УДК 004.054 **Информационные технологии**

Базеева Наталья Алексеевна, *преподаватель факультета довузовской подготовки и среднего профессионального образования, ФГБОУ ВО*
Харитонов Виталий Игоревич, *преподаватель факультета довузовской подготовки и среднего профессионального образования, ФГБОУ ВО «Мордовский государственный университет им. Н.П. Огарёва»*
e-mail: mr.vitallka@mail.ru
Малкина Анастасия Анатольевна, *студентка факультета довузовской подготовкии среднего профессионального образования, ФГБОУ ВО «Мордовский государственный университет им. Н.П. Огарёва»*

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ (TDD)

Аннотация: данная статья предназначена для ознакомления с принципами процесса разработки через тестирование. Описывается реализация данного метода разработки, а также приведены подробные примеры, написанные на языке программирования C#.

Ключевые слова: TDD, разработка через тестирование, тестирование ПО, рефакторинг, баг, программный продукт.

Abstract: This article is intended to introduce the principles of the development process through testing. The implementation of this development method is described and detailed examples written in C# programming language are given.

Keywords: TDD, development through testing, software testing, refactoring, bug, software product.

Тестирование ПО (программного обеспечения) – это деятельность, которая позволяет подтвердить работоспособность кода (программного продукта) или же, наоборот, ее опровергнуть. Суть тестирования заключается в том, что

создаются отдельные сценарии, чтобы выявить слабые места вашей созданной системы (на одну лишь разработку полагаться не стоит, так как на данном этапе невозможно продумать все тест-кейсы). Тестирование предназначено для того, чтобы впоследствии не было багов (ошибок) от клиента, которые нужно срочно чинить (бывает и такое, что у клиента ломаются не отдельные модули, а система в целом, потому что в свое время тестирование было проведено ненадлежащим образом). Тестирование может быть ручным, – когда передаются входные данные и запрашивается выполнение какой-либо команды; после полученные результаты сравниваются с эталоном, и, если они совпадают – тест пройден. Но данный процесс можно автоматизировать, тогда он будет проходить гораздо быстрее и полноценнее, вероятность возникновения ошибок значительно снизится, потому что, как правило, ручное тестирование проводят люди, а человеческий фактор очень сильно может повлиять на данный процесс (невнимательность – главный наш враг).

TDD (Test Driven Development, «разработка через тестирование») – особая методология разработки, которая позволяет быстро и качественно создать программный продукт. Суть разработки через тестирование в том, что сначала пишутся тесты на новый функционал, а только потом код, который реализует данный функционал. После успешного прохождения теста проводится рефакторинг (переработка, перепроектирование) кода, если это необходимо, а затем повторная проверка работоспособности кода, что заметно ускоряет разработку, а также делает ваш код более поддерживаемым [3].

Плюсы данной методологии:

– Разработчик может сам писать тесты, проверяя отдельные модули системы.

– Эффективность. Зачастую тесты пишутся после разработанного функционала, поэтому тесты подстраиваются под написанный функционал, а не под требования в техническом задании, в TDD же все наоборот.

– В приложении, пригодном для автоматического тестирования, равномерно распределяется ответственность между его компонентами.

– Обеспечивается стабильность приложения, так как проблемные места покрыты тестами, которые постоянно проверяются.

– В дальнейшем разработчики могут не переживать за внесение изменений в код, так как тесты предупредят об ошибках в любой момент.

– Быстрота. За счет автоматических тестов сокращается объем ручного тестирования, соответственно, и временной промежуток, отведенный в целом на тестирование (и в будущем на поддержку программного продукта).

– При данной методологии программист начинает «думать по-другому»: продумывать больше тестовых сценариев и писать легко поддерживаемый код.

Цикл разработки согласно TDD:

– создать тест для нового функционала, еще не разработанного, или же для воспроизведения какого-либо бага;

– запустить тест и убедиться в том, что он не проходит;

– написать функционал, при котором этот тест мог бы пройти;

– запустить тест (если есть ранее написанные тесты, их тоже нужно запустить). При прохождении тестов разработчик понимает, что новый функционал готов, а ранее реализованный работает так же корректно;

– провести рефакторинг и оптимизацию. Нужно понимать, что тесты должны не только проходить, но и быстро работать;

– перезапустить тесты, убедиться в их прохождении;

– повторить цикл.

