**Практическая работа №3**

**Модульное тестирование**

**Цель работы:** изучить возможность создания автоматических тестов, для модульного тестирования.

**Модульное тестирование**, или **юнит-тестирование** (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

**Задачи и цели модульного тестирования**

Каждая сложная программная система состоит из отдельных частей - модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе всей системы, необходимо вначале протестировать каждый *модуль* системы *по* отдельности. В случае возникновения проблем при тестировании системы в целом это позволяет проще выявить модули, вызвавшие проблему, и устранить соответствующие дефекты в них. Такое тестирование модулей *по* отдельности получило называние *модульного тестирования (unit testing)*.

Для каждого модуля, подвергаемого тестированию, разрабатывается тестовое окружение, включающее в себя *драйвер* и *заглушки*, готовятся тест-требования и тест-планы, описывающие конкретные тестовые примеры.

Основная цель модульного тестирования - удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его *интеграция* в состав системы.

При этом в ходе модульного тестирования решаются следующие основные задачи :

1. Поиск и документирование несоответствий требованиям
2. Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия
3. Поддержка рефакторинга модулей
4. Поддержка устранения дефектов и отладки

Первая задача - классическая задача тестирования, включающая в себя не только разработку тестового окружения и тестовых примеров, но и выполнение тестов, *протоколирование* результатов выполнения, составление отчетов о проблемах.

Вторая задача больше свойственна "легким" методологиям типа XP, где применяется принцип тестирования перед разработкой (Test-driven *development*), при котором основным источником требований для программного модуля является тест, написанный до реализации самого модуля. Однако, даже при классической схеме тестирования модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные *механизмы* работы с модулем.

Третья задача связана с поддержкой процесса изменения системы. Достаточно часто в ходе разработки требуется проводить рефакторинг модулей или их групп - оптимизацию или полную переделку программного кода с целью повышения его сопровождаемости, скорости работы или надежности. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода выполняет те же функции, что и старый.

Последняя, четвертая, задача сопряжена с обратной связью, которую получают разработчики от тестировщиков в виде отчетов о проблемах. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

В силу того, что модули, подвергаемые тестированию, обычно невелики *по* размеру, *модульное тестирование* считается наиболее простым (хотя и достаточно трудоемким) этапом тестирования системы. Однако, несмотря на внешнюю простоту, с модульным тестированием сопряжены две проблемы.

Первая из них связана с тем, что не существует единых принципов определения того, что в точности является отдельным модулем.

Вторая заключается в различиях в трактовке самого понятия модульного тестирования - понимается ли под ним обособленное тестирование модуля, работа которого поддерживается только тестовым окружением, или речь идет о проверке корректности работы модуля в составе уже разработанной системы. В последнее время термин "*модульное тестирование*" чаще используется во втором смысле, хотя в этом случае речь скорее идет об интеграционном тестировании.

Эти две проблемы рассмотрены в двух следующих разделах.

**Понятие модуля и его границ. Тестирование классов**

Традиционное *определение* модуля с точки зрения его тестирования: "*модуль* - это *компонент* минимального размера, который может быть независимо протестирован в ходе верификации программной системы". В реальности часто возникают проблемы с тем, что считать модулем. Существует несколько подходов к данному вопросу:

* модуль - это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;
* модуль - это программный модуль, т.е. минимальный компилируемый элемент программной системы;
* модуль - это задача в списке задач проекта (с точки зрения его менеджера);
* модуль - это участок кода, который может уместиться на одном экране или одном листе бумаги;
* модуль - это один класс или их множество с единым интерфейсом.

Обычно за тестируемый *модуль* принимается либо программный *модуль* (*единица* компиляции) в случае, если система разрабатывается на процедурном языке программирования, или *класс*, если система разрабатывается на объектно-ориентированном языке.

В случае систем, написанных на процедурных языках, процесс тестирования модуля происходит так, как это было рассмотрено в предыдущих лекциях: для каждого модуля разрабатывается тестовый *драйвер*, вызывающий функции модуля и собирающий результаты их работы, и набор заглушек - они имитируют поведение функций, которые содержатся в других модулях, не попадающих под тестирование данного модуля. При тестировании объектно-ориентированных систем существует ряд особенностей, прежде всего вызванных *инкапсуляцией данных* и методов в классах.

В случае объектно-ориентированных систем более мелкое *деление* классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно, поскольку для тестирования каждого метода потребуется разработка тестового окружения, сравнимого *по* сложности с уже написанным программным кодом класса. Кроме того, *декомпозиция* класса нарушает принцип инкапсуляции, согласно которому объекты каждого класса должны вести себя как единое целое с точки зрения других объектов.

Процесс тестирования классов как модулей иногда называют компонентным тестированием. В ходе такого тестирования проверяется взаимодействие методов внутри класса и правильность доступа методов к внутренним данным класса. Здесь возможно обнаружение не только стандартных дефектов, связанных с выходами за границы диапазона или неверно реализованными требованиями, но и специфических дефектов объектно-ориентированного программного обеспечения:

* дефектов инкапсуляции, в результате которых, например, *сокрытые данные* класса оказываются недоступными для соответствующих публичных методов;
* дефектов наследования, при наличии которых схема наследования блокирует важные данные или методы от классов-потомков;
* дефектов полиморфизма, при которых полиморфное поведение класса оказывается распространенным не на все возможные классы;
* дефектов инстанцирования, при которых во вновь создаваемых объектах класса не устанавливаются корректные значения по умолчанию параметров и внутренних данных класса.

