ЛАБ-1-Защита

ВАЖНО!
Написано @kkettch по лекциям Клименкова и иногда с помощью ChatGPT, использовалась для защиты ЛР. Информация может оказаться старой или недостоверной.

1. Понятие тестирования ПО. Основные определения

Тестирование необходимо, так как:

- Есть человеческий фактор, из-за которого всегда код может содержать ошибки
- Есть сильное давление со стороны бизнеса, так как ПО нужно выпускать как можно быстрее, пока оно не потеряло актуальность на рынке. Чаще всего на рынок выходит ПО, которое почти не проходило никакого тестирования. Таким образом тестирование нужно, чтобы систематически повышать качество ПО

Основные термины:

- **Mistake (error)** - ошибка, просчет человека. Появляются как ошибка человека, которая может быть не специальная. Приводит к тому, что в программе появляется *Fault*
- **Fault** - дефект, изъян. В дальнейшем этот дефект приводит к тому, что программа будет работать не так, как задумано. Из-за этого произойдет *Failure*
- **Failure** - неисправность, отказ, сбой, который является внешним проявлением дефекта (*Fault*). Кроме того отказ может являться следствием какой-то окружающей среды (например, сбой электропитания)
- **Error** - невозможность выполнить задачу вследствие отказа. В результате *Failure* (сбоя), происходит ошибка, которая не позволяет выполнить задачу так, как это должно быть.

Есть еще такой понятия как **Bug** неформальное определение. Может обозначать: :

- Дефект (*Fault*)
- Отказ (*Failure*)
- Невозможность выполнить задачу (*Error*)
- Что-то другое или ничего не обозначать

К определениям можно также отнести **уровни восприятия тестирования** (различное отношение организаций к необходимости тестирования):

1. **Уровень 0**: тестирование == отладка.

- Работой с отладчиком нельзя обеспечить повторяемую работы программы, чтобы перед каждым build'ом проверять наличие дефектов.
- Не отличает некорректное поведение и ошибки программы.
- Не учитывает требования надежности и безопасности

2. **Уровень 1**: предназначение тестирования - показать корректность работы ПО начальству, коллегам и т.д.
   - Физически невозможно доказать, так как успешное тестирование не доказывает отсутствие дефектов
   - Нет формальных правил
3. **Уровень 2**: поиск ошибок разработчиков
   - Тестировщики систематически ищут ошибки разработчиков, что приводит к конфликтам разработчиков и тестировщиков
4. **Уровень 3**: тестирование может показать наличие ошибок
   - Есть попытка минимизировать возможные риски. Тогда можно свести последствия ошибок к незначительным. Если на риски не обращать внимания, то последствия могут быть катастрофическими
   - Тестировщики и разработчики должны совместно снижать риски
5. **Уровень 4**: тестирование - возможный способ оценки качества ПО в терминах найденных дефектов
   - Используется в реальных инженерных компаниях
   - Фиксируется количество дефектов. Если они растут, значит что-то не так с разработчиками или другими отделами. Необходимо стремиться к снижению дефектов
   - Дефекты могут быть *функциональными* (в технической отработке программы) и *не функциональными* (надежность, практичность, эффективность)

Кроме того есть другие способы оценки качества продукты:

1. Разработка стандартов. При использовании определенного стандарта происходит стандартизация интерфейсов. Люди будут пользоваться одним и тем же интерфейсом, что снижает количество дефектов, связанных с реализацией функционала
2. Обучение. Внутренние тренинги внутри компании
3. Систематический анализ дефектов.

2. Цели тестирования. Классификация тестов.

Основные цели тестирования по стандарта ISTQB (International Software Testing Qualifications Board):

- Обнаружение дефектов
- Повышение уверенности в уровне качества продукта

- Предоставление информации для принятия решения (готово ли ПО к выходу, сколько сейчас имеется дефектов, какова плотность дефектов). Это должно помочь менеджерам решить можно ли выпускать проект
- Предотвращение дефектов

В целом, **тестирование** - увеличение приемлемого уровня пользовательского доверия в том, что программа функционирует корректно во всех необходимых обстоятельствах. Необходимо увеличить восприятие пользователя, что его программа работает корректно

Для этого необходимо определить:

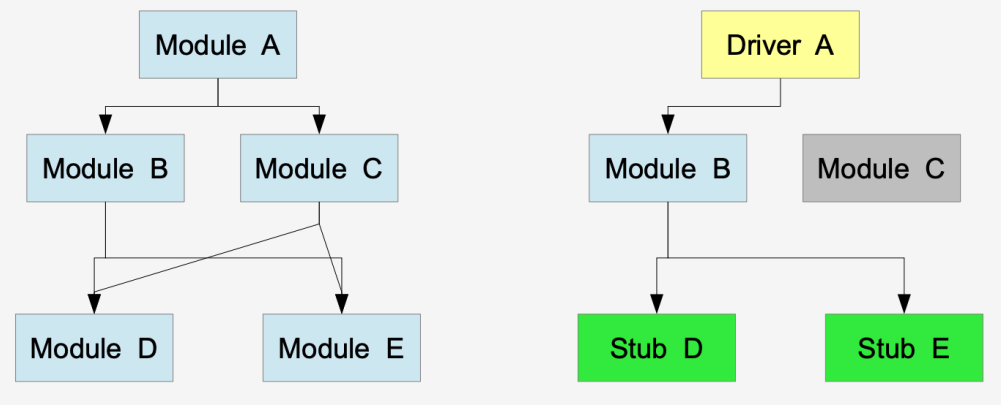
1. Уровень доверия
   - Необходимо *наглядно* продемонстрировать пользователю, что ошибок нет.
   - Можно показывать такую штуку, как *уровень остаточного обнаружение дефектов*. Например, строить график числа дефектов обнаруженных в заданное время (продемонстрировать, что их количество снижается)
   - Сформировать *требования к надежности* и провести испытания.
2. Корректное поведение
   - Необходимо определение из детальных требований
   - Из спецификация, описаний
   - Зависит от условия тестирования
3. Реальное окружение - те данные, которые будут использоваться на самом деле, а не по предположению аналитиков
   - Необходимо реалистичное количество данных (напр. пользователей) - такое же как в целевой системе

Сколько же нужно тестов:

- *Полное тестовое покрытие*, говорит о том, что необходимо ровно такое количество тестов, сколько путей в нашей системе может существовать.

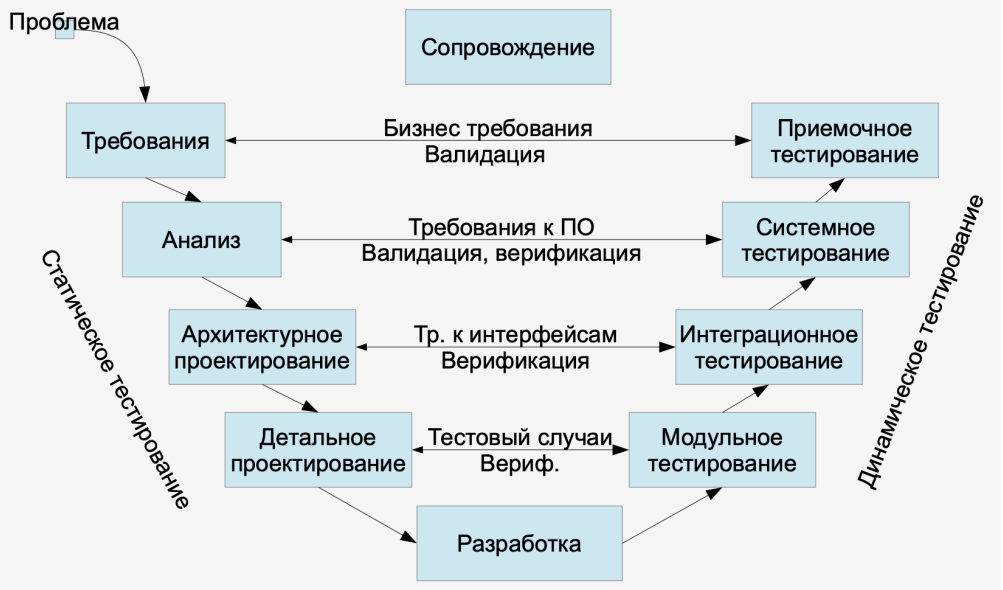
3. Модульное тестирование. Понятие модуля.

Модульное (компонентное) тестирование - тестирование отдельных компонентов ПО. ***Модуль*** - это компонент, который нужно тестировать отдельно от остального программного продукта. Он выполняет некоторую законченную функцию. Совокупность методов и классов формирует ***программный модуль***. Модули определены в дизайне программы.



