УЧЕБНОЕ ПОСОБИЕ
МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Новосибирск

2020

Учебное пособие предназначено для студентов обучающихся по дисциплине «Современные технологии программирования 1», для основной профессиональной образовательной программы по направлению 09.03.01 Информатика и вычислительная техника, направленность (профиль) – Программное обеспечение средств вычислительной техники и автоматизированных систем, квалификация – бакалавр, программа академического бакалавриата, форма обучения – очная, заочная. Материал учебного пособия охватывает часть материала изучаемого по дисциплине. Излагаются вопросы, касающиеся модульного тестирования и средств модульного тестирования Visual Studio, а также приведены указания к выполнению и варианты выполнения лабораторных работ. Примеры упражнений выполнены в среде Visual Studio на языках C++, C#.

Составитель: канд. физ.-мат. наук, доц. *М. Г. Зайцев*

СибГУТИ, 2020 г.

Оглавление

1. Тестирование ПО

1.1. Тестирование - способ обеспечения качества

Качество программного продукта характеризуется набором свойств, определяющих, насколько продукт "хорош" с точки зрения заинтересованных сторон, таких как

- заказчик продукта,
- спонсор,
- конечный пользователь,
- разработчики и тестировщики продукта,
- инженеры поддержки,
- сотрудники отделов маркетинга, обучения и продаж.

Каждый из участников может иметь различное представление о продукте и о том, насколько он хорош или плох, то есть о том, насколько высоко качество продукта. Таким образом, постановка задачи обеспечения качества продукта выливается в задачу определения заинтересованных лиц, их критериев качества и затем нахождения оптимального решения, удовлетворяющего этим критериям.

Тестирование является одним из наиболее устоявшихся способов обеспечения качества разработки программного обеспечения и входит в набор эффективных средств современной системы обеспечения качества программного продукта.

С технической точки зрения тестирование заключается в выполнении приложения на некотором множестве исходных данных и сверке получаемых результатов с заранее известными (эталонными) с целью установить соответствие различных свойств и характеристик приложения заказанным свойствам.

Тестирование программного обеспечения (Software Testing) — проверка соответствия реальных и ожидаемых результатов поведения программы, проводимая на конечном наборе тестов, выбранном определённым образом.

Цель тестирования — проверка соответствия ПО предъявляемым требованиям, обеспечение уверенности в качестве ПО, поиск очевидных ошибок в программном обеспечении, которые должны быть выявлены до того, как их обнаружат пользователи программы.

Для чего проводится тестирование ПО?

- Для проверки соответствия требованиям.
- Для обнаружение проблем на более ранних этапах разработки и предотвращение повышения стоимости продукта.
- Обнаружение вариантов использования, которые не были предусмотрены при разработке. А также взгляд на продукт со стороны пользователя.
- Повышение лояльности к компании и продукту, т.к. любой обнаруженный дефект негативно влияет на доверие пользователей.

Принципы тестирования

- **Принцип 1 — Тестирование демонстрирует наличие дефектов (Testing shows presence of defects)**.Тестирование только снижает вероятность наличия дефектов, которые находятся в программном обеспечении, но не гарантирует их отсутствия.
- **Принцип 2 — Исчерпывающее тестирование невозможно (Exhaustive testing is impossible)**. Полное тестирование с использованием всех входных комбинаций данных, результатов и предусловий физически невыполнимо (исключение — тривиальные случаи).
- **Принцип 3 — Раннее тестирование (Early testing)**. Следует начинать тестирование на ранних стадиях жизненного цикла разработки ПО, чтобы найти дефекты как можно раньше.
- **Принцип 4 — Скопление дефектов (Defects clustering)**. Большая часть дефектов находится в ограниченном количестве модулей.
- **Принцип 5 — Парадокс пестицида (Pesticide paradox)**. Если повторять те же тестовые сценарии снова и снова, в какой-то момент этот набор тестов перестанет выявлять новые дефекты.
- **Принцип 6 — Тестирование зависит от контекста (Testing is context depending)**. Тестирование проводится по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем новостной портал.
- **Принцип 7 — Заблуждение об отсутствии ошибок (Absence-of-errors fallacy)**. Отсутствие найденных дефектов при тестировании не всегда означает готовность продукта к выпуску. Система должна быть удобна пользователю в использовании и удовлетворять его ожиданиям и потребностям.

Обеспечение качества (QA — Quality Assurance) и **контроль качества (QC — Quality Control)** — эти термины похожи на взаимозаменяемые, но разница между обеспечением качества и контролем качества все-таки есть, хоть на практике процессы и имеют некоторую схожесть.

QC (Quality Control) — Контроль качества продукта — анализ результатов тестирования и качества новых версий выпускаемого продукта.

К задачам контроля качества относятся:

- проверка готовности ПО к выпуску;
- проверка соответствия требований и качества данного проекта.

QA (Quality Assurance) — Обеспечение качества продукта — изучение возможностей по изменению и улучшению процесса разработки, улучшению коммуникаций в команде, где тестирование является только одним из аспектов обеспечения качества.

1.2. Место тестирования в программной инженерии

Руководство к Своду Знаний по Программной Инженерии — «SWEBOK V3», которая стала стандартом ISO/IEC TR 19759:2015, было издано в 2013 году и включает базовые определения и описания областей знаний, которые составляют суть профессии инженера-программиста.

Описание областей знаний в SWEBOK построено по иерархическому принципу, как результат структурной декомпозиции, и включает 15 областей знаний в сфере программной инженерии:

- software requirements — требования к ПО;
- software design — проектирование ПО;
- software construction — конструирование ПО;
- software testing — тестирование ПО;
- software maintenance — сопровождение ПО;
- software configuration management — управление конфигурацией;
- software engineering management — управление IT проектом;
- software engineering process — процесс программной инженерии;
- software engineering models and methods — модели и методы разработки;
- software quality — качество ПО;
- software engineering professional practice — описание критериев профессионализма и компетентности;
- software engineering economics — экономические аспекты разработки ПО;
- computing foundations — основы вычислительных технологий, применимых в разработке ПО;
- mathematical foundations — базовые математические концепции и понятия, применимые в разработке ПО;
- engineering foundations — основы инженерной деятельности.

Тестирование выделено в отдельную область знаний в сфере программной инженерии в виду его важности для обеспечения качества ПО.

1.3. Тестирование программного обеспечения

Тестирование ПО— это процесс проверки работоспособности, основанный на использовании конечного набора тестовых данных, сформированных на базе требований к ПО и сравнения полученных результатов с целевыми показателями качества, заложенными в проекте (рис. 1.1).

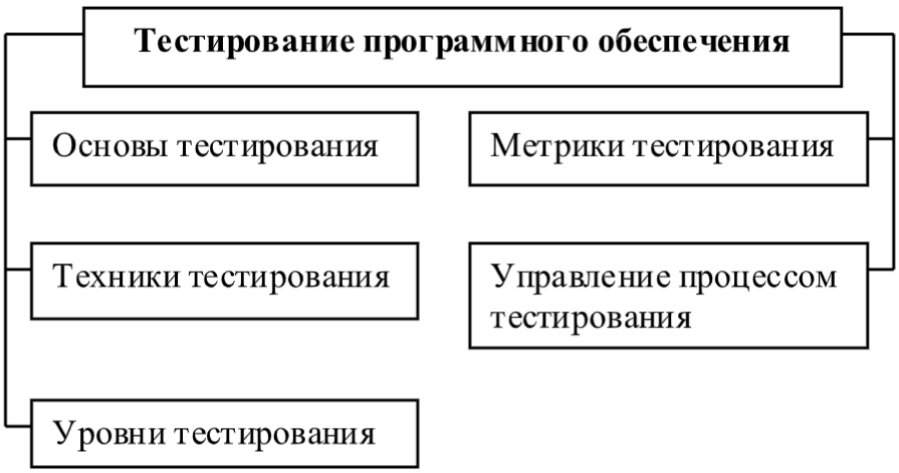


Рис. 1.1 – Область знаний «Тестирование ПО»

1.3.1. Основы тестирования

В соответствии с принятой терминологией при тестировании выявляются следующие недостатки: отказы и дефекты, сбои, ошибки. Важно четко разделять причину нарушения работы прикладных программ, обычно описываемую терминами «недостаток» или «дефект», и наблюдаемый нежелательный эффект, вызываемый этими причинами — сбой. Термин «ошибка», в зависимости от контекста, может описывать и причину сбоя, и сам сбой. Тестирование позволяет обнаружить дефекты, приводящие к сбоям.

Базовым понятием тестирования является также тест, который выполняется в заданных условиях и на проверочных наборах данных. Степень тестируемости зависит от задания критериев покрытия системы тестами и вероятности появления сбоев.

1.3.2. Уровень тестирования

Уровень тестирования определяет, «над чем» производятся тесты: над отдельным модулем, группой модулей или системой в целом. При этом ни один из уровней тестирования не может считаться приоритетным. Выделяют следующие уровни тестирования:

- модульное тестирование, предполагающее проверку отдельных, изолированных и независимых элементов (компонентов) ПО;
- интеграционное тестирование, которое ориентировано на проверку связей и способов взаимодействия (интерфейсов) компонентов друг с другом;
- тестирование программного обеспечения, предназначенное для проверки правильности функционирования в целом, с обнаружением отказов и дефектов, и их устранения. При этом контролируется и выполнение нефункциональных требований (безопасность, надежность и т. д.), а так же правильность задания и выполнения внешних интерфейсов со средой окружения.

Тестирование проводится в соответствии с определенными целями проверки качества и работоспособности ПО. Выделяют следующие, наиболее распространенные цели (и, соответственно виды) тестирования:

- функциональное тестирование, которое заключается в проверке соответствия выполнения функциональных возможностей ПО;
- регрессионное тестирование— тестирование программного обеспечения или его компонентов после внесения в них изменений;
- тестирование эффективности функционирования ПО — проверка в соответствии со спецификациями требований производительности, пропускной способности, максимального объема данных, системных ограничений и т. д.;
- нагрузочное (стресс) тестирование— проверка поведения ПО при максимально допустимой нагрузке;
- внутреннее и внешнее тестирование— альфа (тестирование продукта силами штатных разработчиков или тестировщиков) и бета (тестирование продукта с помощью добровольцев из числа обычных будущих пользователей продукта, которым доступна предварительная версия продукта (так называемая бета-версия).);
- тестирование конфигурации или конфигурационное **тестирование** (Configuration Testing) — — специальный вид **тестирования**, направленный на проверку работы программного обеспечения при различных **конфигурациях** системы (заявленных платформах, поддерживаемых драйверах, при различных **конфигурациях** компьютеров и т.д.);
- приемочное тестирование— проверка в соответствии с заранее подготовленной программой и методикой испытаний поведения системы на этапе приемки сдачи ее заказчику. Приемочное тестирование – это комплексное тестирование, необходимое для определения уровня готовности системы к последующей эксплуатации. Тестирование проводится на основании набора тестовых сценариев, покрывающих основные бизнес-операции системы. Как правило, данный вид тестирования реализуется конечными пользователями системы, однако привлечение опытных тестировщиков сократит время на подготовку к тестированию и позволит повысить качество и надежность проводимых испытаний.

1.3.3. Техники тестирования

Техники тестирования можно условно разбить на следующие группы:

Техники, базирующиеся на интуиции и опыте инженера, рассматривающего проблему с точки зрения имевшихся ранее аналогий.

Техники, базирующиеся на блок-схеме ПО, строятся исходя из покрытия всех условий и решений блок-схемы. Максимальная отдача от тестов на основе блок-схемы получается, когда тесты покрывают различные пути блок-схемы. Адекватность таких тестов оценивается как процент покрытия всех возможных путей блок-схемы.

Тестирование, ориентированное на дефекты, направлено на обнаружение наиболее вероятных ошибок, предсказываемых заранее, например, в результате анализа возможных рисков программного проекта.

Техники, базирующиеся на условиях использования, позволяют оценить поведение системы в реальных условиях. Входные параметры тестов задаются на основе вероятностного распределения соответствующих параметров или их формирования в процессе эксплуатации.

Техники, базирующиеся на природе приложения, находят применение в зависимости от технологической или архитектурной природы приложений, среди таких техник можно выделить:

- Объектно-ориентированное тестирование.
- Компонентно-ориентированное тестирование.
- Web-ориентированное тестирование.
- Тестирование на соответствие протоколам.
- Тестирование систем реального времени.

1.3.4. Метрики тестирования

Метрики тестирования предназначены для измерения процессов и результатов планирования, проектирования и тестирования ПО.

Ме́трика програ́ммного обеспе́чения (англ. software metric) — мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций. Поскольку количественные методы хорошо зарекомендовали себя в других областях, многие теоретики и практики информатики пытались перенести данный подход и в разработку программного обеспечения.

Измерение результатов тестирования касается оценки качества получаемого продукта. Оценка программ в процессе тестирования должна базироваться на размере программ (например, в терминах количества строк кода или функциональных точек) или их структуре (например, с точки зрения оценки ее сложности в тех или иных архитектурных терминах). Структурные измерения могут также включать частоту обращений одних модулей программы к другим.

Эффективность тестирования может быть измерена путем определения, какие типы дефектов найдены в процессе тестирования ПО, и как изменяется их частота во времени. Эта информация позволяет прогнозировать качество ПО и совершенствовать процесс разработки в целом.

Тестируемая программа может оцениваться также на основе подсчета и классификации найденных дефектов. Для каждого класса дефектов можно определить отношение между количеством соответствующих дефектов и размером программы.

В ряде случаев процесс тестирования, требует систематического выполнения тестов для определенного набора элементов программы, задаваемых ее архитектурой или спецификацией. Соответствующие метрики позволяют оценить степень охвата характеристик системы и глубину их детализации. Такие метрики помогают прогнозировать вероятностное достижение заданных параметров качества системы.

Документация на тестирование включает описание тестовых документов, их связи между собой и с процессом тестирования. Без документации на процессы тестирования невозможно провести сертификацию и оценку зрелости программного продукта. После завершения тестирования рассматриваются вопросы стоимости и рисков, связанных с появлением сбоев и недостаточно надежной работой системы. Стоимость тестирования является одним из ограничений, на основе которого принимается решение о прекращении или продолжении тестирования.

1.3.5. Управление тестированием

Концепции, стратегии, техники и измерения тестирования должны быть объединены в единый процесс тестирования (от планирования тестов до оценки их результатов) как деятельности по обеспечению качества ПО, поддерживающей «правила игры» для членов команды тестирования. Только в том случае, если тестирование рассматривать как один из важных процессов всей деятельности по созданию и поддержке программного обеспечения, можно добиться оценки стоимости соответствующих работ и, в конце концов, выполнить те ограничения, которые определены для проекта. Работы по управлению процессом тестирования, ведущиеся на разных уровнях, должны быть организованы в единый процесс, на основе учета четырех элементов и связанных с ними факторов: участников процесса (в том числе, в контексте организационной структуры и культуры), инструментов, регламентов и количественных оценок измерения.

В состав этого процесса должны входить:

а) планирование работ по тестированию (составление планов, тестов, наборов данных) и измерению показателей качества ПО;

б) генерация необходимых тестовых сценариев, соответствующих среде выполнения ПО:

в) проведение тестирования с учетом следующих положений:

- все работы и результаты процесса тестирования должны обязательно фиксироваться;
- форма журналирования работ и их результатов должна быть такой, чтобы соответствующее содержание было понятно, однозначно интерпретируемо и повторяемо другими лицами;
- тестирование должно проводиться в соответствии с заданными и документированными процедурами;
- тестирование должно производиться над однозначно идентифицируемой версией и конфигурацией ПО;
- должен осуществляться сбор данных об отказах, ошибках и др. непредвиденных ситуациях при выполнении программного продукта;
- подготовка отчетов - должны составляться отчеты по результатам тестирования и оценкам характеристик ПО.

Контрольные вопросы

1. Что такое тестирование?

2. Что такое «дефект»?
3. Что такое «сбой»?
4. Что определяет уровень тестирования?
5. Перечислите группы, на которые можно условно разбить техники тестирования?
6. Для чего необходимы метрики тестирования?

2. Основные понятия тестирования

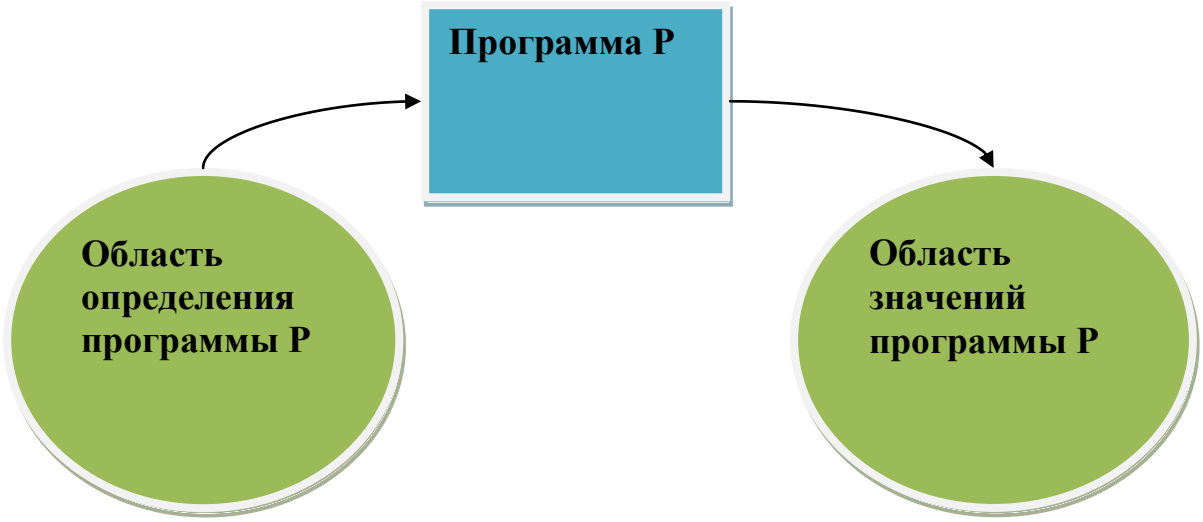
2.1. Концепция тестирования

Программа – это аналог формулы в математике.

Формула для функции f, полученная суперпозицией функций f_1, f_2, ... f_n – выражение, описывающее эту суперпозицию.

$f = f_1 * f_2 * f_3 *... * f_n$

Если аналог f_1, f_2,... f_n – операторы языка программирования, то их формула – программа.



Здесь:

[P] = {(S_X,S_Y)} – программная функция программы P;

S_X – исходное состояние данных программы;

S_Y – результирующее состояние данных программы.

Состояние данных программы S– множество упорядоченных пар {(переменная$_1$ – значение), переменная$_2$ – значение),…(переменная$_n$ – значение)}.

Программа отображает исходное состояние данных S_X в результирующее состояние S_Y путём последовательного выполнения операторов программы, каждый из которых может изменять текущее состояние данных программы. Принципиально возможно спецификацию программы задать в виде формулы. А после того как программа написана, получить формулу программной функции по тексту программы. Тогда задача проверки правильности программы сведётся к доказательству эквивалентности этих двух формул.

Существует два метода обоснования истинности формул:

1. Формальный подход или доказательство применяется, когда из исходных формул-аксиом с помощью формальных процедур (правил вывода) выводятся искомые формулы и утверждения (теоремы). Вывод осуществляется путем перехода от одних формул к другим по строгим правилам, которые позволяют свести процедуру перехода от формулы к формуле к последовательности текстовых подстановок:

A**3 = A*A*A

A*A*A = A -> R, A*R -> R, A*R -> R

Преимущество формального подхода заключается в том, что с его помощью удается избегать обращений к бесконечной области значений и на каждом шаге доказательства оперировать только конечным множеством символов.

2. Интерпретационный подход применяется, когда осуществляется подстановка констант в формулы, а затем интерпретация формул как осмысленных утверждений в элементах множеств конкретных значений. Истинность интерпретируемых формул проверяется на конечных множествах возможных значений. Сложность подхода состоит в том, что на конечных множествах входных данных число комбинаций их возможных значений для реализации исчерпывающей проверки могут оказаться достаточно большим.

Интерпретационный подход используется при экспериментальной проверке соответствия программы своей спецификации.

Применение интерпретационного подхода в форме экспериментов над исполняемой программой составляет суть отладки и тестирования.

2.2. Основная терминология

Отладка (debug, debugging) – процесс локализации и исправления ошибок в программе.

Термин "отладка" в отечественной литературе используется двояко: для обозначения активности по поиску ошибок (собственно тестирование), по нахождению причин их появления и исправлению, или активности по локализации и исправлению ошибок.

Тестирование обеспечивает выявление (констатацию наличия) фактов расхождений с требованиями (ошибок).

Как правило, на фазе тестирования осуществляется и исправление идентифицированных ошибок, включающее локализацию ошибок, нахождение причин ошибок и соответствующую корректировку программы тестируемого приложения (Application Under Testing (AUT) или Implementation Under Testing (IUT)).

Если программа не содержит синтаксических ошибок (прошла трансляцию) и может быть выполнена на компьютере, она обязательно вычисляет какую-либо функцию, осуществляющую отображение входных данных в выходные. Это означает, что компьютер на своих ресурсах доопределяет частично определенную программой функцию до тотальной определенности. Следовательно, судить о правильности или неправильности результатов выполнения программы можно,

только сравнивая спецификацию желаемой функции с результатами ее вычисления, что и осуществляется в процессе тестирования.

Тестирование разделяют на статическое и динамическое:

- Статическое тестирование выявляет формальными методами анализа неверные конструкции или неверные отношения объектов программы (ошибки формального задания) с помощью специальных инструментов контроля кода – CodeChecker без выполнения тестируемой программы. Также к статическому тестированию относят тестирование требований, спецификаций, документации.
- Динамическое тестирование (собственно тестирование) осуществляет выявление ошибок только на выполняющейся программе с помощью специальных инструментов автоматизации тестирования – Testbed или Testbench.

2.2. Организация тестирования

Тестирование осуществляется на заданном заранее множестве входных данных X и множестве предполагаемых результатов Y – (X,Y), которые задают график желаемой функции. Кроме того, зафиксирована процедура Оракул (oracle), которая определяет, соответствуют ли выходные данные – Y_B (вычисленные по входным данным – X) желаемым результатам – Y, т.е. принадлежит ли каждая вычисленная тестируемой программой точка (X,Y_B) графику желаемой функции (X,Y).

Оракул дает заключение о факте появления неправильной пары (X,Yв) и ничего не говорит о том, каким образом она была вычислена или каков правильный алгоритм – он только сравнивает вычисленные и желаемые результаты. Оракулом может быть даже заказчик или программист, производящий соответствующие вычисления в уме, поскольку Оракулу нужен какой-либо альтернативный способ получения функции (X,Y) для вычисления эталонных значений Y.

Пример сравнения словесного описания пункта спецификации требований с результатом выполнения фрагмента кода.

Пункт спецификации требований к методу Convert: метод должен принимать входные параметры: x – неотрицательное целое число, и b – основание позиционной системы счисления, в которую преобразуется число x. Метод должен возвращать вычисленное значение – строку, содержащую представление числа x в позиционной системе счисления с основанием b.

Выполняем метод со следующими параметрами: Convert(161, 16).

Проверка результата выполнения возможна, когда результат вычисления заранее известен – "A1". Если результат преобразования числа 161 в систему счисления с основанием 16– строка "A1", то он соответствует спецификации требований.

В процессе тестирования Оракул последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений (X,Y_B) для идентификации фактов несовпадений (test incident).

При выявлении несовпадений (X,Y_B) и (X,Y) запускается процедура исправления ошибки, которая заключается во внимательном анализе (просмотре) протокола промежуточных вычислений, приведших к (X,Y_B), с помощью следующих методов:

1. "Выполнение программы в уме" (deskchecking).
2. Вставка операторов протоколирования (печати) промежуточных результатов (logging).
3. Пошаговое выполнение программы (single-step running).

Тестирование заканчивается, когда выполнилось или "прошло" (pass) успешно достаточное количество тестов в соответствии с выбранным критерием тестирования.

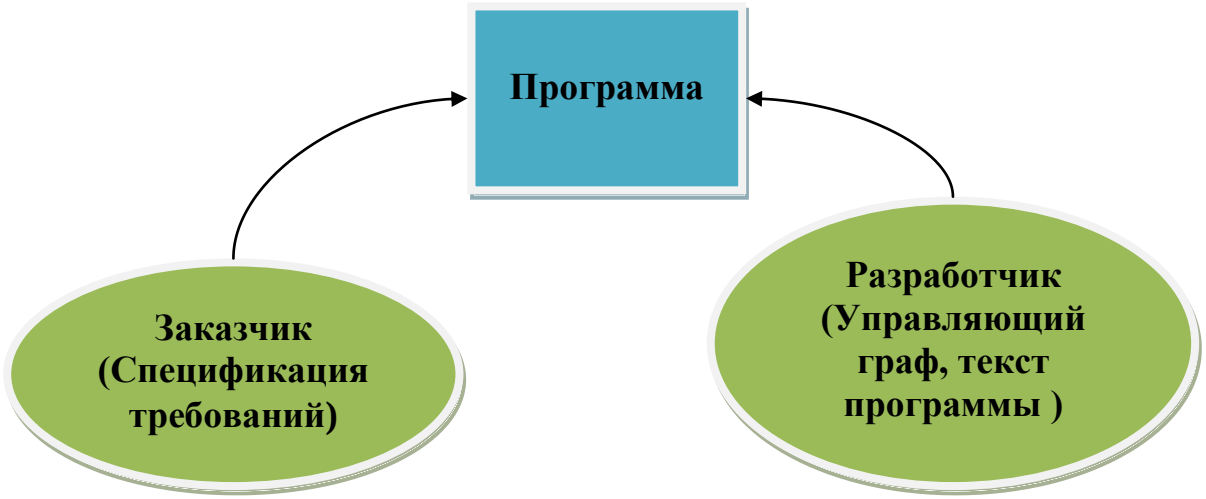
Тестирование – это:

1. Процесс выполнения ПО системы или компонента в условиях анализа или записи получаемых результатов с целью проверки (оценки) некоторых свойств тестируемого объекта.
2. Процесс анализа пункта требований к ПО с целью фиксации различий между существующим состоянием ПО и требуемым (что свидетельствует о проявлении ошибки) при экспериментальной проверке соответствующего пункта требований.
3. Контролируемое выполнение программы на конечном множестве тестовых данных и анализ результатов этого выполнения для поиска ошибок.

