



# Введение в юнит-тестирование

Качество тестов



# Оглавление

<b>Введение</b>	<b>3</b>
<b>Термины, используемые в лекции</b>	<b>3</b>
<b>Как проектировать хорошие тесты и для чего это нужно</b>	<b>4</b>
Выделяют 4 атрибута качественно работающего теста	5
Сквозные (end-to-end) тесты	9
Тривиальный тест	10
Хрупкие тесты	11
<b>Тестирование по принципу чёрного и белого ящика</b>	<b>12</b>
Тестирование «чёрного ящика»	12
Тестирование «белого ящика»	13
Сравнение методов	14
Тестирование «серого ящика»	14
<b>Метрики тестов</b>	<b>15</b>
<b>Инструменты для измерения покрытия тестами</b>	<b>19</b>
<b>Разработка через тестирование</b>	<b>21</b>
<b>Разработка через поведение</b>	<b>26</b>
<b>Подведём итоги</b>	<b>27</b>
<b>Что можно почитать ещё?</b>	<b>27</b>
<b>Используемая литература</b>	<b>28</b>

# Введение

Мы дали определение unit-тестированию; поговорили о его особенностях; о том где принято писать тексты; о структуре тестов; узнали, что такое фреймворки; попрактиковались в написании тестов JUnit 5.

На этой лекции вы найдёте ответы на следующие вопросы:

- Как проектировать хорошие тесты и для чего это нужно.
- Тестирование по принципу чёрного и белого ящика.
- Метрики тестов.
- Инструменты для измерения покрытия тестами.
- Тестирование через разработку.
- Тестирование через поведение.

## Термины, используемые в лекции

**Регрессионное тестирование** — собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Такие ошибки — когда после внесения изменений в программу, перестаёт работать то, что должно было продолжать работать, — называются **регрессионными ошибками**. Термины «регрессия», «программная ошибка» и «баг» — синонимы.

**SUT** (System under test) — представляет все классы в тесте, которые не являются предопределёнными фрагментами кода, такими как заглушки.

**Бизнес-логика** — совокупность правил, принципов, зависимостей поведения объектов предметной области (области человеческой деятельности, которую система поддерживает).

**Рефакторинг** — это изменение деталей имплементации без изменения наблюдаемого поведения.

**Метрика покрытия (coverage metric)** — метрика, которая показывает, какая доля исходного кода была выполнена хотя бы одним тестом — от 0 до 100%.

# Как проектировать хорошие тесты и для чего это нужно

Хорошие юнит-тесты помогают предотвратить стагнацию и сохранить темп разработки со временем. С такими тестами вы будете уверены в том, что изменения не приведут к багам. В свою очередь, это упростит рефакторинг кода или добавление новой функциональности.

💡 Хорошие тесты становятся надёжной системой предупреждения об ошибках, что способствует стабильному росту проекта.

Целью юнит-тестирования, как вы, наверное, помните из первой лекции, является **обеспечение стабильного роста проекта**. Так вот, невозможно её добиться, просто написав много тестов — необходимо учитывать, как ценность теста для проекта, так и цену дальнейшего сопровождения тестов.




На графике изображена динамика роста проекта без тестов (**красная линия**) и проекта с тестами (**зелёная линия**).

Как видите, проект без тестов стартует быстрее, так как вы просто не тратите время на написание тестов, а решаете основные бизнес-задачи, не отвлекаясь на возможные ошибки.

Далее, по мере роста проекта, код становится сложнее, обрастает зависимостями, которые уже не удержать в голове и скорость развития проекта заметно снижается, иногда проект вообще перестаёт развиваться. Вам нужно много времени на поиск и устранение ошибок, поддержку старого кода.

Тесты помогают справиться с этими проблемами. Они становятся своего рода «подушкой безопасности» — средством, которое обеспечивает защиту против большинства ошибок.

Но как вы могли заметить проект с тестами требует больших начальных вложений, стартует он позже проекта без тестов, но это окупается за счёт того, что с тестами вы становитесь увереннее, когда ваш продукт растёт и развивается. Потому что уверены, что ваш код не содержит ошибок, и вы можете смело переходить к более сложным задачам.

 **Проекты с плохими тестами, как и проекты без тестов вообще приходят к одинаковому результату:** либо стагнация, либо множество багов с каждым новым релизом.

Ошибки поджидают нас на каждом этапе разработки. Ошибки, которые возникают после внесения изменений в код новой функциональности, называют **регрессионными ошибками**. Термин произошёл от понятия «регресс» — движение назад, отход, откат, возврат.

Регрессионное тестирование может потребоваться в следующих ситуациях:

- Изменение ранее написанного программного кода.
- Добавление нового функционала в программу.
- Устранение ранее обнаруженных дефектов и багов.
- Корректировка проблем, связанных с производительностью.

Сложность в том, что чем больше кода вы пишете, тем больше вероятность возникновения регрессивных ошибок и хороший тест должен максимально защищать от них, поэтому важно знать, как определить, насколько полезен тест в отношении защиты от регрессий.

## **Выделяют 4 атрибута качественно работающего теста**

### **Атрибут 1. Защита от регрессий.**

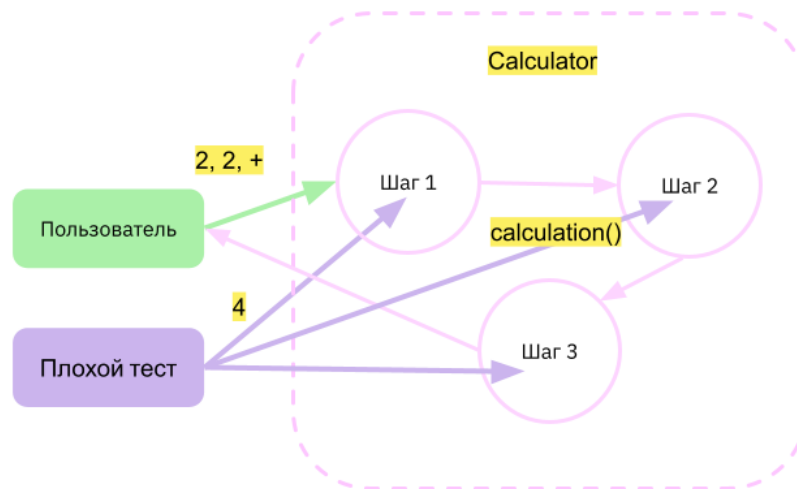
У этого атрибута есть два важных критерия:

- **Чем больше кода проверяет тест, тем выше вероятность выявить баг.** Само собой, тест также должен иметь актуальный и логичный набор проверок (assertions).

- **Важен не только объём проверяемого кода, но и его сложность.** Конечно, проверять нужно наиболее сложные участки кода, цена поломки которого высока. Простой код проверять не нужно.

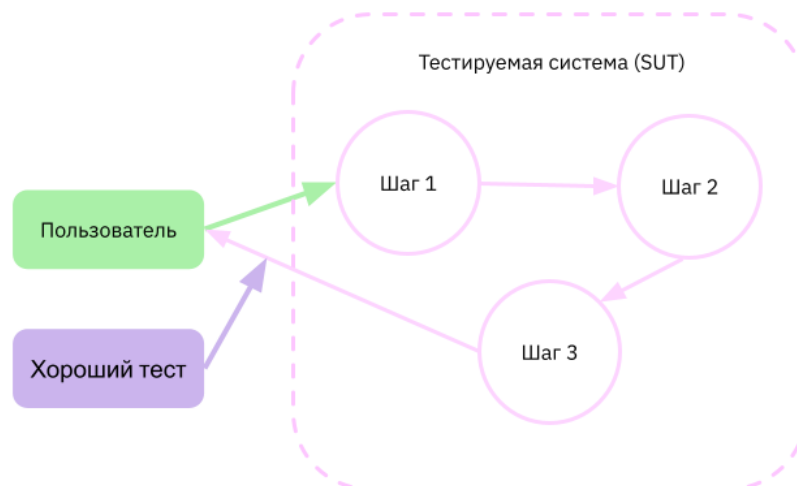
## Атрибут 2. Устойчивость к новым изменениям

Определяет, насколько хорошо тест может пережить рефакторинг тестируемого им кода без ошибок.



Выше показано взаимодействие системы с пользователем, пользователь посылает запрос, тестируемая система производит вычисления (шаг 1 - 2 - 3 – внутренняя логика системы) и возвращает пользователю ответ.

**Плохая практика** — привязываться при проектировании теста к тому, **как достигается результат, а не к самому результату**. При изменении внутри системы тест становится бесполезным и его приходится переписывать.



Обратите внимание на принципиальное улучшение этого теста по сравнению с исходной версией. **Он соотносится с требованиями бизнеса, проверяя только результат**, который имеет смысл для конечного пользователя. Падения таких

тестов всегда указывают на проблему: они сообщают об изменении поведения приложения, что может отразиться на пользователе, и поэтому должны быть рассмотрены разработчиком.

Тест должен проверять результат работы, а не способ достижения результата. Он должен подходить к проверке SUT с точки зрения конечного пользователя и проверять только результат, имеющий смысл для этого пользователя.

Этот пример теста сравнивает не результат работы метода, а дословно текст всего класса с тем, как он реализован на самом деле. То есть при любом изменении исходного кода класса, мы будем получать падающий тест:

```
@Test
void calculationImplementedCorrectly() throws IOException {
    String checkedCode = Files.readString(Path.of("%Путь до файла
класса%\Calculator.java"));

    assertThat("package ru.gb.units;\n" +
        "import java.util.Scanner;\n"
        "public class Calculator {\n" +
        "private static Scanner scanner = new
Scanner(System.in);\n" +
        "public static void main(String[] args) {\n" +
        "int firstOperand = getOperand();\n" +
        "int secondOperand = getOperand();\n" +
        "char operator = getOperator();\n" +
        "int result = calculation(firstOperand, secondOperand,
operator);\n"+
        "...
... Весь код класса Calculator ...
        ...
    ).isEqualTo(checkedCode);
}
```

Конечно, такой тест не идеален, потому что в систему всё ещё могут быть внесены изменения, которые нарушат работу теста. Например, можно добавить дополнительный шаг или изменить получаемый ответ, и это приведёт к ошибке компиляции.

Например, мы захотели вводить входные данные не отдельно, а вместе, для вычисления сложных выражений, для этого изменили конструктор метода.

С технической точки зрения такая ошибка тоже будет считаться ложным срабатыванием, ведь падение теста произошло не из-за изменения в поведении приложения.

```
public static int calculation(String expression) {...} //  
(2+2) * 4
```

Но такой тест легко отрефакторить. В нашем случае достаточно просто следовать рекомендациям компилятора и добавить новый параметр во все тесты, вызывающие метод:

```
@Test  
void additionExpressionEvaluation() {  
    Calculator calculator = new Calculator();  
    assertThat(calculator.calculation( expression: 2, 8, '+')).isEqualTo( expected: "10");  
}
```

Expected 1 arguments but found 3  
Create method 'calculation' in 'Calculator'

ru.gb.units.Calculator

```
public static int calculation(  
    String expression  
)
```

- **Атрибут 3. Простота поддержки**

Оценивает затраты на сопровождение кода (чем понятнее написан тест, тем затраты меньше)

- **Атрибут 4. Быстрая обратная связь**

Чем быстрее работают тесты, тем больше их можно включить в проект и тем чаще вы их сможете запускать.

Быстро выполняемые тесты сильно ускоряют обратную связь. В идеальном случае тесты начинают предупреждать вас об ошибках сразу же после их внесения, в результате чего затраты на исправление этих ошибок уменьшаются почти до нуля.

