);  DateTime dt2 = new DateTime(2016, 12, 24, 22, 30, 0);  DateTime dt3 = new DateTime(2016, 12, 24, 23, 0, 0);  if (sc.DatesEmailTexts.Keys.First<DateTime>().ToShortTimeString() == dt1.ToShortTimeString())  {  Debug.WriteLine("Body " + sc.DatesEmailTexts[sc.DatesEmailTexts.Keys.First<DateTime>()]);  Debug.WriteLine("Subject " + $"Рассылка от {sc.DatesEmailTexts.Keys.First<DateTime>().ToShortDateString()} {sc.DatesEmailTexts.Keys.First<DateTime>().ToShortTimeString()}");  sc.DatesEmailTexts.Remove(sc.DatesEmailTexts.Keys.First<DateTime>());  }  if (sc.DatesEmailTexts.Keys.First<DateTime>().ToShortTimeString() == dt2.ToShortTimeString())  {  Debug.WriteLine("Body " + sc.DatesEmailTexts[sc.DatesEmailTexts.Keys.First<DateTime>()]);  Debug.WriteLine("Subject " + $"Рассылка от {sc.DatesEmailTexts.Keys.First<DateTime>().ToShortDateString()} {sc.DatesEmailTexts.Keys.First<DateTime>().ToShortTimeString()}");  sc.DatesEmailTexts.Remove(sc.DatesEmailTexts.Keys.First<DateTime>());  }  if (sc.DatesEmailTexts.Keys.First<DateTime>().ToShortTimeString() == dt3.ToShortTimeString())  {  Debug.WriteLine("Body " + sc.DatesEmailTexts[sc.DatesEmailTexts.Keys.First<DateTime>()]);  Debug.WriteLine("Subject " + $"Рассылка от {sc.DatesEmailTexts.Keys.First<DateTime>().ToShortDateString()} {sc.DatesEmailTexts.Keys.First<DateTime>().ToShortTimeString()}");  sc.DatesEmailTexts.Remove(sc.DatesEmailTexts.Keys.First<DateTime>());  }  Assert.AreEqual(0, sc.DatesEmailTexts.Count);  } |

Также мы можем добавить метод с атрибутом ClassCleanup, который будет выполняться после выполнения последнего метода в классе.

Ещё есть атрибут AssembleInitialize и AssembleCleanup, которые, соответственно, будут выполняться перед первым методом в сборке с тестами и после последнего метода в сборке с тестами.

Есть три класса, которые помогают тестировать различные участки кода:

1. Assert.
2. CollectionAssert.
3. StringAssert.

У класса Assert есть метод AreEqual, который есть 18 различных вариантов перегрузок.

Например, можно добавить в качестве параметра строковый тип message. В нём можно указать, что мы сравниваем и какая возможна ошибка. Это сообщение будет передано в output.

Есть параметр delta, в котором можно указать погрешность сравнивания.

Есть для сравнения подаём строки, то третьим параметром можно передать параметр с типом bool, который будет означать, учитываем ли мы регистр.

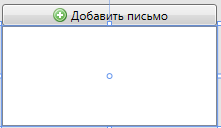
Метод из класса Assert.AreSame помогает сравнить ссылки.

Класс CollectionAssert имеет ряд методов, которые помогают сравнивать коллекции между собой. Части коллекций и т.д.

Класс StringAssert помогает сравнивать строки, части строк и т.д.

# Домашнее задание

1. Задание направлено на повторение предыдущих трех уроков. Мы добавили реализацию возможности отправлять серию писем в один день при помощи класса SchedulerClass. Так вон необходимо добавить такую возможность на весь проект.
   1. Необходимо убрать параметр dtSend из сигнатуры метода SendEmails, класса SchedulerClass
   2. Добавить возможность добавить несколько вариантов времени отправления писем на одну дату и соответствующих текстов писем.
      1. Вместо поля tbTimePicker (или tbTimePicker, для тех, кто смог заменить текстовое поле на элемент TimePicker) нужно добавить Элемент ListView и сверху кнопку «Добавить письмо», по клику на которую будет добавляться новый элемент в ListView. Выглядеть это должно вот так



* + 1. В качестве элемента в ListView добавлять самописный контрол. Для которого нужно создать свой проект ListViewItemScheduler. Выглядеть контрол должен вот так



Где посередине поле, куда записываются даты. Кнопка «-» удаляет элемент из списка. Кнопка с карандашиком вызывает окно с RichTextBox, в который можно записать текст письма.

* 1. При клике на кнопку «Отправить запланированно»



Нужно забирать информацию из ListView, передавать её в класс SchedulerClass в поле Dictionary<DateTime, string> и обрабатывать так как мы прописали на уроке.

1. Добавить UnitTest для метода getPassword проекта CodePasswordDLL.
   1. Методы с тестами добавить в проект CodePasswordDLL.Test класс CodePasswordTest по аналогии с тестами для метода getCodPassword.
   2. Добавить в класс CodePasswordTest методы TestInitialize и TestCleanup.
   3. Расширить метод Assert.AreEqual параметром с сообщением для output.
   4. Добавить Debug.WriteLine внутри тестовых методов и посмотреть, как записываются текстовые сообщения
   5. Добавить тестовые методы для этого же класса с использованием методов StringAssert .
2. Добавить тестовые методы для класса SchedulerClassTest с использованием тестовой коллекции sc.DatesEmailTexts и методов класса CollectionAssert

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы: