

Лекция 1.

**Введение в тестирование
программных продуктов**

Что такое тестирование ПО?

Что такое тестирование ПО?

Тестирование программного обеспечения — процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта.

Что такое тестирование ПО?

Тестирование программного обеспечения — это процесс проверки того, соответствует ли программа заданным требованиям и работает ли она корректно в различных условиях. Цель тестирования — **найти ошибки (баги)** до того, как программа попадёт к конечному пользователю. Тестирование может включать:

- запуск программы с разными входными данными,
- проверку, как она ведёт себя при ошибках или нестандартных ситуациях,
- сравнение фактического результата с ожидаемым.

Тестирование = отладка ?

Тестирование и отладка

Отладка (debugging) — это уже **процесс исправления** ошибок, которые были обнаружены в ходе тестирования (или при эксплуатации). Если тестирование отвечает на вопрос *«где и когда программа ломается?»*, то отладка — на вопрос *«почему она ломается и как это починить?»*.

Ключевое различие:

Аспект	Тестирование	Отладка	↓
Цель	Обнаружить ошибки	Исправить ошибки	
Когда проводится	До и после изменений в коде (часть процесса разработки)	После обнаружения ошибки	
Кто участвует	Тестировщики, разработчики, автоматизированные системы	В основном разработчики	
Результат	Отчёт об ошибках	Исправленный код	

Эволюция подходов к тестированию

- **1950–1960-е:** тестирование отождествлялось с **отладкой** — целью было «починить программу», а не оценить её качество.
- **1970–1980-е:** появление понятия **тестирования как отдельной дисциплины**. Разработаны формальные методы (например, критерии покрытия). Акцент на обнаружение ошибок.
- **1990-е:** формирование **процессного подхода** — тестирование как часть жизненного цикла ПО (V-Model, IEEE 829).
- **2000-е:** рост популярности **гибких методологий** (Agile, Scrum). Тестирование становится **непрерывным**, интегрируется в CI/CD.
- **2010-е – настоящее время:** смещение в сторону **проактивного обеспечения качества**:
 - Shift-left testing (тестирование «влево» — на ранних этапах)
 - TDD/BDD
 - автоматизация, observability, quality culture
 - тестирование рассматривается как **коллективная ответственность всей команды**, а не только QA-инженеров

Цели и задачи тестирования

Основная цель тестирования:

- Повышение качества программного продукта за счёт раннего выявления и устранения дефектов.

Ключевые задачи:

- Верификация: «делаем ли мы продукт правильно?» (соответствие спецификациям)
- Валидация: «делаем ли мы правильный продукт?» (соответствие ожиданиям пользователя)
- Обеспечение уверенности в качестве ПО
- Предотвращение дефектов через раннее участие в жизненном цикле
- Поддержка принятия управленческих решений (на основе метрик и отчётности)
- Снижение рисков, связанных с эксплуатацией ПО

Важно: тестирование **не доказывает отсутствие дефектов**, а лишь уменьшает вероятность их проявления в эксплуатации.

Основные понятия и определения

Основные понятия и определения

- **Ошибка (error)** — неверное действие или решение человека (например, разработчика), приведшее к некорректному коду.
- **Дефект (defect / bug)** — несоответствие между ожидаемым и фактическим поведением ПО; может быть в коде, требованиях, дизайне.
- **Сбой (failure)** — проявление дефекта во время выполнения программы, приводящее к неправильному результату или аварийному завершению.

Ошибка приводит к появлению **дефекта**, который при определенных условиях может вызвать **сбой**.

Основные понятия и определения

- **Качество ПО** — степень, в которой система, компонент или процесс удовлетворяет:
 - установленным требованиям,
 - ожиданиям заинтересованных лиц,
 - атрибутам, определённым в стандартах (например, ISO/IEC 25010: функциональная пригодность, производительность, совместимость, удобство использования, надёжность, безопасность и др.)

Ошибка (Error)

Ошибка (Error)

Что это?

Ошибка — это **человеческий просчёт** на любом этапе жизненного цикла ПО: при анализе требований, проектировании, написании кода, настройке окружения и т.д.

Ключевые черты:

- Происходит **до** появления дефекта.
- Это **причина**, а не следствие.
- Не всегда сразу видна — может проявиться позже, как дефект или сбой.

Ошибка (Error)

ЭТАП	ОШИБКА
Анализ требований	Аналитик неправильно понял бизнес-правило: «Скидка 10% для всех пользователей» вместо «Скидка 10% только для премиум-пользователей».
Проектирование	Архитектор решил хранить пароли в открытом виде, не зная о требованиях безопасности.
Кодирование	Программист написал <code>if (age > 18)</code> вместо <code>if (age >= 18)</code> , хотя по требованиям лицам с 18 лет регистрация разрешена.