С другой стороны, медленные тесты увеличивают время, в течение которого ошибки остаются не обнаруженными, что приводит к увеличению затрат на их исправление.

Дело в том, что медленные тесты отбивают у разработчика желание часто запускать их, поэтому в итоге он тратит больше времени, двигаясь в ошибочном направлении.

Требований, которые нужно соблюсти, чтобы обеспечить высокую скорость выполнения всего два:

- **Тест должен быть небольшим.**



- **Тест должен быть изолирован от окружения, то есть на него не должны влиять зависимости.**

Чтобы тест был эффективным, он должен демонстрировать результативность в каждой из четырёх категорий. К сожалению, создать такой тест невозможно. Дело в том, что первые три атрибута (защита от багов, устойчивость к рефакторингу и быстрая обратная связь) являются взаимоисключающими. Невозможно довести их до максимума одновременно: одним из трёх придётся пожертвовать для максимизации двух остальных. Четвёртый атрибут (простота поддержки) не так сильно связан с первыми тремя, за исключением сквозных (end-to-end) тестов, поэтому будем рассматривать его, как часть атрибута устойчивость к изменениям.



Атрибуты нужно максимизировать так, чтобы ни один из них не падал слишком низко.

Примеры тестов, которые стараются максимизировать два из трёх атрибутов за счёт третьего:

#### Атрибуты качества юнит-теста. Хрупкие тесты



## Сквозные (end-to-end) тесты

Сквозные тесты рассматривают систему с точки зрения конечного пользователя. Они обычно проходят через все компоненты системы, включая пользовательский интерфейс, базу данных и внешние приложения.

**Сквозные тесты задействуют большой объём кода, они обеспечивают наилучшую защиту от багов.** Так как используют ваш код и код, написанный не

вами, но используемый в проекте (внешние библиотеки, фреймворки и сторонние приложения).

Они практически не выдают ложных срабатываний, а следовательно, обладают хорошей устойчивостью к рефакторингу. Правильно проведённый рефакторинг не изменяет наблюдаемого поведения системы и поэтому не влияет на сквозные тесты.

Другое преимущество таких тестов в том, что они не настаивают на какой-то конкретной имплементации. Единственное, на что смотрят сквозные тесты — это поведение приложения с точки зрения конечного пользователя. Они настолько отделены от деталей имплементации, насколько это возможно.

Наряду с преимуществами, у сквозных тестов имеется крупный недостаток: **они очень медленные**. Любой проект, который полагается исключительно на такие тесты, не сможет получить быструю обратную связь. Именно поэтому невозможно обеспечить покрытие кода только сквозными тестами.

## Тривиальный тест

В отличие от сквозных тестов, тривиальные тесты предоставляют быструю обратную связь. Кроме того, вероятность ложных срабатываний также мала, поэтому они обладают хорошей устойчивостью к рефакторингу.

Тем не менее **тривиальные тесты вряд ли смогут выявить какие-либо ошибки, потому что покрываемый ими код слишком прост**. Такие тесты покрывают простой фрагмент кода, вероятность сбоя в котором невелика.

### Пример.

Класс пользователь, с полем имя, и два метода, метод `getName` который возвращает имя, и `setName`, с помощью которого можно задать пользователю новое имя:

```
public class User {
    private String name;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

В тесте для этого класса, мы проверяем что имя пользователя соответствует John Smith. Тест проходит, но он проверяет очень простые участки кода, ошибка в которых очень маловероятна:

```
class UserTest {
    @Test
    void checkUserName() {
        User testUser = new User();
        testUser.setName("John Smith");
        assertEquals("John Smith", testUser.getName());
    }
}
```

## Хрупкие тесты

Хрупкие тесты работают быстро и хорошо выявляют ошибки в коде, но делает это с множеством ложных срабатываний. Такие тесты называются хрупкими. Они падают при любом рефакторинге тестируемого кода независимо от того, изменилась тестируемая ими функциональность или нет. Тривиальные тесты обладают хорошей устойчивостью к рефакторингу и обеспечивают быструю обратную связь, но не защищают от багов.

Пример теста, проверяющего, какая команда SQL (запрос к базе данных) была выполнена:

```
public class UserRepository {
    public User getUserById(int id) {
        // Метод ищет в базе данных пользователя
        // по его идентификатору и возвращает его
        return user;
    }

    public String getLastSQLQuery() {
        String sqlQuery;
        // Метод возвращает последний использованный запрос к
        базе данных
        return sqlQuery;
    }
}
```

Этот тест проверяет, генерирует ли класс UserRepository правильную команду SQL при выборке пользователя от базы данных:

```
@Test
```

```
void getByIdCheckSqlquery() {  
    UserRepository userRepository = new UserRepository();  
    User user = userRepository.getUserById(5);  
    assertEquals("SELECT * FROM dbo.[User] WHERE UserID = 5",  
userRepository.getSqlquery());  
}
```

Подобный тест может обнаружить разные ошибки. Например, разработчик может ошибиться в SQL-коде и использовать ID вместо UserID, тогда тест упадёт, сообщив об этой ошибке. Но он не обладает хорошей устойчивостью к рефакторингу. Существует несколько разновидностей команды SQL, которые приводят к одному и тому же результату:

```
SELECT * FROM dbo.[User] WHERE UserID = 5  
SELECT * FROM dbo.User WHERE UserID = 5  
SELECT UserID, Name, Email FROM dbo.[User] WHERE UserID = 5  
SELECT * FROM dbo.[User] WHERE UserID = @UserID
```

Тест `getByIdCheckSqlquery` упадёт при замене SQL-кода любой из этих разновидностей, хотя сама функциональность остаётся работоспособной.

## Тестирование по принципу чёрного и белого ящика

### Тестирование «чёрного ящика»

Устойчивость к рефакторингу формируется за счёт проектирования теста на проверку результата, а не реализацию. Такой принцип проектирования тестовых методов называется тестированием «чёрного ящика» (англ. Black-box testing).



Мы ничего не знаем о системе

В основе метода лежит принцип, согласно которому мы для написания тестов **используем документацию, а не знания о реализации**. Принцип используется не только в юнит-тестировании, также и в интеграционных и сквозных тестах, в тестировании графического интерфейса. Как следует из названия этой методологии, вы рассматриваете систему «чёрный ящик».

Представьте, что перед вами стоит закрытая коробка с кнопками и светодиодами. Вы не знаете, что находится внутри или как работает система, а знаете, что при правильном вводе система выдаёт желаемый результат. Всё, что вам нужно знать для правильного тестирования системы — это функциональная спецификация системы. Вы нажимаете кнопки, получаете результат и сравниваете его с результатом в спецификации.

Проблема в том, что эти изменения всё ещё могут нарушать непроверенную функциональность. Чтобы устранить эту проблему, вам необходимо точно знать, какой код выполняется, когда вы (или сборка) вызываете тесты.

В тестировании системы может принять участие любой желающий, например, инженер по контролю качества, разработчик или даже заказчик.



Мы все знаем о системе

## Тестирование «белого ящика»

Тестирование по принципу **«белого ящика»** работает по противоположному принципу. Этот метод тестирования проверяет внутренние механизмы приложения. Тесты строятся **на основе исходного кода, а не на основе требований или спецификаций**.

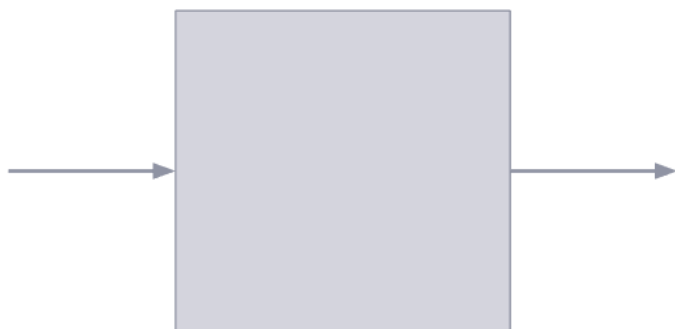
В этом типе тестирования мы используем подробные знания о реализации для создания тестов. Кроме понимания реализации компонента мы должны знать, как этот процесс тестирования взаимодействует с другими компонентами. По этим причинам разработчики являются лучшими кандидатами для создания тестов белого ящика.

## Сравнение методов

	Защита от багов	Устойчивость к изменениям
Тестирование по принципу белого ящика	Хорошая	Плохая
Тестирование по принципу чёрного ящика	Плохая	Хорошая

У обоих методов есть как достоинства, так и недостатки.

**Тестирование по принципу «белого ящика»** обычно получается более **тщательным**. Анализируя исходный код, можно выявить множество ошибок, которые часто упускаются, когда вы полагаетесь исключительно на внешние спецификации. С другой стороны, результаты тестирования по принципу «белого ящика» часто оказываются хрупкими, поскольку они часто завязаны на детали имплементации тестируемого кода. Такие тесты генерируют много ложных срабатываний и поэтому не имеют хорошей устойчивости к изменениям. Тестирование по принципу «чёрного ящика» обладает противоположными достоинствами и недостатками, это показано на таблице.



Мы знаем о системе **не все**

В некоторых ситуациях нужны ориентированные на пользователя тесты, а в других нужно протестировать детали реализации системы. Чаще всего используют что-то среднее, комбинацию методов «чёрного ящика» и «белого ящика» — это тестирование **«серого ящика»**, оно предполагает частичную осведомлённость о внутренних процессах.

## Тестирование «серого ящика»

Тестирующий серого ящика использует **ориентированный на код подход тестирования белого ящика** и объединяет его с различными подходами тестирования **чёрного ящика**, такими как функциональное тестирование и

регрессионное тестирование. Тестировщик оценивает как внутреннюю работу программного обеспечения, так и его пользовательский интерфейс.

## Метрики тестов

Существуют специальные методики подсчёта степени того, насколько протестировано и приложение, их называют **метриками покрытия**.

Техника покрытия кода была одной из первых методик, изобретённых для систематического тестирования программного обеспечения (особенно, при тестировании белого ящика).

При внесении изменений вы запускаете тесты, которые дают вам обратную связь о новых тестируемых методах и о том, нарушают ли изменения существующие тесты. Проблема в том, что эти изменения всё ещё могут нарушать непроверенную функциональность, а вы можете об этом не знать. Чтобы устранить эту проблему, вам необходимо точно знать, какой код выполняется, когда вы или сборка вызываете тесты. В идеале тесты должны охватывать 100% кода вашего приложения.

Метрики могут сказать о качестве вашей программы, но это также спорный показатель. Высокое покрытие кода тестами ничего не говорит вам о качестве тестов. Хороший программист должен уметь видеть дальше чистого процента, полученного в результате выполнения тестов.

Существует несколько различных способов измерения покрытия, вот самые популярные из них:

- Покрытие строк (**code coverage**): сколько строк исходного кода было протестировано.
- Покрытие ветвей (**branch coverage**): сколько ветвей управляющих структур (например, операторов if) было выполнено.
- Покрытие требований (**requirements coverage**): доля требований, покрытых набором тестов и реализованных в коде.

Наиболее часто используемая метрика покрытия — code coverage (другое название — test coverage). Эта метрика равна отношению количества строк кода, выполняемых одним тестом, к общему количеству строк в основном коде проекта. Code coverage вычисляется, как отношение количества строк кода, выполняемых тестами, к общему количеству строк в основном коде проекта.

$$\text{Code coverage (test coverage)} = \frac{\text{Количество выполненных строк кода}}{\text{Общее количество строк кода}}$$

Рассмотрим метод, который считает длину переданной строки и возвращает false, если длина превышает 3 символа. Напишем тест. Посчитаем покрытие. Общее количество строк в методе checkStringLength равно 5 (вместе с фигурными скобками, они тоже считаются):

```
public static boolean checkStringLength(String s)
{
    if (s.length() <= 3) {
        return true;
    }
    return false;
}
```

```
@Test
void checkStringLengthTest() {
    String s = "abc";
    assertEquals(checkStringLength(s), true);
}
```

Количество строк, выполняемых в тесте 3, так как мы не выходим из условия. Таким образом, покрытие равно: **Code coverage = 3 / 5 \* 100% = 60%**

Помните о том, что процент покрытия не совсем объективная метрика? Вот хороший пример: выполняем метод checkStringLength так, чтобы не было конструкции if-else.