Однако, согласно [[10](https://intuit.ru/studies/courses/1040/209/literature#literature.10)], выбор класса в качестве тестируемого модуля имеет и ряд сопряженных проблем.

* **Определение степени полноты тестирования класса**. В том случае, если в качестве тестируемого модуля выбран класс, не совсем ясно, как определять степень полноты его тестирования. С одной стороны, можно использовать классический критерий полноты покрытия программного кода тестами: если полностью выполнены все структурные элементы всех методов, как публичных, так и скрытых, то тесты можно считать полными.

Однако существует альтернативный подход к тестированию класса, в котором все публичные методы должны предоставлять пользователю данного класса согласованную схему работы - тогда достаточно проверить типичные корректные и некорректные сценарии работы с данным классом. Т.е., например, в классе, объекты которого представляют записи в телефонной книжке, одним из типичных сценариев работы будет "Создать запись \to искать запись и найти ее \to удалить запись \to искать запись вторично и получить сообщение об ошибке".

Различия в этих двух методах напоминают различия между тестированием черного и белого ящиков, но, на самом деле, второй подход отличается от черного ящика тем, что функциональные требования на систему могут быть составлены на уровне более высоком, чем отдельные классы, и установление адекватности тестовых сценариев требованиям остается на откуп тестировщику.

* **Протоколирование состояний объектов и их изменений**. Некоторые методы класса предназначены не для выдачи информации пользователю, а для изменения внутренних данных объекта класса. Значение внутренних данных объекта определяет его состояние в каждый отдельный момент времени, а вызов методов, изменяющих данные, изменяет и состояние объекта. При тестировании классов необходимо проверять, что класс адекватно реагирует на внешние вызовы в любом из состояний. Однако, зачастую из-за инкапсуляции данных невозможно определить внутреннее состояние класса программными способами внутри драйвера.

В этом случае может помочь составление схемы поведения объекта как конечного автомата с определенным набором состояний. Такая схема может входить в низкоуровневую проектную документацию (например, в составе описания архитектуры системы), а может составляться тестировщиком или разработчиком на основе функциональных требований к системе. В последнем случае для определения всех возможных состояний может потребоваться ручной анализ программного кода и определение его соответствия требованиям. Автоматизированное тестирование в этом случае может лишь определить, по всем ли выявленным состояниям осуществлялись переходы и все ли возможные реакции проверялись.

* **Тестирование изменений**. Как уже упоминалось выше, модульные тесты - мощный инструмент проверки корректности изменений, внесенных в исходный код при рефакторинге. Однако, в результате рефакторинга только одного класса, как правило, не меняется его внешний интерфейс с другими классами (интерфейсы меняются при рефакторинге сразу нескольких классов). В результате обычных эволюционных изменений системы у класса может меняться внешний интерфейс, причем как по формальным (изменяются имена и состав методов, их параметры), так и по функциональным (при сохранении внешнего интерфейса меняется логика работы методов) признакам. Для проведения модульного тестирования класса после таких изменений потребуется изменение драйвера и, возможно, заглушек. Но только модульного тестирования в данном случае недостаточно, необходимо также проводить и интеграционное тестирование данного класса вместе со всеми классами, которые связаны с ним по данным или по управлению.

Вне зависимости от того, на какие модули, подвергаемые тестированию, разбивается система, рекомендуется изложить принципы выделения тестируемых модулей в плане и стратегии тестирования, а также составить на базе структурной схемы архитектуры системы новую структурную схему, на которой нужно отметить все тестируемые модули. Это позволит спрогнозировать состав и сложность драйверов и заглушек, требуемых для модульного тестирования системы. Такая схема также может использоваться позже на этапе модульного тестирования для выделения укрупненных групп модулей, подвергаемых интеграции.

**Организация модульного тестирования**

*Модульное тестирование*, с точки зрения тестировщика, - это комплекс *работ* *по* выявлению дефектов в тестируемых модулях. В эти работы включается *анализ* требований, разработка тест-требований и тест-планов, разработка тестового окружения, выполнение тестов, сбор информации об их прохождении.

Однако, с точки зрения руководителя группы тестирования (или с точки зрения руководителя проекта, если в нем не выделена отдельная *группа* тестирования), *модульное тестирование* является более широким понятием. Для того, чтобы процесс модульного тестирования мог функционировать совместно с другими процессами разработки, он должен включать в себя несколько фаз: планирование процесса, разработку тестов, выполнение тестов, сбор статистики, управление отчетами о выявленных дефектах.

Согласно стандарту *IEEE* 1008 [[19](https://intuit.ru/studies/courses/1040/209/literature#literature.19)] процесс модульного тестирования состоит из трех фаз, в состав которых входит 8 видов деятельности (этапов).

* Фаза планирования тестирования
  1. Этап планирования основных подходов к тестированию, ресурсное планирование и календарное планирование
  2. Этап определения свойств, подлежащих тестированию
  3. Этап уточнения основного плана, сформированного на этапе (1)
* Фаза получения набора тестов
  1. Этап разработки набора тестов
  2. Этап реализации уточненного плана
* Фаза измерений тестируемого модуля
  1. Этап выполнения тестовых процедур
  2. Этап определения достаточности тестирования
  3. Этап оценки результатов тестирования и тестируемого модуля.

Во время этапа планирования основных подходов в качестве входных данных используется общий план проекта (*модульное тестирование*, как часть проектных *работ*, должно укладываться в общий *график*) и требования к системе (для оценки трудоемкости *работ* и любого планирования необходимо проводить *анализ* сложности системы на основании требований к ней).

Основные задачи, решаемые в ходе этапа планирования, включают в себя:

* **определение общего подхода к тестированию модулей** - определяются риски и на их основе - степень полноты и охвата тестирования системы. Определяются источники входных и выходных данных. Определяются технологии проверки результатов тестирования и форматы записи данных о проведенном тестировании. Описывается внешний интерфейс тестируемых модулей и их информационное окружение;
* **определение требований к полноте тестирования** - определяется необходимая степень покрытия программного кода различных участков тестируемого модуля, определяется подходы к классам эквивалентности (требуется ли тестирование за границами диапазона);
* **определение требований к завершению тестирования** - определяются условия, проверка которых позволяет утверждать, что тестирование модуля завершено, и условия, при которых дальнейшее тестирование модуля считается невозможным до его изменения и доработки. Примером таких условий может служить достижение определенного уровня покрытия исходного кода тестами и невозможность компиляции модуля соответственно;
* **определение требований к ресурсам** - для разработки и выполнения тестов, а также для анализа результатов тестирования необходимы ресурсы - как технические (компьютеры и программное обеспечение), так и людские (тестировщики). При решении этой задачи необходимо указывать требования к программному и аппаратному обеспечению, требования к необходимой квалификации людей, а также должно определяться необходимое для проведения количество ресурсов и время их занятости;
* **определение общего плана-графика работ** - на основании общего плана проекта составляется план работ по модульному тестированию. Основной критерий начала работ по тестированию - готовность модулей, т.е. общий план работ по тестированию согласуется по датам начала работ с датами окончания работ общего плана разработки.

После завершения этапа планирования начинается этап определения свойств системы, подлежащих тестированию.

Основные задачи, которые решаются в ходе деятельности *по* определению свойств системы, подлежащих тестированию, включают в себя:

* **изучение функциональных требований** - определение тестопригодности требований, при необходимости запрашивается уточнение требований;
* **определение дополнительных требований и связанных процедур** - определение требований, которые не попадают под функциональные требования, но могут быть протестированы на уровне модульного тестирования (например, это могут быть требования к производительности системы, входящие в состав системных требований);
* **определение состояний тестируемого модуля** - если тестируемый модуль может быть представлен в виде конечного автомата с определенным набором состояний, то каждое состояние должно быть идентифицировано, а также должны быть выделены все требования, относящиеся к этому состоянию;
* **определение характеристик входных и выходных данных** - для всех данных, которые поступают в модуль, а также выходят из него, должны быть определены форматы, частота поступления, допустимые значения и т.п.;
* **выбор элементов, подвергаемых тестированию** - в случае, когда не может применяться полное тестирование, необходимо выбрать элементы тестируемого модуля, которые будут подвергаться тестированию. Основной источник информации здесь - данные о рисках, проанализированные на уровне структуры исходного кода тестируемого модуля. Для тестирования в первую очередь должны отбираться элементы с максимальной степенью риска.

И, наконец, в завершение фазы планирования производится уточнение основного плана - уточняется общий подход к тестированию, формулируются специальные и дополнительные требования к ресурсам, составляется детальный план-*график* *работ*.

*По* завершению этих этапов фаза планирования считается оконченной и начинается фаза разработки тестов. При этом процесс разработки тестов подчиняется тем планам и требованиям, которые были созданы на предыдущем этапе. Таким образом, если на первом этапе основную роль выполнял руководитель группы тестирования, то на втором этапе основную роль начинает играть *тестировщик*, действующий в согласии с указаниями руководителя.

Фаза разработки тестов начинается с собственно разработки набора тестов, который будет использован для тестирования модуля. Основные документы, которые используются на этом этапе: *функциональные требования* к модулю, *архитектура* модуля, *список* элементов, подвергаемых тестированию, план-*график* *работ*, определения тестовых примеров от предыдущей версии модуля (если они существовали) и результаты тестирования прошлой версии (если они существовали).

В ходе этого этапа должны быть решены следующие задачи:

* **разработка архитектуры тестового набора** - под тестовым набором здесь понимается не набор конкретных тестовых примеров, а общая структура системы тестов для проверки функциональности тестируемого модуля. Организация тестов в такой системе как правило отражает структуру функциональных требований и зачастую представляет собой иерархию, на каждом уровне которой определяется свой набор тестов;
* **разработка явных тестовых процедур (тест-требований)** - в случае достаточно подробных функциональных требований и четко прописанной концепции разработки тестов явные тестовые процедуры могут и не разрабатываться. Однако, при наличии необходимых ресурсов, разработка тест-требований позволит более четко интерпретировать подвергаемые тестированию функциональные требования - тест-требования снижают риск неоднозначной трактовки функциональных требований;
* **разработка тестовых примеров** - тестовые примеры должны соответствовать требованиям к полноте тестирования и составляться либо на базе тест-требований, либо на основании функциональных требований. Данный вид деятельности наиболее продолжителен во времени;
* **разработка тестовых примеров, основанных на архитектуре (в случае необходимости)** - некоторые тестовые примеры основываются не на функциональных требованиях, а на особенностях архитектуры тестируемого модуля. Для разработки тестовых примеров, основанных на архитектуре, необходимо использовать подход стеклянного ящика. Эти тестовые примеры пишутся либо на основе низкоуровневых требований к системе, либо на основе низкоуровневых тест-требований, если они разрабатывались на одном из предыдущих этапов;
* **составление спецификации тестовых примеров** - результатом деятельности тестировщика в ходе данного этапа составляется документ *Test Design* Specification (формат которого описан в стандарте IEEE 829 [[15](https://intuit.ru/studies/courses/1040/209/literature#literature.15)]).