Спецификация программы

Требования к программному обеспечению — совокупность утверждений относительно атрибутов, свойств или качеств программной системы, подлежащей реализации. Создаются в процессе разработки требований к программному обеспечению (ПО), в результате анализа требований.

Спецификация требований программного обеспечения — структурированный набор требований (функциональность, производительность, конструктивные ограничения и атрибуты) к программному обеспечению и его внешним интерфейсам. Спецификация требований предназначена для того, чтобы установить основу для соглашения между заказчиком и разработчиком (или подрядчиками) о том, как должен функционировать программный продукт.



Имеются различия во взгляде на программу со стороны заказчика и разработчика. Для заказчика программа представлена спецификацией требований, для разработчика, помимо этого, она представлена управляющим графом, блок-схемой или текстом на заданном языке программирования (реализацией спецификации требований).

Для нашего примера спецификация требований к программе имеет следующий вид.

На вход программа принимает два целочисленных числовых параметра: x – исходное десятичное целое число, b – основание позиционной системы счисления. Программа преобразует десятичное целое число x в систему счисления с основанием b. Результат вычисления выводится на консоль.

Значения числа и основания системы счисления должны быть целыми.

Значения числа x, подлежащего преобразованию, должны лежать в диапазоне – [0..255].

Значения основания b должны лежать в диапазоне – [2..16].

Если числа, подаваемые на вход, лежат за пределами указанных диапазонов, то в программе необходимо предусмотреть вывод сообщения об ошибке.

Сквозной пример тестирования

Возьмем следующую программу:

```
class Program
{
    /*  Метод переводит целое неотрицательное x
       в систему счисления с основание b */
    static public string convert (int x, int b)
    {
        int d;
        char ch;//Текущий разряд в системе счисления с основанием b.
        string s = "";//Строка результата.
        while(x > 0)
        {
            d = x % b;
```

```
                if (d <= 9)
                     ch = (char)(d + '0');
                else ch = (char)(d - 10 + 'A');
                s = ch.ToString() + s;
                x = x / b;
            }
            if (s == "")
                s = "0";
            return s;
        }
        static void Main(string[] args)
        {
            int x;//Преобразуемое число.
            int b;//Основание системы счисления.
            try
            {
                Console.WriteLine("Введите x:");
                x = Convert.ToInt32(Console.ReadLine());
                if ((x >= 0) & (x <= 255))
                {
                    Console.WriteLine("Введите b:");
                    b = Convert.ToInt32(Console.ReadLine());
                    if ((b >= 2) & (b <= 16))
                    {
                        Console.WriteLine("{0} - ичное представление {1} это {2}",
b, x, convert(x, b));
                        Console.ReadLine();
                    }
                    else
                    {
                        Console.WriteLine("Ошибка: значение b должно лежать от
[2..16]");
                        Console.ReadLine();
                    }
                }
                else
                {
                    Console.WriteLine("Ошибка: значение x должно лежать от
[0..255]");
                    Console.ReadLine();
                }
            }
            catch (Exception)
            {
                Console.WriteLine("Ошибка: пожалуйста введите десятичное целое
число.");
                Console.ReadLine();
            }
        }

    }
}
```

Пример 2.1. Преобразования десятичного представления целого числа в систему счисления с заданным основанием на C#

```
#include "stdafx.h"
```

```
#include <string>
#include <iostream>
#include <windows.h>

using namespace std;

/*  Метод переводит целое неотрицательное x
в систему счисления с основание b */
string convert(int x, int b)
{
    int d;
    char ch;//Текущий разряд в системе счисления с основанием b.
    string s = "";//Строка результата.
    while (x > 0)
    {
          d = x % b;
          if (d <= 9)
                ch = (char)(d + '0');
          else ch = (char)(d - 10 + 'A');
          s = ch + s;
          x = x / b;
    }
    if (s == "")
          s = "0";
    return s;
}

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int x;//Преобразуемое число.
    int b;//Основание системы счисления.

    cout << "Введите x: ";
    if (scanf_s("%d", &x))
          if ((x >= 0) & (x <= 255))
          {
                cout << "Введите b: ";
                if (scanf_s("%d", &b))
                      if ((b >= 2) & (b <= 16))
                      {
                            cout << b << " - ичное представление " << x << " это "
<< convert(x, b);
                            cout << endl;;
                      }
                      else
                      {
                            cout << "Ошибка : значение b должно лежать от [2..16]";
                            cout << endl;
                      }
                else
                {
                      cout << "Ошибка : пожалуйста введите десятичное целое
число.";
```

```
                        cout << endl;
                }
        }
        else
        {
                cout << "Ошибка : значение x  должно лежать от  [0..255]";
                cout << endl;
        }
    else
    {
        cout << "Ошибка : пожалуйста введите десятичное целое число.";
        cout << endl;
    }
    system("pause");
}
```

2.1.1. Преобразования десятичного представления целого числа в систему счисления с заданным основанием на C++

Для приведенной программы, осуществляющей преобразования десятичного представления целого числа в систему счисления с заданным основанием (Пример 2.1), воспроизведем последовательность действий, необходимых для тестирования.

2.5. Разработка тестов на основе классов эквивалентности входных параметров

Для разработки тестового набора воспользуемся одним из частных видов функциональных критериев тестирования (класс II) - Тестирование классов входных данных (см. раздел 3.2.2). Для разработки тестового набора в случае выбора этого критерия тестирования достаточно наличия спецификации на тестируемую программу.

Разработка тестового набора начинается с анализа области определения программы и определения областей эквивалентности входных параметров.

Для x – числа, подлежащего преобразованию, определим классы возможных значений:

1. x < 0 (ошибочное)
2. x > 255 (ошибочное)
3. x - не число (ошибочное)
4. 0 <= x <= 255 (корректное)

Для b – основания системы счисления:

5. b < 2 (ошибочное)
6. b > 16 (ошибочное)
7. b - не число (ошибочное)
8. 2 <= b <= 16 (корректное)

Анализ тестовых случаев

1. Входные значения: (x = 161, b = 16) (покрывают классы 4, 8).

Ожидаемый результат: (16 - ичное представление 161 это A1.

2. Входные значения: {(x = -1, b = 2), (x = 256, b = 5)} (покрывают классы 1, 2).

Ожидаемый результат: Ошибка : значение x должно лежать от [0..255].

3. Входные значения: {(x = 100, b = 0), (x = 100, b = 200)} (покрывают классы 5,6).

Ожидаемый результат: Ошибка : значение b должно лежать от [2..16].

4. Входные значения: (x = ADS, n = ASD) (покрывают классы эквивалентности 3, 7).

Ожидаемый результат: Ошибка: пожалуйста, введите десятичное целое число.

5. Проверка на граничные значения:
    1. Входные значения: (x = 255, b = 2).

Ожидаемый результат: 2 - ичное представление 255 это 11111111.

    2. Входные значения: (x = 0, b = 16).

Ожидаемый результат: 2 - ичное представление 0 это 0.

Выполнение тестовых случаев

Запустим программу с заданными значениями аргументов.

Оценка результатов выполнения программы на тестах

В процессе тестирования Оракул последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений (X,Y_B). В процессе тестирования производится оценка результатов выполнения путем сравнения получаемого результата с ожидаемым.

Три фазы тестирования

Реализация тестирования разделяется на три этапа:

1. Создание тестового набора (test suite) путем ручной разработки или автоматической генерации для конкретной среды тестирования (testing environment).
2. Прогон программы на тестах, управляемый тестовым монитором (test monitor, test driver) с получением протокола результатов тестирования (test log).
3. Оценка результатов выполнения программы на наборе тестов с целью принятия решения о продолжении или остановке тестирования.

Основная проблема тестирования – определение достаточности множества тестов для истинности вывода о правильности реализации программы, а также нахождения множества тестов, обладающего этим свойством.

Пример построения управляющего графа программы

Рассмотрим вопросы тестирования на примере программы (Пример 2.1) на языке C#. Текст этой программы и некоторых других несколько видоизменен с целью сделать иллюстрацию описываемых фактов более прозрачной.

```
         /*  Метод переводит целое неотрицательное x
            в систему счисления с основание b */
1  static public string convert (int x, int b){
2  int d; char ch;
3  string s = "";//
4  while(x > 0){
5      d = x % b;
6      if (d <= 9)
7           ch = (char)(d + '0');
8      else ch = (char)(d - 10 + 'A');
9      s = ch.ToString() + s; x = x / b;} /* Возврат в п.4 */
10  if (s == "")
11     s = "0";
12  return s; }
```

2.3. Пример программы на языке C#

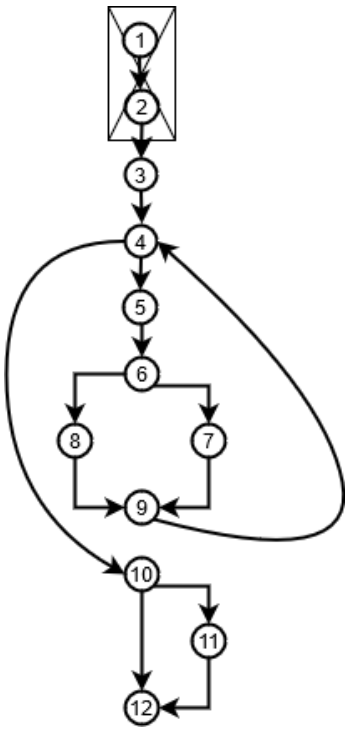


Рис. 2.1. Управляющий граф программы

Управляющий граф программы (УГП) на Рис. 2.1 отображает поток управления программы. Нумерация узлов графа совпадает с нумерацией строк программы. Узлы 1 и 2 не включаются в УГП, поскольку отображают строки описаний, т.е. не содержат управляющих операторов.

2.5. Управляющий граф программы

Управляющий граф программы (УГП) – граф G(V, A), где V(V_1,… V_m) – множество вершин (операторов), A(A_1,… A_n) – множество дуг (управлений), соединяющих операторы-вершины.

Путь – последовательность вершин и дуг УГП, в которой любая дуга выходит из вершины V_i и приходит в вершину V_j, например:(3, 4, 10, 12), (3, 4, 10, 11 ,12), (3, 4, 5, 6, 7, 9, 4, 10, 12), (3, 4, 5, 6, 7, 9, 4, 10, 11, 12), (3, 4, 5, 6, 8, 9, 4, 10, 12), (3, 4, 5, 6, 8, 9, 4, 10, 11, 12), (3, 4, 5, 6, 8, 9, 4, 5, 6, 8, 9, 4, 10, 12)

Ветвь – путь (V_1, V_2, … V_k), где V_1 - либо первый, либо условный оператор программы, V_k - либо условный оператор, либо оператор выхода из программы, а все остальные операторы – безусловные, например: (3, 4), (4, 5, 6), (6, 7, 9, 4), (6, 8, 9, 4), (4, 10), (10, 11, 12), (10, 12). Пути, различающиеся хотя бы числом прохождений цикла – разные пути, поэтому число путей в программе может быть не ограничено. Ветви - линейные участки программы, количество ветвей в программе конечное число.

Существуют реализуемые и нереализуемые пути в программе, в нереализуемые пути в обычных условиях попасть нельзя.

```
static public double H( double x)
  {
   double R;
1  if ( x * x < 0)
2     R = 2;
3  else R = 5;
4  return R * x + x * x;
  }
```

2.5. Пример описания функции с реализуемыми и нереализуемыми путями

```
double H( double x)
  {
   double R;
1  if ( x * x < 0)
2     R = 2;
3  else R = 5;
4  return R * x + x * x;
```

2.5.1. Пример описания функции с реализуемыми и нереализуемыми путями

Например, для функции Пример 2.5 путь (1, 3, 4) реализуем, путь (1, 2, 4) **нереализуем** в условиях нормальной работы. Но при сбоях даже нереализуемый путь может реализоваться.

2.6. Основные проблемы тестирования

Рассмотрим пример тестирования приведённой ниже программы на всей области определения её программной функции.

Пусть программа H(x: int, y: int) реализована в машине с 64 разрядными словами, тогда мощность множества тестов ||(X, Y)||=2**128.

Это означает, что компьютеру, работающему на частоте 1Ггц, для прогона этого набора тестов (при условии, что один тест выполняется за 100 команд) потребуется ~ 3К лет.

Отсюда вывод:

1. Тестирование программы на всех наборах входных значений невозможно.
2. Невозможно тестирование и на всех путях, если программа содержит циклы, управляемые событиями.
3. Следовательно, надо отбирать конечный набор тестов, позволяющий проверить программу на основе наших интуитивных представлений.

Требование к тестам - программа на любом из них должна останавливаться, т.е. не попадать в бесконечный цикл («зацикливаться»). Можно ли заранее гарантировать останов на любом тесте?

В теории алгоритмов доказано, что не существует общего метода для решения этого вопроса, а также вопроса, достигнет ли программа на данном тесте заранее фиксированного оператора.

Задача о выборе конечного набора тестов (X, Y) для проверки программы в общем случае неразрешима.

Поэтому на практике приходится искать частные случаи решения этой задачи.

Контрольные вопросы

1. Как можно рассматривать программу с точки зрения математики?
2. Какие два метода обоснования истинности формул существуют?
3. Что такое отладка?
4. Что такое тестирование?
5. Что собой представляет процедура тестирование?
6. В чём назначение Оракула?
7. Что такое статическое тестирование?
8. Что такое динамическое тестирование?
9. Что такое УПГ?
10. Что такое путь?
11. Что такое ветвь?
12. Назовите три фазы тестирования?
13. В чём состоит проблема тестирования?
14. В чём состоит требование к тестам?

3. Критерии выбора тестов

3.1. Требования к идеальному критерию тестирования

Требования к идеальному критерию следующие:

1. Критерий должен быть достаточным, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.
2. Критерий должен быть полным, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.
3. Критерий должен быть надежным, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы.
4. Критерий должен быть легко проверяемым, например вычисляемым на тестах.

Для нетривиальных классов программ в общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций.

Поэтому мы стремимся к идеальному общему критерию через реальные частные.

3.2. Классы критериев

Выделяют следующие классы критериев тестирования ПО.

1. Структурные критерии используют информацию о структуре программы (критерии так называемого "белого ящика")
2. Функциональные критерии формулируются в терминах описания требований к программному изделию (критерии так называемого "черного ящика").
3. Критерии стохастического тестирования формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.
4. Мутационные критерии ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

3.2.1. Структурные критерии (класс I).

Структурные критерии используют модель программы в виде "белого ящика", что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный класс критериев часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing).

Структурные критерии базируются на основных элементах УГП, операторах, ветвях и путях.

- Условие критерия тестирования команд (критерий C0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного

раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.

- Условие критерия тестирования ветвей (критерий C1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах автоматизации тестирования.
- Условие критерия тестирования путей (критерий C2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или числом классов выходных путей).

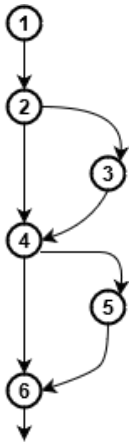
Рассмотрим пример 3.1, в котором приведён пример простой программы. Рассмотрим условия ее тестирования в соответствии со структурными критериями.

```
public static int M(int x, int y, int z) {
1  int m = x;
2  if (m > y)
3      m = y;
4  if (m > z)
5      m = z;
6  return m; }
```

3.1. Пример простой программы, для тестирования по структурным критериям на C#

```
int M(int x, int y, int z){
1  int m = x;
2  if (m > y)
3      m = y;
4  if (m > z)
5      m = z;
6  return m;}
```

3.1.1. Пример простой программы, для тестирования по структурным критериям на C++



3.1 УПГ программы, для тестирования по структурным критериям

УПГ этой программы содержит следующие ветви: 1-2, 2-4, 2-3-4, 4-6, 4-5-6;

и следующие пути: 1-2-4-6, 1-2-3-4-5-6, 1-2-3-4-6, 1-2-4-5-6.

Тестовый набор из одного теста, удовлетворяет критерию команд (C0):

(X, Y) = {(x = 3, y = 2, z = 1, m = 1)} покрывает все операторы трассы 1-2-3-4-5-6.

Тестовый набор из двух тестов, удовлетворяет критерию ветвей (C1):

(X, Y) = {(x = 3, y = 2, z = 1, m = 1), (x = 1, y = 2, z = 3, m = 1)} добавляет 1 тест к множеству тестов для C0 и трассу 1-2-4-6. Трасса 1-2-3-4-5-6 проходит через все ветви достижимые в операторах if при условии true, а трасса 1-2-4-6 через все ветви, достижимые в операторах if при условии false.

Тестовый набор из четырех тестов, удовлетворяет критерию путей (C2):

(X, Y)={ (x = 3, y = 2, z = 1, m = 1), (x = 1, y = 2, z = 3, m = 1), (x = 3, y = 1, z = 2, m = 1), (x = 2, y = 3, z = 1, m = 1)}

Набор условий для двух операторов if с метками 2 и 4 приведен в таблица 3.1

Таблица 3.1. Условия операторов if

	(3,2,1,1)	(1,2,3,1)	(3,1,2,1)	(2,3,1,1)
2 if (m > y)	>	≤	>	≤
4 if (m > z)	>	≤	≤	>

Критерий путей C2 проверяет программу более тщательно, чем критерии - C1, однако даже если он удовлетворен, нет оснований утверждать, что программа реализована в соответствии со спецификацией.

Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию C2 мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

3.2.2. Функциональные критерии (класс II)

Функциональный критерий - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При функциональном тестировании преимущественно используется модель "черного ящика". Проблема функционального тестирования - это, прежде всего, трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (Software requirement specification, Functional specification и т.п.), как правило, достаточно объемны, тем не менее, соответствующая проверка должна быть всеобъемлющей.

Ниже приведены частные виды функциональных критериев.

- Тестирование пунктов спецификации - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза.

Спецификация требований может содержать сотни и тысячи пунктов требований к программному продукту и каждое из этих требований при тестировании должно быть проверено в соответствии с критерием не менее чем одним тестом

- Тестирование классов входных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза.

При создании тестов классы входных данных сопоставляются с режимами использования тестируемого компонента или подсистемы приложения, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов. Следует заметить, что перебирая в соответствии с критерием величины входных переменных (например, различные файлы - источники входных данных), мы вынуждены применять мощные тестовые наборы. Действительно, наряду с ограничениями на величины входных данных, существуют ограничения на величины входных данных во всевозможных комбинациях, в том числе проверка реакций системы на появление ошибок в значениях или структурах входных данных. Учет этого многообразия - процесс трудоемкий, что создает сложности для применения критерия.

- Тестирование правил - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики.

Следует заметить, что грамматика должна быть достаточно простой, чтобы трудоемкость разработки соответствующего набора тестов была реальной (вписывалась в сроки и штат специалистов, выделенных для реализации фазы тестирования).

- Тестирование классов выходных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (time out).

При создании тестов классы выходных данных сопоставляются с режимами использования тестируемого компонента или подсистемы, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов.

- Тестирование функций - набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза.

Очень популярный на практике критерий, который, однако, не обеспечивает покрытия части функциональности тестируемого компонента, связанной со структурными и поведенческими свойствами, описание которых не сосредоточено в отдельных функциях (т.е. описание рассредоточено по компоненту).

Критерий тестирования функций объединяет отчасти особенности структурных и функциональных критериев. Он базируется на модели "полупрозрачного ящика", где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.

- Комбинированные критерии для программ и спецификаций - набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза.

При этом все комбинации непротиворечивых условий надо подтвердить, а условия противоречий следует обнаружить и ликвидировать.

3.2.3. Стохастические критерии (класс III)

Стохастическое тестирование применяется при тестировании сложных программных комплексов - когда набор детерминированных тестов (X,Y) имеет громадную мощность. В случаях, когда подобный набор невозможно разработать и исполнить на фазе тестирования, можно применить следующую методику.

- Разработать программы - имитаторы случайных последовательностей входных сигналов {x}.
- Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y).
- Протестировать приложение на тестовом наборе (X,Y), используя два способа контроля результатов:
    - Детерминированный контроль - проверка соответствия вычисленного значения $y_B \in \{y\}$ значению y, полученному в результате прогона теста на наборе {x} - случайной последовательности входных сигналов, сгенерированной имитатором.
    - Стохастический контроль - проверка соответствия множества значений$\{y_B\}$, полученного в результате прогона тестов на наборе входных значений {x}, заранее известному распределению результатов F(Y).

В этом случае множество Y неизвестно (его вычисление невозможно), но известен закон распределения данного множества.

Критерии стохастического тестирования:

- Статистические методы окончания тестирования - стохастические методы принятия решений о совпадении гипотез о распределении случайных величин. К ним принадлежат широко известные: метод Стьюдента (St), метод Хи-квадрат (χ^2) и т.п.
- Метод оценки скорости выявления ошибок - основан на модели скорости выявления ошибок, согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для фазы тестирования приложения.

3.2.4. Мутационный критерий (класс IV).

Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа – перестановка местами максимальных значений индексов в

описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы цикла на 1 и т.п.

Предлагается подход, позволяющий на основе мелких ошибок оценить общее число ошибок, оставшихся в программе.

Подход базируется на следующих понятиях:

Мутации - мелкие ошибки в программе.

Мутанты - программы, отличающиеся друг от друга мутациями.

Метод мутационного тестирования - в разрабатываемую программу P вносят мутации, т.е. искусственно создают программы-мутанты P_1, P_2... Затем программа P и ее мутанты тестируются на одном и том же наборе тестов (X,Y).

Если на наборе (X,Y) подтверждается правильность программы P и, кроме того, выявляются все внесенные в программы-мутанты ошибки, то набор тестов (X,Y) соответствует мутационному критерию, а тестируемая программа объявляется правильной.

Если некоторые мутанты не выявили всех мутаций, то надо расширять набор тестов (X,Y) и продолжать тестирование.

Контрольные вопросы

1. Перечислите требования к идеальному критерию тестирования?
2. Раскройте понятие достаточность критерия тестирования.
3. Раскройте понятие полноты критерия тестирования.
4. Раскройте понятие надёжности критерия тестирования.
5. Раскройте понятие проверяемость критерия тестирования.
6. Перечислите классы критериев тестирования?
7. Что такое структурные критерии тестирования?
8. Что такое функциональные критерии тестирования?
9. Что такое мутационные критерии тестирования?
10. Что такое критерии стохастического тестирования?
11. На чём базируются структурные критерии тестирования?
12. В чём состоит условие критерия тестирования команд (критерий C0)?
13. В чём состоит условие критерия **тестирования ветвей** (критерий C1)?
14. В чём состоит условие критерия **тестирования путей** (критерий C2)?
15. Приведите частные виды функциональных критериев.
16. Что такое Стохастическое тестирование?
17. В чём сущность Мутационного критерия (класс IV) тестирования?

4. Средства модульного тестирования Visual Studio

4.1. Модульное тестирование или юнит тестирование (unit testing)

Виды тестирования:

- Модульное тестирование выполняется программистом по мере разработки отдельных частей кода, которые можно проверить по отдельности: объекты, классы, функции, программные модули. Тесты пишутся отдельно для каждой функции или метода. На этом этапе проверяют работоспособность части кода, нет ли регрессии— не появились ли после изменения кода ошибки там, где раньше всё работало нормально. Это самый нижний уровень тестирования, часто это делают те, кто пишет код.
- К интеграционному тестированию переходят после модульной проверки. Здесь тестируют связи между проверенными элементами и то, как программа взаимодействует с операционной системой, оборудованием.
- Системное тестирование показывает, соответствует ли готовая система функциональным и нефункциональным требованиям.
- Приёмочное тестирование выполняется заказчиком, на этапе приёмки программного обеспечения от разработчиков. Его цель— убедиться, что продукт удовлетворяет требованиям заказчика. На основании этого заказчик принимает решение, о готовности программы или необходимости её доработки.

Модульное тестирование или юнит тестирование (unit testing) — заключается в изолированной проверке каждого отдельного модуля путем запуска тестов в искусственной среде. Для этого необходимо использовать драйверы и заглушки. Модульное тестирование — первый шаг в испытании разработанного исходного кода. Оценивая каждый модуль изолированно, и подтверждая корректность его работы, точно установить проблему значительно проще чем, если бы элемент был частью системы.

Unit (модуль, элемент) — наименьший компонент, который можно откомпилировать – функция, метод, класс.

Unit тест – блок кода (обычно метод), который вызывает тестируемый блок кода и проверяет правильность его работы. Если результат юнит-теста не совпадает с ожидаемым результатом, тест считается не пройденным.

Драйверы — модули тестов, которые запускают тестируемый элемент.

Заглушки — заменяют недостающие компоненты, которые вызываются модулем и выполняют следующие действия:

- возвращаются к элементу, не выполняя никаких других действий;
- отображают трассировочное сообщение и иногда предлагают тестировщику продолжить тестирование;
- возвращают постоянное значение или предлагают тестировщику самому ввести возвращаемое значение;
- осуществляют упрощенную реализацию недостающей компоненты;
- имитируют исключительные или аварийные условия.

Цель модульного тестирования:

- выявить локализованные в модуле ошибки в реализации алгоритмов, а также определить степень готовности системы к переходу на следующий уровень разработки и тестирования;
- получить работоспособный код с наименьшими затратами. Его применение оправдано тогда и только тогда, когда оно дает больший эффект, нежели другие методы.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, таких как работа с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования. Модульное тестирование проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Для тестирования любого метода класса необходимо:

- Определить, какая часть функциональности метода должна быть протестирована, то есть при каких условиях он должен вызываться. Под условиями здесь понимаются параметры вызова методов, значения полей и свойств объектов, наличие и содержимое используемых файлов и т. д.
- Создать тестовое окружение, обеспечивающее требуемые условия.
- Запустить тестовое окружение на выполнение.
- Обеспечить сохранение результатов в файл для их последующей проверки.
- После завершения выполнения сравнить полученные результаты со спецификацией.

4.2. Средства автоматизации модульного тестирования Visual Studio

Средства автоматизации модульного тестирования Visual Studio, разработаны для поддержки отдельных пользователей и групп пользователей. Модульные тесты позволяют разработчикам и тест-инженерам быстро искать логические ошибки в функциях и методах классов для проектов на языках Visual C#, Visual Basic и Visual C++.

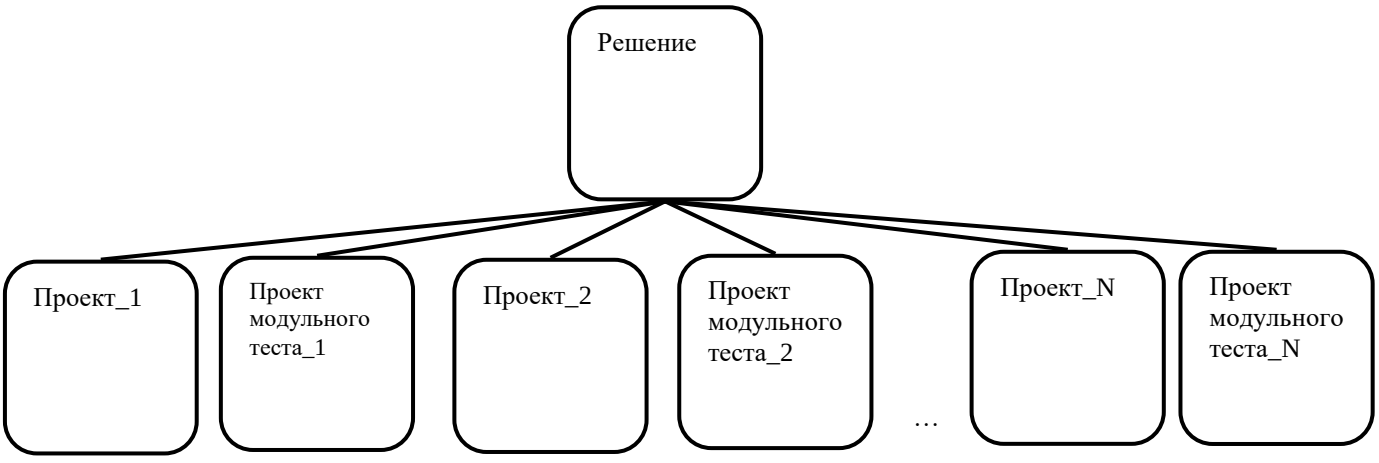
Средства модульного тестирования Visual Studio включают:

1. Обозреватель тестов. Обозреватель тестов позволяет выполнять модульные тесты и просматривать их результаты. Обозреватель тестов может использовать любую структуру, в том числе тестирования модулей сторонних расширений, которые имеют адаптер для обозревателя.
2. Платформа модульного тестирования Майкрософт для управляемого кода— Платформа для тестирования Майкрософт для управляемого кода устанавливается с Visual Studio и предоставляет среду для тестирования кода в .NET.

3. Платформа модульного тестирования Майкрософт для C++ — платформа модульного тестирования Майкрософт для C++ устанавливается в составе рабочей нагрузки Разработка классических приложений на C++ . Эта платформа обеспечивает тестирование машинного кода. Вдобавок включаются платформы Google Test, Boost.Test и CTest, а также сторонние адаптеры для дополнительных платформ тестирования. Дополнительные сведения см. в статье Создание модульных тестов для C/C++.
4. Инструменты покрытия кода — можно определить объем кода продукта, который покрывают модульные тесты, при помощи одной команды в обозревателе тестов.
5. Платформа изоляции Microsoft Fakes — границы изоляции Microsoft Fakes могут создать постановочные классы и методы для рабочего кода и систем, которые создают зависимости в тестируемом коде. Путем реализации подставных делегатов для функции можно контролировать поведение и возвращаемые значения объекта зависимости.

Модульное тестирование наиболее эффективно, когда оно является неотъемлемой частью рабочего процесса разработки программного обеспечения. Как только создается функция или блок кода приложения, необходимо создать модульные тесты, проверяющие поведение кода в ответ на стандартные, граничные и неверные случаи входных данных. В практике разработки программного обеспечения, известной как разработка через тестирование, модульные тесты создаются перед разработкой кода, поэтому модульные тесты используются как проектная документация и функциональная спецификация.

Модульные тесты зачастую отражают структуру тестируемого кода. Например, отдельный проект модульных тестов может быть создан для каждого проекта в продукте. Тестовый проект может находиться в том же решении, что и рабочий код, или он может находиться в отдельном решении.



Visual Studio включает платформы модульного тестирования Майкрософт для управляемого и неуправляемого кода.

Управляемый код — термин, введённый Microsoft для обозначения кода программы, исполняемой под «управлением» виртуальной машины .NET —

Common Language Runtime. При этом обычный машинный код называется неуправляемым кодом.

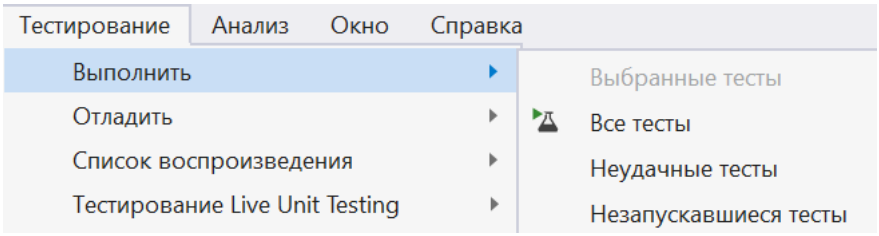
Тестовые проекты могут использовать различные платформы модульного тестирования(xUnit и пр.). Если тестируемый код написан для платформы .NET, то тестовый проект может быть написан на любом языке .NET, независимо от языка тестируемого кода. Проекты с неуправляемым кодом C/C++ должны тестироваться с помощью среды модульного тестирования C++.

Таким образом, процесс модульного тестирования средствами Visual Studio, предполагает наличие:

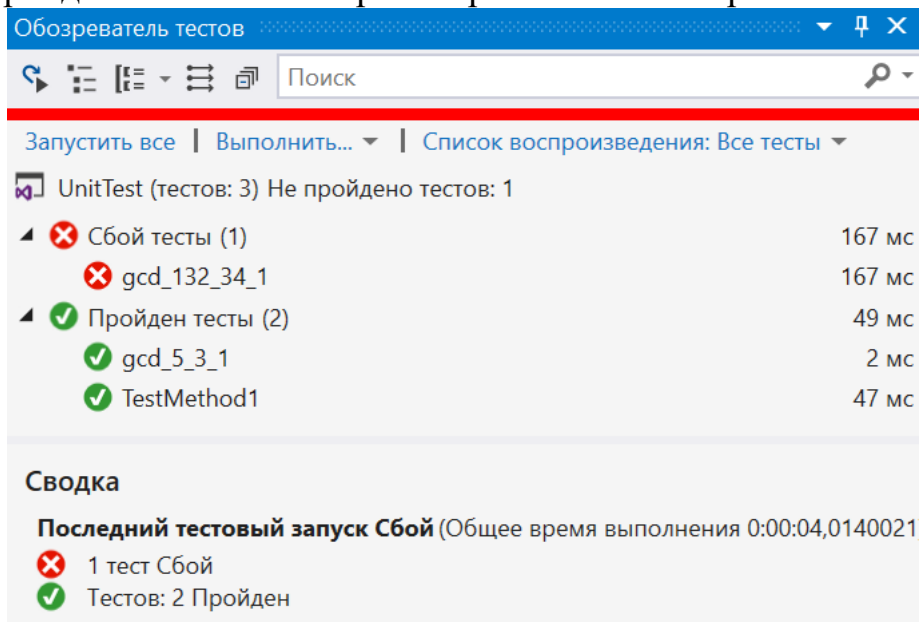
- проекта, функции или классы (модули) которого подлежат тестированию;
- набора тестов предварительно разработанного на основе выбранного критерия, с помощью которых будет осуществляться тестирование модулей.

Для проведения модульного тестирования, необходимо:

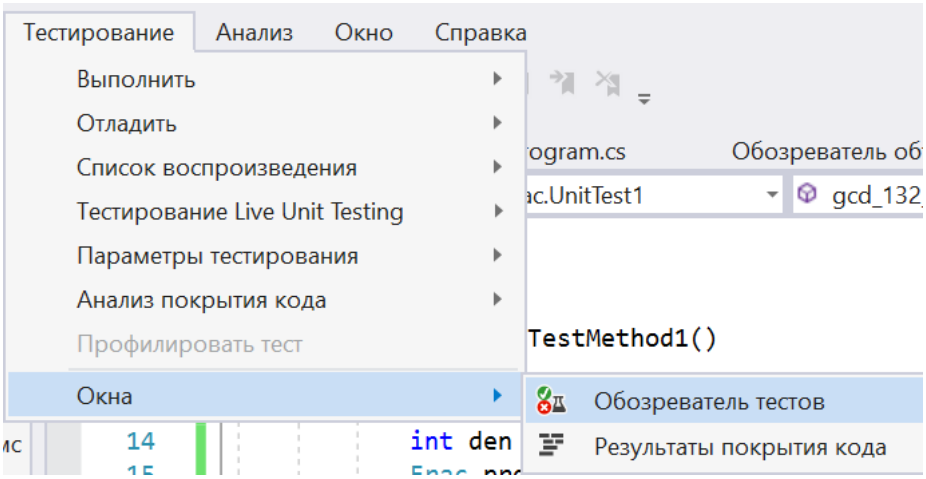
- создать проект модульного теста, который можно добавить в состав решения, включающего тестируемый проект, или сделать в отдельном решении;
- включить в состав проекта модульного теста реализацию тестов, ранее разработанного тестового набора;
- выполнить один или несколько тестов и проанализировать результат их выполнения. Это можно сделать с помощью меню Тестирование.



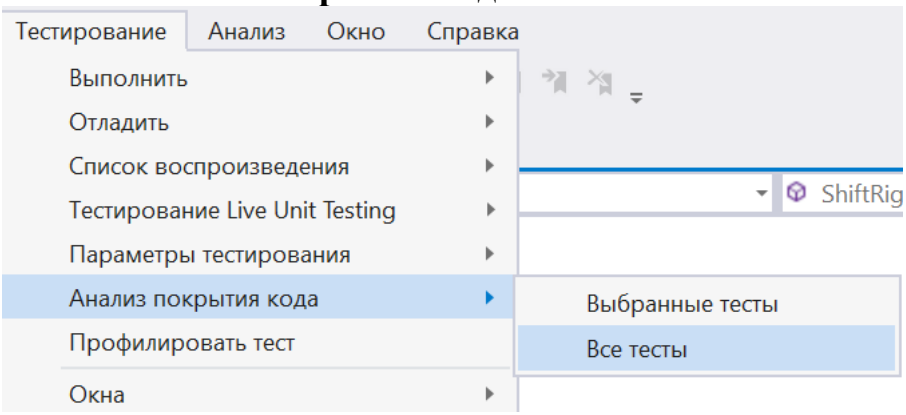
Результат выполнения теста принимает одно из двух значений: «сбой» или «пройден». Его можно просмотреть в окне обозревателя тестов



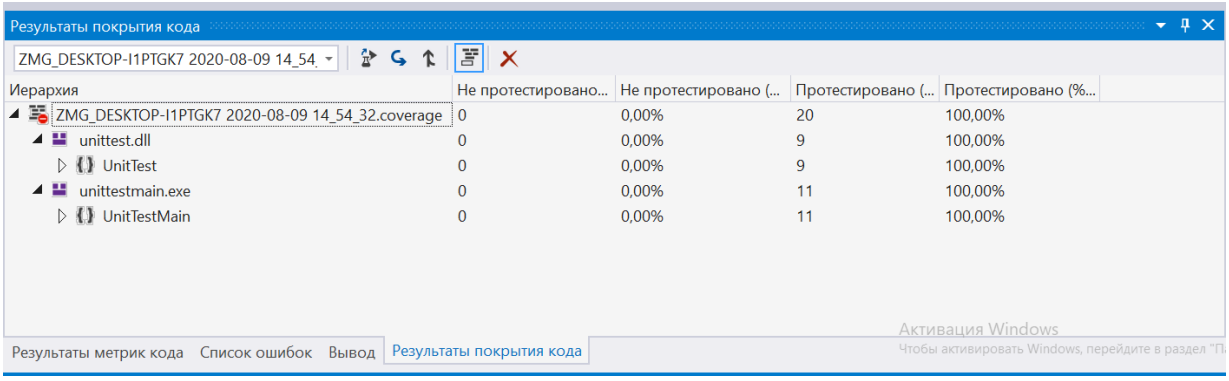
Доступ к этому окну можно получить через меню Тестирование



- проанализировать результат покрытия кода тестами для определения объема протестированного кода. Это можно выполнить с помощью меню **Тестирование > Анализ покрытия кода > Все тесты**



В окне **Результаты тестирования кода** можно проанализировать процент покрытия кода тестами.



4.2.1. Проект модульного теста.

Проект модульного теста, по умолчанию содержит один тестовый класс, в состав которого включён один тестовый метод.

Пример тестового класса на языке C# имеет следующий вид:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTestProject1
{
    [TestClass]
```

```
    public class UnitTest1//Класс, реализующий тестовый набор.
    {
        [TestMethod]
        public void TestMethod1()//Метод, реализующий один из тестов.
        {
               //arrange
               //act
               //assert
        }
    }
}
```

Здесь TestClass – обязательный атрибут тестового класса, поименованного как UnitTest1, а TestMethod –. обязательный атрибут тестового метода, поименованного как TestMethod1.

Тестовые методы, которые вы можете включать в состав тестового класса, включаются с обязательным атрибутом [TestMethod] – тестовые методы. В тестовый метод помещается программный код, реализующий выполнение теста из тестового набора. Код тестового метода, в общем случае, включает в себя:

- подготовку исходных данных для вызова тестируемого модуля (arrange),
- вызов тестируемого модуля для его выполнения (act) и
- сопоставление результата выполнения тестируемого модуля с ожидаемым результатом (assert).

Кроме указанных атрибутов, тестовые методы могут иметь следующие атрибуты:

[ClassInitialize()] (ИнициализироватьКласс) - используется для выполнения кода перед запуском первого теста в классе.

[ClassCleanup()] (КлассОчистка) - используется для запуска кода после выполнения всех тестов в классе.

[TestInitialize()] (ТестИнициализировать) - используется для выполнения кода перед выполнением каждого теста.

[TestCleanup()] (ТестОчистка) - используется для запуска кода после выполнения каждого теста.

Методы, помеченные атрибутом [ClassInitialize()] или [TestInitialize()] создаются, чтобы подготовить среду окружения, в которой будет выполняться модульный тест. Это необходимо для создания требуемого состояния данных для выполнения модульного теста. Например, можно использовать метод [ClassInitialize()] или [TestInitialize()] для копирования, изменения или создания определенных файлов данных, которые будут использоваться в тесте.

Методы, помеченные атрибутом [ClassCleanup()] или [TestCleanUp{}] создаются, чтобы вернуть среду в необходимое состояние после выполнения теста. Это может означать удаление файлов в папках или возвращение базы данных в необходимое состояние.

Для сопоставления результата вызова тестируемого модуля с ожидаемым результатом, необходимо обязательно использовать статические методы класса Assert.

Пример тестового класса на языке C++ имеет следующий вид:

```
#include "stdafx.h"
#include "CppUnitTest.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
    TEST_CLASS(UnitTest1) //Класс, реализующий тестовый набор.
    {
    public:

          TEST_METHOD(TestMethod1) //Метод, реализующий один из тестов.
          {
                // TODO: Разместите здесь код своего теста
          }
    };
}
```

Здесь TEST_CLASS – обязательный атрибут тестового класса, поименованного как UnitTest1, а TEST_METHOD –. обязательный атрибут тестового метода, поименованного как TestMethod1.

TEST_CLASS(className)

Требуется для каждого класса, содержащего методы тестов. Определяет className как тестовый класс. TEST_CLASS должен быть объявлен в области пространства имен.

TEST_METHOD(methodName)
{
  // test method body
}

Определяет methodName как метод теста. TEST_METHOD необходимо объявить в области класса метода.

Атрибуты методов тестов инициализации и очистки:

TEST_METHOD_INITIALIZE(methodName)
{
  // method initialization code
}

Определяет methodName в качестве метода, который выполняется перед выполнением каждого метода теста. TEST_METHOD_INITIALIZE может быть определен только один раз в тестовом классе и должен быть определен в его области.

TEST_METHOD_CLEANUP(methodName)
{
  // test method cleanup code
}

Определяет methodName в качестве метода, который выполняется после выполнения каждого метода теста. TEST_METHOD_CLEANUP может быть определен только один раз в тестовом классе и должен быть определен в его области.

Тестовые классы:

```
TEST_CLASS_INITIALIZE(methodName)
{
   // test class initialization  code
}
```

Определяет methodName в качестве метода, который выполняется перед созданием каждого тестового класса. TEST_CLASS_INITIALIZE может быть определен только один раз в тестовом классе и должен быть определен в его области.

```
TEST_CLASS_CLEANUP(methodName)
{
   // test class cleanup  code
}
```

Определяет methodName в качестве метода, который выполняется после создания каждого тестового класса. TEST_CLASS_CLEANUP может быть определен только один раз в тестовом классе и должен быть определен в его области.

4.2.2. Классы Assert

Методы классов Assert играют роль Оракула, который отвечает на вопрос: соответствует ли результат вызова тестируемого модуля ожидаемому результату.

Пространство имен Microsoft.VisualStudio.TestTools.UnitTesting (C#) и Microsoft::VisualStudio::CppUnitTestFramework (C++) предоставляет несколько типов классов Assert.

Класс Assert.

В методе теста можно вызывать любое число методов класса Assert, таких как Assert.AreEqual(). Класс Assert содержит набор методов, и многие из этих методов имеют несколько перегрузок.

Класс CollectionAssert.

Класс CollectionAssert служит для сравнения коллекций объектов и проверки состояния одной или нескольких коллекций.

Класс StringAssert.

Класс StringAssert служит для сравнения строк. Этот класс содержит различные полезные методы, такие как StringAssert.Contains, StringAssert.Matches и StringAssert.StartsWith.

Класс AssertFailedException.

Исключение AssertFailedException возникает в случае невыполнения теста. Причиной невозможности выполнения теста может быть истечение времени ожидания, непредвиденное исключение или оператор Assert, создающий результат "Ошибка".

Класс AssertInconclusiveException.

Исключение AssertInconclusiveException возникает при каждом завершении теста с неопределенным результатом. Как правило, оператор Assert.Inconclusive

добавляется к тесту, над которым еще ведется работа, для обозначения его неготовности к выполнению.

Класс UnitTestAssertException.

При написании нового класса исключения Assert наследование этого класса от базового класса UnitTestAssertException упрощает выявление исключения как ошибки подтверждения, а не непредвиденного исключения, выдаваемого тестом или продуктивным кодом.

Класс ExpectedExceptionAttribute.

Если необходимо, чтобы метод теста проверял, что исключение, возникающее в этом методе, на самом деле является требуемым исключением, включите в метод теста атрибут ExpectedExceptionAttribute.

Основные методы класса Assert

Класс Assert содержит статические методы, необходимые для сравнения результата выполнения тестируемого модуля с ожидаемым результатом в тестовом методе.

Метод модульного теста использует код метода тестируемого проекта, но выдает результаты по поведению кода только в том случае, если включены операторы вызова методов класса Assert.

Основные методы Assert:

AreEqual(Object, Object). Проверяет два указанных объекта на равенство. Утверждение не выполняется, если объекты не равны.

AreEqual(Double, Double, Double). Проверяет, равны ли два указанных значения с двойной точностью, или лежит ли расхождение между ними в пределах заданной точности. Утверждение не выполняется, если расхождение этих значений выходит за пределы заданной точности.

AreEqual<T>(T, T). Проверяет, что два указанных элемента данных универсального типа равны, используя оператор равенства. Утверждение не выполняется, если они не равны.

AreNotEqual(Object, Object, String). Проверяет два указанных объекта на неравенство. Утверждение не выполняется, если объекты равны. Если утверждение не выполняется, выводит сообщение.

AreNotSame(Object, Object). Проверяет, ссылаются ли две указанные объектные переменные на разные объекты. Утверждение не выполняется, если переменные ссылаются на один и тот же объект.

AreSame(Object, Object). Проверяет, ссылаются ли две указанные объектные переменные на один и тот же объект. Утверждение не выполняется, если переменные ссылаются на разные объекты.

Fail(String). Отменяет выполнение утверждения без проверки каких-либо условий. Выводит сообщение.

Inconclusive(String). Указывает, что утверждение не может быть проверено. Выводит сообщение.

IsFalse(Boolean). Проверяет, имеет ли указанное условие значение false. Утверждение не выполняется, если условие имеет значение true.

IsInstanceOfType(Object, Type). Проверяет, является ли указанный объект экземпляром заданного типа. Утверждение не выполняется, если этот тип не обнаруживается в иерархии наследования объекта.

IsNotNull(Object, String). Проверяет, не имеет ли указанный объект значение ссылка NULL (Nothing в Visual Basic).Утверждение не выполняется, если объект имеет значение ссылка NULL (Nothing в Visual Basic).Если утверждение не выполняется, выводит сообщение.

IsTrue(Boolean, String). Проверяет, имеет ли указанное условие значение true. Утверждение не выполняется, если условие имеет значение false. Если утверждение не выполняется, выводит сообщение.

В нашем примере используется метод AreEqual (параметры которого проверяет, что два указанных элемента данных универсального типа равны, используя оператор равенства. Утверждение не выполняется, если они не равны. Если утверждение не выполняется, выводит сообщение).

4.2.3. Использование покрытия кода для определения объема протестированного кода

Чтобы определить, какая часть кода проекта в действительности тестируется закодированными тестами, такими как модульные тесты, можно воспользоваться возможностью покрытия кода в Visual Studio. Для обеспечения эффективной защиты от ошибок тесты должны выполнять ("покрывать") большую часть кода.

Анализ покрытия кода может применяться и к управляемому (CLI), и к неуправляемому (машинному) коду.

Проанализировать покрытие кода тестами можно с помощью окна **Результаты покрытия кода**. В таблице результатов отображается процент кода, который был выполнен в каждой сборке, классе и методе. Кроме того, редактор исходного кода показывает, какой код был протестирован.

Функция проверки объема протестированного кода доступна только в выпуске Visual Studio Enterprise.

Чтобы после выполнения тестов увидеть, какие строки были выполнены, щелкните **Цвета отображения покрытия кода** в окне **Результаты покрытия кода**. По умолчанию код, охватываемый тестами, выделяется голубым цветом.

Если результаты показывают низкое покрытие, проверьте, какие части кода не обрабатываются, и напишите несколько дополнительных тестов для их покрытия. Команды разработчиков обычно стремятся покрыть около 80% кода. В некоторых случаях допустимо более низкое покрытие. Например, более низкое покрытие допустимо, когда некоторый код создается из стандартного шаблона.

Если полученный результат отличается от ожидаемого, добавьте тесты в тестовый набор и не забудьте снова выполнить покрытие кода после обновления кода. Результаты покрытия и цвета кода не обновляются автоматически после изменения кода или при выполнении тестов.

Отчеты в блоках или строках

Покрытие кода измеряется в ***блоках***. Блок — это часть кода с одной точкой входа и точкой выхода. Если поток управления программы проходит через блок во время тестового запуска, то этот блок учитывается как покрытый. Количество раз, когда используется блок, не влияет на результат.

Результаты также можно отобразить в виде строк, щелкнув **Добавить или удалить столбцы** в заголовке таблицы. Некоторые пользователи предпочитают считать в строках, поскольку процентные соотношения более точно соответствуют размеру фрагментов, которые можно увидеть в исходном коде. Длинный блок вычислений учитывается как единый блок, даже если он занимает большое количество строк.

Строка кода может содержать более одного блока кода. В этом случае, если во время тестового запуска все блоки кода были обработаны в любой строке, она учитывается как одна строка. Если обработаны только некоторые блоки в строке, она считается частичной строкой.

4.3. Создание модульных тестов на C#

В этом разделе описывается создание проектов модульного теста на языке C#.

4.3.1. Создание модульного теста для тестирования метода Main класса Program

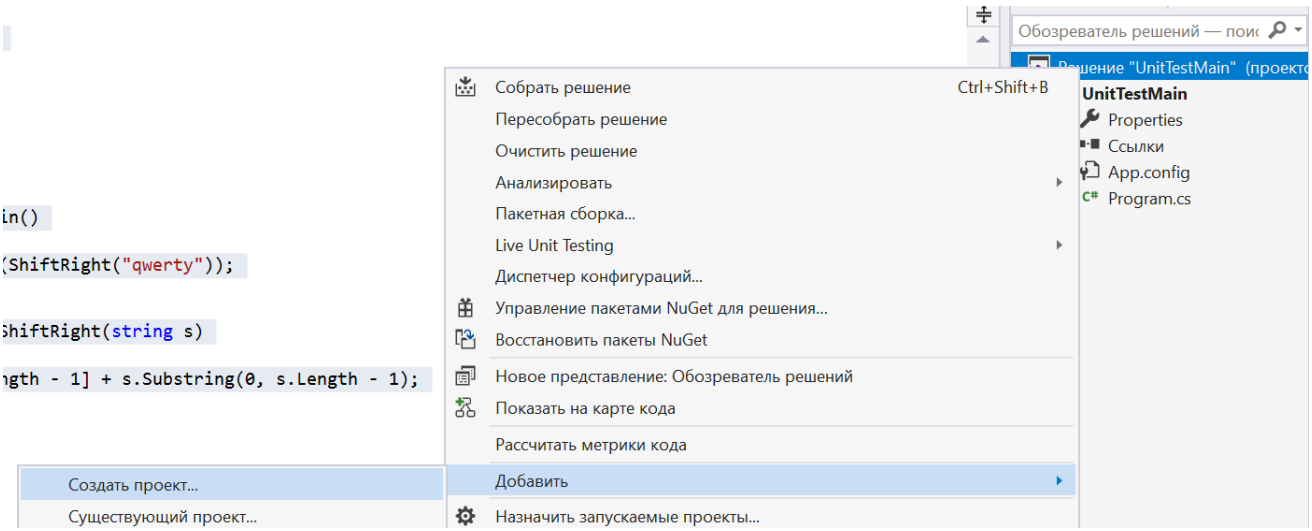
1. Откройте проект, который хотите протестировать в Visual Studio.

Для наглядности приведём модульный тест для тестирования простого проекта консольного приложения с именем **UnitTestMain**. Пример кода для такого проекта выглядит следующим образом:

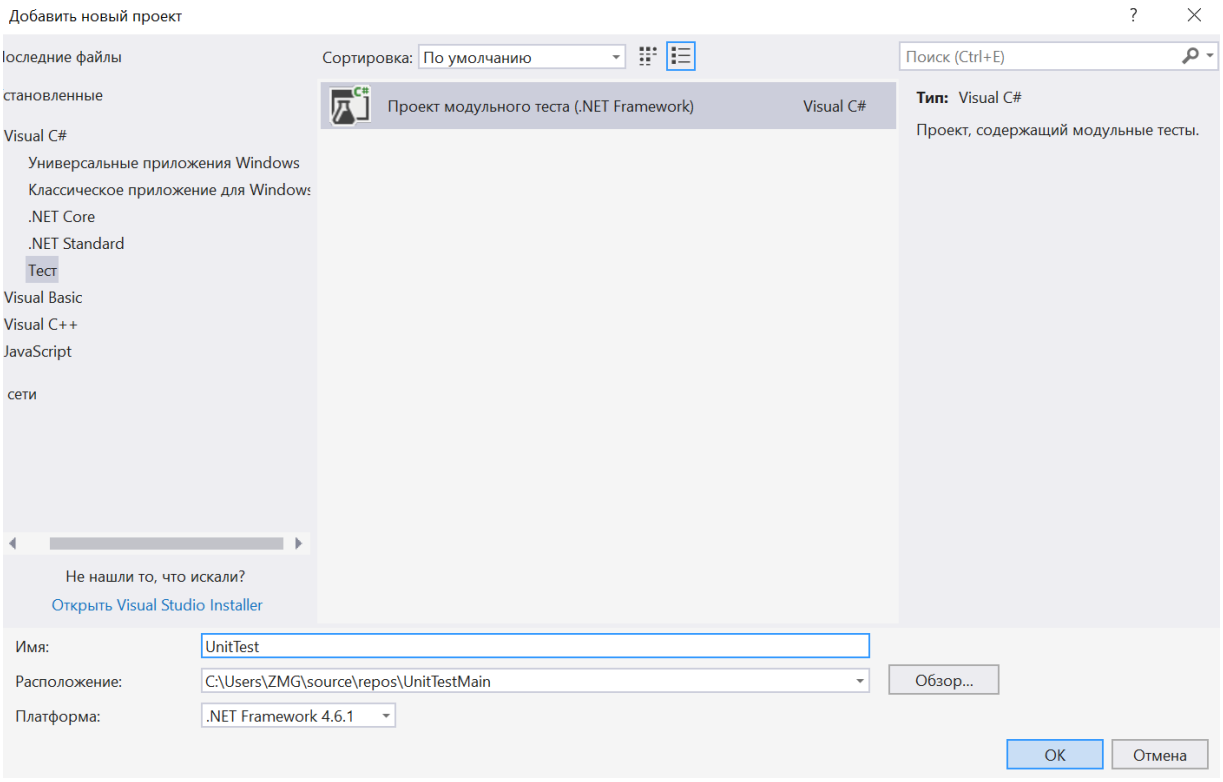
```
namespace UnitTestMain
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine(ShiftRight(args[0]));
        }
        public static string ShiftRight(string s)
        {
            return s = s[s.Length - 1] + s.Substring(0, s.Length - 1);
        }
    }
}
```

Функция `ShiftRight` выполняет сдвиг символов строки на одну позицию вправо.

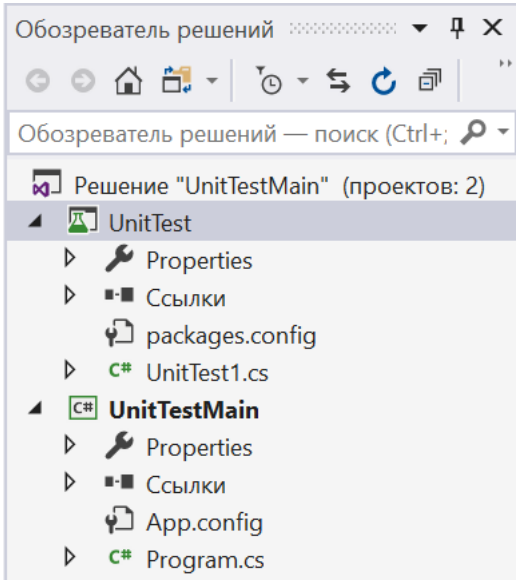
2. Выберите узел решения в окне **Обозревателе решений**. Затем в верхней строке меню выберите **Файл** > **Добавить** > **Создать проект**.



3. В диалоговом окне создания проекта найдите шаблон проекта модульных тестов, который требуется использовать, и выберите его.



Выберите имя для тестового проекта и нажмите **ОК**. Проект добавляется в решение.



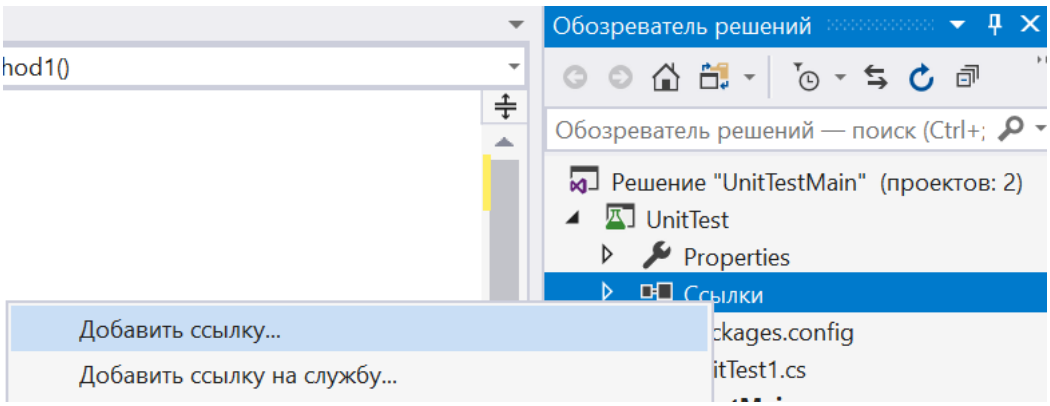
В состав проекта входит файл исходного кода `UnitTest1.cs`. Этот файл содержит текст на языке C# тестового класса UnitTest1, содержащего единственный тестовый метод TestMethod1, следующего вида:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

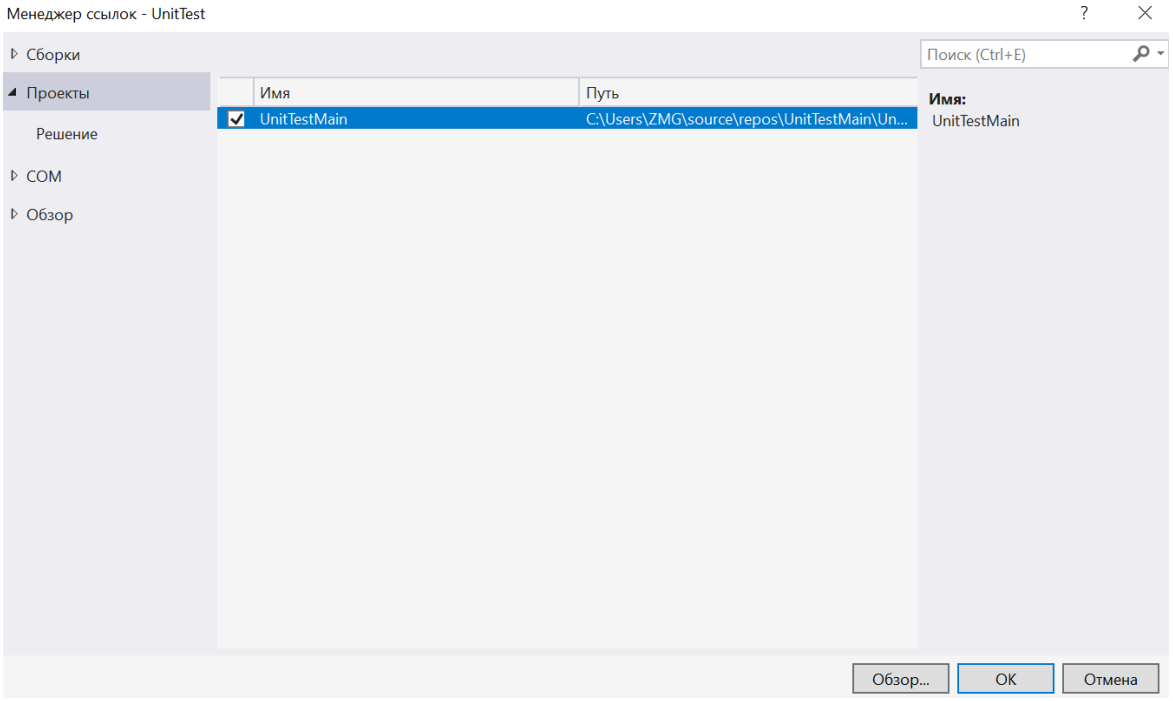
Можно добавить в состав проекта модульного теста необходимое количество тестовых классов, а также классов C#. Причём в тестовый класс можно добавить помимо тестовых методов обычные методы.

Тестовый класс и тестовый метод определяются компилятором по наличию перед их заголовками атрибутов `[TestClass]` и `[TestMethod]` соответственно.

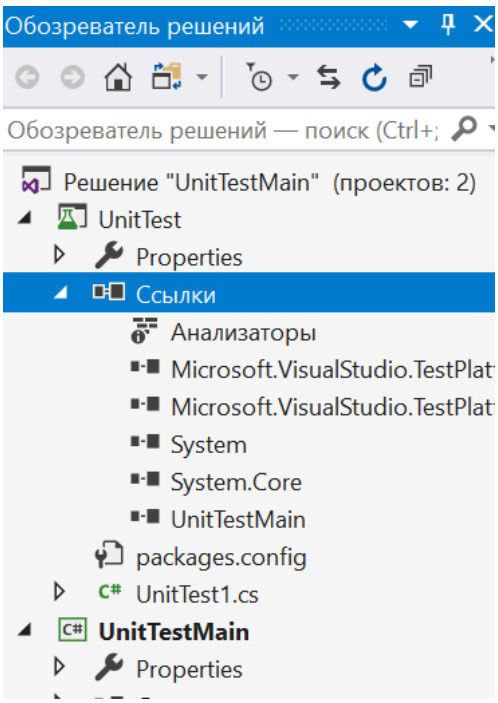
4. В проекте модульных тестов добавьте ссылку на проект, который необходимо протестировать, щелкнув правой кнопкой мыши **Ссылки** или **Зависимости**, и выберите **Добавить ссылку**.



5. Выберите проект, содержащий код, который будет тестироваться, и нажмите **OK**.



В **Обозревателе решений** можно увидеть, что в проекте модульного теста в разделе **Ссылки**, появилась ссылка на тестируемый проект.



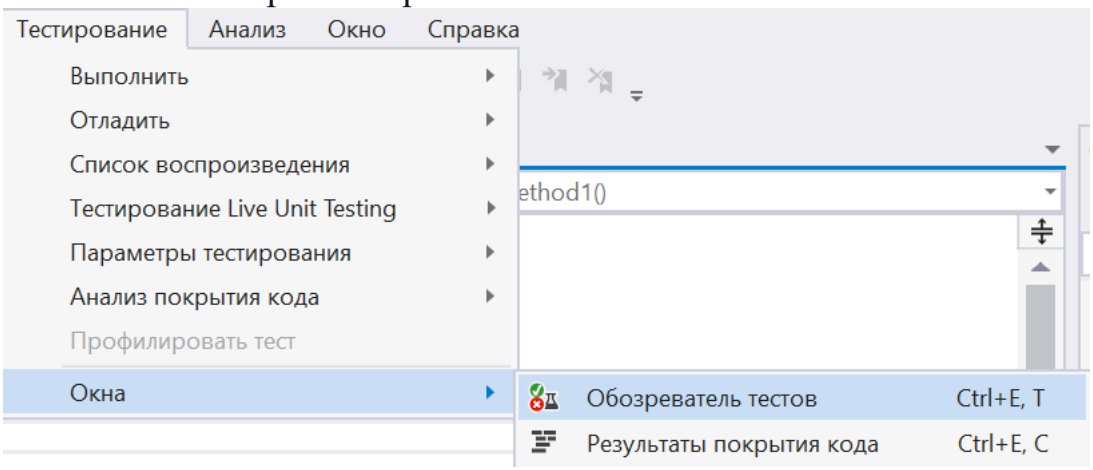
6. Добавьте код в метод модульных тестов.

Для нашего проекта мы можем использовать приведенный ниже код.

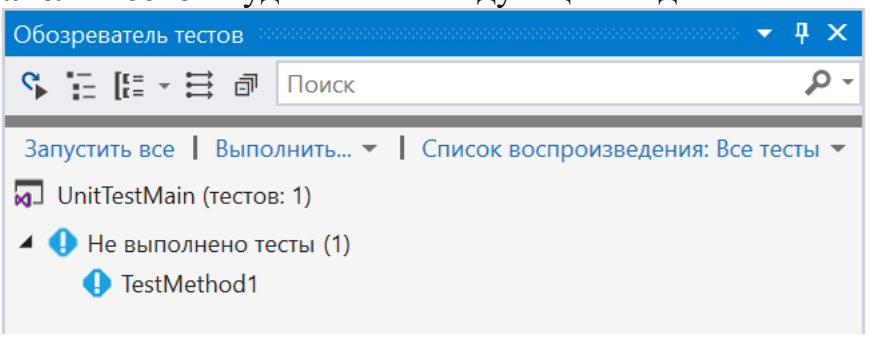
```
using System;
using System.IO;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
            using (var sw = new StringWriter())
            {
             //arrange(обеспечить)
                string[] args = { "qwerty" };//Исходная строка.
                string Expected = "yqwert";//Ожидаемое значение.
                Console.SetOut(sw);//Связывание консоли со строковым потоком.
             //act(выполнить)
                UnitTestMain.Program.Main(args);//Вывод в строковый поток.
                var result = sw.ToString().Trim();//Вычисленное значение.
             //assert(доказать)
                Assert.AreEqual(Expected, result);//Оракул.
            }
        }
    }
}
```

Запуск модульных тестов

1. Откройте окно **Обозревател тестов**, выбрав **Тестирование** > **Окна** > **Обозреватель тестов** в верхней строке меню.

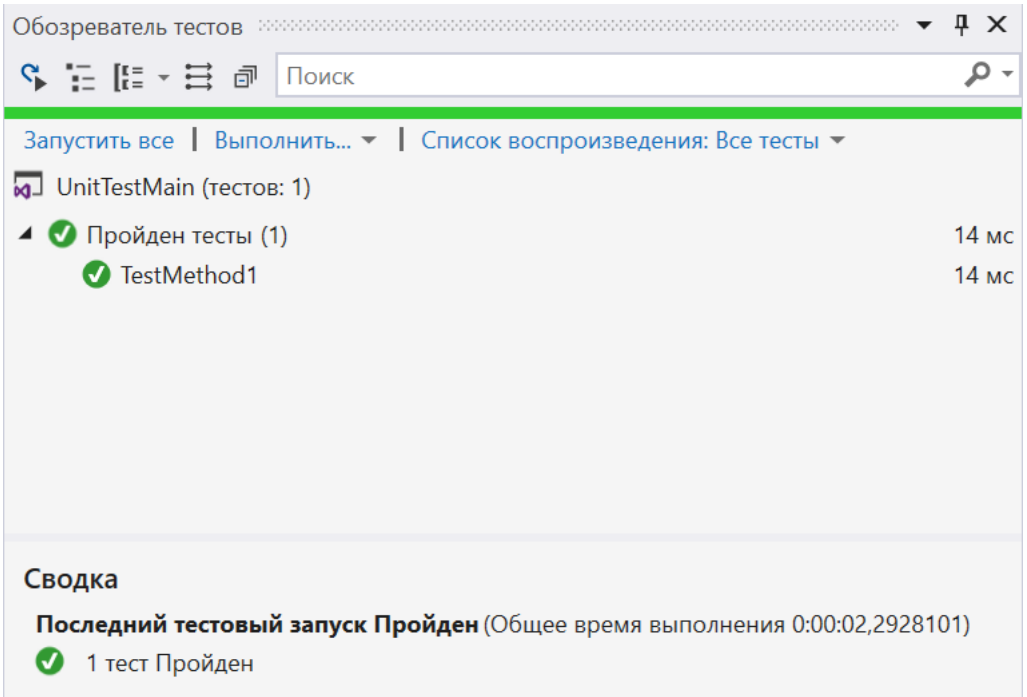


Окно **Обозреватель тестов** будет иметь следующий вид:



2. Выполните модульные тесты, щелкнув **Запустить все**.

После завершения зеленый флажок указывает, что тест пройден. Красный значок "x" указывает на сбой теста.



Используйте окно **Обозреватель тестов** для запуска модульных тестов из встроенной платформы тестирования (MSTest) или сторонней платформы тестирования. Вы можете группировать тесты по категориям, фильтровать список

тестов, а также создавать, сохранять и запускать списки воспроизведения тестов. Кроме того, с его помощью можно выполнять отладку тестов и анализировать производительность тестов и покрытие кода.

Анализ покрытия кода

Чтобы определить, какая часть кода проекта в действительности тестируется закодированными тестами, такими как модульные тесты, можно воспользоваться возможностью покрытия кода в Visual Studio. Для обеспечения эффективной защиты от ошибок тесты должны выполнять ("покрывать") большую часть кода.

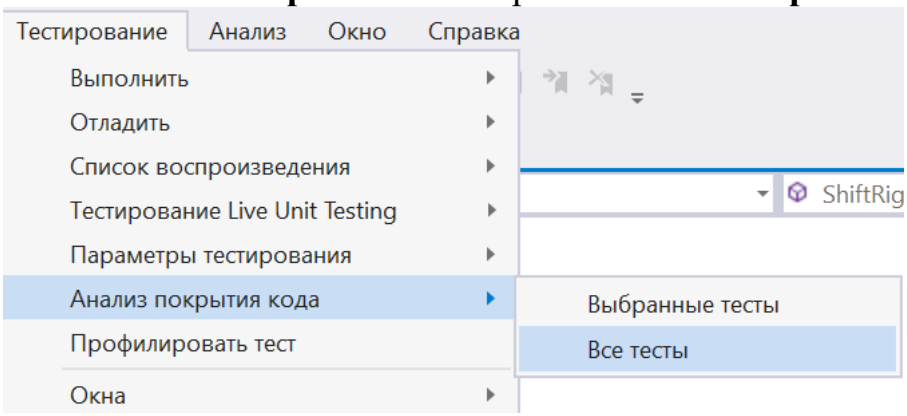
Анализ покрытия кода может применяться и к управляемому (CLI), и к неуправляемому (машинному) коду.

Покрытие кода возможно при выполнении методов тестов с помощью окна **Обозреватель тестов**. В окне **Результаты покрытия кода** в таблице результатов отображается процент кода, который был выполнен в каждой сборке, классе и методе. Кроме того, редактор исходного кода показывает, какой код был протестирован.

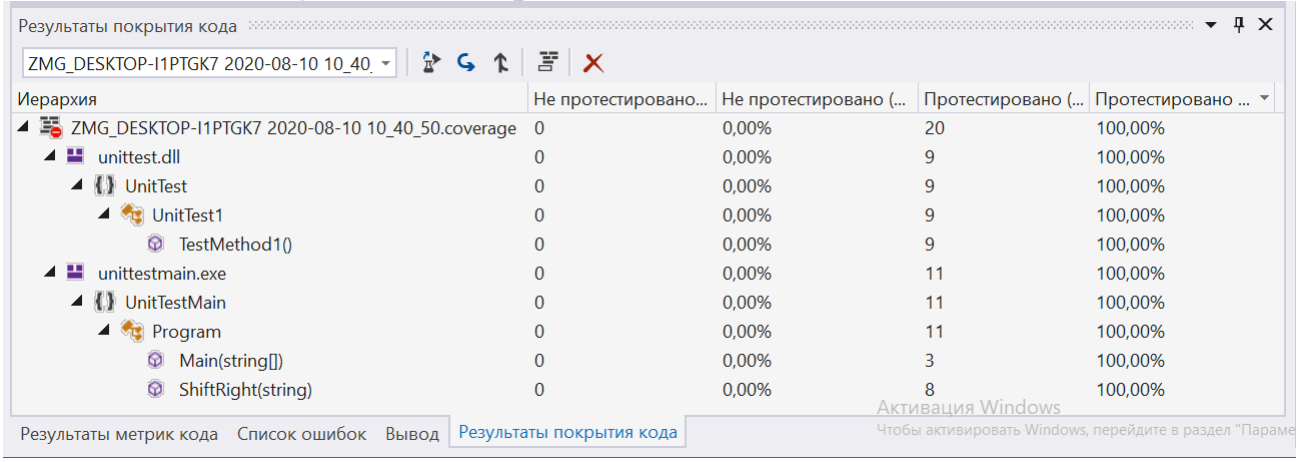
Функция проверки объема протестированного кода доступна только в выпуске Visual Studio Enterprise.

Для анализа покрытия кода необходимо:

1. В меню **Тестирование** выберите **Анализ покрытия кода>Все тесты**.

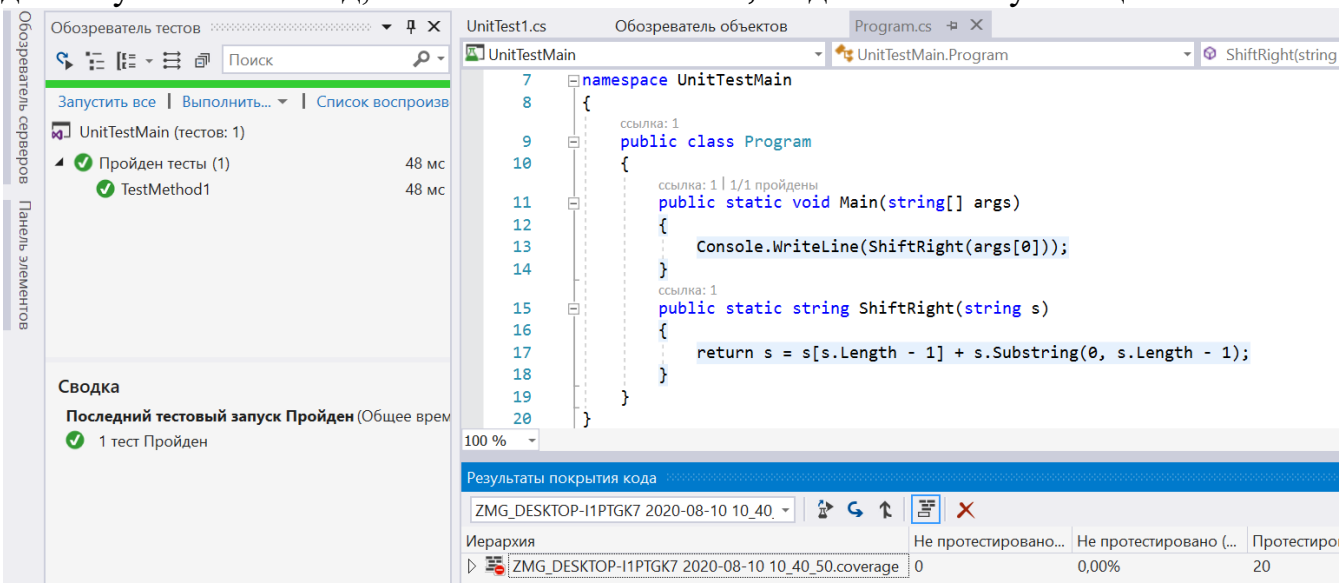


Появится окно **Результаты покрытия кода**.



В нём мы можем просмотреть результат покрытия кода по каждому классу и методу тестируемого проекта.

2. Чтобы после выполнения тестов увидеть, какие строки были выполнены, щелкните **Цвета отображения покрытия** кода в окне **Результаты покрытия кода**. По умолчанию код, охватываемый тестами, выделяется голубым цветом.



3. Если результаты показывают низкое покрытие, проверьте, какие части кода не обрабатываются, и напишите несколько дополнительных тестов для их покрытия. Команды разработчиков обычно стремятся покрыть около 80% кода. В некоторых случаях допустимо более низкое покрытие. Например, более низкое покрытие допустимо, когда некоторый код создается из стандартного шаблона.

4.3.2. Создание модульного теста для тестирования метода класса в проекте консольного приложения

1. Откроем проект, который нам необходимо протестировать в Visual Studio.

Для примера возьмём проект консольного приложения по имени **MyProject_1**, в составе которого имеется класс **MyClass** с методом **Shift_Right**, исходный текст которого на языке C# находится в файле **Shift_Right.cs** и представлен ниже.

```
namespace MyProject_1
{
    class MyClass
    {
        public static void Shift_Right(int[] v, int p)
        {
            for (int i = 1; i <= p; i++)
            {
                int t = v[v.Length - 1];
                for (int j = v.Length - 2; j >= 0; j--)
                {
                    v[j + 1] = v[j];
                }
                v[0] = t;
            }
        }
    }
}
```

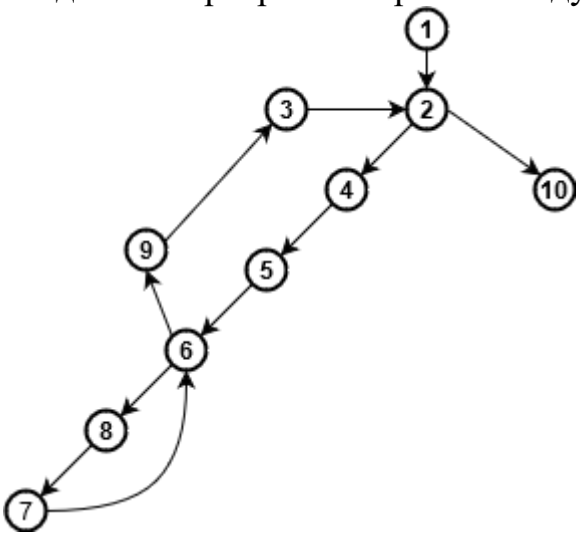
Функция **Shift_Right** выполняет циклический сдвиг значений в массиве **int[] v** на **p** позицию вправо.

2. Разработаем тестовый набор данных для тестирования метода **Shift_Right.**

Для тестирования этого метода необходимо разработать тестовый набор данных, построенный по одному из критериев тестирования. Если в качестве критерия мы будем использовать структурный критерий, то нам понадобится УГП метода. Чтобы его построить представим текст в виде удобном для построения УГП программы.

```
  public static void Shift_Right(int[] v, int p) {
1   for (int i = 1;
2        i <= p;
3        i++)
4        {int t = v[v.Length - 1];
5         for (int j = v.Length - 2;
6              j >= 0;
7              j--)
8              {v[j + 1] = v[j];}
9         v[0] = t; }
10 }
```

Тогда УПГ программы примет следующий вид.



Тестовый набор из одного теста, удовлетворяет критерию команд (C0):

(X, Y) = {({1, 2}$_{\text{вх. массив}}$, p = 1, {2, 1$_{\text{вых. массив}}$})} покрывает все *операторы* трассы **1-2-4-5-6-8-7-6-9-3-2-10**.

Это же тестовый набор удовлетворяет критерию ветвей (C1). Он покрывает все ветви:

1-2, 2-10, 2-4-5-6, 6-9-3-2, 6-8-7-6.

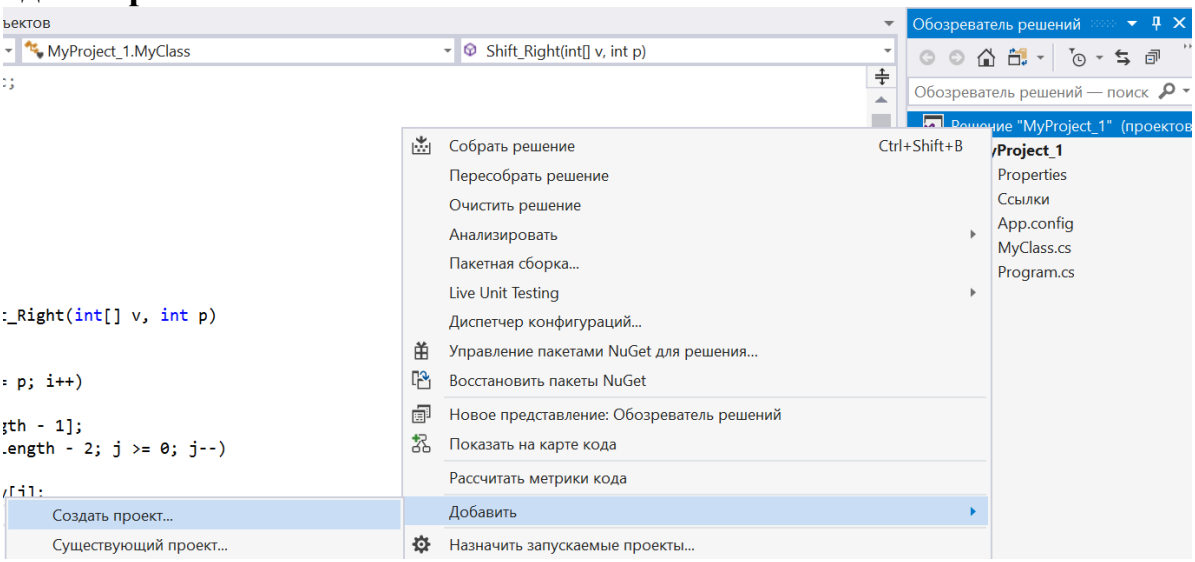
Тестовый набор из трёх тестов, удовлетворяет критерию путей (C2):

(X, Y) = { ({1, 2}$_{\text{вх. массив}}$, p = 0, {2, 1 } $_{\text{вых. массив}}$), ({1}$_{\text{вх. массив}}$, p = 1, { 1 }) $_{\text{вых. массив}}$, ({1, 2}$_{\text{вх. массив}}$, p = 1, { 2, 1} $_{\text{вых. массив}}$) }. На первом тесте первый цикл **for** не выполняется ни разу. На втором тесте второй цикл **for** не выполняется ни разу. На третьем тесте оба цикла **for** выполняется по одному разу.

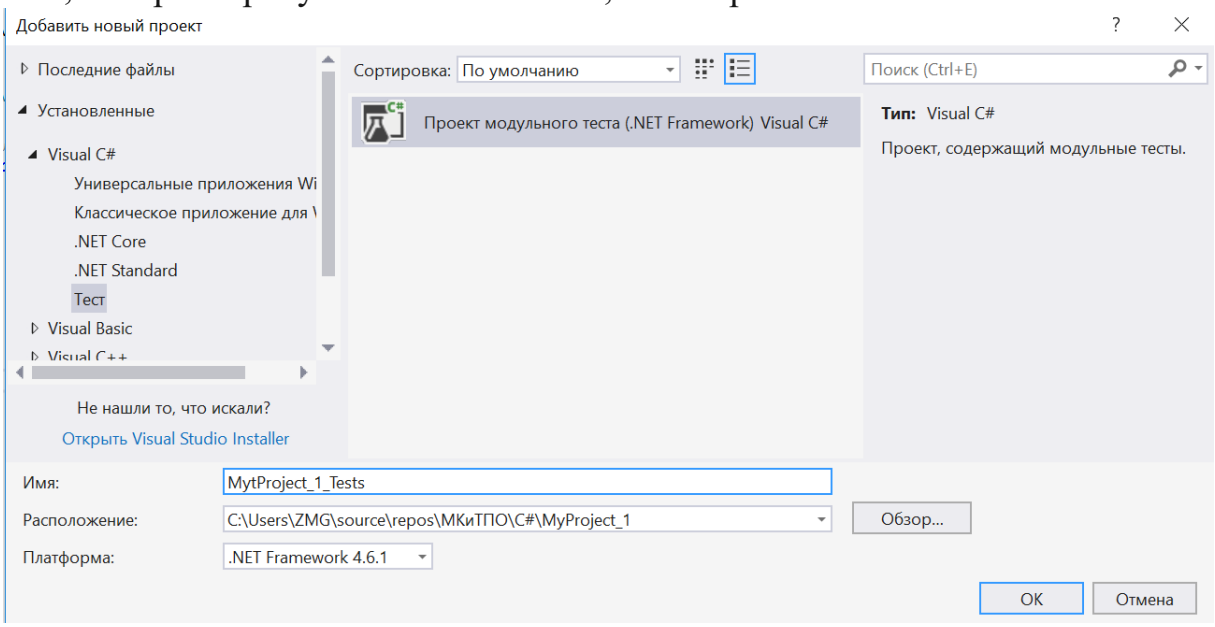
В программе по этому критерию тестируются следующие пути:

1-2-10, 1-2-4-5-6-9-3-2-10, 1-2-4-5-6-8-7-6-9-3-2.

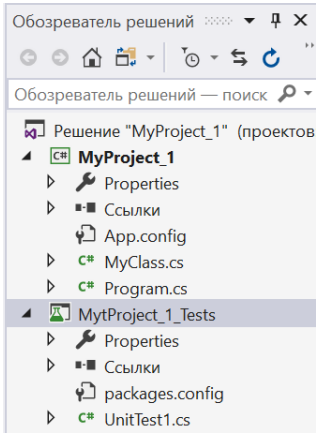
Для тестирования входящих в проект **MyProject_1** классов и методов необходимо в наше решение, имя которого совпадает с именем проекта **MyProject_1,** добавить проект модульного теста. Выберите узел решения в окне **Обозреватель решений**. Затем в верхней строке меню выберите **Добавить** > **Создать проект**.



3. В диалоговом окне создания проекта найдите шаблон проекта модульных тестов, который требуется использовать, и выберите его.



Выберите имя для тестового проекта и нажмите **ОК**. Проект добавляется в решение.



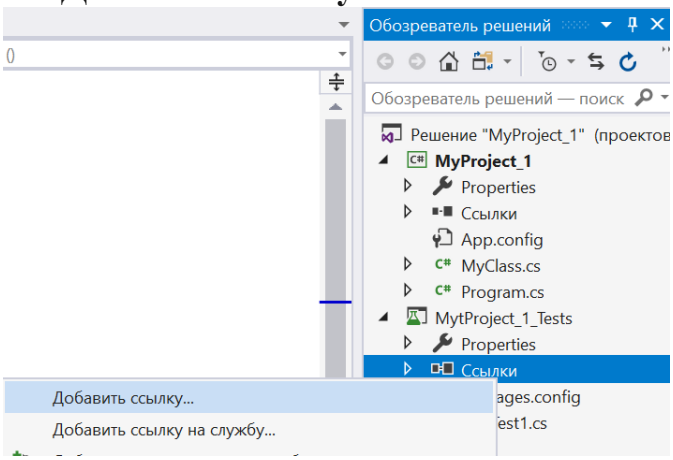
В состав проекта входит файл исходного кода модульного проекта **UnitTest1.cs**. Этот файл содержит текст на языке C# тестового класса **UnitTest1**, содержащего единственный тестовый метод TestMethod1, следующего вида:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MytProject_1_Tests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

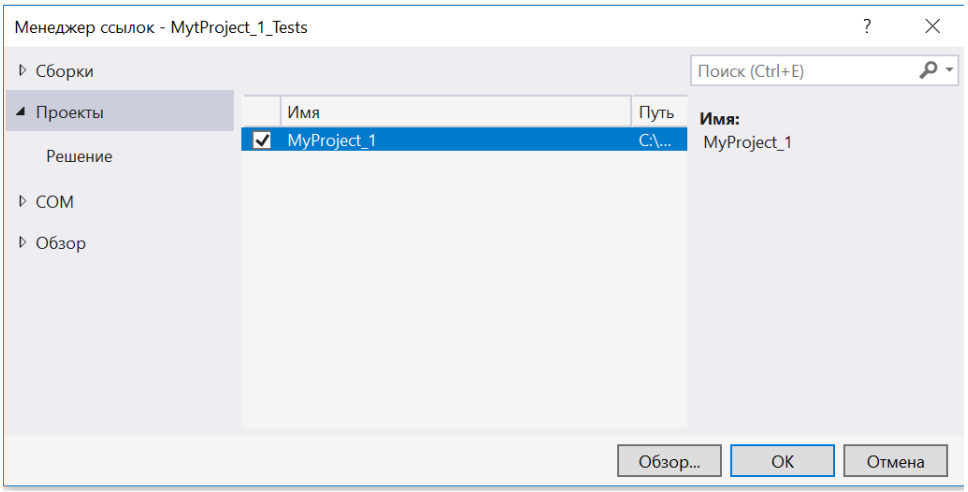
Можно добавить в состав проекта модульного теста необходимое количество тестовых классов, а также классов C#. Причём в тестовый класс можно добавить помимо тестовых методов обычные методы.

Тестовый класс и тестовый метод определяются компилятором по наличию перед их заголовками атрибутов [**TestClass**] и [**TestMethod**] соответственно.

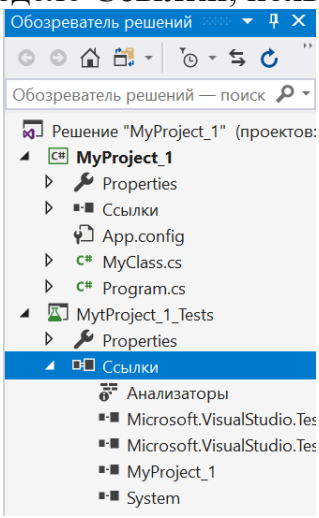
4. Теперь необходимо сделать видимым класс **MyClass** в проекте модульного теста. Для этого в проект модульного теста добавьте ссылку на проект, который необходимо протестировать, щелкнув правой кнопкой мыши **Ссылки** или **Зависимости**, и выберите **Добавить ссылку**.



Выберите проект **MyProject_1**, содержащий код, который будет тестироваться, и нажмите **OK**.



В окне **Обозреватель решений** можно увидеть, что в проекте модульного теста в разделе **Ссылки**, появилась ссылка на тестируемый проект.



В файле **MyClass.cs** перед классом **MyClass** добавьте уровень доступа **public:**

```
public class MyClass
```

В файл **UnitTest1.cpp** добавьте следующее предложение использования:

```
using MyProject_1;
```

Теперь в проекте модульного теста нам будет виден класс **myClass** и мы сможем вызывать его методы.

Содержимое файла UnitTest1.cpp представлено ниже.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MyProject_1;
namespace MytProject_1_Tests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
```

```
    }
}
```

Переименуем класс модульных тестов **UnitTest1** в класс **MyClassTests.** Теперь его имя отражает имя тестируемого класса.

Тестовый метод **TestMethod1** переименуем в **Shift_Right_result_2.** Тогда его имя будет отражать имя тестируемого метода и входные данные для его вызова.

Реализуем тест, ранее разработанный нами по критерию С0, для тестирования метода **Shift_Right** и поместим разработанный код в метод модульных тестов **Shift_Right_result_2**.

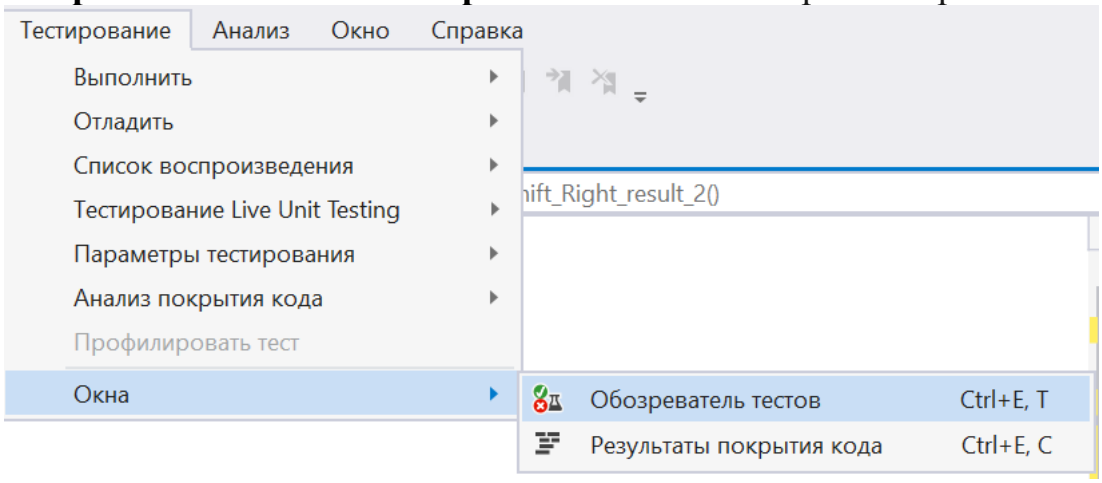
Текст класса модульного теста примет следующий вид:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MyProject_1;//Ссылка на тестируемый проект.
namespace MytProject_1_Tests
{
    [TestClass]
    public class MyClassTests
    {
        [TestMethod]
        public void Shift_Right_result_2()
        {
            //arrange(обеспечить)
            int[] result = new int[]  { 1, 2};//Исходный массив.
            int p = 1;
            int[] expected = new int[] { 2, 1 };//Ожидаемое значение.
            //act(выполнить)
            MyClass.Shift_Right(result, p);
            //assert(доказать)
            CollectionAssert.AreEqual(expected, result);//Оракул.
        }
     }
}
```

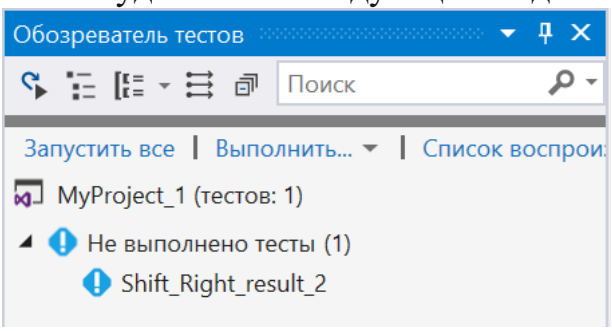
В качестве Оракула мы использовали метод **AreEqual** класса **CollectionAssert**, который позволяет сравнивать две коллекции на равенство.

Запуск модульного теста

5. Для запуска модульного теста воспользуемся окном обозревателя тестов, выбрав **Тестирование** > **Окна** > **Обозреватель тестов** в верхней строке меню.

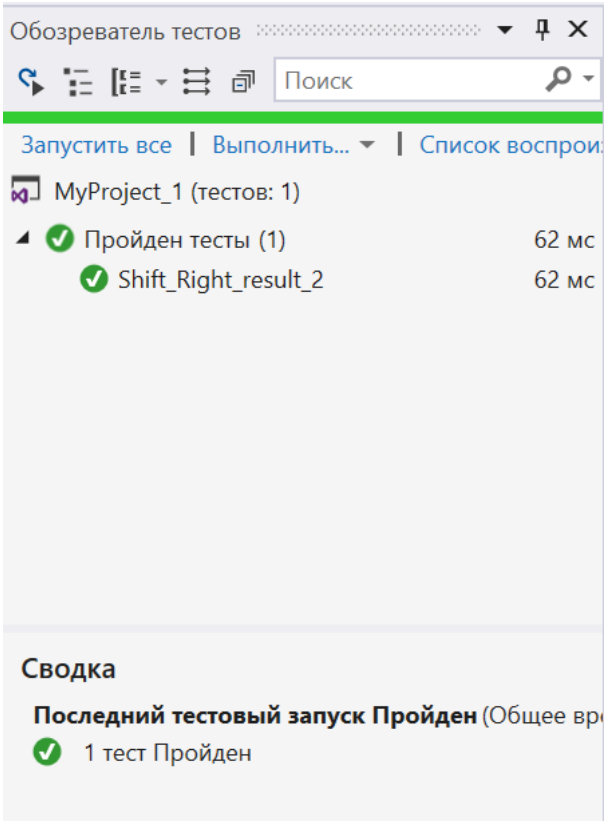


Окно **Обозреватель тестов** будет иметь следующий вид:



6. Для выполнения модульного теста, необходимо выполнить команду **Запустить все**.

После завершения выполнения зеленый флажок указывает, что тест пройден. Красный значок "x" указывает на сбой теста.



Используйте обозреватель тестов для запуска модульных тестов из встроенной платформы тестирования (MSTest) или сторонней платформы тестирования. Вы можете группировать тесты по категориям, фильтровать список тестов, а также создавать, сохранять и запускать списки воспроизведения тестов. Кроме того, с его помощью можно выполнять отладку тестов и анализировать производительность тестов и покрытие кода.

Анализ покрытия кода

Чтобы определить, какая часть кода проекта в действительности тестируется закодированными тестами, такими как модульные тесты, можно воспользоваться средством анализ покрытия кода в Visual Studio. Для обеспечения эффективной защиты от ошибок тесты должны выполнять ("покрывать") большую часть кода.

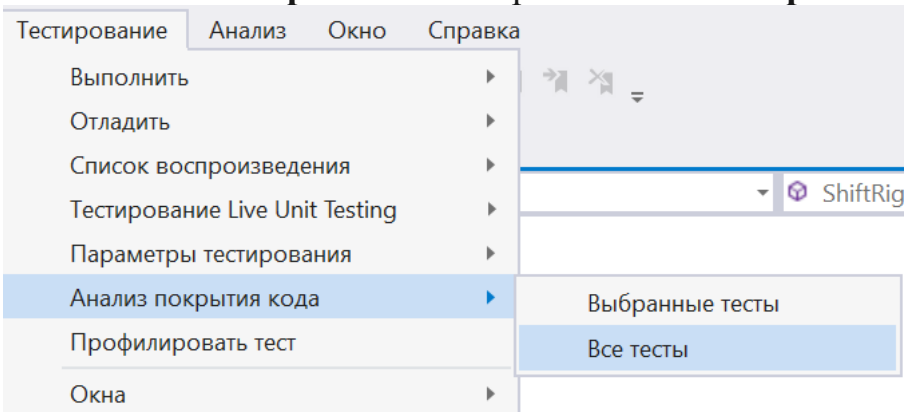
Анализ покрытия кода может применяться и к управляемому (CLI), и к неуправляемому (машинному) коду.

Проанализировать покрытие кода тестами можно с помощью окна **Результаты покрытия кода**. В этом окне в таблице результатов отображается процент кода, который был выполнен в каждой сборке, классе и методе. Кроме того, редактор исходного кода показывает, какой код был протестирован.

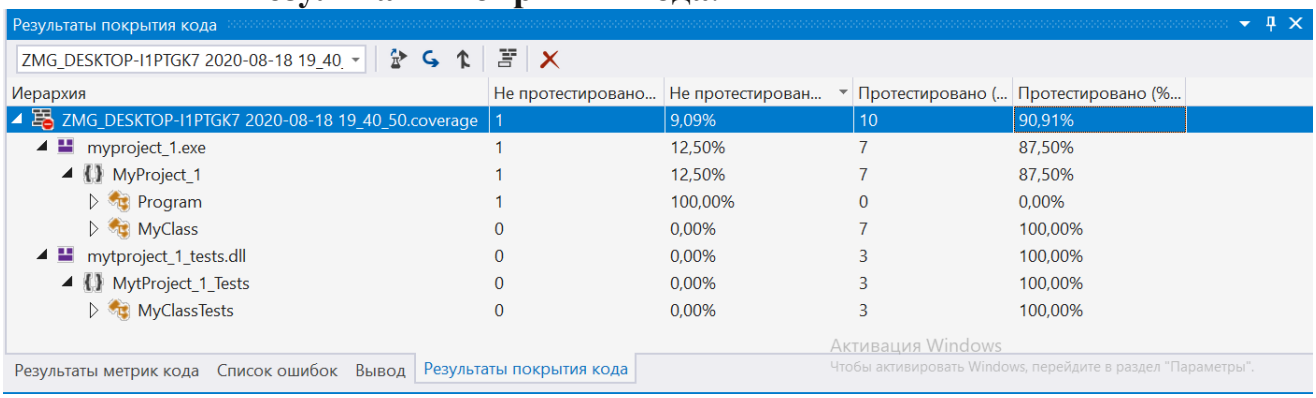
Функция проверки объема протестированного кода доступна только в выпуске Visual Studio Enterprise.

Для анализа покрытия кода необходимо:

7. В меню **Тестирование** выберать **Анализ покрытия кода** для всех тестов.

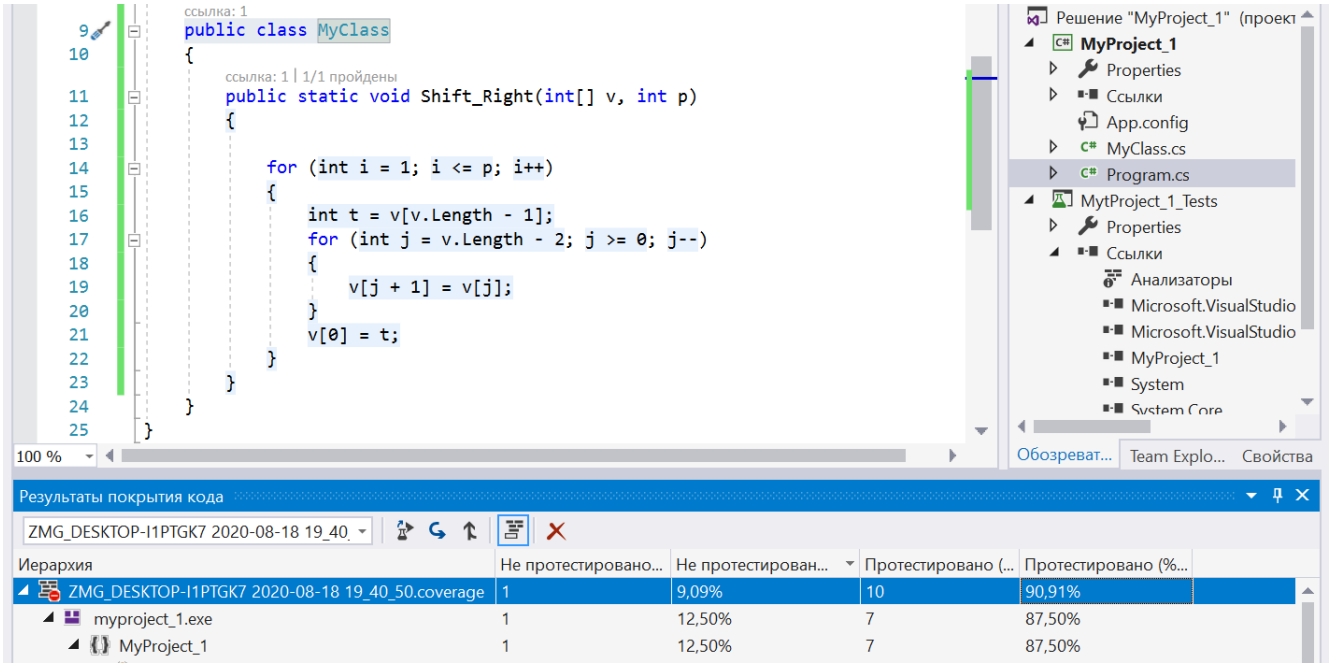


Появится окно **Результаты покрытия кода**.



В нём мы можем просмотреть результат покрытия кода по каждому классу и методу тестируемого проекта.

8. Чтобы после выполнения тестов увидеть, какие строки были выполнены, щелкните Цвета отображения покрытия кода в окне Результаты покрытия кода. По умолчанию код, охватываемый тестами, выделяется голубым цветом.



9. Если результаты показывают низкое покрытие, проверьте, какие части кода не обрабатываются, и напишите несколько дополнительных тестов для их покрытия. Команды разработчиков обычно стремятся покрыть около 80 % кода. В некоторых случаях допустимо более низкое покрытие. Например, более низкое покрытие допустимо, когда некоторый код создается из стандартного шаблона.

Тестирование исключительных ситуаций

Пускай, нам известен следующий пункт спецификации требований для функции **Shift_Right**:

Циклический сдвиг массива вправо может осуществляться на число позиций **p**, находящееся в диапазоне от 0 до количества элементов в массиве минус единица. Если **p** выходит за границы указанного диапазона, функция должна выбрасывать исключение.

Для тестирования исключения добавим в класс **MyClass** вложенный класс исключений **invalid_argument_arg_2**, наследующий от библиотечного класса **ArgumentException**, а в функцию **Shift_Right** - оператор возбуждения исключения. Тогда класс MyClass примет следующий вид:

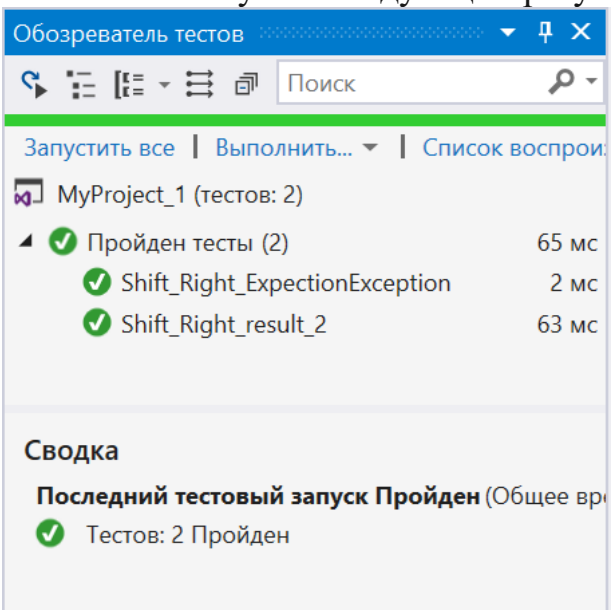
```
public class MyClass
{
    public class invalid_argument_2: ArgumentException
    {
        public invalid_argument_2(string mes): base(mes)
        {
        }
    }
    public static void Shift_Right(int[] v, int p)
    {
        if ((p < 0) | (p > v.Length - 1)) throw new
invalid_argument_2("Неверный параметр сдвига!");
        for (int i = 1; i <= p; i++)
        {
```