```
public static boolean checkStringLength(String s)
{
    return s.length() <= 3;
}
```

```
public static boolean checkStringLength(String s)
{
    return s.length() <= 3;
}
```

И посчитаем процент покрытия:

**Code coverage = 3 / 3 \* 100% = 100%**



Покрытие увеличилось до 100%, но разве мы что-то улучшили в методе? Он работает по-прежнему. Поэтому не стоит слепо полагаться на процент покрытия.

Другая метрика покрытия называется **branch coverage** (покрытием ветвей). Она показывает более точные результаты, чем code coverage. Вместо использования количества строк кода, эта метрика ориентируется на управляющие структуры, такие как команды if и switch. Она показывает, какое количество таких управляющих структур обходится, по крайней мере, одним тестом в проекте.

$$\text{Branch coverage} = \frac{\text{Количество покрытых ветвей}}{\text{Общее количество ветвей}}$$

Branch coverage вычисляется, как отношение количества ветвей кода, выполненных хотя бы одним тестом, к общему количеству ветвей в коде. Чтобы вычислить метрику branch coverage, необходимо подсчитать все возможные ветви (branches) в коде и посмотреть, сколько из них выполняются тестами.

Вернёмся к предыдущему примеру:

```
public static boolean checkStringLength(String s)
{
    if (s.length() <= 3) {      1  1
        return true;
    }
    return false;              2
}
```

```
@Test
void checkStringLengthTest() {
    String s = "abc";
    assertEquals(checkStringLength(s), true);
}
```

Метод checkStringLength содержит две ветви: одна для ситуации, в которой длина строкового аргумента превышает три символов, и другая для строк, длина которых менее или равна 3 символам. Тест покрывает только одну из этих ветвей, поэтому метрика покрытия составляет  $1/2 = 0,5 = 50\%$ . При этом неважно, какое представление будет выбрано для тестируемого кода — будете ли вы использовать команду if, как прежде, или выберете более короткую запись. Метрика branch coverage принимает во внимание только количество ветвей; она не учитывает, сколько строк кода понадобилось для реализации этих ветвей.

**Branch coverage =  $1 / 2 * 100\% = 50\%$**

Ещё одна метрика — покрытие требований (англ. **requirements coverage**), показывает процент бизнес-требований, проверяемых тестом.

Вспомним наши тесты калькулятора. Допустим, что из требований нам нужно было написать программу калькулятор с функциями сложения, вычитания, умножения, деления и вычисления факториала. Допустим, мы написали всё, кроме вычисления факториала, и реализовали только 4 из 5 требований. То есть нет ни теста, ни функции. Все тесты проходят корректно, но согласно данной метрике тестами покрыто только 80%, для 100% покрытия нужно реализовать функцию вычисления факториала и протестировать её.

```
public static int calculation(...) {  
    int result;  
    switch (operator) {  
        case '+':    1 из 5 требований  
            ...  
        case '-':    2 из 5  
            ...  
        case '*':    3 из 5  
            ...  
        case '/':    4 из 5  
            ...  
    }  
    return result;  
}
```

**Requirements coverage = 4 / 5 \* 100% = 80%**

Кажется, что одна метрика даёт результаты лучше, чем другая. Но вы не сможете положиться на одну метрику для определения качества тестов по двум причинам:

🔥 Невозможно гарантировать, что тест проверяет все компоненты результата работы тестируемой системы. Пример теста `checkStringTest`. Его метрика равна 100%, но он ничего не проверяет:

