МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федерального государственного бюджетного образовательного учреждения

высшего образования

«РОССИЙСКИЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Г.В.ПЛЕХАНОВА»

Техникум Пермского института (филиала)

СОПРОВОДИТЕЛЬНАЯ ЗАПИСКА

к практической работе по дисциплине «Поддержка и тестирование программных модулей»

Тема: Создание и запуск модульных тестов для управляемого кода

Руководитель

_____ /Д.Б.Берестов/

«____» _____ 2026 г.

Исполнитель

_____ /К.А.Черепнев/

«____» _____ 2026 г.

Пермь, 2026 г.

ОГЛАВЛЕНИЕ

Введение

Данная работа представляет собой комплексное упражнение, моделирующее реальный процесс разработки программного обеспечения — от написания кода до его всесторонней проверки и улучшения.

Актуальность темы обусловлена тем, что современная разработка программных продуктов (особенно по методологиям DevOps и CI/CD) немыслима без использования автоматизированных тестов. Они являются не просто способом поиска ошибок, а механизмом, повышающим общую надежность и стабильность приложения, позволяющим разработчикам уверенно вносить изменения в код без страха «сломать» существующую функциональность.

В рамках работы передо мной стояла цель изучить жизненный цикл модульного тестирования на практике. Это включает в себя создание тестовой среды, написание набора тестов для проверки конкретного метода (Debit), запуск тестов и анализ результатов. Ключевым этапом становится ситуация, когда тест обнаруживает намеренно допущенную ошибку — это позволяет не только зафиксировать навыки отладки, но и проследить путь улучшения кода: от простой констатации факта ошибки к написанию более "умных" тестов, проверяющих не только факт исключения, но и его причину.

Теоретические сведения о модульном тестировании

Прежде чем перейти к практической реализации, важно определить место модульного тестирования в процессе разработки и понять, какие задачи оно решает. В широком смысле, модульное тестирование (или unit-тестирование) представляет собой метод проверки программного обеспечения, при котором объектом испытания являются максимально изолированные, мельчайшие компоненты программы — так называемые «модули» или «юниты». В контексте объектно-ориентированного программирования, которым является C#, таким модулем чаще всего выступает метод класса.

Суть этого подхода заключается не в простом «запуске кода», а в создании специальной среды (тестового проекта), которая автоматически проверяет, соответствует ли поведение каждого модуля ожиданиям разработчика. Модульное тестирование служит своего рода «страховочной сеткой»: оно используется для непрерывного обеспечения качества кода на всех этапах его жизни, позволяя с высокой точностью локализовать и предотвращать ошибки еще до того, как код попадет в общий проект или к конечному пользователю.

Упрощение сопровождения и рефакторинга. Разработчики часто боятся менять «хрупкий» код, так как любое изменение может непредсказуемым образом повлиять на другие части системы. Наличие набора модульных тестов действует как успокаивающий фактор: после внесения изменений достаточно прогнать все тесты, и их «зеленый» статус станет объективным доказательством того, что существующая функциональность не пострадала. Это дает уверенность и свободу для улучшения кода.

Улучшение архитектуры и дизайна кода. Этот пункт менее очевиден, но крайне важен. Код, который трудно тестировать, как правило, сам по себе имеет проблемы с архитектурой: он перегружен зависимостями, делает слишком много дел сразу или плохо структурирован. Стремление написать простой и понятный тест часто невольно подталкивает разработчика к созданию более модульного, слабосвязанного и логичного кода. Тесты выступают в роли «контролеров» качества архитектуры.

Документирование кода. Хороший модульный тест — это живая документация. Взглянув на тест, например,

Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange, другой разработчик (или вы сами через несколько месяцев) мгновенно поймет, какое поведение ожидается от метода Debit в пограничной ситуации, не вникая в детали его внутренней реализации. Тесты показывают, как предполагается использовать код и что он должен делать в ответ на различные входные данные.

Таким образом, модульное тестирование в современной разработке — это не просто галочка в списке требований, а фундаментальная практика, которая делает процесс создания программного обеспечения более предсказуемым, безопасным и управляемым. В рамках данной работы эти теоретические принципы будут закреплены на практике путем разработки и выполнения конкретных тестовых сценариев.

Создание проекта

Отправной точкой практической части работы стала подготовка среды для разработки и создание тестируемого приложения. Все действия выполнялись в интегрированной среде разработки Visual Studio, которая предоставляет исчерпывающий набор инструментов для написания кода, его сборки и последующего тестирования.

Для обеспечения актуальности полученных навыков и соответствия современным стандартам разработки в качестве целевой платформы была выбрана .NET (версия 6.0). Использование современной версии .NET гарантирует, что применяемые подходы и API не устарели, а также позволяет задействовать последние улучшения производительности и безопасности, что немаловажно даже в учебном проекте.

Процесс создания проекта был выполнен пошагово в соответствии с руководством:

После запуска Visual Studio на начальном экране был выбран пункт «Создать проект».

В галерее шаблонов найден и выбран шаблон «Консольное приложение на C#».

Проекту было присвоено имя Bank — лаконичное и отражающее предметную область будущего кода.

На завершающем этапе была выбрана целевая платформа .NET (рекомендуемая 6 версия), после чего проект был успешно создан и отобразился в окне «Обозреватель решений».

Таким образом, на этом этапе была сформирована основа для дальнейшей работы — структурная единица (решение Visual Studio), готовая к наполнению программным кодом и последующему подключению тестовой инфраструктуры.

Разработка класса банковского счета

Следующим логическим шагом после создания пустого проекта стало наполнение его предметной логикой — разработка класса, который будет выступать в роли объекта тестирования. В качестве такого класса был выбран BankAccount (банковский счет), поскольку его функциональность интуитивно понятна и содержит четкие бизнес-правила, что делает его идеальным кандидатом для демонстрации возможностей модульного тестирования.

Класс BankAccount был реализован в файле BankAccount.cs и включает в себя базовый набор элементов, характерных для любого финансового инструмента:

Свойства класса: CustomerName (имя владельца) и Balance (текущий баланс), которые хранят состояние счета.

Конструктор: Специальный метод, который задает начальные значения при создании нового экземпляра счета — имя владельца и стартовую сумму на счету.

Основные методы: Credit (пополнение счета) и Debit (списание средств со счета), которые реализуют основную бизнес-логику работы со счетом.

Однако ключевой особенностью реализованного класса является не просто наличие методов пополнения и списания, а встроенная защита от некорректного использования. При разработке было учтено, что финансовая логика требует особой надежности, поэтому в методы были добавлены механизмы валидации входных данных.

В частности, метод списания средств Debit содержит две важные проверки:

Проверка на отрицательную сумму: Снятие отрицательной суммы (что эквивалентно пополнению через операцию списания) является некорректной операцией с точки зрения бизнес-логики.

Проверка на превышение баланса: нельзя снять со счета больше средств, чем на нем имеется.

В случае нарушения этих правил метод не выполняет операцию, а генерирует исключение ArgumentOutOfRangeException. Это важнейший механизм обеспечения безопасности и надежности: вместо того чтобы молча выполнить неверное действие (например, уйти в отрицательный баланс) или вернуть

некорректный код ошибки, программа сигнализирует о проблеме громким "сигналом" (исключением), который невозможно проигнорировать. Такой подход вынуждает вызывающий код (и тестировщика) явно обрабатывать подобные ситуации.

Создание и настройка тестового проекта

После того как был разработан основной проект Bank с его ключевым классом BankAccount, следующим важнейшим этапом работы стала подготовка инфраструктуры для тестирования. В современной разработке тесты никогда не размещают внутри самого проверяемого проекта — это нарушило бы его чистоту и смешало бы основную логику с вспомогательными механизмами проверки. Вместо этого создается полностью отдельный проект, специально предназначенный для размещения тестов.

В данной работе в соответствии с методическими указаниями был создан проект с названием BankTests. При его создании в качестве шаблона был выбран тип «Проект модульного теста MSTest». Выбор именно MSTest (фреймворка модульного тестирования от Microsoft) обусловлен его полной интеграцией со средой Visual Studio и используемым языком C#, что обеспечивает максимальную совместимость и удобство работы без необходимости установки дополнительных компонентов.

Однако просто создать тестовый проект недостаточно — необходимо объяснить среде разработки, какой именно код мы собираемся тестировать. Для этого была выполнена процедура настройки связи (ссылки) между двумя проектами. В обозревателе решений в проекте BankTests был открыт раздел «Зависимости», через который с помощью «Диспетчера ссылок» была добавлена ссылка на основной проект Bank. После выполнения этого действия тестовому проекту становится «виден» внутренний код класса BankAccount, и тесты могут его вызывать так, как если бы он был частью самого тестового проекта.

Такая организация работы — разделение на два связанных проекта — несет в себе важный смысл. Она позволяет разработчику вносить изменения в код программы, не боясь случайно задеть тесты, и наоборот. Но самое главное — такая архитектура позволяет запускать проверку в автоматическом режиме. После того как тесты написаны и настроены, их выполнение больше не требует ручного вмешательства. Достаточно нажать кнопку в среде разработки (или настроить автоматический запуск при сохранении кода), и Visual Studio сама выполнит все тестовые методы, соберет результаты и покажет, какие части программы работают корректно, а какие требуют внимания.

Разработка тестовых методов

С завершением настройки тестового проекта и установки связи между BankTests и основным проектом Bank работа перешла в наиболее важную и содержательную стадию — непосредственное написание тестовых методов. Основное внимание в рамках данной работы было сосредоточено на проверке метода списания средств Debit, так как он содержит наиболее интересную с точки зрения тестирования логику с несколькими ветвлениями и потениальными ошибками.

Прежде чем приступить к написанию кода, был выполнен важный подготовительный шаг: стандартный тестовый класс UnitTest1, автоматически созданный шаблоном, был переименован в BankAccountTests. Такое именование сразу дает понять, к какому именно классу относится данный набор тестов, следуя широко распространенной практике именования ИмяТестируемогоКлассаTests. Аналогично был переименован и сам файл на диске, что поддерживает порядок в структуре проекта.

Для обеспечения чистоты кода и удобства его чтения в начало тестового файла была добавлена директива using BankAccountNS;. Это позволило обращаться к тестируемому классу BankAccount напрямую, без использования полного имени с пространством имен, что сделало код тестов более лаконичным и читаемым.

При разработке тестовых методов использовался фундаментальный принцип организации тестового кода, известный как Arrange-Act-Assert (Подготовка-Действие-Проверка). Согласно этому принципу, каждый тест логически разделен на три четкие части:

Arrange (Подготовка): на этом этапе создаются необходимые условия для теста. В данном случае это создание экземпляра класса BankAccount с начальным балансом и определение тестовых значений (сумм списания).

Act (Действие): выполняется само тестируемое действие — вызов метода Debit с подготовленными параметрами.

Assert (Проверка): производится сравнение фактического результата выполнения метода с тем результатом, который ожидался.

В соответствии с требованиями к полноте тестового покрытия было разработано три ключевых тестовых сценария, охватывающих различные аспекты работы метода Debit:

Тестирование корректных данных (Debit_WithValidAmount_UpdatesBalance): Первый и самый очевидный тест проверяет, что при вводе допустимой суммы (больше нуля, но меньше баланса) метод правильно выполняет свою основную функцию. В этом тесте после вызова метода Debit с помощью утверждения Assert.AreEqual проверяется, что итоговый баланс уменьшился ровно на сумму списания.

Тестирование отрицательной суммы (Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange): Второй тест проверяет защиту метода от некорректных входных данных. Он вызывает метод Debit с отрицательным значением и ожидает, что будет выброшено исключение ArgumentOutOfRangeException. Для этого используется механизм Assert.ThrowsException, который специально создан для подобных сценариев и считается тест пройденным только в том случае, если нужное исключение действительно возникло.

Тестирование превышения баланса (Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange): Третий тест проверяет еще одно защитное условие — невозможность снятия средств сверх остатка. Он также ожидает возникновения исключения ArgumentOutOfRangeException, но уже по другой причине.

Тесты не просто подтверждают, что метод "работает", они документируют и проверяют все заложенные в него бизнес-правила: как правильное поведение, так и корректную реакцию на ошибки. Это создает надежный защитный барьер, который гарантирует, что метод Debit будет вести себя предсказуемо во всех предусмотренных ситуациях.

Запуск и анализ тестирования

После завершения разработки тестовых методов наступил ключевой момент практической работы — их выполнение и анализ полученных результатов. Этот этап наглядно демонстрирует, как теоретические принципы модульного тестирования воплощаются в реальный инструмент контроля качества кода.

Для запуска тестов был использован встроенный инструмент среды Visual Studio — «Обозреватель тестов» (Test Explorer). Этот компонент представляет собой специализированную панель, которая автоматически обнаруживает все методы, помеченные атрибутом [TestMethod] в проекте, и предоставляет удобный интерфейс для управления их выполнением. Запуск тестов осуществляется буквально одной командой — достаточно выбрать опцию «Запустить все», и среда разработки самостоятельно выполняет всю дальнейшую работу.

Процесс выполнения тестов визуализируется в Обозревателе тестов с помощью анимированной строки состояния, что позволяет наглядно наблюдать за ходом проверки. По завершении тестового запуска строка состояния окрашивается в определенный цвет, давая мгновенную обратную связь о состоянии проекта:

Зеленый цвет сигнализирует об успешном прохождении всех тестов.

Красный цвет указывает на то, что один или несколько тестов не пройдены.

В данном случае, при первом запуске тестов, строка состояния окрасилась в красный цвет — один из тестов завершился неудачей. Это не свидетельствует о неисправности тестов, а, напротив, подтверждает их работоспособность и чувствительность. Конкретно тест Debit_WithValidAmount_UpdatesBalance, проверяющий корректное списание средств, показал отрицательный результат.

Анализ деталей, представленных в Обозревателе тестов, позволил точно понять причину ошибки. Сообщение, сгенерированное методом Assert.AreEqual, содержало ценную отладочную информацию: ожидалось, что после списания баланс уменьшится и составит 7.44 условных единицы, однако фактическое значение баланса оказалось больше исходного. Это прямо указывало на характер ошибки — метод списания Debit не вычитал сумму, а прибавлял ее.