Изолирование модулей производиться ради исключения стороннего воздействия.
Драйвер - компонент, вызывающий модули и обеспечивающий последовательность тестирования. Драйвер должен последовательно вызывать тестируемый модуль с разными входными данными.
Заглушка ведет себя подобно подчиненному модулю, имеет тот же интерфейс, но гораздо более простую реализацию. При вызове заглушка возвращает заранее определённые значения.

4. V-образная модель. Статическое и динамическое тестирование.



V-модель предложена Барри Боемом и Джэком Мансоном в то же время, когда Ройс выложил статью о своем подходе. “version of the V-model came up as something that was ‘in the air’ among Southern California aerospace companies at the time”

В основе лежит та же последовательность шагов, что и в водопадной модели, но каждому уровню разработки соответствует свой уровень тестирования. Модульное, интеграционное и системное тестирования проводятся последовательно. Последним этапом является приемочное тестирование - соответствие основным функциональным требованиям.

Для проведения тестирования сначала необходимо определить корректное поведение программы - оно должно быть четко задано вне кода разработанной системы.

- Левая половина - статические тесты. Проводятся на ранней стадии проекта для выявления грубых ошибок в проектировании. Проверка артифактов разработки без их компьютерного исполнения.
- Правая половина - динамические тесты. Компьютерное исполнение тестов для проверки функциональности системы.
  Каждому уровню разработки ПО ставится свой уровень тестирования.

Валидация - сделали ли то, что было нужно?
Верификация - сделали ли правильно?

- Статическое (рецензирование)
  - Не включает выполнения кода
  - Ручное, автоматизированное
  - Неформальное, сквозной контроль, инспекция
- Динамическое
  - Запуск модулей, групп модулей, всей системы
  - После появления первого кода (а иногда перед!)

Статическое тестирование - не связано с запуском набора тестов, разработанных для ПО Включает в себя рецензирование и инспекции кода.

Динамическое тестирование - требует уже созданной программной архитектуры (осуществляется сборка и запуск модулей или всей системы)

TDD - Test Driven Develpment - тесты разрабатываются перед разработкой кода. По мере разработки отчет о проведении тестирования начинает “зеленеть”

5. Валидация и верификация. Тестирование методом "чёрного" и "белого" ящика.

- Валидация
  - Проверка на соответствие ожиданиями
  - ПО выполняет требования пользователя?
  - Пирожок (мясной, вегетарианский, сладкий)
  - Have we done the right thing?
- Верификация
  - Внутреннее управление качеством
  - ПО выполняет требования спецификации?
  - Пирожок (размер, степень прожарки, начинка, …)
  - Have we done the thing right?

Валидация - сделали ли то, что было нужно?
Верификация - сделали ли правильно?

Метод “черного ящика”: содержимое программы от нас скрыто. Проверка тестами проходит на основе спецификации, требований и дизайна.

Метод “белого ящика”: содержимое программы нам доступно. Мы можем проверить код аналитически, а также оценить размер тестового покрытия. Для этого из приложения строиться граф, где части кода - это узлы, циклы и ветвления - переходы.

После определяется цикломатическая сложность программы, которая определяет число путей обхода кода программы. Цикломатическая сложность определяет максимальное необходимое число тестов.

Также ***опыт разработчика*** тестов позволяет минимизировать затраты и ускорить работы. ***UML-диаграммы*** также в этом помогают.

Роли и деятельность в тестировании:

- Проектирование тестов (на основании формальных критериев, на основании знаний предметной области, опыта и экспертизы). Высокая квалификация специалистов.
- Автоматизация тестов (знание средств, скриптов). Программисты, занимающиеся разработкой тестов скриптов или программ.
- Непосредственное исполнение тестов. Нет специальных требований к квалификации
- Анализ результатов. Необходимо знание предметной области.

6. Тестовый случай, тестовый сценарий и тестовое покрытие.

Тестовый случай состоит из набора входных значений, предусловий выполнения, ожидаемых результатов и постусловий. В идеальных условиях ожидаемые результаты должны быть определенны до момента выполнения теста. Также характерной чертой тестового случая является его ***повторяемость***. Должны учитываться ***состояния внутри ПО*** и переходы между ними.

- Входные значение
  - Данные или управляющие воздействия
- Предусловия, условия выполнения, постусловия
- Ожидаемый результат
  - Выходные данные и состояния, изменения в них, и другие последствия теста
  - Определен до запуска теста! (в идеале и TDD)

Тестовый сценарий - это последовательность тестовых случаев. Он должен обрабатывать как корректное поведение, так и вариант ошибки.

Тестовый план (Test Plan) - это документ, описывающий весь объем работ по тестированию программного продукта, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения. Перед составлением **тестового плана** необходимо разобраться, каковы результаты, ожидаемые от тестирования проекта.

Тестовое покрытие

При выборе количества тестов требуется баланс.

- Много тестов → больше покрытие → качество выше.
- Меньше тестов → выше скорость разработки → быстрее выход на рынок.

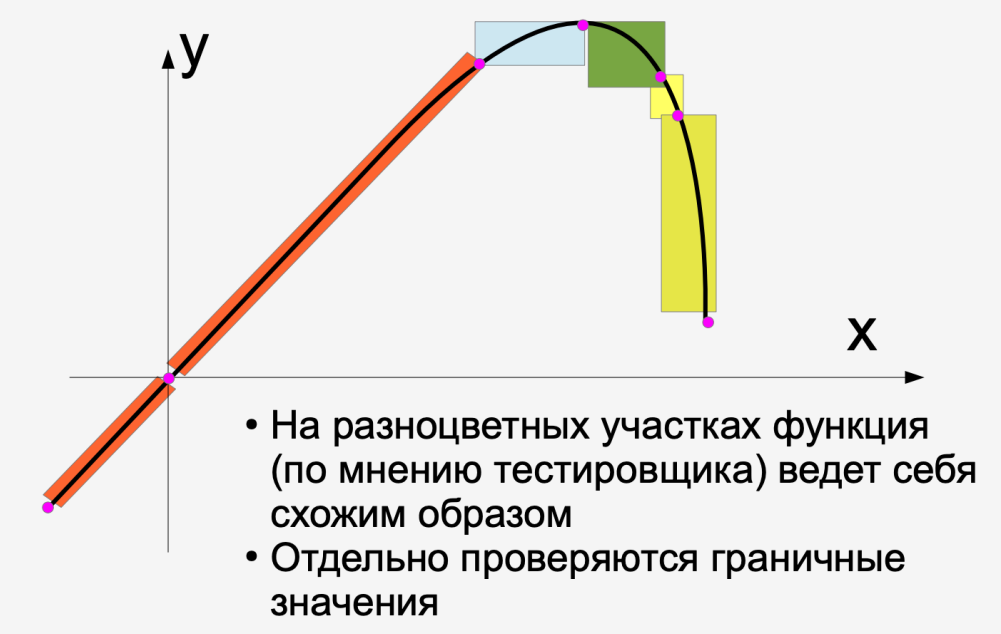
Необходимо учитывать, что нельзя тестировать вечность и полное покрытие недостижимо. Необходимо выбрать специфические значения для тестирования.

Тестовое покрытие:

- Эквивалентное разбиение (партиции эквивалентности)
- Таблица решений
- Таблица переходов

- Сценарии использования

7. Анализ эквивалентности.

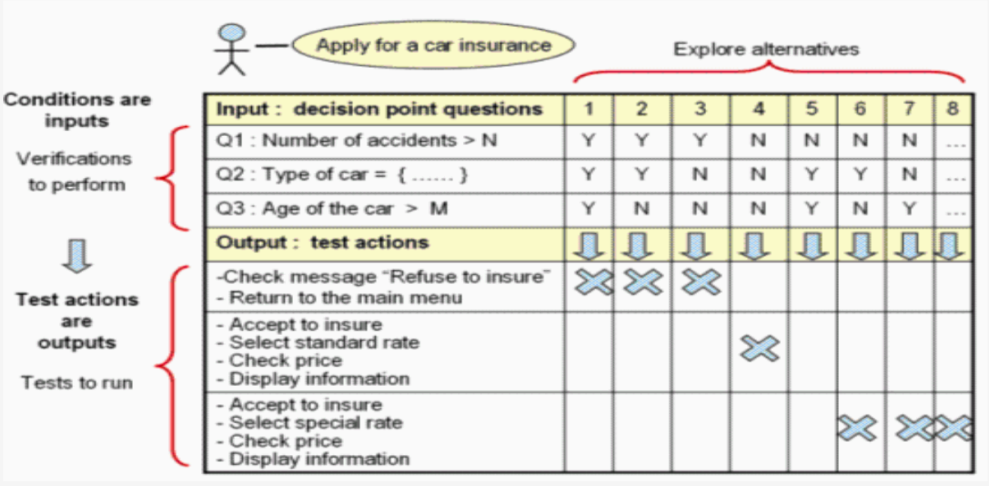


При анализе тестируемая функция или модуль разбиваются на отрезки, где программа ведет себя одинаково. Также отдельно проверяются граничные значения.