На следующем этапе проводится реализация тестов (например, в виде тестового окружения и формализованных описаний тестовых примеров). В ходе этого этапа формируются тестовые наборы данных, которые используются в тестовых примерах, создается тестовое окружение, и также осуществляется *интеграция* тестового окружения с тестируемым модулем.

После того, как все тесты реализованы, они выполняются на тестовом стенде в ручном или автоматическом режиме. Вне зависимости от вида тестирования в ходе этого этапа решаются две задачи: выполнение тестовых примеров, и сбор и *анализ* результатов тестирования.

Сбору подлежит следующая *информация*:

* результат выполнения каждого тестового примера (прошел/не прошел);
* информация об информационном окружении системы в случае, если тест не прошел;
* информация о ресурсах, которые потребовались для выполнения тестового примера.

*По* результатам анализа этой информации составляются запросы на изменение проектной документации, программного кода тестируемого модуля или тестового окружения.

Этапы разработки (доработки), реализации и выполнения тестов продолжаются до тех пор, пока не будет достигнут критерий завершения модульного тестирования. Примером такого критерия может служить отсутствие не прошедших тестовых примеров при 75% покрытии строк исходного кода.

После прекращения тестирования выполняются работы *по* оценке проведенного тестирования, в ходе которых:

* описываются отличия реального процесса тестирования от запланированного;
* отличия поведения тестируемого модуля от описанного в требованиях (с целью дальнейшей коррекции требований);
* составляется общий отчет о прохождении тестов, включающий в себя и информацию о покрытии.

В завершение модульного тестирования необходимо проверить, что все созданные в его ходе артефакты - документы, программный код, файлы отчетов и данных - помещены в базу данных проекта, которая хранит все данные, используемые и создаваемые в процессе разработки программной системы.

***ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ И***

***ФОРМА ОТЧЕТНОСТИ:***

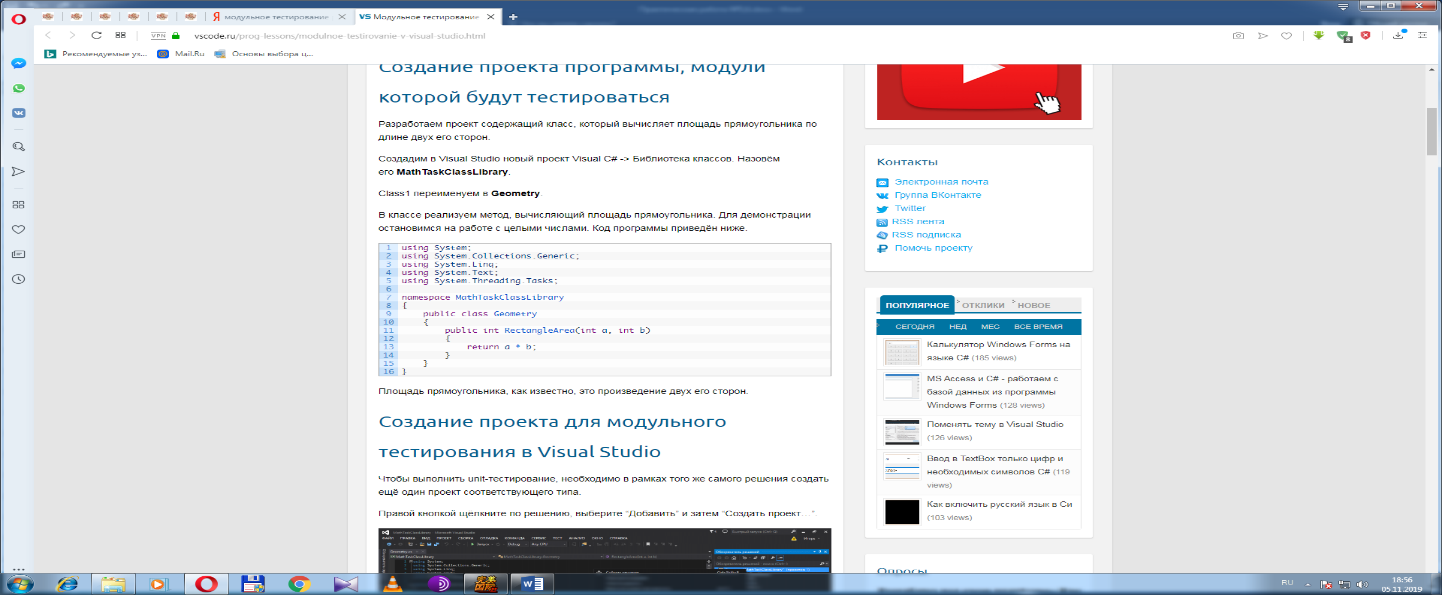
**Задание 1.** Создание проекта программы, модули которого будут тестироваться.