После получения необходимых знаний для понимания TDD в теории, следует разобрать это на практике. В данной статье будет рассмотрено модульное тестирование на языке программирования C#. Модульное тестирование – это тестирование программного продукта, при котором проверяются его отдельные модули или компоненты.

Для начала нужно понимать, что означает «чистый тест» и из чего состоит тело теста.

Чистый тест должен быть написан с соблюдением **5 правил**:

– тест не должен зависеть от другого теста;

– тест должен быстро выполняться при быстрых запусках;

– воспроизведение должно осуществляться в любой среде;

– должен возвращаться результат для быстрого заключения;

– тест должен писаться своевременно, когда это действительно актуально.

В качестве примера разработано консольное приложение «Calculator» (калькулятор). Логика простая: пользователь будет выбирать арифметическую операцию и вводить нужные числа, над которыми будет выполняться выбранная операция. Так как не имеется никакого разработанного функционала, пишется тест для нового. Для наглядности будет разработана операция деления, так как тут имеется как минимум два тестовых случая (первый – числа делятся, а второй – при делении на ноль программа не завершается с ошибкой). Первый тест (рисунок 1) проверяет, что, если дать калькулятору 2 целых числа и выполнить метод деления, то тест пройдет. Это можно понять из названия теста: через нижнее подчеркивание перечисляем входные данные, тестируемый сценарий и ожидаемый результат.

```
[Test]
0 references
public void Div_IntNumbers_TestPass()
{
    // arrange
    var calculator = new CalculatorLogic();
    calculator.FirstNumber = 3;
    var secondNumber = 2;

    // act
    var result = calculator.Div(secondNumber);

    // assert
    Assert.AreEqual(1.5, result);
}
```

Рисунок 1 – Тест деления двух целых чисел

Как видно из комментариев на рисунке 1, тело теста состоит из 3 частей:

1. создается экземпляр класса калькулятора, а также задаются необходимые тестовые данные. Согласно этой логике, первое число – это свойство класса, а второе число передается уже в метод каждой операции, поэтому оно создается отдельной переменной;

2. выполняется действие (операция деления в данном случае), в ходе которого мы получим какой-то результат;

3. сравнение результата с эталоном (ожидаемым результатом).

Первый этап пройден. Запускаем тест: проект не скомпилировался из-за отсутствия класса с требуемой логикой, соответственно, второй этап тоже пройден. Переходим к 3 этапу, разработке функционала (рисунок 2).

```
public class CalculatorLogic
{
    private double _firstNumber;

    1 reference
    public double FirstNumber
    {
        get { return _firstNumber; }
        set { _firstNumber = value; }
    }

    1 reference
    public double Div(double secondNum)
    {
        return _firstNumber / secondNum;
    }
}
```

Рисунок 2 – Новый функционал

Ошибки при компиляции в классе с тестами исчезли, теперь запускается сам тест. Тест пройден (рисунок 3), значит, можно переходить к следующему этапу.

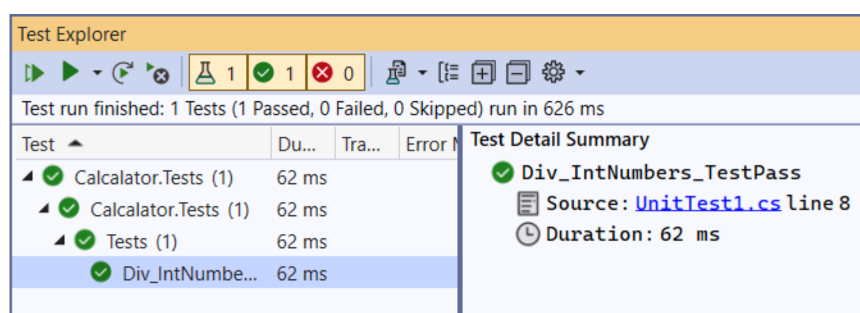

Рисунок 3 – Пройденный тест

Так как функционал метода слишком примитивный, то в проведении рефакторинга и оптимизации нет необходимости, значит, осуществляется переход к следующей итерации цикла разработки: пользователь ввел в качестве второго числа значение 0, из-за этого на консоль вывелся непонятный результат (в C# это «плюс бесконечность»). Следует написать тест и под данную проблему (рисунок 4).

```
[Test]
| 0 references
public void Div_SeconNumberIsZero_TestPass()
{
    // arrange
    var calculator = new CalculatorLogic();
    calculator.FirstNumber = 3;
    var secondNumber = 0;

    // act
    var result = calculator.Div(secondNumber);

    // assert
    Assert.That(result == double.PositiveInfinity);
}
```