💡 Ошибка — это то, что сделал человек неправильно. "

Дефект (bug)

Дефект (Defect/Bug)

Что это?

Дефект — это несоответствие между тем, что должно быть (требования, спецификации, ожидания), и тем, что есть в системе (код, документация, дизайн, данные).

Ключевые черты:

- Местонахождение: в артефактах ПО — коде, требованиях, макетах, конфигурации.
- Может не проявляться при выполнении (например, если условие для сбоя не наступило).
- Обнаруживается при ревью, статическом анализе или тестировании.

Дефект (Defect/Bug)

ТИП ДЕФЕКТА	ОПИСАНИЕ
В коде	В функции расчёта скидки используется <code>price * 0.1</code> , но должно быть <code>price * discount_rate</code> , где <code>discount_rate</code> берётся из настроек.
В требованиях	В ТЗ указано: «Пользователь может загружать файлы до 10 МБ», но в UI-дизайне — ограничение 5 МБ. Противоречие = дефект в документации.
В логике	Система начисляет бонусы за покупку, но забывает учитывать отменённые заказы → бонусы начисляются неправильно.

💡 Дефект — это «баг» в системе, спрятанный до тех пор, пока не вызовет сбой.

Сбой (failure)

Сбой (Failure)

Что это?

Сбой — это наблюдаемое неправильное поведение системы во время выполнения, вызванное активацией дефекта при определённых условиях.

Ключевые черты:

- Происходит только при запуске ПО.
- Виден пользователю или тестировщику.
- Не всегда легко воспроизводится (например, только при высокой нагрузке или в определённой локали).

Сбой (Failure)

СЦЕНАРИЙ	СБОЙ
Пользователь вводит возраст 18 → система отказывает в регистрации, хотя по требованиям это разрешено.	Неправильный результат (ожидался успех, получена ошибка).
При одновременном редактировании документа тремя пользователями приложение падает с ошибкой 500	Аварийное завершение (crash).
После обновления ПО кнопка «Оплатить» в мобильном приложении не реагирует на нажатие	Отсутствие реакции на действие пользователя.

💡 Сбой — это то, что пользователь *видит и переживает.*

Цепочка «Ошибка → Дефект → Сбой»

1. **Ошибка:** Программист неверно реализует условие:

```
python
1  if user.age > 18: # ошибка: должно быть >=
2      allow_registration()
```

2. **Дефект:** В коде заложена логическая ошибка — 18-летние пользователи не могут зарегистрироваться.
(Дефект существует даже если никто не пытался зарегистрироваться.)
3. **Сбой:** Пользователь с возрастом **18** пытается зарегистрироваться → получает сообщение «Вам должно быть больше 18».
→ Это наблюдаемый сбой.

“ **!** Важно: не всякий дефект приводит к сбою!

Например, если в системе никто никогда не вводит возраст 18, сбой не произойдёт — но дефект останется. ”

Дополнительные нюансы

- Дефект в требованиях может привести к системе, которая работает «без ошибок», но не решает задачу (см. 7-й принцип тестирования).
- Сбой может быть вызван внешней причиной (например, обрыв сети), но если система не обрабатывает такие ситуации — это тоже проявление внутреннего дефекта (отсутствие обработки ошибок).
- Обнаружение дефекта \neq сбой: при code review можно найти дефект до запуска программы — тогда сбоя не было.

Заключение

Понимание разницы между ошибкой, дефектом и сбоем помогает:

- точнее описывать проблемы в отчётах,
- находить корневые причины сбоев,
- внедрять профилактические меры (например, ревью, статический анализ),
- говорить на одном языке в команде.

Эта тройка — основа профессионального мышления тестировщика и инженера по качеству.

Классификация дефектов ПО

Классификация дефектов

Зачем классифицировать дефекты?

- Чтобы **рационально распределять ресурсы** разработки и тестирования.
- Для **анализа качества** — какие типы ошибок чаще возникают?
- Чтобы **улучшать процессы** — если много дефектов безопасности, нужно усилить ревью или обучение.

Классификация дефектов

1. По уровню серьёзности (Severity)

Отражает, насколько сильно дефект влияет на работу системы:

- **Critical (Критический)**

Система неработоспособна: краш, потеря данных, невозможность запуска ключевой функции.

Пример: нельзя авторизоваться в системе.

- **Major (Серьёзный)**

Основная функция работает неправильно, но есть обходные пути.

Пример: не сохраняются изменения в профиле, но можно повторить попытку.

- **Minor (Незначительный)**

Нарушение второстепенной функции, не влияющее на основной сценарий.

Пример: неправильное отображение иконки.