Разработаем проект содержащий класс, который вычисляет площадь прямоугольника по длине двух его сторон.

Создадим в Visual Studio новый проект Visual C# - Библиотека классов. Назовём его **MathTaskClassLibrary**.

Class1 переименуем в **Geometry**.

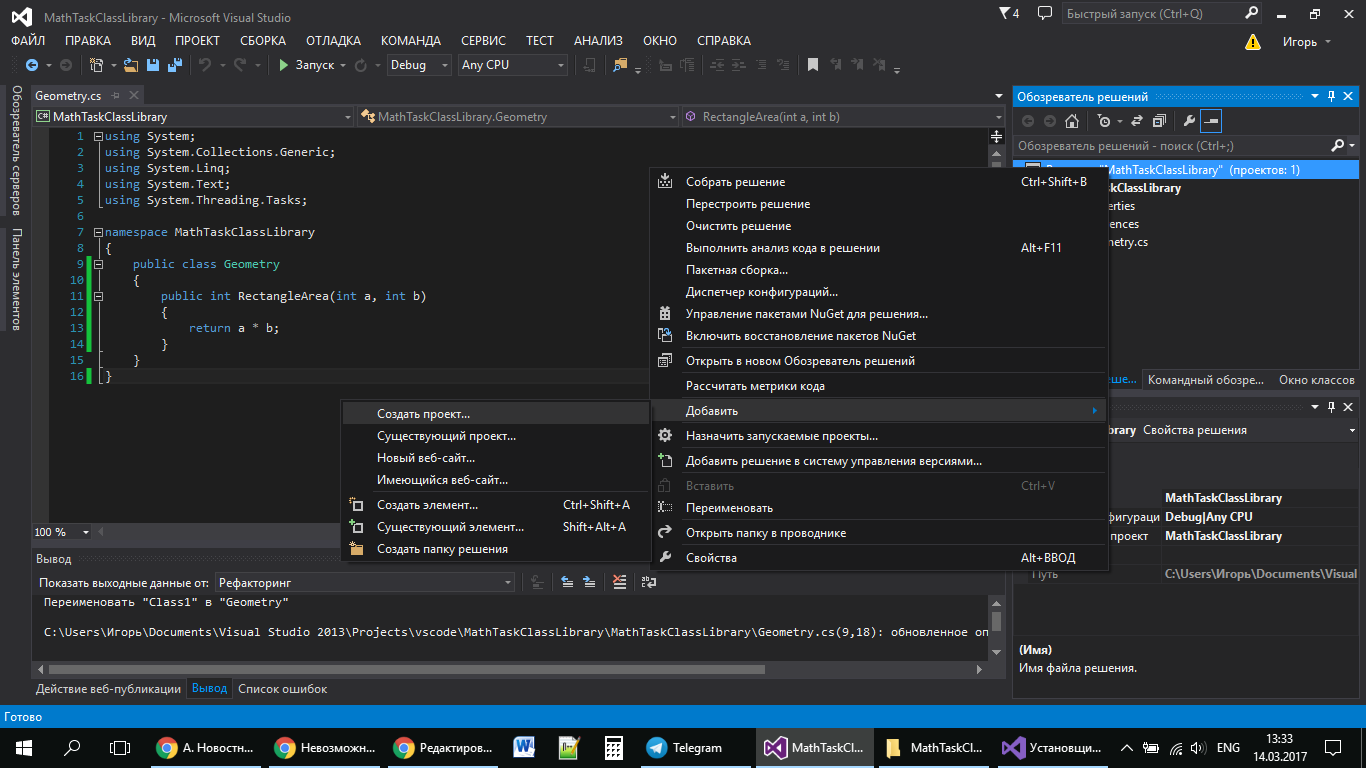
В классе реализуем метод, вычисляющий площадь прямоугольника. Для демонстрации остановимся на работе с целыми числами. Код программы приведён ниже.



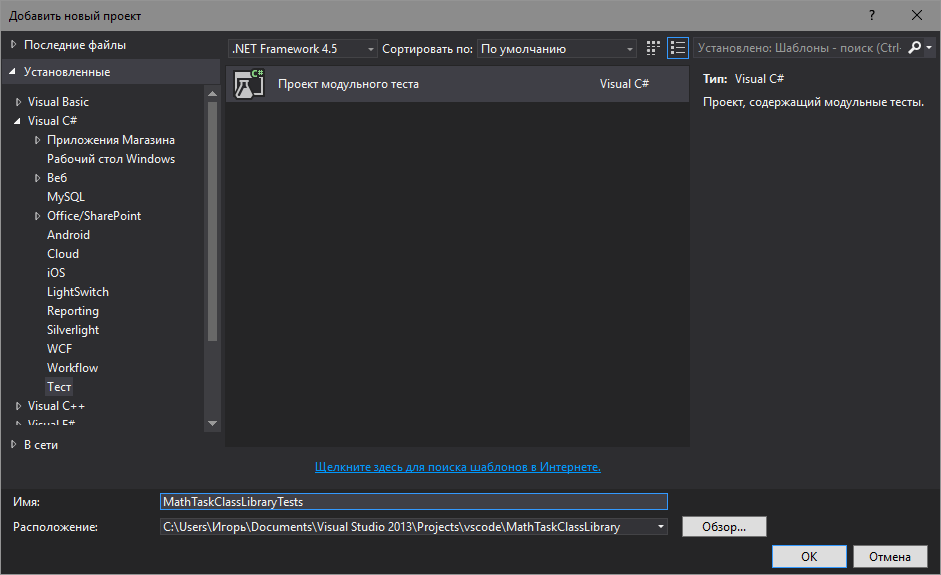
**Создание проекта для модульного тестирования в Visual Studio.**

Чтобы выполнить unit-тестирование, необходимо в рамках того же самого решения создать ещё один проект соответствующего типа.