8. Таблицы решений и таблицы переходов.

Таблица решений позволяет сократить количество тестовых случаев.

- Используется в системах со сложной логикой, описание конечного автомата
- Может включать большой набор условий (обычно true-false), и действий



Input : decision point questions	1	2	3	4	5	6	7	8
Q1 : Number of accidents > N	Y	Y	Y	N	N	N	N	...
Q2 : Type of car = { ..... }	Y	Y	N	N	Y	Y	N	...
Q3 : Age of the car > M	Y	N	N	N	Y	N	Y	...
Output : test actions	⇩	⇩	⇩	⇩	⇩	⇩	⇩	⇩
-Check message "Refuse to insure" - Return to the main menu	X	X	X					
- Accept to insure - Select standard rate - Check price - Display information				X				
- Accept to insure - Select special rate - Check price - Display information						X	X	X

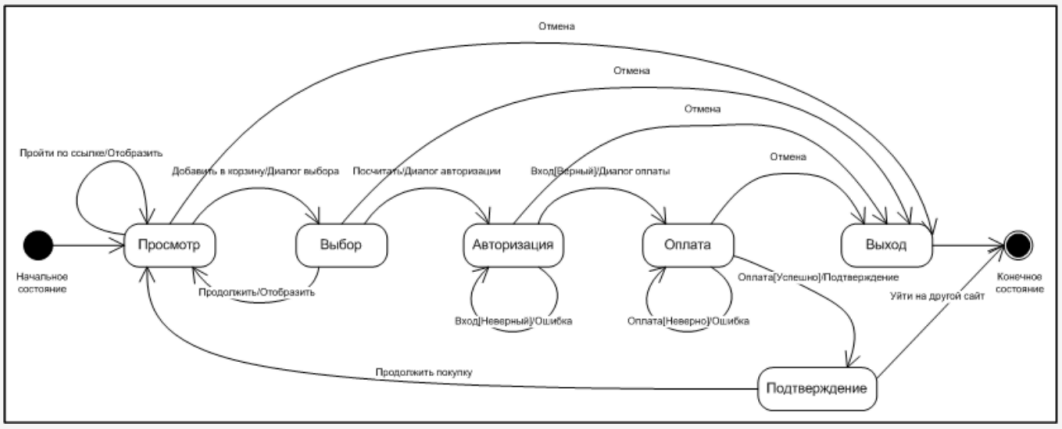
Перебираются все важные вопросы по решению успешного прохождения или нет:
При попытке купить страховку на машину система задает следующие вопросы:

- Сколько аварий было с этой машиной
- Какой тип машины (скоростная, медленная и т.п.)
- Возраст машины
  Исходя из комбинации ответов будет приниматься решение о том какую страховку выдать и выдать ли вообще.

Таким образом, можно не проверять все тестовое покрытие, а ограничиться лишь основными главными вопросами. Необходимо выбирать тестовые случаи, которые будут соответствовать тому или иному выбору

Таблица переходов

- Позволяют выбрать состояния и их комбинации, которые можно опустить
  - Рекс Блэк «Advanced Software Testing».
  - http://inrecolan.ru/blog/viewpost/361



Основной принцип: применяет в веб тестировании. У нас есть состояние корзины/ покупки и она проходит некоторое количество действий - валидное изменение состояний этой системы. Всегда есть переход в состояние отмена. Таким образом, определяется граф состояний и он всегда такой

- Прокрыть определенные строки тестами
- while (есть тесты с «непокрытыми» строками)
  - Выбрать состояния «не определено»
  - Попытаться покрыть тестом