Рисунок 4 – Тест «При делении на ноль результат – бесконечность»

Чтобы не возникало такой проблемы, в консоль должно выводиться сообщение о некорректности введенных данных, а сам метод деления должен выбрасывать исключение. Для этого пишется еще один тест, который будет проверять, выбрасывается ли исключение определенного типа (рисунок 5).

```
[Test]
| 0 references
public void Div_SeconNumberIsZero_ExceptionThrows()
{
    // arrange
    var calculator = new CalculatorLogic();
    calculator.FirstNumber = 3;
    var secondNumber = 0;

    // act
    TestDelegate testDelegate = () => calculator.Div(secondNumber);

    // assert
    Assert.Throws<DivideByZeroException>(testDelegate);
}
```

Рисунок 5 – Тест на вылет исключения

Так как функционал не обновлен, то тест не проходит, – соответственно, расширяем его с последующим запуском тестов (рисунок 6).

```
4 references | 3/3 passing
public double Div(double secondNum)
{
    if (secondNum == 0)
        throw new DivideByZeroException();
    return _firstNumber / secondNum;
}
```

Рисунок 6 – Обновленный метод деления

Чтобы на консоль не выводилось исключение, оно будет поймано в классе программы и вместо него будет выведено сообщение (рисунок 7).

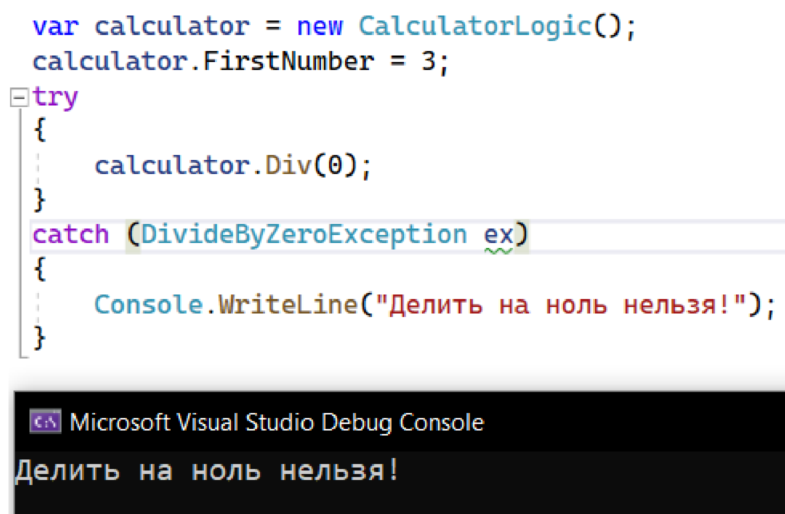

Рисунок 7 – Обработка исключения, вывод сообщения

Подводя итоги вышесказанному, можно сказать, что даже в маленькой примитивной программе могут быть ошибки в каждой строчке кода, поэтому тестирование необходимо проводить всегда. Благодаря тестированию код будет проще поддерживать и подвергать рефакторингу. Однако стоит отметить, что данный метод разработки применим только в определенных случаях, когда

специфика разрабатываемой программы позволяет использовать данный метод разработки.

Библиографический список:

1 Принципы юнит-тестирования / В. Хориков. – СПб.: Питер, 2021. – 320 с.: ил. – («Серия для профессионалов»).

2 Экстремальное программирование: разработка через тестирование / К. Бек – СПб.: Питер, 2017 – 219 с.

3 Правильное TDD [Электронный ресурс] Habr – Режим доступа: https://habr.com/ru/post/573016/.