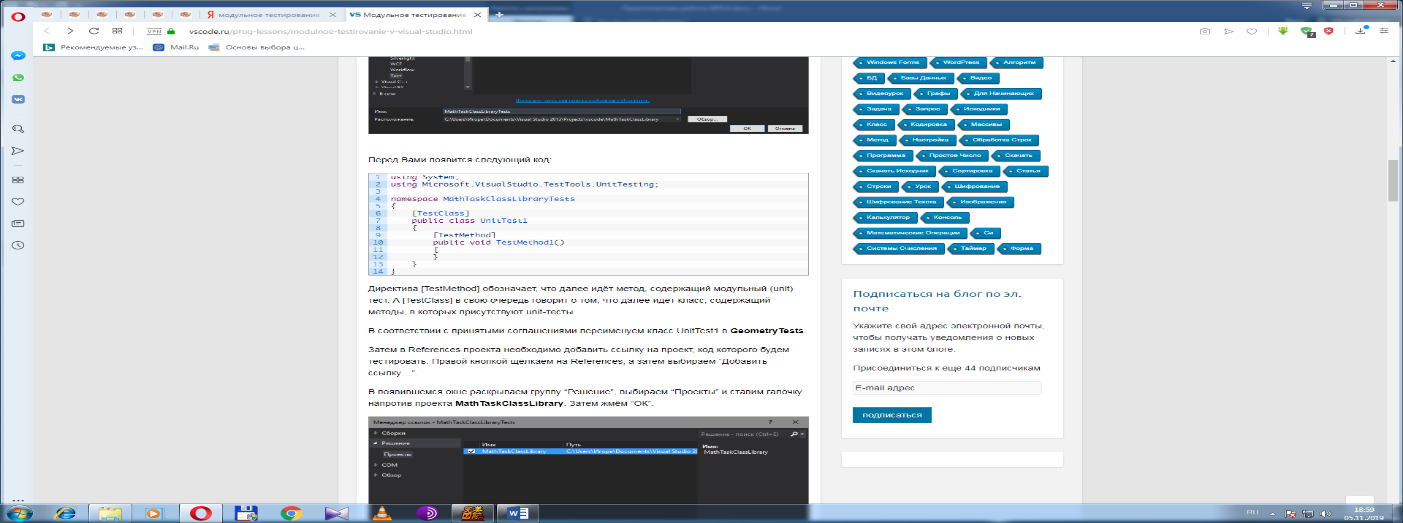
Правой кнопкой щёлкните по решению, выберите “Добавить” и затем “Создать проект…”.



В открывшемся окне в группе Visual C# щёлкните “Тест”, а затем выберите “Проект модульного теста”. Введите имя проекта **MathTaskClassLibraryTests**и нажмите “ОК”. Таким образом проект будет создан.



Перед Вами появится следующий код:

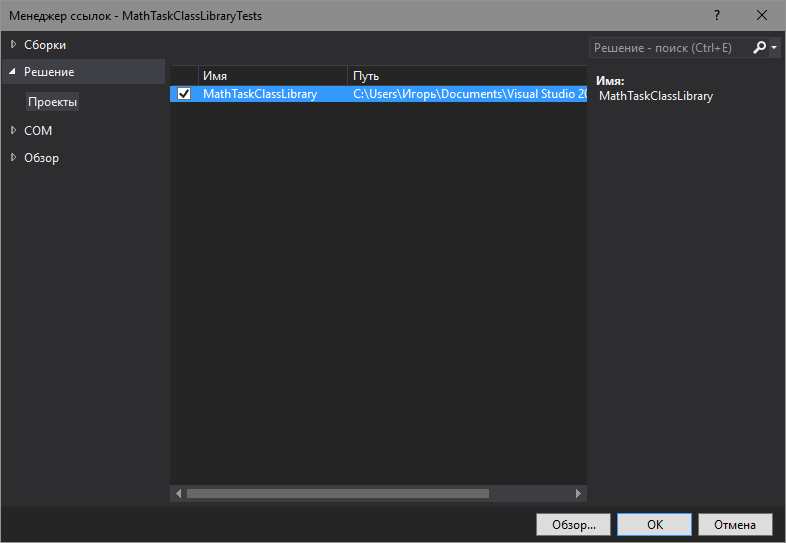


Директива [TestMethod] обозначает, что далее идёт метод, содержащий модульный (unit) тест. А [TestClass] в свою очередь говорит о том, что далее идёт класс, содержащий методы, в которых присутствуют unit-тесты.

В соответствии с принятыми соглашениями переименуем класс UnitTest1 в **GeometryTests**.

Затем в References проекта необходимо добавить ссылку на проект, код которого будем тестировать. Правой кнопкой щёлкаем на References, а затем выбираем “Добавить ссылку…”.