```
                int t = v[v.Length - 1];
                for (int j = v.Length - 2; j >= 0; j--)
                {
                    v[j + 1] = v[j];
                }
                v[0] = t;
            }
        }

    }
```

В класс модульного теста добавим метод **Shift_Right_ExpectionException**, проверяющий возникновение исключения заданного в атрибуте **ExpectedException**, класса

```
        [TestMethod]
        [ExpectedException(typeof(MyClass.invalid_argument_2))]
        public void Shift_Right_ExpectionException()
        {
            //arrange(обеспечить)
            int[] result = new int[] { 1, 2 };//Исходный массив.
            int p = 2;
            int[] expected = new int[] { 2, 1 };//Ожидаемое значение.
            //act(выполнить)
            MyClass.Shift_Right(result, p);
        }
```

Запустим тесты на выполнение и получим следующий результат:



Окно **Обозреватель тестов** показывает, что тесты успешно пройдены.

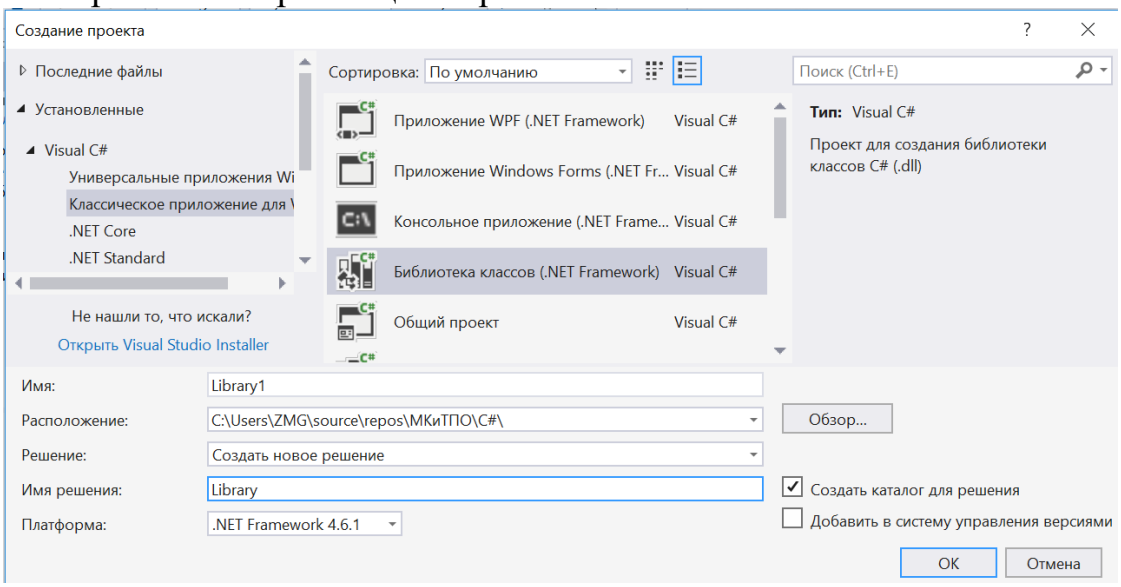
4.3.3. Модульное тестирование библиотеки классов

Тестирование библиотеки классов можно выполнять двумя способами. Первый состоит в том, что мы создаём проект модульного теста в составе того же решения, в которое включена библиотека классов. Второй состоит в том, что мы создаём проект модульного теста в составе отдельного решения. Будем создавать проект модульного теста первым способом.

Создание проекта Библиотека классов C#.

Открываем Visual Studio. Выполняем команды **Файл > Создать > Проект**.

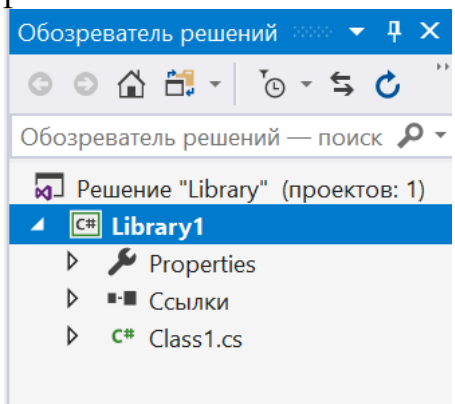
В появившемся окне **Создание проектов** именуем проект Library1, решение – Library и выбираем место размещения решения.



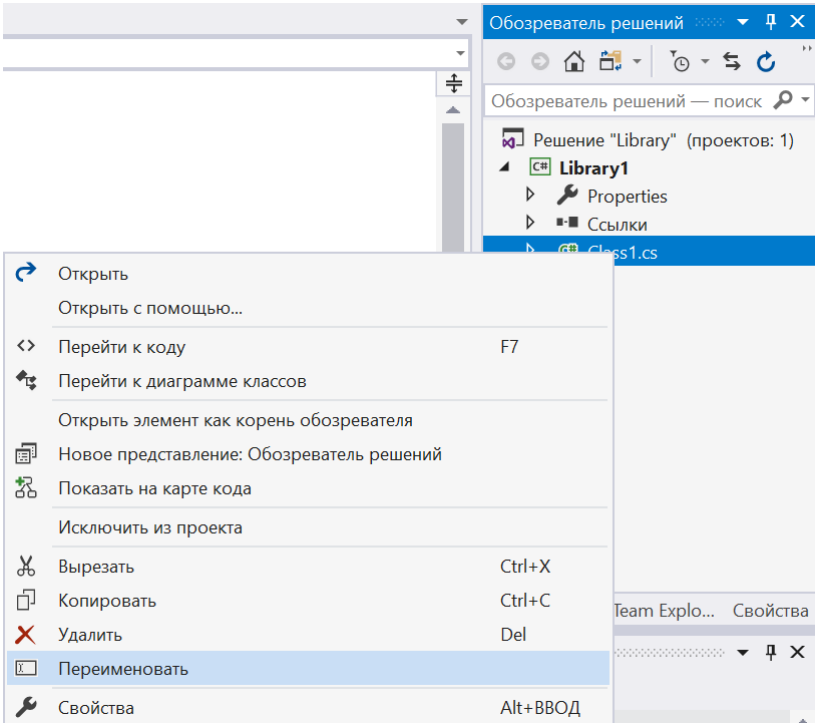
Открывается окно редактора, в котором находится заготовка библиотечного класса Class1.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Library1
{
    public class Class1
    {
    }
}
```

В окне **Обозреватель решений** видно, что в составе библиотеки имеется единственный класс Class1 в файле Class1.cs.

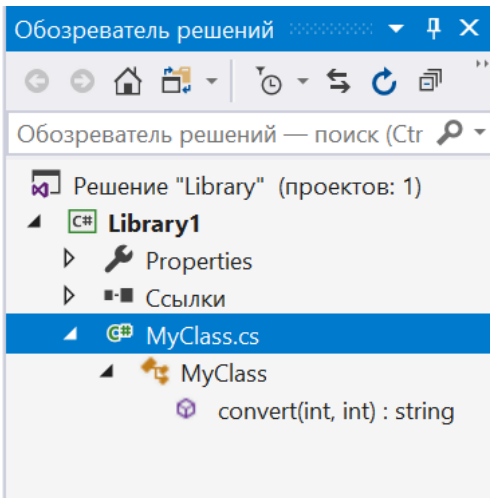


Используя окно **Обозреватель решений** переименуем Class1 в MyClass. Для этого необходимо на имени MyClass.cs кликнуть правой кнопкой мыши и выбрать команду **Переименовать**.



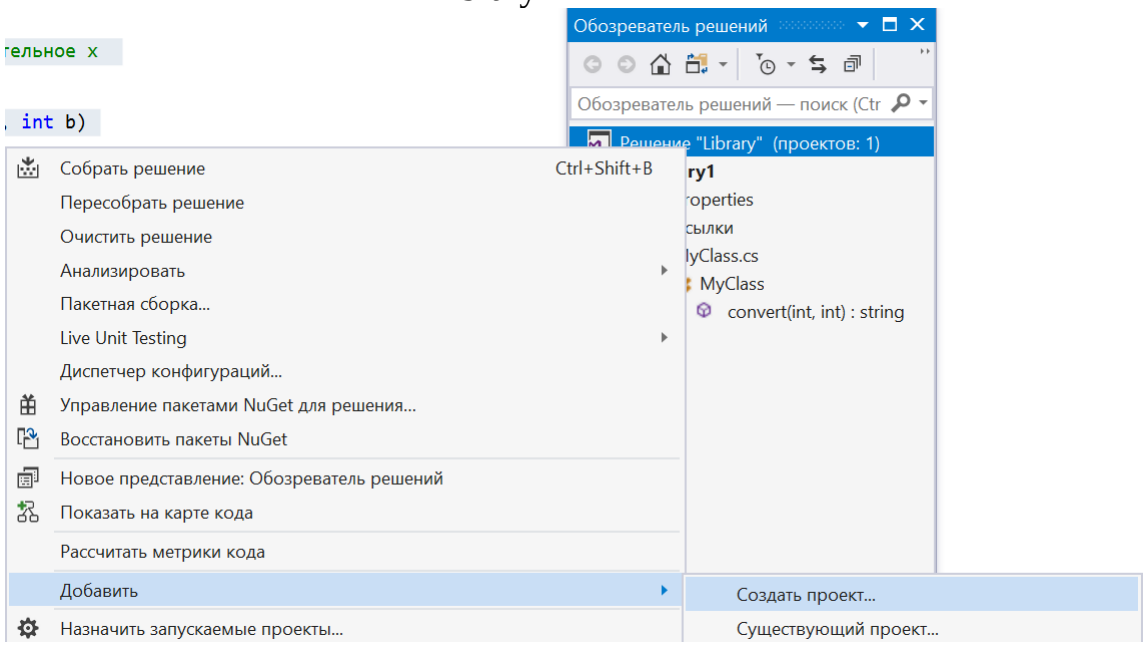
Добавим в этот класс статическую функцию convert, которая переводит целое неотрицательное x в систему счисления с основание b.

```
namespace Library1
{
    public class MyClass
    {
        /*  Метод переводит целое неотрицательное x
   в систему счисления с основание b */
        static public string convert(int x, int b)
        {
            int d;
            char ch;//Текущий разряд в системе счисления с основанием b.
            string s = "";//Строка результата.
            while (x > 0)
            {
                d = x % b;
                if (d <= 9)
                    ch = (char)(d + '0');
                else ch = (char)(d - 10 + 'A');
                s = ch.ToString() + s;
                x = x / b;
            }
            if (s == "")
                s = "0";
            return s;
        }
    }
}
```

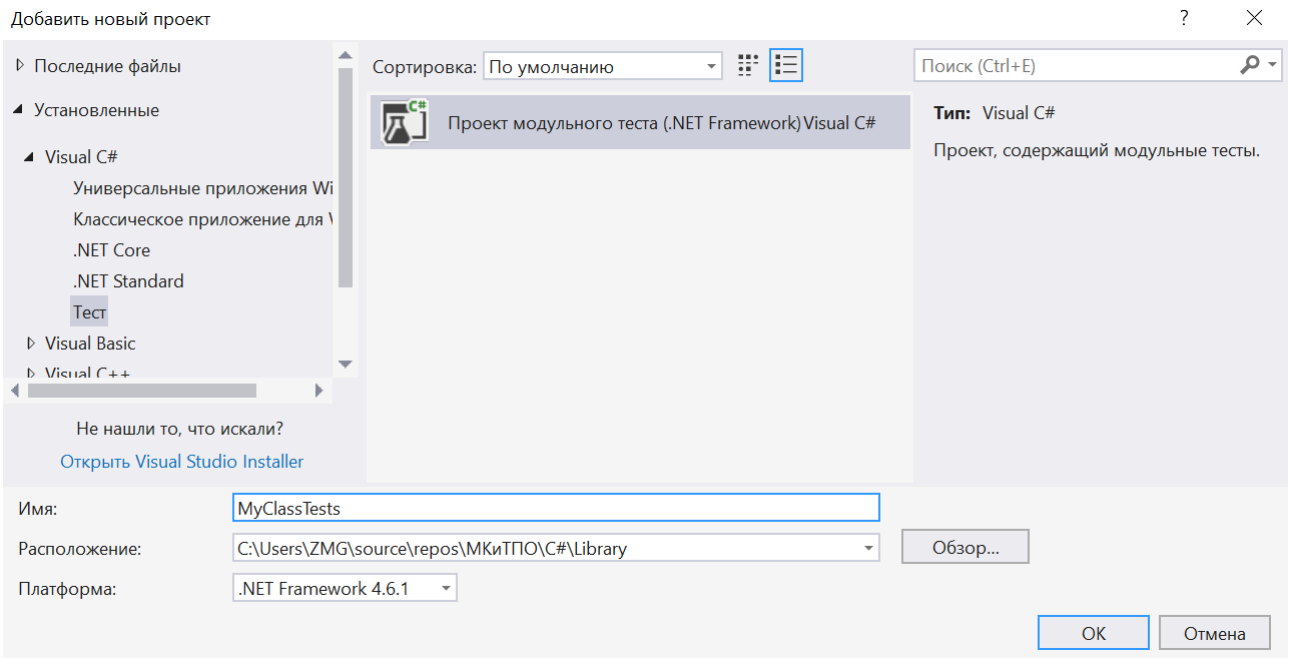


Создание приложения модульного теста для тестирования функции convert.

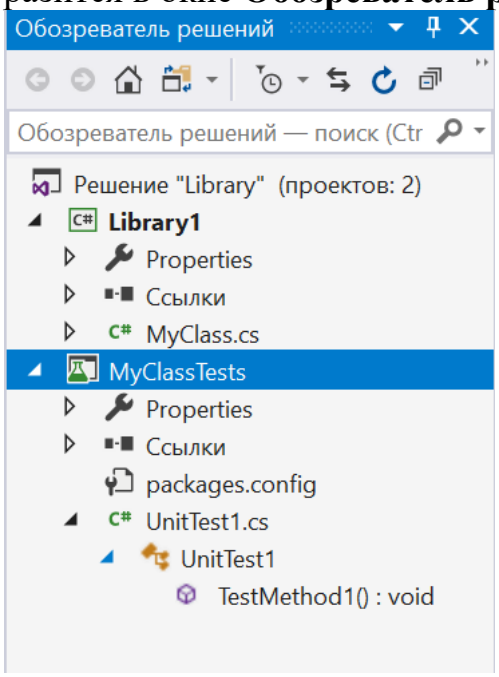
Чтобы протестировать эту функцию, добавим в наше решение **Проект модульного теста**. Для этого в окне **Обозреватель решений** кликнем правой кнопкой мыши на Решение «Library»



и выполним команды меню: **Добавить > Создать проект…** Выберем в пункте **Тест Проект модульного теста** и поименуем его MyClassTests.



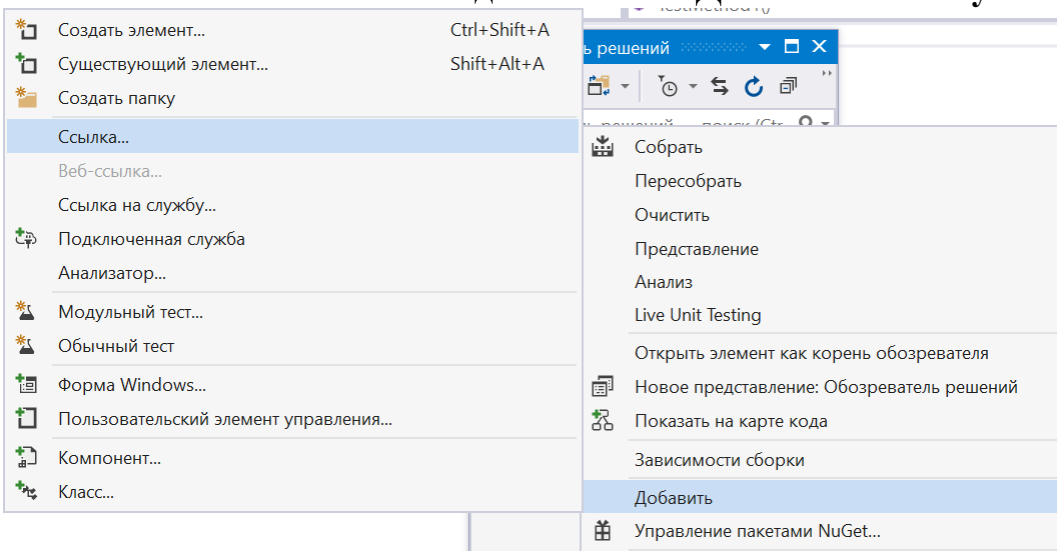
Добавленный проект отобразится в окне **Обозреватель решений**.



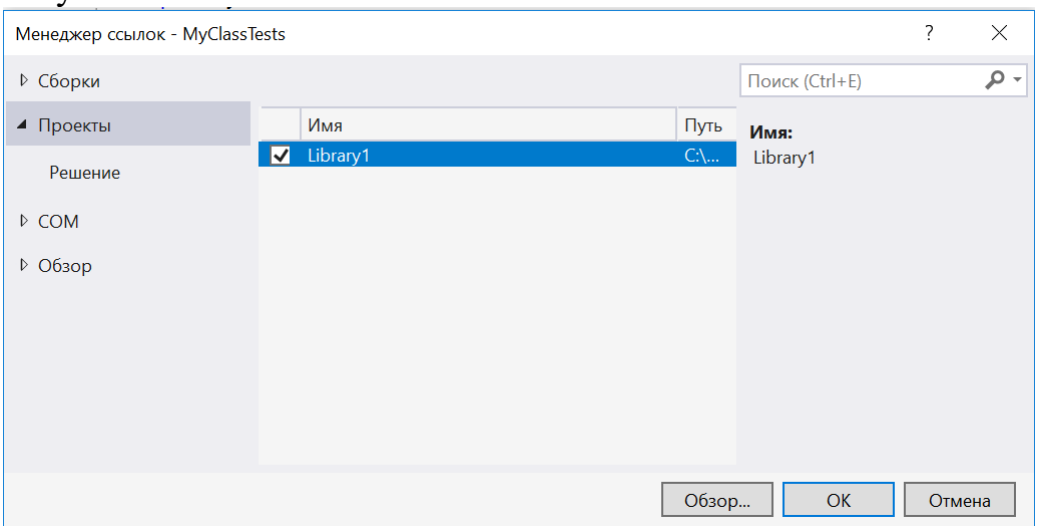
А в окне редактора откроется файл UnitTest1.cs со следующим кодом:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace MyClassTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

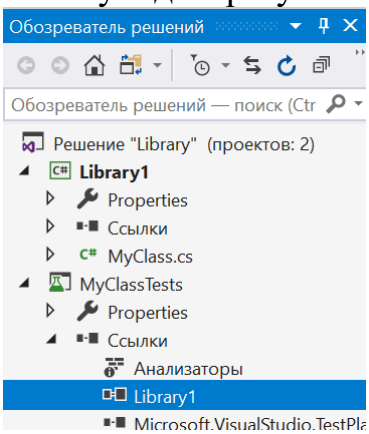
Теперь к нашему проекту MyClassTests необходимо подключить в окне **Обозреватель решений** ссылку на проект Library1. Для этого необходимо кликнуть правой кнопкой мыши на команде **Ссылки > Добавить ссылку**.



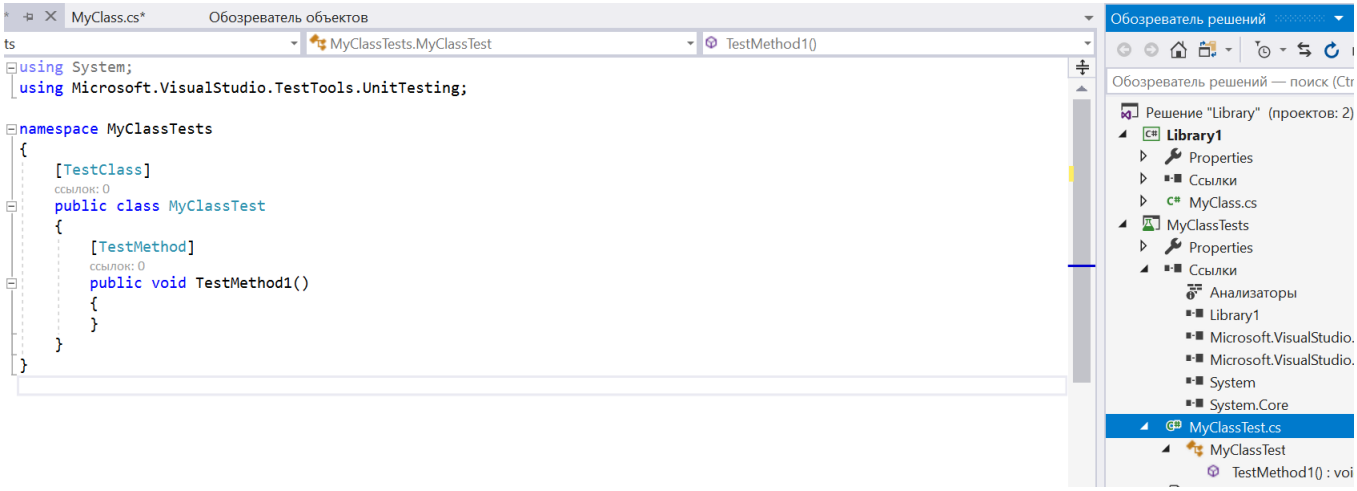
В появившемся окне необходимо поставить галочку слева от Library1 и нажать кнопку OK.



В окне **Обозреватель решений** мы увидим результат:



В окне **Обозреватель решений** переименуем файл UnitTest1.cs в MyClassTests.cs. Вместе с файлом будет переименован и класс, содержащийся в нём.



Теперь в файл MyClassTests.cs добавим предложение using Library1;

После этого в методах тестового класса MyClassTests станут доступны методы библиотечного класса MyClassTests.

Разработка тестового набора для тестирования метода convert.

Для тестирования этого метода необходимо разработать тестовый набор данных, построенный по одному из критериев тестирования. Выберем для построения тестового набора структурный критерий тестирования (C1). Проанализировав УГП метода, представленный на Рис. 2.1, можно выделить следующие ветви: (3, 4), (4, 5, 6), (6, 7, 9, 4), (6, 8, 9, 4), (4, 10), (10, 11, 12), (10, 12).

Тестовый набор из двух тестов, удовлетворяет критерию ветвей (C1):

(X, Y) = {(x = 0, b = 2, "0"),(x = 161, b = 16, "A1")} покрывает все ветви: (3, 4), (4, 5, 6), (6, 7, 9, 4), (6, 8, 9, 4), (4, 10), (10, 11, 12), (10, 12).

Реализуем оба теста на C# и поместим созданный код в тестовые методы с именами covert_0_2_returned_0, covert_161_16_returned_A1.