Таким образом, модульные тесты выполнили свою главную задачу — они выявили логическую ошибку, которая была намеренно допущена в коде на

этапе разработки. Этот момент является ключевым в понимании ценности тестирования: ошибка была обнаружена автоматически, сразу после написания кода, в безопасной среде разработки, а не в процессе эксплуатации программы реальными пользователями.

Корректировка кода и повторное тестирование

Выявление ошибки — важный, но лишь промежуточный этап работы. Главная ценность модульного тестирования раскрывается в тот момент, когда разработчик, вооруженный информацией от тестов, приступает к исправлению дефекта. В этом заключается циклический характер современной разработки: код пишется, тестируется, анализируется и улучшается.

Получив от Обозревателя тестов четкий сигнал о том, что метод Debit работает некорректно (баланс увеличивается вместо уменьшения), я приступил к анализу исходного кода класса BankAccount. Внимательное изучение метода Debit быстро подтвердило выводы, сделанные на основе тестов: в строке, отвечающей за изменение баланса, действительно содержалась логическая ошибка. Вместо операции вычитания была ошибочно запрограммирована операция сложения: (см. рисунок 1)

```
C#

m_balance += amount;
```

на:

```
C#

m_balance -= amount;
```

Рисунок 1 – исправление ошибки

В соответствии с бизнес-логикой банковского счета списание средств должно уменьшать баланс, поэтому исправление оказалось простым и очевидным — оператор сложения += был заменен на оператор вычитания -=

После внесения этого изменения настал самый важный и показательный момент — повторный запуск тестов. Снова была использована команда «Запустить все» в Обозревателе тестов. На этот раз результат оказался принципиально иным: строка состояния, ранее горевшая красным, окрасилась в торжествующий зеленый цвет. Тест Debit_WithValidAmount_UpdatesBalance, который до этого неизменно завершался неудачей, теперь был успешно пройден.

Заключение

Выполненная практическая работа по созданию и запуску модульных тестов для управляемого кода на платформе .NET позволила не только освоить конкретные инструменты и технологии, но и сформировать целостное понимание роли автоматизированного тестирования в современном процессе разработки программного обеспечения.

В ходе работы было последовательно реализовано несколько ключевых этапов. Начав с создания консольного приложения и разработки класса BankAccount, имитирующего реальный программный модуль с бизнес-логикой, я перешел к формированию полноценной тестовой инфраструктуры. Был создан отдельный проект на базе фреймворка MSTest, настроены необходимые ссылки и связи, обеспечивающие изолированное, но взаимосвязанное существование основного и тестового кода.

Центральное место в работе заняла разработка набора тестовых методов, построенных по принципу Arrange-Act-Assert и охватывающих различные сценарии использования метода Debit. Важно отметить, что тесты проверяли не только "идеальный" вариант работы с корректными данными, но и граничные состояния — попытку списания отрицательной суммы и суммы, превышающей остаток на счете. Такой подход обеспечил комплексную проверку функциональности и продемонстрировал важность тестирования защитных механизмов программы.

Ключевым моментом, наглядно подтвердившим ценность проделанной работы, стал запуск тестов и анализ их результатов. Благодаря интеграции с Обозревателем тестов Visual Studio процесс выполнения проверок оказался максимально прозрачным и информативным. Тест, проверяющий корректное списание средств, закономерно выявил логическую ошибку, намеренно допущенную в коде. Это стало убедительной демонстрацией того, как автоматизированные тесты способны обнаруживать дефекты на самых ранних этапах, когда их исправление требует минимальных усилий.

Исправление обнаруженной ошибки (замена операции сложения на вычитание) и повторный запуск тестов, завершившийся их успешным прохождением, замкнули цикл разработки и подтвердили правильность внесенных изменений. Зеленый индикатор в Обозревателе тестов выступил объективным

свидетельством того, что код теперь функционирует в полном соответствии с ожиданиями.

Таким образом, в рамках данной работы методология модульного тестирования полностью доказала свою эффективность. Она позволила не просто проверить готовый код, но и выступила в роли инструмента, направляющего процесс разработки и улучшающего качество конечного продукта. Полученные навыки — от написания тестовых методов до интерпретации их результатов и последующей корректировки кода — формируют фундамент профессионального подхода к созданию надежного и сопровождаемого программного обеспечения. Модульное тестирование, как показала практика, перестает быть просто "проверкой" и становится неотъемлемой частью культуры разработки, обеспечивающей стабильность и уверенность в работе с кодом.

Список использованных источников

1. Документация Microsoft.

2. Учебные материалы по тестированию программ - https://cloud.mail.ru/public/JaXA/BUKbRzZoN