- **Trivial / Cosmetic (Косметический)**


Опечатки, мелкие визуальные недочёты.

Пример: текст кнопки написан с маленькой буквы.

2. По приоритету (Priority)

Показывает, **насколько срочно дефект нужно исправить** (может отличаться от severity):

- **High** — исправить нужно до релиза.
- **Medium** — желательно исправить, но можно отложить.
- **Low** — можно пофиксить в будущих версиях.

 Важно: дефект может быть **критическим по severity**, но **низким по priority**, если он проявляется только в редком сценарии, который никто не использует.

3. По типу дефекта

Помогает понять **природу ошибки**:

- **Функциональный** — нарушена логика работы (программа делает не то, что должна).
- **Интерфейсный** — проблемы с UI/UX (неверное расположение элементов, плохая читаемость).
- **Производительности** — система работает медленно или потребляет слишком много ресурсов.
- **Безопасности** — уязвимости, утечки данных, отсутствие проверок доступа.
- **Совместимости** — ошибка проявляется только в определённой ОС, браузере или устройстве.
- **Локализации** — проблемы с переводом, кодировкой, форматами дат/чисел.
- **Регрессионный** — ранее работавшая функция перестала работать после изменений.

4. По стадии жизненного цикла (когда обнаружен)

- Требования (например, противоречивые или неполные)
- Проектирование (архитектурные недочёты)
- Кодирование (ошибки в реализации)
- Тестирование (выявлены в ходе проверок)
- Эксплуатация (найлены пользователями)

Основные принципы тестирования

Основные принципы тестирования

Семь основных принципов тестирования были сформулированы в рамках ISTQB (International Software Testing Qualifications Board) и лежат в основе современной практики обеспечения качества программного обеспечения. Вот они с краткими пояснениями и примерами:

1. Тестирование показывает наличие дефектов, но не доказывает их отсутствие

Суть: Тесты могут выявить ошибки, но даже при успешном прохождении всех тестов нельзя гарантировать, что система полностью свободна от дефектов.

Пример:

Вы протестировали калькулятор на сложение 1000 пар чисел — всё работает. Но при вводе очень большого числа (например, 10^{100}) программа падает. Тесты не охватили этот сценарий.

Основные принципы тестирования

2. Исчерпывающее (полное) тестирование невозможно

Суть: Проверить все возможные комбинации входных данных, состояний и путей выполнения — нереально, особенно в сложных системах.

Пример:

Веб-форма с 10 полями, каждое из которых может принимать 100 значений. Общее число комбинаций — 100^{10} (10^{20}). Даже при скорости 1 млн тестов/сек это займёт тысячелетия.

“Поэтому применяют **техники редуцирования тестов**: классы эквивалентности, анализ граничных значений, попарное тестирование и т.д.”

Основные принципы тестирования

3. Раннее тестирование экономит время и деньги

Суть: Чем раньше в жизненном цикле ПО начинается тестирование (даже на этапе требований), тем дешевле и проще устранять дефекты.

Пример:

Анализ требований выявил противоречие: «Пользователь может удалить аккаунт, но данные должны храниться 10 лет». Это логическая ошибка. Исправить её на этапе анализа — минуты. А после релиза — дорого и сложно.

Основные принципы тестирования

4. Дефекты распределяются неравномерно

Суть: 80% ошибок обычно сосредоточено в 20% модулей (принцип Парето). Такие модули — «горячие точки» — требуют повышенного внимания.

Пример:

В мобильном приложении модуль оплаты вызывает 90% жалоб пользователей, в то время как экран «О нас» работает без сбоев. Тестирование должно быть сфокусировано на критических зонах.

Основные принципы тестирования

5. Пестицидный парадокс

Суть: Если повторно использовать одни и те же тесты, со временем они перестанут находить новые дефекты («устойчивость» к старым тестам). Тесты нужно регулярно пересматривать и обновлять.

Пример:

Автоматизированный набор UI-тестов годами проверяет стандартный сценарий «авторизация → заказ → оплата». Он не находит новых ошибок, потому что не проверяет, например, сценарий с отменой оплаты или возвратом товара.

“Решение: регулярно расширять и ротировать тестовые наборы, добавлять негативные и граничные сценарии.”

Основные принципы тестирования

6. Тестирование зависит от контекста

Суть: Подходы, методы и уровень тестирования различаются в зависимости от типа системы, домена, рисков и требований.

Пример:

- **Медицинское ПО** (например, ПО для рентгена): требуется строгое документирование, регрессионное тестирование, соответствие стандартам (FDA, ISO).
- **Социальная сеть для подростков:** важнее UX, быстрая доставка новых фич, допустимы риски в non-critical функциях.