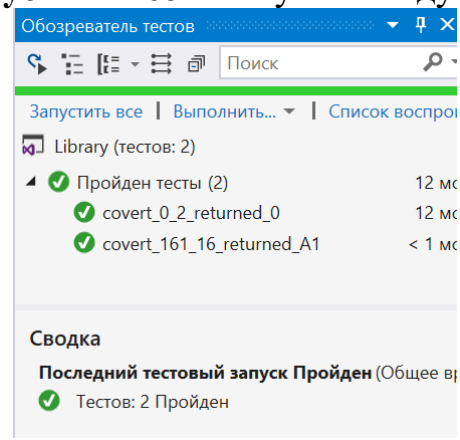
Переименуем в классе MyClassTests метод TestMethod1 на covert_0_2_returned_0. И добавим в тестовый класс новый метод под именем covert_161_16_returned_A1. Тогда текст тестового класса примет следующий вид:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Library1;
namespace MyClassTests
{
    [TestClass]
    public class MyClassTest
    {
        [TestMethod]
        public void covert_0_2_returned_0()
        {
            //arrange(обеспечить)
            string expected = "0";
            //act(выполнить)
            string actual = MyClass.convert(0, 2);
            //assert(доказать)
            Assert.AreEqual(expected, actual);
        }
        [TestMethod]
        public void covert_161_16_returned_A1()
```

```
        {
            //arrange(обеспечить)
            string expected = "A1";
            //act(выполнить)
            string actual = MyClass.convert(161, 16);
            //assert(доказать)
            Assert.AreEqual(expected, actual);
        }
    }
}
```

Здесь функция AreEqual класса Assert сравнивает два значения expected (ожидаемое) и actual (возвращаемое) и возвращает значение true, если они равны.

Для выполнения тестов откроем окно **Обозреватель тестов**, выполнив команды **Тестирование > Выполнить > Все тесты**. В открывшемся окне **Обозреватель тестов** выберем команду **Запустить все** и получим следующий результат:



Зелёный кружок показывает, что тест выполнен и полученный результат соответствует ожидаемому значению.

Тестирование исключительных ситуаций

Пускай, нам известен следующий пункт спецификации требований для функции convert:

Основание системы счисления b должно находиться в диапазоне [2..16].

Число x должно находиться в диапазоне [0..255]. Если x или b выходят за границы указанных диапазонов, функция должна выбрасывать исключения.

Нам понадобятся два тестовых набора для тестирования выхода x за границы диапазона и два тестовых набора для тестирования выхода b за границы диапазона. Это такие тесты:

(x = -1, b = 2, исключение invalid_argument_1),(x = 300, b = 16, исключение invalid_argument_1), (x = -10, b = 1, исключение invalid_argument_2),(x = 200, b = 17, исключение invalid_argument_2).

Для тестирования исключения добавим в класс MyClass вложенные классы исключений invalid_argument_arg_1, invalid_argument_arg_2, наследующие от библиотечного класса ArgumentException, а в функцию convert – операторы возбуждения исключений. Тогда класс MyClass примет следующий вид:

```
namespace Library1
{
    public class MyClass
```

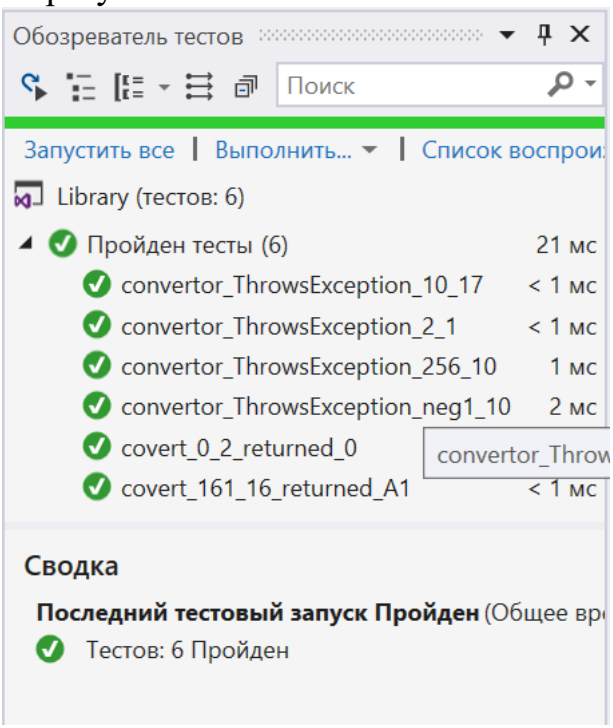
```
    {
        public class invalid_argument_1 : ArgumentException
        {
            public invalid_argument_1(string mes) : base(mes)
            {
            }
        }
        public class invalid_argument_2 : ArgumentException
        {
            public invalid_argument_2(string mes) : base(mes)
            {
            }
        }
        /*  Метод переводит целое неотрицательное x
   в систему счисления с основание b */
        static public string convert(int x, int b)
        {
            string s1 = "Ошибочное число. Функция convert.";
            string s2 = "Ошибка в осноавинии с. сч.. Функция convert.";
            if ((x < 0) || (x > 255)) throw new invalid_argument_1(s1);
            if ((b < 2) || (b > 16)) throw new invalid_argument_2(s2);
            int d;
            char ch;//Текущий разряд в системе счисления с основанием b.
            string s = "";//Строка результата.
            while (x > 0)
            {
                d = x % b;
                if (d <= 9)
                    ch = (char)(d + '0');
                else ch = (char)(d - 10 + 'A');
                s = ch.ToString() + s;
                x = x / b;
            }
            if (s == "")
                s = "0";
            return s;
        }
    }
}
```

Теперь добавим в тестовые методы код реализующий тесты для тестирования исключений, возникающих при недопустимых значения x и b.

```
        [TestMethod]
        [ExpectedException(typeof(MyClass.invalid_argument_1))]
        public void convertor_ThrowsException_neg1_10()
        {
            //arrange(обеспечить)
            //act(выполнить)
            string actual = MyClass.convert(-1, 10);
            //assert(доказать)
        }
        [TestMethod]
        [ExpectedException(typeof(MyClass.invalid_argument_1))]
        public void convertor_ThrowsException_256_10()
        {
            //arrange(обеспечить)
```

```
        //act(выполнить)
        string actual = MyClass.convert(256, 10);
        //assert(доказать)
    }
    [TestMethod]
    [ExpectedException(typeof(MyClass.invalid_argument_2))]
    public void convertor_ThrowsException_2_1()
    {
        //arrange(обеспечить)
        //act(выполнить)
        string actual = MyClass.convert(2, 1);
        //assert(доказать)
    }
    [TestMethod]
    [ExpectedException(typeof(MyClass.invalid_argument_2))]
    public void convertor_ThrowsException_10_17()
    {
        //arrange(обеспечить)
        //act(выполнить)
        string actual = MyClass.convert(10, 17);
        //assert(доказать)
      }
```

Для выполнения теста в окне **Обозреватель тестов** выберем команду **Запустить все**. Получим следующий результат:

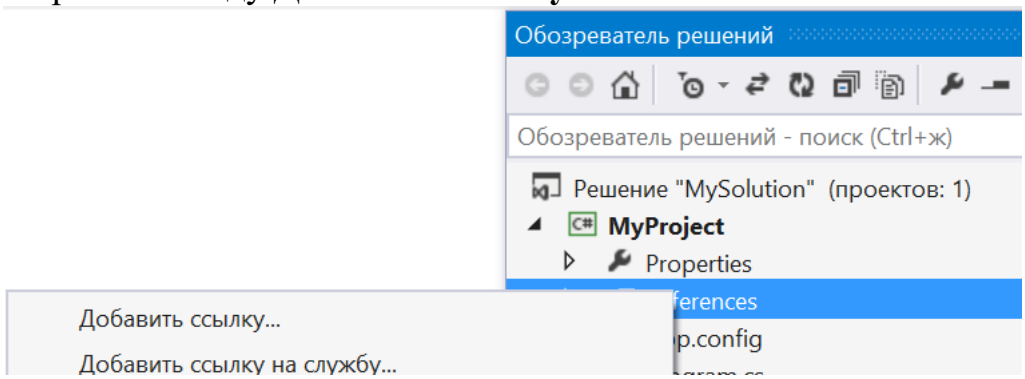


Зелёный кружок показывает, что тест выполнен и полученный результат соответствует ожидаемому значению.

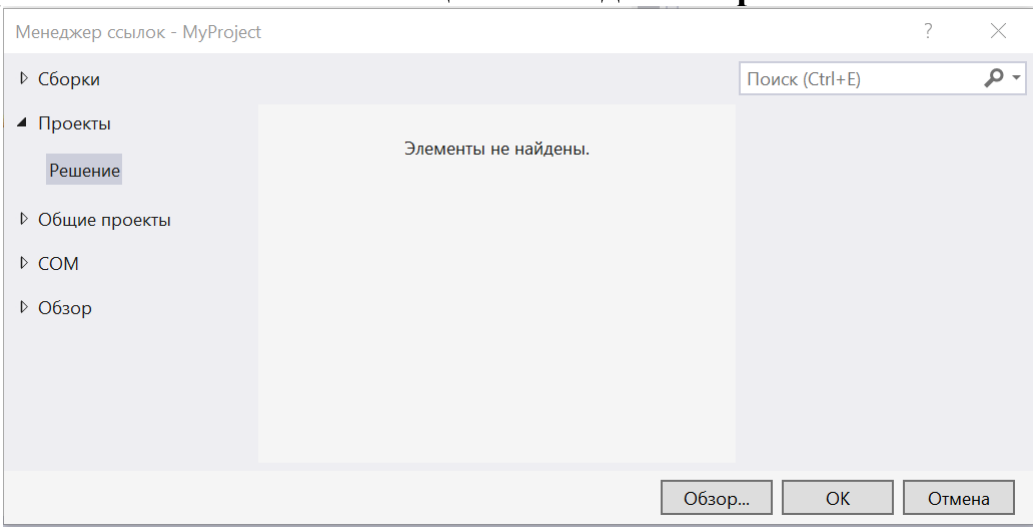
Подключение библиотеки классов к консольному приложению.

Создадим новое решение MySolution и включим в его состав проект консольное приложение MyProject. Подключим к нему библиотечный класс MyClass, чтобы можно было использовать метод convert. Для этого к проекту MyProject необходимо подключить в окне **Обозреватель решений** ссылку на проект библиотеки классов

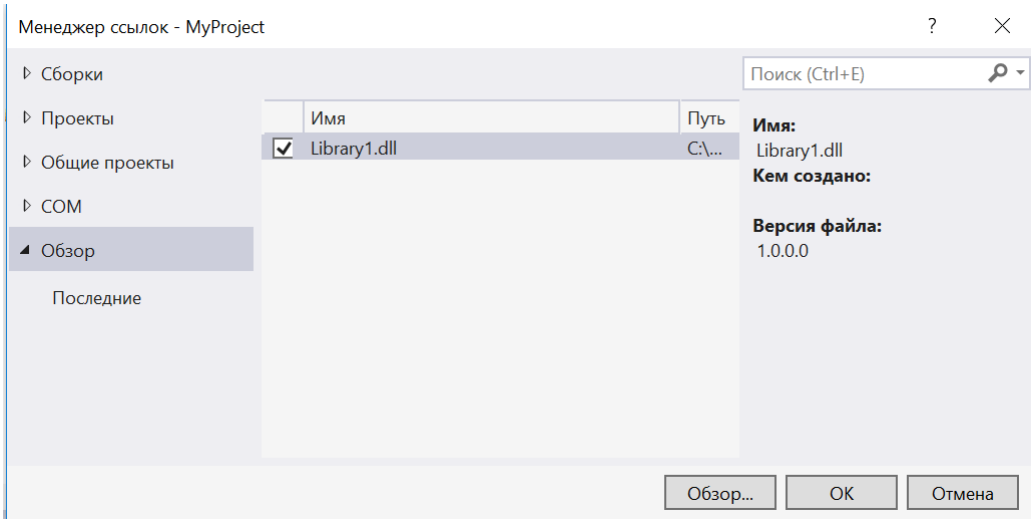
Library1. Для этого необходимо кликнуть правой кнопкой мыши на разделе **Ссылки** и выбрать команду **Добавить ссылку**.



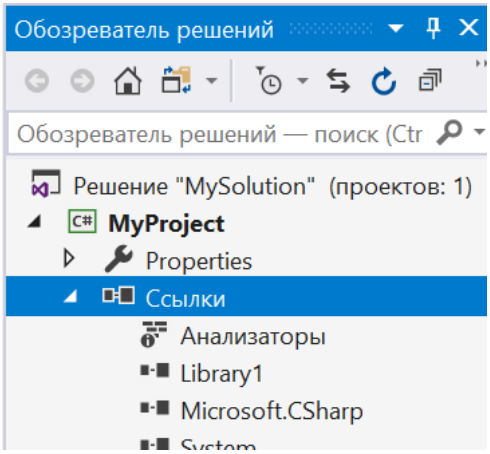
В появившемся окне **Менеджер ссылок** – MyProject, выберем **Проекты>Решения**. С помощью команды **Обзор**



необходимо найти папку, в которой находится файл Library1.dll (путь к файлу: Library>Library1>bin>Debug> Library1.dll) и выбрать его. Наш выбор отобразится в окне **Менеджер ссылок** – MyProject.



Нажимаем кнопку OK и окно **Обозреватель решений** отразит подключение библиотеки.



Добавим предложение uses Library1 в файл Program.cs проекта MyProject. Добавим в функцию Main код вызова функции convert. Вызываем функцию MyClass.convert, указывая имя класса и имя функции.

```
using Library1;
namespace MyProject
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int x = 127, b = 8;
                string y = MyClass.convert(x, b);
                Console.WriteLine("convert({0}, {1}) = {2}", x, b, y);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.WriteLine("Продолжение работы программы.");
        }
    }
}
```

Запустив нашу программу на выполнение, получим следующий результат:

```
convert(127, 8) = 177
Продолжение работы программы.
Для продолжения нажмите любую клавишу . . .
```

Если мы поменяем значение x = 3 на ошибочное x = -1, возникнет исключение. И мы получим следующий результат работы программы:

```
Ошибочное число. Функция convert.
Продолжение работы программы.
Для продолжения нажмите любую клавишу . . .
```

4.4. Создание модульных тестов на C++

В этом разделе описывается создание проекта модульного теста на языке C++.

4.4.1. Создание модульного теста для тестирования проекта Консольное приложение Widows

Рассмотрим пример создания проекта модульного теста для тестирования проекта **MyProject** консольного приложения Widows. В составе проекта имеется один класс **MyClass**, текст которого приведён ниже.

Заголовочный файл класса:

```
class MyClass
{
public:

      static double Power(int x, int n);
};
```

Файл определений класса:

```
#include "stdafx.h"
#include "MyClass.h"


double MyClass::Power(int x, int n)
{
      double z = 1; int i;
      for ( i = 1;  i <= n;  i++)
      {
            z = z * x;
      }
      return z;
}
```

Пусть нам необходимо протестировать функцию **Power.** Для этого нам предварительно построить набор тестов, на которых мы будем осуществлять тестирование.

Если в нашем распоряжении имеется спецификация требований для функции **Power,**.мы можем разработать тестовый набор на основе спецификации, выделяя области эквивалентности входных параметров.

Спецификация требований к функции Power

На вход функция принимает два параметра: x - число, n – степень. Результат вычисления выводится в месте вызова.

Значения числа и степени должны быть целыми.

Значения числа, возводимого в степень, должны лежать в диапазоне – [0..999].

Значения степени должны лежать в диапазоне – [1..100].

Если числа, подаваемые на вход, лежат за пределами указанных диапазонов, то должно возбуждаться исключение.

Разработка тестов

Будем разрабатывать тестовый набор на основе покрытия тестами областей эквивалентности входных параметров.

Определим области эквивалентности входных параметров.

Для x – числа, возводимого в степень, определим классы возможных значений:

1. x < 0 (ошибочное)
2. x > 999 (ошибочное)
3. 0 <= x <= 999 (корректное)

Для n – степени числа:

1. n < 1 (ошибочное)
2. n > 100 (ошибочное)
3. 1 <= n <= 100 (корректное)

Анализ тестовых случаев.

1. Входные значения: (x = 2, n = 10) (покрывают классы 3, 6).

Ожидаемый результат: 1024.

2. Входные значения: {(x = -1, n = 2),(x = 1000, n = 5)} (покрывают классы 1, 2).

Ожидаемый результат: исключение класса invalid_argument_1 с сообщением «Ошибка : x должна принадлежать интервалу [0..999]».

3. Входные значения: {(x = 100, n = 0),(x = 100, n = 200)} (покрывают классы 4, 5).

Ожидаемый результат: исключение класса invalid_argument_2 с сообщением «Ошибка : n должна принадлежать интервалу [1..100]».

Проверка на граничные значения:

4. Входные значения: (x = 999, n = 1).

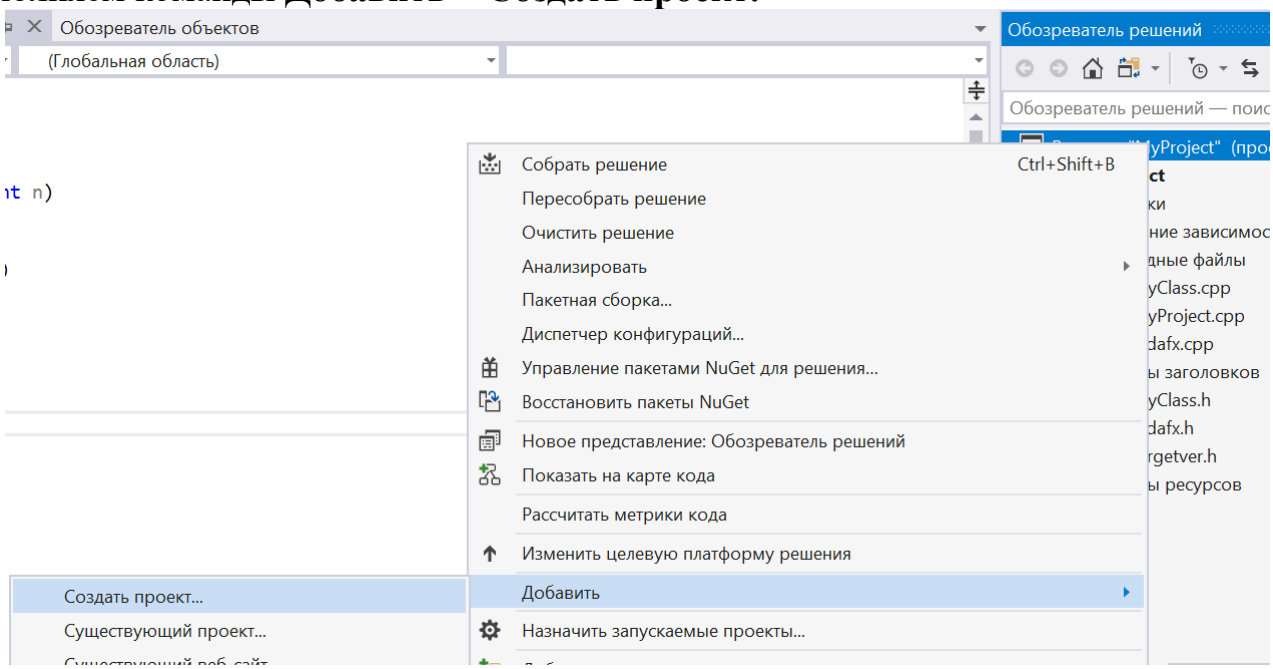
Ожидаемый результат: 999.

5. Входные значения: (x = 0, n = 100).

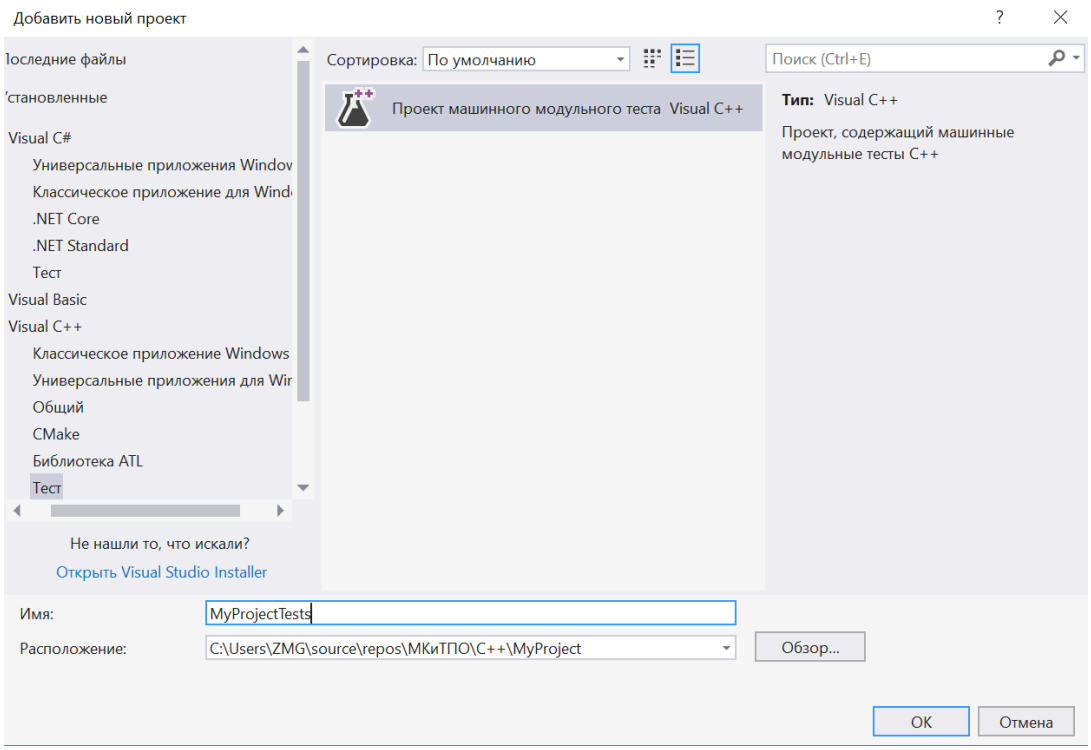
Ожидаемый результат: 0.

Для тестирования класса **MyClass** и функции **Power**, входящей в состав класса нам необходимо добавить в решение **MyProject** проект модульного теста **MyProjectTests**. Это можно сделать следующим образом:

В Обозревателе решений устанавливаем курсор на имя решения **MyProject** и выполняем команды **Добавить** > **Создать проект.**



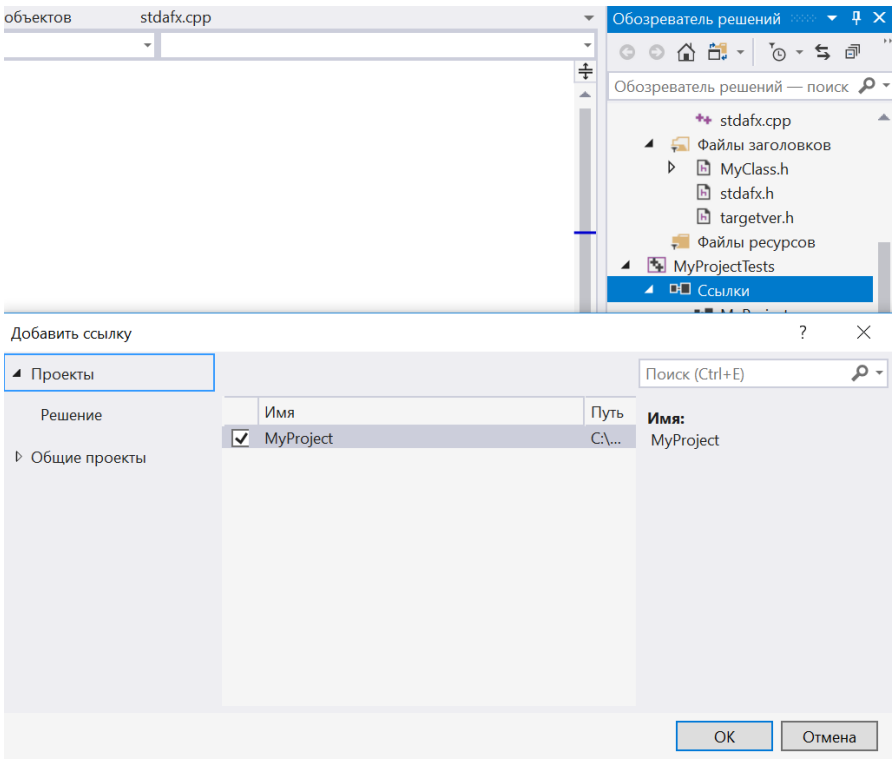
Выбираем **Проект машинного модульного теста** и именуем его **MyProjectTests**.



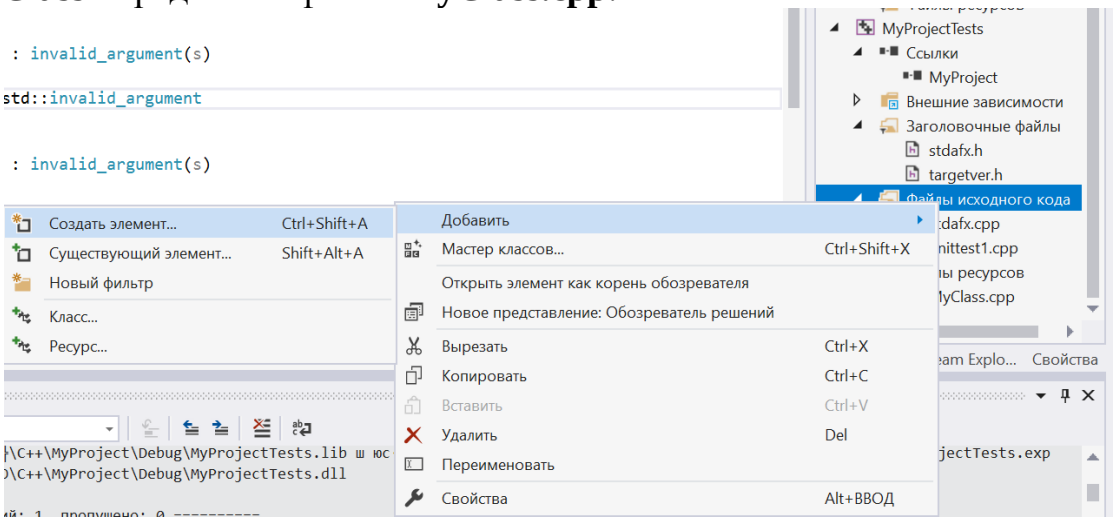
Получаем проект модульного теста с классом модульного теста следующего вида:

```
#include "stdafx.h"
#include "CppUnitTest.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace MyProjectTests
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(TestMethod1)
		{
			// TODO: Разместите здесь код своего теста
		}

	};
}
```

В окне **Обозреватель решений** добавим к проекту **MyProjectTests** ссылку на проект **MyProject**. Для этого выполним команду **Ссылки > Добавить ссылку**.

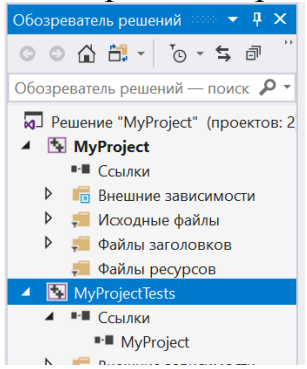


Также добавим к файлам ресурсов ссылку на файл **MyClass.cpp**. Иначе невозможно будет собрать исполняемый файл модульного теста, потому что класс **MyClass** определён в файле **MyClass.cpp**.



Выполняем команды **Добавить > Существующий элемент** и с помощью окна проводника находим файл **MyClass.cpp**.

В обозревателе решений можно видеть результат:



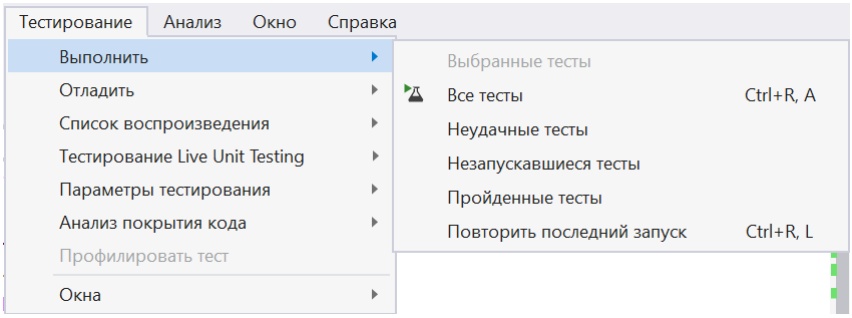
Также необходимо добавить в файл **unittest1.cpp** директиву include с полным путём к заголовочному файлу **MyClass.h**

```
#include "stdafx.h"
#include "CppUnitTest.h"
#include "../MyProject/MyClass.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace MyClassTests
```

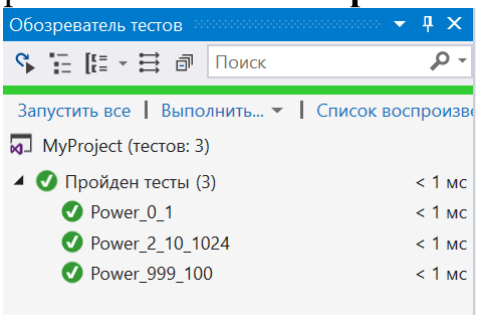
Переименуем тестовый класс в **MyClassTests** и добавим в него три тестовых метода **Power_2_10_1024, Power_999_1_1, Power_0_1_0**. Эти методы реализуют 1, 4, 5 тесты из разработанного тестового набора.