Состояния		События/Условия		Текущее состояние	Событие/Условие	Действие	Новое состояние
				Просмотр	Пройти по ссылке	Отобразить	Просмотр
		Пройти по ссылке		Просмотр	Добавить в корзину	Диалог выбора	Выбор
		Добавить в корзину		**Просмотр**	**Продолжить**	**Не определено**	**Не определено**
		Продолжить		**Просмотр**	**Выписать**	**Не определено**	**Не определено**
Просмотр		Выписать		**Просмотр**	**Вход [неверный]**	**Не определено**	**Не определено**
Выбор		Вход [неверный]		**Просмотр**	**Вход [верный]**	**Не определено**	**Не определено**
Авторизация	X	Вход [верный]	=	**Просмотр**	**Оплата [неверно]**	**Не определено**	**Не определено**
Оплата		Оплата [неверно]		**Просмотр**	**Оплата [успешно]**	**Не определено**	**Не определено**
Подтверждение		Оплата [успешно]		**Просмотр**	**Отмена**	**Нет действия**	**Выйти**
Выйти		Отмена		**Просмотр**	**Продолжить покупку**	**Не определено**	**Не определено**
		Продолжить покупку		**Просмотр**	**Уйти на другой сайт**	**Не определено**	**Не определено**
		Уйти на другой сайт		**Выбор**	**Пройти по ссылке**	**Не определено**	**Не определено**
				{53 строки, сгенерированных по шаблону, не показаны}			
				Выйти	**Уйти на другой сайт**	**Не определено**	**Не определено**

Затем берутся имеющиеся состояния и создается таблица всех возможных состояний и событий. Получается огромное количество строк и каждая отдельная строка анализируется отдельно и проверяется возможность покрыть ее каким-либо тестом. Если да, то для этого состояния разрабатывается тест.

9. Регрессионное тестирование

Автоматизация тестов - это хорошо, но не всегда. Потому что ручное тестирование может оказаться проще и дешевле.

Регрессионное тестирование - это применение старых тестов для изменённой программы. Это позволяет понять, не влияют ли внесенные изменения на работы программы. Для автоматизации и проведения регрессионного тестирования необходимо обеспечить однозначное ***повторение тестового сценария*** (одинаковые входные данные порождают одинаковые выходные).

Тестирование совместимости - это тестирование ПО в разных окружениях.

10. Библиотека JUnit. Особенности API. Класс `junit.framework.Assert`.

JUnit - простейший фреймворк для создания модульных тестов и выполнения их в определённом окружении. JUnit4 построен на аннотациях.

@Test - аннотация, которой помечаются все методы тестирования. При этом фреймворк тестирования последовательно просматривает все загружаемые классы на наличие этой аннотации (при помощи reflection API). Внутри метода проверяется тестовое покрытие на соответствие определённым условиям при помощи специальных функций проверки, которые называются ***assertion***. Для организации тестового окружения существуют аннотации, позволяющие выполнить код перед всеми тестами,

после всех тестов, а также при загрузке тестового класса в память и его выгрузке из неё (@Before, @After, @BeforeClass, @AfterClass). Считается, что все тесты выполняются параллельно и независимо.

11. JUnit 4 от JUnit5

JUnit используется для модульного тестирования на Java.Kotlin. Модуль - компонент, который необходимо протестировать отдельно от остального прогаммного продукта. В разных ЯП модуль может быть методом, классом или совокупностью методов и классов, формирующих программный модуль. Для проведения модульного тестирования необходимо изолировать модуль из системы.

1. Вместо аннотации JUint4 `@Ignore` для пропуска теста в JUnit5 используется аннотация `@Disabled`
2. JUnit5 использует класс Assertions вместо Assert в JUint4:

```
Assertions.assertTrue(expectedValue, actualValue, message)
Assert.assertTrue(message, expectedValue, actualValue)
```

3. Аннотация `@Before` из JUint4 заменяется на `@BeforeEach` в JUnit5 (используется для обозначения того, что аннотированный метод должен выполняться перед каждым методом @Test, @RepeatedTest, @ParameterizedTest, или @TestFactory в текущем классе). Аналогично `@After` заменяется на `@AfterEach`
4. Аннотация `@BeforeClass` из JUint4 заменяется на `@BeforeAll` в JUnit5 (позволяет добавить метод, который будет вызван один раз перед всеми тестовыми методами).Аналогично `@AfterClass` заменяется на `@AfterAll`
5. JUnit5 ввел новые аннотации, чтобы сделать тесты более понятными и удобными для использования. Например, `@DisplayName` позволяет задавать читаемые названия для ваших тестов

```
@DisplayName("Тест проверки четности числа")
@ParameterizedTest(name = "Проверка, что число {0} – четное")
@ValueSource(ints = { 1, 2, 3, 4, 5 })
void testIsEven(int number) {
    assertTrue(number % 2 == 0, () -> "Число " + number + " должно
быть четным");
}
```