```
@Test  
void checkStringTest() {  
    checkStringLength("abc");  
    checkStringLength("abcd");  
}
```

🔥 Ни одна метрика покрытия не может учитывать ветвления кода во внешних библиотеках. Влияния библиотек могут быть учтены только, если они могут быть извлечены из исходного кода и проанализированы, как часть метрики покрытия.

Полезно иметь высокое покрытие в наиболее важных частях системы. Плохо превращать такое высокое покрытие в требование. Аналогичным образом стремление к конкретному проценту покрытия создаёт неверный стимул, противоречащий цели юнит-тестирования. Вместо того чтобы сосредоточиться на тестировании действительно важных вещей, люди начнут искать способы для достижения этой искусственной цели.

💡 Часто в компаниях останавливаются на уровне в 80%, так обеспечивается оптимальное соотношение между временем, потраченным на написание тестов, и пользой юнит-тестирования.

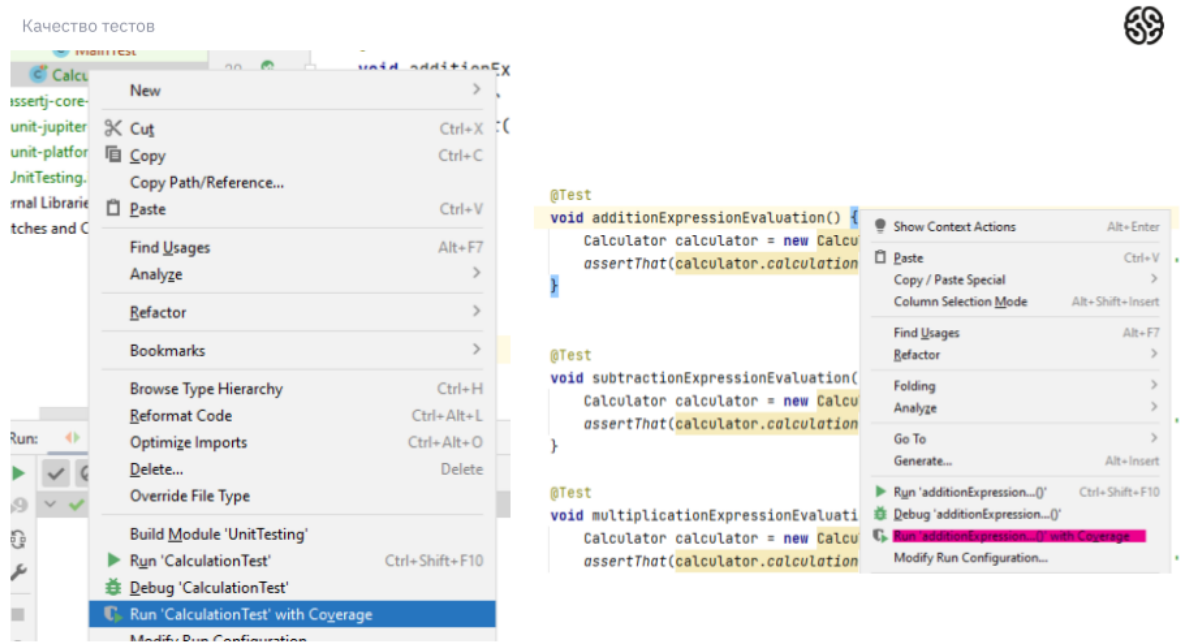
Аналогичным образом метрики покрытия служат хорошим негативным, но плохим позитивным признаком. Низкий процент покрытия — хороший признак проблем с тестами, но высокий процент покрытия ещё не означает высокого качества тестов. Branch coverage предоставляет более качественную информацию о полноте тестов, чем code coverage, но по нему нельзя судить о том, достаточно хороши ваши тесты или нет. Ни одна из метрик покрытия не учитывает наличия проверок (assertions) и ветвей выполнения в сторонних библиотеках, используемых в вашем проекте.

Но метрики остаются очень полезным инструментом для оценки тестирования кода, особенно учитывая, что сегодня есть много инструментов для расчёта метрик.

## Инструменты для измерения покрытия тестами

Для измерения покрытия тестами можно воспользоваться встроенным в среду разработки IntelliJ IDEA инструментом. Для этого перейдём в класс с тестами, наведём либо на отдельный тест либо на класс и выберем «Run 'Test' with Coverage» (англ. «запустить [имя класса] с покрытием»). После выполнения тестов, в контекстном меню справа можно будет увидеть отчёт о покрытии кода тестами. Покрытие тестами строки кода будут выделяться зелёным цветом, не

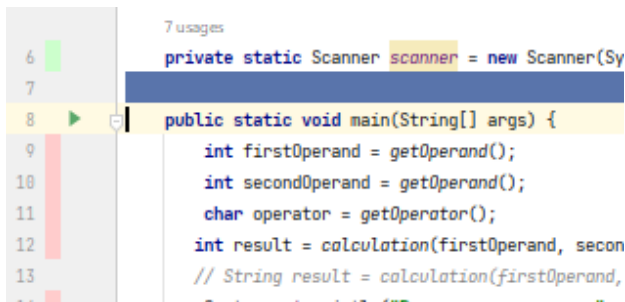
покрытые — красным. Можно запускать как отчёты для отдельных тестов, так и для всего тестового класса.



Результаты тестирования будут представлены в таблице справа:

Element	Class, %	Method, %	Line, %
ru	50% (1/2)	28% (2/7)	38% (14/36)
gb	50% (1/2)	28% (2/7)	38% (14/36)
units	50% (1/2)	28% (2/7)	38% (14/36)

Код, не покрытый тестами, выделен красным цветом, а покрытый — зелёным.



Существует множество инструментов, предназначенных для оценки тестового покрытия, наиболее популярные из них:

- **JaCoCo** — это бесплатная библиотека покрытия кода Java.

- **Cobertura** — это бесплатный инструмент Java, который вычисляет процент кода, доступного тестам. Его можно использовать для определения того, какие части вашей Java-программы не охвачены тестированием.
- **OpenClover** — измеряет охват кода для Java и Groovy и собирает более 20 показателей кода. Он не только показывает вам непроверенные области вашего приложения, но и объединяет охват и метрики для поиска наиболее рискованного кода.

	OpenClover	Atlassian Clover	Cobertura	JaCoCo	JCov	Code Cover	PIT
<b>Report types</b>							
HTML	✓ <i>details</i>	✓ <i>details</i>	✓	✓	?	✓	✓
PDF	✓	✓	✗	✗	?	✗	✗
XML	✓	✓	✓	✓	✓	✗	✓
JSON	✓	✓	✗	✗	?	✗	✗
Text	✓	✓	✓ <i>via cobertura:check</i>	✗	?	✗	✗
CSV	✗	✗	✗	✓	?	✓	✗
<b>Supported languages</b>							
Java	✓	✓	✓	✓	✓	✓	✓
Groovy	✓		✓	✓	?	✗	✗
AspectJ	✓ <i>(details)</i>	✓ <i>(details)</i>	?	?	?	?	?

Существует ещё довольно много инструментов для оценки покрытия тестами, они отличаются поддерживаемыми библиотеками тестирования, видами экспортируемых отчетов и другими специфическими функциями. В таблице представлена часть сравнительной таблицы от OpenClover.

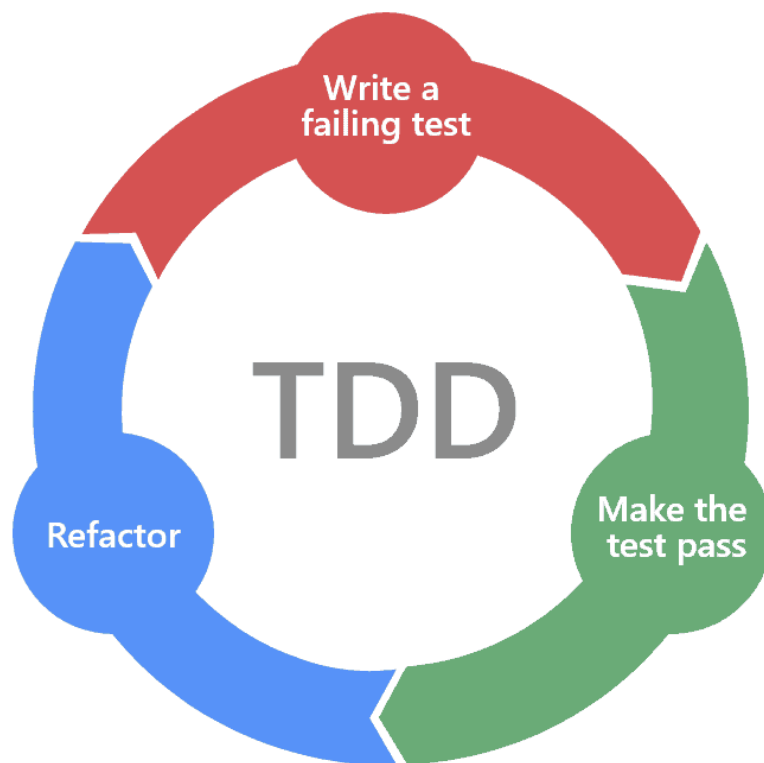
## Разработка через тестирование

Разработка через тестирование (англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки:

- Сначала пишется тест, покрывающий желаемое изменение.
- Затем пишется код, который позволит пройти тест.
- Под конец проводится рефакторинг нового кода к соответствующим стандартам.

Кент Бек, считающийся изобретателем этой техники, утверждал в 2003 году, что разработка через тестирование поощряет простой дизайн, и внушает уверенность.

TDD-методология может использоваться на уровне Юнит тестирования и интеграционного тестирования. Суть методологии в проектировании приложения по результатам тестов. Процесс написания тестов по такой методологии представляет собой цикл. Пишется тест, который не проходит, затем дописывается код, удовлетворяющий данному тесту. После этого всё повторяется. Так, шаг за шагом, строится приложение.



Процесс состоит из трёх (четырёх, по мнению некоторых авторов) стадий, которые повторяются для каждого тестового сценария:

1. Написать падающий тест (Write a failing test), который покажет, какую функциональность необходимо добавить и каким поведением она должна обладать.
2. Написать код, минимально достаточный для прохождения теста. На этой стадии код не обязан быть элегантным или чистым.
3. Провести рефакторинг кода. Вы можете безопасно «чистить» код, защищённый написанными ранее тестами, сделать его более читаемым и простым в сопровождении.

Добавим новую функцию вычисления площади круга, в зависимости от радиуса:

1. Добавим новую функцию вычисления площади круга, в зависимости от радиуса. Начнём с написания метода-заглушки, это нужно, чтобы не сталкиваться с ошибками компиляции. Метод принимает радиус, пока возвращает 0:

```
public double computeAreaCircle(double radius){  
    return 0;  
}
```

2. Создаём неудачный тест для этого метода. **314.1592653589793** — это значение площади круга с радиусом 10, оно рассчитано заранее, такое же значение должен вернуть нам наш метод. Проверяем, используя ассерты JUnit:

```
@Test  
void computeAreaCircle() {  
    Calculator calculator = new Calculator();  
    assertEquals(314.1592653589793,  
calculator.computeAreaCircle(10), "Should return right circle  
area");  
}
```

3. Запускаем тест. Он ожидаемо провалился, так как метод возвращает ноль. Переходим на вторую итерацию, рефакторинг основного кода:

**Expected: 314.1592653589793**

**Actual: 0.0**

4. Теперь возвращаемся к методу, и рефакторим код: вместо return 0 используем формулу расчёта площади круга по радиусу, пи на радиус в квадрате, в качестве пи используем дробное число 3.14:

```
public double computeAreaCircle(double radius){  
    return 3,14 * radius * radius;  
}
```

5. Ожидаемое и фактическое значение уже почти совпадают, по крайней мере, целые части:

**Expected: 314.1592653589793**

**Actual: 314.0**

Но тест не проходит. Это связано со спецификой сравнения чисел с плавающей точкой (double) в java. Если переменные типа int сравниваются на точное

совпадение, то для неточных чисел нужно учитывать погрешность при сравнении:

Дело в том, что в двоичной системе невозможно точно представить число 314,1592..., поэтому сравнение таких чисел производят с некоторой погрешностью, которую можно задать, это будет точность сравнения. Это можно сделать, используя класс `Math`. Этот класс располагается в пакете `java.lang` и предоставляет набор статических методов для осуществления ряда различных математических вычислений.

Так сравниваются два числа: **`Math.abs(a - b) < delta`**

**`abs`** — метод для вычисления модуля числа (разницы, в нашем случае).

**`delta`** — степень точности сравнения (погрешность).

Поэтому правильно в данном тесте будет сравнивать погрешности ожидаемую и фактическую. Но, нам уже заранее известна ожидаемая от метода точность, в виде числа 314.1592653589793, поэтому воспользуемся константой `Pi` из того же класса `Math` — это статическая конечная двойная константа в Java, эквивалентная в  $\pi$ -математике. Используем `Math.PI` и запускаем тест. Возвращаемся к методу и проводим рефакторинг снова, на этот раз вместо 3.14 используем переменную класса `Math`.