В появившемся окне раскрываем группу “Решение”, выбираем “Проекты” и ставим галочку напротив проекта **MathTaskClassLibrary**. Затем жмём “ОК”.



Также в коде необходимо подключить с помощью директивы using следующее пространство имён:  
using MathTaskClassLibrary;

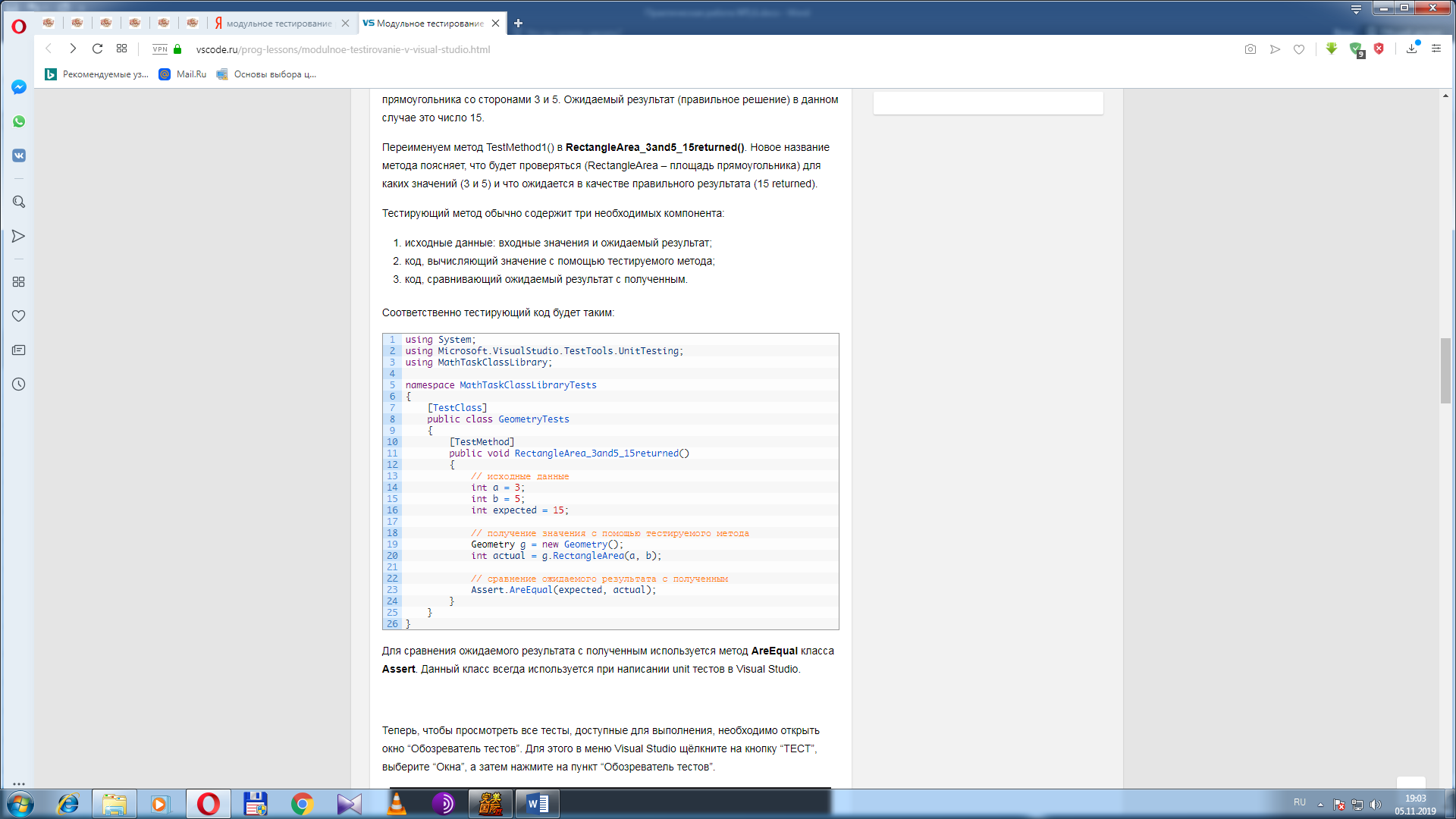
Займёмся написание теста. Проверим правильно ли вычисляет программа площадь прямоугольника со сторонами 3 и 5. Ожидаемый результат (правильное решение) в данном случае это число 15.

Переименуем метод TestMethod1() в **RectangleArea\_3and5\_15returned()**. Новое название метода поясняет, что будет проверяться (RectangleArea – площадь прямоугольника) для каких значений (3 и 5) и что ожидается в качестве правильного результата (15 returned).

Тестирующий метод обычно содержит три необходимых компонента:

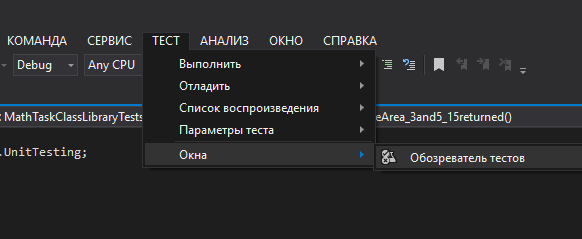
1. исходные данные: входные значения и ожидаемый результат;
2. код, вычисляющий значение с помощью тестируемого метода;
3. код, сравнивающий ожидаемый результат с полученным.