```
    TEST_CLASS(MyClassTests)
    {
    public:

        TEST_METHOD(Power_2_10_1024)
        {
            // arrage(окружение)
            int x = 2;
            int n = 10;
            double expected = pow(x, n);
            // act(действие)
            double actual = MyClass::Power(x, n);
            // assert(доказать)
            Assert::AreEqual(expected, actual);
        }
        TEST_METHOD(Power_999_100)
        {
            // arrage(окружение)
            int x = 999;
            int n = 100;
            MyClass a;
            double expected = pow(x, n);
            // act(действие)
            double actual = a.Power(x, n);
            // assert(доказать)
            Assert::IsTrue(expected = actual);
        }
        TEST_METHOD(Power_0_1)
        {
            // arrage(окружение)
            int x = 0;
            int n = 1;
            double expected = pow(x, n);
            // act(действие)
            double actual = MyClass::Power(x, n);
            // assert(доказать)
            Assert::AreEqual(expected, actual);
        }
    };
}
```

Выполняем тестирование. Выполняем команды **Тестирование > Выполнить > Все тесты.**



Результат тестирования отображается в окне **Обозревателя тестов**.



Для того чтобы реализовать тесты 2, 3 нам необходимо добавить в объявление **MyClass** два класса для обработки исключений, которые будут возникать при выходе параметров за границы допустимых диапазонов. В описание класса добавляем возбуждение исключений при выходе параметров за границы допустимых диапазонов.

Тогда код класса **MyClass** примет следующий вид:

```
#pragma once
#include <stdexcept>
class MyClass
{
public:

	static double Power(int x, int n);
	class invalid_argument_1 : public std::invalid_argument
	{
	public:
		invalid_argument_1(const char* s) : invalid_argument(s)
		{}
	}; class invalid_argument_2 : public std::invalid_argument
	{
	public:
		invalid_argument_2(const char* s) : invalid_argument(s)
		{}
	};
};
```

В класс **MyClass** добавлены два класса исключений **invalid_argument_1, invalid_argument_2**, наследник библиотечного класса **invalid_argument**.

```
#include "stdafx.h"
#include "MyClass.h"

double MyClass::Power(int x, int n)
{
	const char* s = "Ошибочное основание. Функция Power.";
```

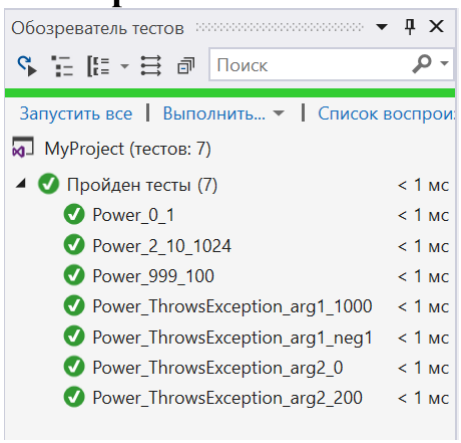
```
    const char* s1 = "Ошибка в показатели степени. Функция Power.";
    if ((x < 0) || (x > 999)) throw invalid_argument_1(s);
    if ((n < 1) || (n > 100)) throw invalid_argument_2(s1);
    double z = 1; int i;
    for ( i = 1;  i <= n;  i++)  { z = z * x; }
    return z;
}
```

В функцию **Power** добавлены операторы возбуждения классов исключений **invalid_argument_1, invalid_argument_2**.

Теперь добавим тестовые методы, реализующие тесты из тестового набора для тестирования исключений, возникающих при недопустимых значения x и n.

```
TEST_METHOD(Power_ThrowsException_arg1_neg1)
{
      // arrage(окружение)
      int x = -1;
      int n = 2;
      // act(действие)
      auto func = [ x, n] { MyClass::Power(x, n); };
      // assert (доказать)
      Assert::ExpectException<MyClass::invalid_argument_1>(func);
}
TEST_METHOD(Power_ThrowsException_arg1_1000)
{
      // arrage(окружение)
      int x = 1000;
      int n = 5;
      MyClass a;
      // act(действие)
      auto func = [a, x, n] { a.Power(x, n); };
      // assert (доказать)
      Assert::ExpectException<MyClass::invalid_argument_1>(func);
}
TEST_METHOD(Power_ThrowsException_arg2_0)
{
      // arrage(окружение)
      int x = 100;
      int n = 0;
      // act(действие)
      auto func = [x, n] { MyClass::Power(x, n); };
      // assert (доказать)
      Assert::ExpectException<MyClass::invalid_argument_2>(func);
}
TEST_METHOD(Power_ThrowsException_arg2_200)
{
      // arrage(окружение)
      int x = 100;
      int n = 200;
      MyClass a;
      // act(действие)
      auto func = [a, x, n] { a.Power(x, n); };
      // assert (доказать)
      Assert::ExpectException<MyClass::invalid_argument_2>(func);
}
```

Запускаем тестирование и просматриваем результат прохождения всех тестов в окне **Обозреватель тестов**:



Все тесты успешно пройдены.

Контрольные вопросы

1. Как создать проект модульного теста?
2. Куда можно включить проект модульного теста для тестирования классов проекта?
3. Как запустить тесты на выполнение?
4. Как просмотреть объём кода, покрытого тестами?
5. Как просмотреть покрытые тестами блоки кода?
6. Для чего предназначено окно **Обозреватель тестов**?
7. Для чего предназначено окно **Результаты покрытия кода**?
8. Как подключить тестируемыq проект к модульному тесту на C#?
9. Как подключить тестируемый проект к модульному тесту на C++?
10. Как подключить библиотеку классов к проекту C#?

Приложение. Практические задания для закрепления

Практическая работа №1. Модульное тестирование программ на языке C# средствами Visual Studio

Цель: сформировать практические навыки разработки модульных тестов для тестирования функций классов и выполнения модульного тестирования на языке C# с помощью средств автоматизации Visual Studio.

Задание

Разработайте на языке C# класс, содержащий функции в соответствии с вариантом задания.

Разработайте тестовые наборы данных по критерию C0 для тестирования функций класса.

Протестируйте созданный класс с помощью средств автоматизации модульного тестирования Visual Studio.

Напишите отчёт о результатах проделанной работы.

Варианты заданий

Вариант №	Спецификация функци
1	Функция получает одномерный массив вещественных переменных. Вычисляет и возвращает произведение значений компонентов массива с нечетными значениями индексов.
	Функция получает одномерный массив вещественных переменных и целое – параметр сдвига. Функция изменяет массив циклическим сдвигом значений его элементов вправо на число позиций, равное параметру сдвига.
	Функция получает целое число b – основание системы счисления и строку s, содержащую представление дробной части числа в системы счисления с основанием b. Функция формирует и возвращает из строки s целое число.
2	Функция получает два одномерных массив a, b одинаковой длины. Возвращает массив с полученный суммированием значений массивов a,b.
	Функция получает одномерный массив вещественных переменных и целое – параметр сдвига. Функция изменяет массив циклическим сдвигом значений его элементов влево на число позиций, равное параметру сдвига.
	Функция находит и возвращает индекс начала первого вхождения последовательности целых чисел, представленных массивом int[] seq в другую последовательность, представленную массивом int[] vec.
3	Функция получает два одномерных массива. В массиве int[] ind хранятся индексы компонентов массива v. Найдите произведение ненулевых компонентов массива v, индексы которых хранятся в массиве ind.
	Функция получает одномерный массив целых переменных. Вычисляет и возвращает минимальный по значению элемент этого массива и номер его индекса.
	Функция получает одномерный массив вещественных переменных. В полученном массиве функция переставляет значения компонентов массива в обратном порядке.
4	Функция получает целое число b – основание системы счисления и строку s, содержащую представление целой части числа в системы счисления с основанием b. Функция формирует и возвращает из строки s целое число.
	Функция получает одномерный массива целых переменных. Вычисляет и возвращает максимальный по значению элемент этого массива и номер его индекса.
	Функция получает одномерный массив целых переменных. Вычисляет и возвращает максимальное значение среди нечётных элементов массива с нечётными значениями индекса и значение индекса (через параметр)
5	Функция получает одномерный массив вещественных переменных. Вычисляет и возвращает произведение значений компонентов массива с чётными значениями индексов.
	Функция получает одномерный массив вещественных переменных и целое – параметр сдвига. Функция изменяет массив циклическим сдвигом его элементов на заданное число позиций, равное параметру сдвига в

	указанном направлении.
	Функция получает одномерный массив целых переменных. Вычисляет и возвращает максимальное значение среди чётных элементов массива с чётными значениями индекса и значение индекса (через параметр).

Рекомендации к выполнению

1. Создайте проект Консольное приложение, и добавить в него класс.
2. Разработайте и включить в класс статические функции в соответствии с вариантом задания.
3. Постройте УГП и разработайте тестовые наборы данных для тестирования функций разработанных по критерию С0.
4. Создайте проект «Модульный тест» свяжите его с проектом «Консольное приложение».
5. Добавить в проект «Модульный тест» код тестовых методов, реализующих разработанные тесты.
6. Выполните модульный тест с помощью окна «Обозреватель тестов».
7. Проведите анализ выполненного теста и, если необходимо отладку кода.
8. Проанализируйте результаты выполненных тестов по объёму покрытия тестируемого кода с помощью окна «Результаты покрытия кода».
9. Составьте отчёт о результатах проделанной работы. Содержание отчёта приводится ниже.

Содержание отчета

1. Задание.
2. УГП и тестовые наборы данных для тестирования функций класса.
3. Исходные тексты программ на языке C#.
4. Результаты выполнения модульных тестов.
5. Результаты покрытия разработанного кода тестами.
6. Выводы по выполненной работе.

Контрольные вопросы

1. Проектирование модульных тестов.
2. Критерий тестирования команд С0.
3. Что такое УГП?
4. Как определена «ветвь УГП»?
5. Как определен «путь УГП»?
6. Этапы модульного тестирования.
7. Состав и структура тестового класса.
8. Что может содержать проект модульного теста?
9. Как рекомендуется именовать тестовые классы?
10. Как рекомендуется именовать тестовые методы?
11. Как рекомендуется организовывать операторы тестового метода?
12. Как сделать видимым тестируемый класс в модульном тесте?

Практическая работа №2. Модульное тестирование библиотеки классов на C# средствами Visual Studio.

Цель: сформировать практические навыки разработки модульных тестов для библиотек классов C# и выполнения модульного тестирования с помощью средств автоматизации Visual Studio.

Задание

Разработайте на языке C# класс, содержащий функции в соответствии с вариантом задания.

Разработайте тестовые наборы данных для тестирования функций класса, по критерию C1.

Протестируйте созданный класс с помощью средств автоматизации модульного тестирования Visual Studio.

Проанализируйте результаты выполненных тестов по объёму покрытия тестируемого кода.

Напишите отчёт о результатах проделанной работы.

Рекомендации к выполнению

1. Создайте проект Библиотека классов, содержащую класс статических функций на языке программирования C#.
2. Реализуйте статические функции в соответствии с вариантом задания и поместить их в класс.
3. Постройте УГП и разработайте тестовые наборы данных по критерию C1 для тестирования функций.
4. Создайте проект Модульный тест свяжите его с проектом Библиотека классов.
5. Добавить в проект Модульный тест код тестовых методов, реализующих разработанные тесты.
6. Выполните модульный тест с помощью окна Обозреватель тестов.
7. Проведите анализ выполненного теста и, если необходимо отладку кода.
8. Проанализируйте результаты выполненных тестов по объёму покрытия тестируемого кода с помощью окна Результаты покрытия кода.
9. Составьте отчёт о результатах проделанной работы. Содержание отчёта приводится ниже.

Варианты задания.

Вариант №	**Функции**
1	Поиск минимума из двух чисел
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает максимальное значение компонентов массива.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает максимальное значение компонентов массива, лежащих на и выше побочной диагонали
2	Поиск максимума из двух чисел
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму значений компонентов массива, у

	которых сумма значений индексов равна максимальному значению второго индекса (индекса столбца). Пояснение: Сумма = A[0,2] + A[2,0] + A[1,1].
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает минимальное значение компонентов массива, лежащих на и выше побочной диагонали
3	Поиск минимума из трёх чисел
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму значений компонентов массива, у которых сумма значений индексов – чётная.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает максимальное значение компонентов массива, лежащих на и ниже главной диагонали
4	Поиск максимума из трёх чисел
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает произведение значений компонентов массива, у которых сумма значений индексов – чётная.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает минимальное значение компонентов массива, лежащих на и ниже главной диагонали
5	Упорядочивает два числа в порядке убывания
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает произведение чётных значений компонентов массива.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму чётных компонентов массива, лежащих на и выше побочной диагонали
6	Упорядочивает два числа в порядке возрастания
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму чётных значений компонентов массива.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму нечётных компонентов массива, лежащих на и выше побочной диагонали
7	Упорядочивает три числа в порядке убывания
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму нечётных значений компонентов массива.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму нечётных значений компонентов массива, лежащих на и ниже главной диагонали
8	Упорядочивает три числа в порядке возрастания
	Функция получает двумерный массив вещественных переменных A.

	Отыскивает и возвращает произведение значений компонентов массива, у которых сумма значений индексов – нечётная.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму нечётных значений компонентов массива, лежащих на и выше главной диагонали

Содержание отчета

1. Задание.
2. Тестовые наборы данных для тестирования класса.
3. Исходные тексты программ на языке C#.
4. Результаты выполнения модульных тестов.
5. Результаты покрытия разработанного кода тестами.
6. Выводы по выполненной работе.

Контрольные вопросы

1. Что входит в состав приложения «библиотека классов»?
2. Как добавить класс в библиотеку классов?
3. Как подключить библиотеку к проекту, находящемуся в том же решении?
4. Как обеспечить доступ к функции библиотечного класса без использования имени пространства имён класса?
5. Как подключить библиотеку к проекту, находящемуся в другом решении?
6. Как просмотреть операторы кода покрытые тестами?

Практическая работа №3. Модульное тестирование программ на языке C++ в среде Visual Studio

Цель: сформировать практические навыки разработки тестов и модульного тестирования на языке C++ с помощью средств автоматизации Visual Studio.

Задание

Разработайте на языке C++ класс, содержащий набор функций в соответствии с вариантом задания.

Разработайте тестовые наборы данных по критерию C2 для тестирования функций класса.

Протестировать функции с помощью средств автоматизации модульного тестирования Visual Studio.

Провести анализ выполненного теста и, если необходимо отладку кода.

Написать отчёт о результатах проделанной работы.

Рекомендации к выполнению

1. Создайте проект Консольное приложение, и добавить в него класс.
2. Разработайте и включить в класс статические функции в соответствии с вариантом задания.
3. Постройте УГП и разработайте тестовые наборы данных для тестирования функций разработанных по критерию C2.

4. Создайте проект Модульный тест свяжите его с проектом Консольное приложение.
5. Добавить в проект Модульный тест код тестовых методов, реализующих разработанные тесты.
6. Выполните модульный тест с помощью окна Обозреватель тестов.
7. Проведите анализ выполненного теста и, если необходимо отладку кода.
8. Составьте отчёт о результатах проделанной работы. Содержание отчёта приводится ниже.

Варианты заданий

Вариант №	**Описание функции**
1	Функция упорядочивает значения переменных x,y,z в порядке убывания их значений, так чтобы x>=y>=z.
	Функция получает два положительных целых числа a и b Вычисляет и возвращает их наибольший общий делитель.
	Функция получает целое числа a. Формирует и возвращает целое число b из значений чётных разрядов целого числа a. Например: a = 12345, b = 24.
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму нечётных значений компонентов массива, лежащих выше главной диагонали
2	Функция находит максимальное из трёх значений целых переменных.
	Функция получает целое числа a. Формирует и возвращает целое число b из значений чётных разрядов целого числа a, следующих в обратном порядке. Например: a = 12345, b = 42.
	Функция получает целое числа a. Находит и возвращает минимальное значение r среди разрядов целого числа a. Например, a = 62543, r = 2.
	Функция получает двумерный массив целых переменных A. Отыскивает и возвращает сумму нечётных значений компонентов массива, лежащих ниже главной диагонали.
3	Функция получает целое числа a. Формирует и возвращает целое число b из значений нечётных разрядов целого числа a, следующих в обратном порядке. Например: a = 12345, b = 531.
	Функция получает целое числа a. Находит и возвращает номер разряда, в котором находится максимальное значение r среди чётных разрядов целого числа a с чётным значением. Разряды числа, пронумерованы справа налево, начиная с единицы. Например, a = 62543, r = 4.
	Функция получает целое числа a. Возвращает число, полученное циклическим сдвигом значений разрядов целого числа a на заданное число позиций вправо. Например, сдвиг на две позиции: Исходное число: 123456

	Результат: 561234
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму чётных значений компонентов массива, лежащих выше побочной диагонали.
4	Функция получает целое числа a. Находит и возвращает номер разряда, в котором находится минимальное значение r среди нечётных разрядов целого числа a с нечётным. Разряды числа, пронумерованы справа налево, начиная с единицы. Например, a = 12543, r = 3.
	Функция получает целое числа a. Возвращает число, полученное циклическим сдвигом значений разрядов целого числа a на заданное число позиций влево. Например, сдвиг на две позиции: Исходное число: 123456 Результат: 345612
	Функция получает целые числа a, b и n. Возвращает число, полученное путём вставки разрядов числа b в целое число a после разряда, заданного числом n. Разряды нумеруются слева направо, начиная с 1. Например, вставить после 2 разряда значение 6: Исходное число: 123457 вставить 6 после 2 разряда Результат: 1263457
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму чётных значений компонентов массива, лежащих ниже побочной диагонали
5	Функция получает целое числа a. Возвращает число, полученное путём циклического сдвига значения разрядов a на заданное число позиций в заданном направлении. Например, сдвиг на две позиции влево: Исходное число: 123456 Результат: 345612
	Функция вычисляет и возвращает число Фибоначчи по его номеру.
	Функция получает целое числа a, p, n. Возвращает число, полученное путём удаления из a, начиная с позиции p, число разрядов n. Разряды нумеруются слева направо. Например, удалить, начиная с 3 разряд числа 123456 два разряда: Исходное число: 123456 Результат: 1256
	Функция получает двумерный массив вещественных переменных A. Отыскивает и возвращает сумму компонентов массива, лежащих выше побочной диагонали с чётной суммой значений индексов.

Содержание отчета

1. Задание.
2. УГП и тестовые наборы данных для тестирования функций класса.

3. Исходные тексты программ на языке C++.
4. Результаты выполнения модульных тестов.
5. Выводы по выполненной работе.

Контрольные вопросы

1. Понятие и цель модульного тестирования.
2. Проектирование модульных тестов.
3. Критерий тестирования C2.
4. Основные атрибуты теста.
5. Класс Assert. Методы.
6. Тестирование исключений.

Глоссарий

«Недостаток» или «Дефект» программы» — причина нарушения работы прикладной программы.

Сбой – наблюдаемый нежелательный эффект, вызываемый недостатком или дефектом программы.

Тест – это набор входных данных, на которых выполняется тест, результатов, которые ожидаются после выполнения теста, и условий выполнения теста.

Тестовый набор – набор тестов построенных по выбранному критерию тестирования.

Критерий тестирования – определяет правила, определяющие размер тестового набора, подход к разработке тестов, момент завершения тестирования.

Тестирование ПО — это процесс проверки работоспособности, основанный на использовании конечного набора тестовых данных, сформированных на базе требований к ПО и сравнения полученных результатов с целевыми показателями качества, заложенными в проекте.

Статическое тестирование – выявление формальными методами анализа неверных конструкций или неверных отношений объектов программы (ошибок формального задания) с помощью специальных инструментов контроля кода – CodeChecker без выполнения тестируемой программы.

Динамическое тестирование – выявление ошибок только на выполняющейся программе с помощью специальных инструментов автоматизации *тестирования* – *Testbed* или Testbench.

Отладка — процесс локализации и исправления ошибок в программе.

Оракул – тот, кто дает заключение о факте появления не совпадения ожидаемого результата с результатом работы программы. Оракулом может быть даже Заказчик или программист, производящий соответствующие вычисления в уме, поскольку Оракулу нужен какой-либо альтернативный способ получения ожидаемого результата работы программы.

Спецификация требований программного обеспечения — структурированный набор требований (функциональность, производительность, конструктивные ограничения и атрибуты) к программному обеспечению и его внешним интерфейсам. Предназначен для того, чтобы установить базу для

соглашения между заказчиком и разработчиком (или подрядчиками) о том, как должен функционировать программный продукт.

Структурные критерии— критерии, которые используют информацию о структуре программы (критерии так называемого "белого ящика")

Функциональные критерии— критерии, которые формулируются в терминах описании требований к программному изделию (критерии так называемого "черного ящика").

Критерии стохастического тестирования — критерии, которые формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.

Мутационные критерии— критерии, которые ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

Критерий тестирования команд (критерий C0) – это критерий, в соответствии с которым набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.

Критерий тестирования ветвей (критерий C1) – это критерий, в соответствии с которым набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах *автоматизации тестирования*.

Критерий тестирования путей (критерий C2)– это критерий, в соответствии с которым набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или *числом классов* выходных путей).

Функциональный критерий - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При функциональном тестировании преимущественно используется модель "черного ящика".

Тестирование пунктов спецификации - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза.

Тестирование классов входных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза.

Тестирование правил - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики.

Тестирование классов выходных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем

отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (time out).

Тестирование функций - набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза.

Комбинированные критерии для программ и спецификаций - набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза.

Модульное тестирование (юнит тестирование или unit testing) — заключается в изолированной проверке каждого отдельного модуля путем запуска тестов в искусственной среде.

Модуль (unit, элемент)— наименьший компонент, который можно откомпилировать – функция, класс.

Драйверы — модули тестов, которые запускают тестируемый элемент.

Заглушки — заменяют недостающие компоненты, которые вызываются модулем.

Обозреватель тестов— средство модульного тестирования Visual Studio, которое позволяет выполнять модульные тесты и просматривать их результаты.

Средства покрытия кода— средство модульного тестирования Visual Studio, которое позволяет определить объем протестированного кода продукта.

Литература

Основная литература

1. Павловская Т.А. C#. Программирование на языке высокого уровня: Учебник для вузов. - СПб. : Питер, 2014. - 432 с. : ил. - (Серия "Учебник для вузов").
2. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft.NET Framework 4.0 на языке C# . 3-е изд.: - СПб.:Питер, 2012. - 928 с. : ил.

В электронном виде

3. Котляров В.П. Основы тестирования программного обеспечения [Электронный ресурс]/ В.П. Котляров— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 334 c.— Режим доступа: http://www.iprbookshop.ru/62820.html.— ЭБС «IPRbooks»
4. Котляров, В. П. Основы тестирования программного обеспечения : учебное пособие для СПО / В. П. Котляров. — Саратов : Профобразование, 2019. — 335 с. — ISBN 978-5-4488-0364-2. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/86202.html (дата обращения: 21.08.2020). — Режим доступа: для авторизир. пользователей
5. Павловская Т.А. Программирование на языке высокого уровня C# [Электронный ресурс]/ Павловская Т.А.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 245 c.— Режим доступа: http://www.iprbookshop.ru/73713.html.— ЭБС «IPRbooks»

6. Microsoft. Руководство по программированию на C# [Электронный ресурс] URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/ (дата обращения 21.08.20)
7. ИНТУИТ. Лаврищева Е.М., Петрухин В. Курс Методы и средства инженерии программного обеспечения. Лекция 2: Области знаний программной инженерии и стандарты ЖЦ программного обеспечения. [Электронный ресурс] URL: https://www.intuit.ru/studies/courses/2190/237/lecture/6118 (дата обращения 21.08.20)
8. Microsoft. Справочник по языку C++ [Электронный ресурс] URL: https://docs.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=vs-2019