```
public double computeAreaCircle(double radius){  
    return Math.PI * radius * radius;  
}
```

6. Тест пройден, мы завершили разработку функции с помощью TDD. В итоге у нас появился правильно работающий код основной функции и корректный тест, который сработает, если что-то снова пойдёт не так или мы захотим как-то изменить функцию.

### Преимущества TDD:

1. **Уменьшает зависимость от отладки:** поскольку вначале основное внимание уделяется написанию тестов, а затем — созданию кода, предназначенного для прохождения указанных тестов, многие разработчики считают, что жизненный цикл может значительно сократить потребность в отладке. Поскольку во время написания и кодирования теста требуется более глубокое понимание логических и функциональных требований, причину неудачного теста часто можно быстро распознать и устранить.



2. **Анализирует пользовательский опыт:** процесс первоначального осмысления и написания теста заставляет мозг программиста работать наоборот: сначала продумывать, как будет использоваться функция и как она может быть реализована, а только потом, как должен быть написан тест. Это побуждает учитывать особенности пользователя.
3. **Сокращает общее время разработки:** практика показывает, что уменьшается общее время разработки проекта по сравнению с традиционными методами написания кода. В то время как общее количество строк кода увеличивается (из-за тестов), частое тестирование исключает ошибки в процессе и выявляет существующие намного раньше, предотвращая возникновение проблем в дальнейшем.

#### **Недостатки TDD:**

1. **Не любит большой дизайн.** Разработчики часто пишут самый простой из возможных тестов, а затем проходят этот тест с помощью максимально простого кода, это приводит к серьезному дефициту возможностей в отношении общего дизайна функции или всего проекта. При использовании подобной практики слишком легко не заметить крупных ошибок, так как всё внимание сосредоточено на мелочах.
2. **Не подходит для всех.** Метод отлично зарекомендовал себя для работы с небольшими проектами или с небольшими компонентами или функциями крупных проектов. К сожалению, методика может давать сбои при применении к очень объёмным и сложным проектам. Написание тестов для сложной функции, которую программист ещё полностью не представляет, может быть трудным, и даже невозможным. Написание тестов – это хорошо, но если эти новые тесты не совсем точно соответствуют требованиям функции, они бесполезны (или даже могут активно препятствовать разработке). Более того, некоторые проекты, особенно использующие устаревший код или сторонние системы, просто не поддаются практическим методам. Для них практически невозможно создать тесты, которые должным образом интегрируются с этими системами или с устаревшим кодом.
3. **Требует дополнительных затрат времени.** Время, затрачиваемое на предварительную генерацию тестов, экономится позднее в жизненном цикле разработки. Тем не менее, для подготовки и написания тестов требуется значительное время. Его многие разработчики предпочитают потратить на написание нового кода или рефакторинг существующего.

# Разработка через поведение

**BDD** (сокр. от англ. Behavior-driven development, дословно «разработка через поведение») — это методология разработки программного обеспечения, являющаяся ответвлением от методологии разработки через тестирование (TDD).

BDD фокусируется на следующих вопросах:

- С чего начинается процесс.
- Что нужно тестировать, а что нет.
- Сколько проверок должно быть совершено за один раз.
- Что можно назвать проверкой.
- Как понять, почему тест не прошёл.

Поведенческая разработка (BDD), созданная Дэном Нормом в середине 2000-х годов, представляет собой методологию разработки ИТ-решений, непосредственно удовлетворяющих бизнес-требованиям.

Основной идеей этой методологии является совмещение в процессе разработки чисто технических интересов и интересов бизнеса, позволяя тем самым управляющему персоналу и программистам говорить на одном языке.

Для общения между этими группами персонала используется предметно-ориентированный язык, основу которого представляют конструкции из естественного языка, понятные неспециалисту, обычно выражающие поведение программного продукта и ожидаемые результаты.

## Пример разработки с использованием TDD и BDD:

**Задача:** Нам нужно сделать форму, в которую мы вводим возраст котика и его вес, а в ответ получаем, сколько корма котик должен съесть в сутки.

Как подойти к этой задаче, используя **TDD** подход:

- Пишем тест, в котором проверяем, что функция `getCatFood()` возвращает нужные значения в разных ситуациях.
- Проверяем, что тесты упали (кода ещё нет).
- Пишем код функции очень просто — так чтобы тесты прошли.
- Проверяем, что тесты прошли.
- На этом шаге можем задуматься о качестве кода. Можем спокойно рефакторить и изменять код как угодно, т.к. у нас есть тесты, которые с уверенностью скажут, что мы где-то ошиблись.
- Повторяем все вышеуказанные шаги ещё раз.

Вот как будет выглядеть решение с **BDD** подходом:

- Процесс начинается с того, что пользователь открывает форму.
- Нам нужно протестировать числа, которые выдаёт форма.
- Нам нужно ввести 10–20 разных значений.
- Проверка в этом случае это нажатие на Submit и проверка значения.
- Тест пройдёт если результат на форме соответствует «правильным» значениям.
- Далее мы это описываем с помощью специального синтаксиса (он зависит от используемого инструмента). Например:

**Функция:** Расчёт количества корма.

**Сценарий:** При вводе валидных параметров отображается правильный ответ.

Когда я нахожусь на странице с формой, то ввожу возраст 5 лет и вес 5 кг. Тогда мне отображается количество корма 500 г.

Потом эти шаги реализуются в коде.

## Подведём итоги

На этом уроке мы узнали про:

- Как проектировать хорошие тесты и для чего это нужно.
- Тестирование по принципу чёрного и белого ящика.
- Метрики тестов.
- Инструменты для измерения покрытия тестами.
- Тестирование через разработку.
- Тестирование через поведение.

## Что можно почитать ещё?

1. [Тестирование по стратегии чёрного ящика — Википедия](#)
2. [Cobertura](#)
3. [JaCoCo Java Code Coverage Library](#)
4. [TDDx2, BDD, DDD, FDD, MDD и PDD, или всё, что вы хотите узнать о Driven Development](#)

## Используемая литература

1. Хориков Владимир «Принципы юнит-тестирования». — СПб.: Питер, 2021. — 320 с.
2. Бек Кент «Экстремальное программирование: разработка через тестирование». — СПб.: Питер, 2022. — 250 с.
3. Catalin Tudose «JUnit in Action, 3rd Edition». — Manning, 2020. — 560 с.