Соответственно тестирующий код будет таким:

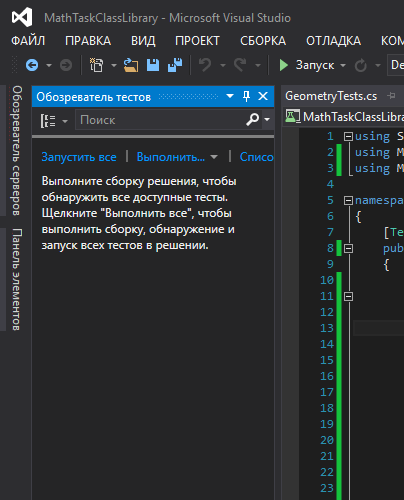


Для сравнения ожидаемого результата с полученным используется метод **AreEqual** класса **Assert**. Данный класс всегда используется при написании unit тестов в Visual Studio.

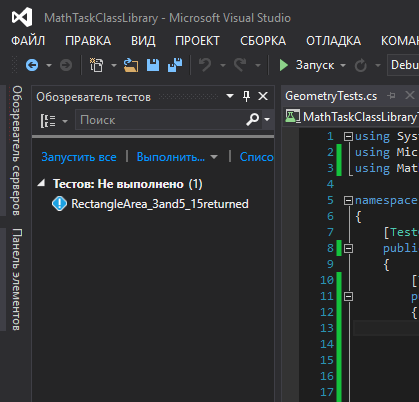
Теперь, чтобы просмотреть все тесты, доступные для выполнения, необходимо открыть окно “Обозреватель тестов”. Для этого в меню Visual Studio щёлкните на кнопку “ТЕСТ”, выберите “Окна”, а затем нажмите на пункт “Обозреватель тестов”.



В студии появится следующее окно:

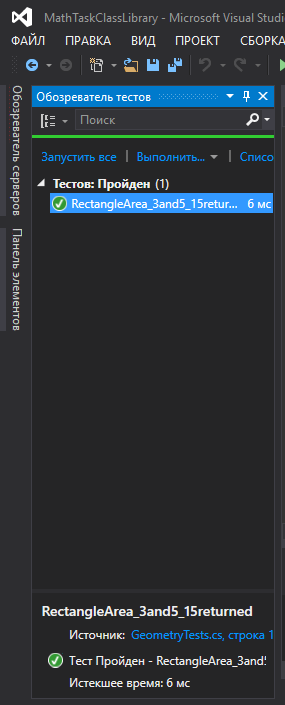


В данный момент список тестов пуст, поскольку решение ещё ни разу не было собрано. Выполним сборку нажатием клавиш Ctrl + Shift + B. После её завершения в “Обозревателе тестов” появится наш тест.



Синяя табличка с восклицательным знаком означает, что указанный тест никогда не выполнялся. Выполним его.

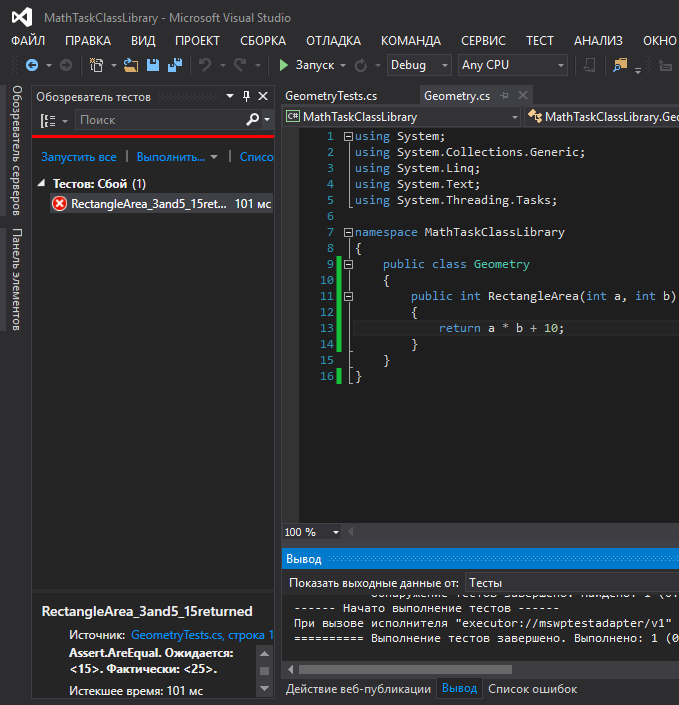
Для этого нажмём правой кнопкой мыши на его имени и выберем “Выполнить выбранные тесты”.



Зелёный кружок с галочкой означает, что модульный тест успешно пройден: ожидаемый и полученный результаты равны.

Изменим код метода **RectangleArea**, вычисляющего площадь прямоугольника, чтобы сымитировать провал теста и посмотреть, как поведёт себя Visual Studio. Прибавим к возвращаемому значению 10.

Запустим unit-тест.



Как Вы видите, красный круг с крестиком показывает провал модульного теста, а ниже указано, что при проверке ожидалось значение 15, а по факту оно равно 25.

**Задание 2**. Разработать программу на C# для подсчета объема цилиндра и создать модульный тест.

**Задание 3**. Разработать программу на C# для подсчета процента от числа и создать модульный тест.