“Нельзя применять один и тот же план тестирования ко всем проектам.”

Основные принципы тестирования

7. Отсутствие ошибок — иллюзия

Суть: Даже если система технически «без багов», она может не удовлетворять потребностям пользователя и быть бесполезной.

Пример:

Приложение для бронирования отелей работает без сбоев, но не позволяет фильтровать по цене или рейтингу. Пользователи уходят к конкурентам. Технически — всё «зелёное», но продукт **не решает задачу**.

“Тестирование должно проверять не только **корректность**, но и **полезность** (валидацию).”

Заключение

Эти семь принципов напоминают, что тестирование — это не просто «запуск тестов», а **стратегическая деятельность**, направленная на управление рисками, понимание контекста и постоянное улучшение качества. Они помогают избежать иллюзий и сосредоточиться на том, что действительно важно для пользователя и бизнеса.

Кто занимается тестированием?

Основные роли в тестировании

- Тестировщик (Tester)
- Инженер QA (Quality Assurance Engineer)
- Инженер QC (Quality Control Engineer)

Основные роли в тестировании

1. Тестировщик (Tester)

Фокус: Выявление дефектов в уже написанном коде.

Основные задачи:

- Выполнение ручных или автоматизированных тестов.
- Составление отчётов об ошибках.
- Проверка соответствия функциональности требованиям.

Особенности:

- Чаще всего работает на уровне QC (контроля качества конкретного продукта).
- Может не участвовать в процессах разработки или улучшения процессов.
- Часто это начальный уровень в карьере QA/QC.

Основные роли в тестировании

2. Инженер QA (Quality Assurance Engineer)

Фокус: Предотвращение дефектов путём улучшения процессов разработки и тестирования.

Основные задачи:

- Разработка и внедрение стандартов качества.
- Участие в планировании жизненного цикла разработки ПО (SDLC).
- Проектирование тестовых стратегий и процессов.
- Автоматизация тестирования и интеграция в CI/CD.

Особенности:

- Работает **до и во время** разработки, а не только после.
- QA — это **процессно-ориентированный** подход.
- Чаще имеет технический бэкграунд и пишет код (особенно в автоматизации).

3. Инженер QC (Quality Control Engineer)

Фокус: Выявление дефектов в конечном продукте.

Основные задачи:

- Проведение тестирования (функционального, регрессионного, интеграционного и т.д.).
- Верификация и валидация продукта.
- Обеспечение соответствия продукта спецификациям.

Особенности:

- QC — это **продукт-ориентированный** подход.
- Деятельность происходит **после** или **в конце** этапа разработки.
- По сути, QC — это часть QA, но более узкая и ориентированная на проверку, а не на улучшение процессов.

Основные роли в тестировании

КРИТЕРИЙ	ТЕСТИРОВЩИК	ИНЖЕНЕР QA	ИНЖЕНЕР QC
Фокус	Поиск багов	Предотвращение багов	Поиск багов
Когда действует	После разработки	На всех этапах SDLC	В конце/после разработки
Подход	Продукт-ориент.	Процесс-ориент.	Продукт-ориент.
Участие в процессах	Минимальное	Активное	Ограниченное
Автоматизация	Иногда	Часто	Иногда

Роли на практике

- Во многих компаниях (особенно в IT) **QA-инженер** — это обобщённый термин, который может включать и тестирование, и автоматизацию, и участие в процессах.
- **QC-инженеры** чаще встречаются в традиционных отраслях (например, производство), а в IT роль QC обычно выполняет тестировщик или QA.
- **Тестировщик** — более узкая и часто ручная роль, тогда как **QA-инженер** — более широкая и стратегическая.

Профессиональные навыки тестировщика

- Понимание моделей разработки ПО, их связей со стадиями тестирования
- Работа с документацией
 - Анализ и тестирование требований
 - Управление требованиями и бизнес-анализ
- Оценка и планирование тестирования
 - Создание плана и стратегии тестирования
 - Оценка трудозатрат
- Работа с тест-кейсами
 - Создание чек-листов
 - Создание тест-кейсов и управление ими
- Знание и владение методологиями тестирования
- Работа с отчетами о дефектах
- Работа с отчетами о результатах тестирования

Технические навыки тестировщика

- Настройка тестового окружения в Windows, Linux, MacOS
- Работа с базами данных
 - Анализ ER-диаграмм и UML-схем баз данных
 - Составление SQL-запросов
- Работа с компьютерными сетями
 - Знание протоколов различного уровня
 - Владение утилитами
- Знание веб-технологий
 - Работа с API и HTTP-запросами
 - Настройка веб-серверов
 - Знание HTML, CSS, JavaScript
 - Владение серверными языками веб-программирования