

Лекция 5.

**Классификация видов  
тестирования**

# Зачем нужна классификация?

## **Проблема:**

В современной разработке ПО существует десятки, а то и сотни различных подходов к тестированию — от проверки отдельных функций до оценки безопасности и удобства использования.

## **Решение:**

Классификация помогает:

- структурировать разнообразие методов тестирования,
- понять, **когда, зачем и как** применять тот или иной вид,
- избежать хаоса в процессе обеспечения качества.

Без системы — только шум. С классификацией — стратегия.

# Зачем нужна классификация?

Классификация видов тестирования позволяет:

- **Чётко распределять ответственность** между участниками проекта (разработчики, тестировщики, аналитики),
- **Планировать тестовые активности** на разных этапах жизненного цикла ПО,
- **Выбирать правильные инструменты и техники** (например, unit-тесты vs. нагрузочное тестирование),
- **Оценивать полноту покрытия** требований и рисков.

Это не просто список — это карта пути к качественному продукту.

# Что дает понимание видов тестирования?

Для **команды разработки и тестирования**:

- Упрощается коммуникация: все говорят на одном языке.
- Повышается эффективность: меньше дублирования и пропусков.

Для **бизнеса и заказчика**:

- Прозрачность: можно оценить, какие аспекты качества проверяются.
- Снижение рисков: системный подход помогает выявлять дефекты раньше и дешевле.

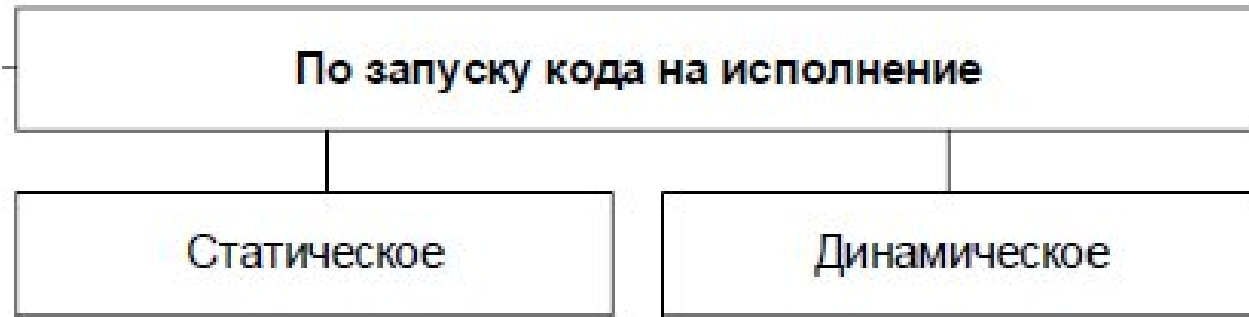
Классификация — основа осознанного тестирования и управления качеством.

# Упрощенная классификация

---

- По запуску кода на исполнение
- По доступу к коду и архитектуре приложения
- По степени автоматизации
- По уровню детализации приложения
- По степени важности тестируемых функций
- По принципам работы с приложением
- По целям и задачам

# **Классификация по запуску кода на исполнение**



- Статическое тестирование — без запуска.
- Динамическое тестирование — с запуском.

# Статическое тестирование

---

## Основная идея:

Проверка программного обеспечения **без его выполнения** — анализ кода, документации и других артефактов «на бумаге» или с помощью инструментов.

## Что проверяется:

- Требования, спецификации, дизайн-документы
- Исходный код (на соответствие стандартам, наличие потенциальных ошибок)
- Тест-кейсы и чек-листы

# Статическое тестирование

---

## Методы и техники:

- Рецензирование (review), инспекции, walkthrough
- Статический анализ кода (например, с помощью SonarQube, ESLint)

## Область применения:

- Ранние этапы разработки (до написания/запуска кода)
- Профилактика дефектов, повышение качества артефактов
- Соответствие стандартам и требованиям безопасности

**Цель:** найти ошибки как можно раньше — когда их дешевле всего исправить.

# Динамическое тестирование

---

## Основная идея:

Проверка ПО **в процессе его выполнения** — запуск программы с целью наблюдения за её поведением при различных входных данных.

## Что проверяется:

- Функциональность (работает ли система по требованиям?)
- Надёжность, производительность, безопасность, удобство использования
- Поведение в граничных и аварийных ситуациях

# Динамическое тестирование

---

## Методы и техники:

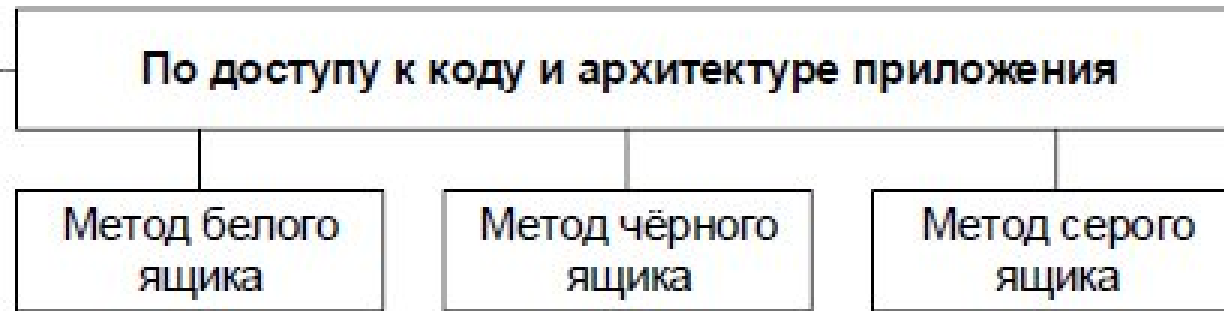
- Ручное и автоматизированное тестирование
- Модульное, интеграционное, системное, приёмочное тестирование
- Нагрузочное, стрессовое, регрессионное тестирование

## Область применения:

- После сборки исполняемого кода
- На всех этапах, где доступна рабочая версия ПО
- Подтверждение соответствия реального поведения ожидаемому

**Цель:** убедиться, что система **работает правильно** в реальных условиях.

# **Классификация по доступу к исходному коду**



- Метод белого ящика — доступ к коду есть.
- Метод чёрного ящика — доступа к коду нет.
- Метод серого ящика — к части кода доступ есть, к части — нет.

# Тестирование методом «чёрного ящика»

---

Тестирование **без знания внутренней структуры** программы — проверяется только соответствие входных данных ожидаемым выходным результатам.

## Что проверяется:

- Функциональность с точки зрения пользователя
- Соответствие требованиям и спецификациям
- Поведение системы при различных сценариях использования

# Тестирование методом «чёрного ящика»

---

## Особенности:

- Тестировщик не видит исходный код
- Акцент на **что делает система**, а не **как**
- Часто используется для системного и приёмочного тестирования

## Область применения:

- Приёмочное тестирование (заказчик/пользователь)
- UI-тестирование, сквозные сценарии
- Валидация требований и пользовательских историй

| **Цель:** убедиться, что система делает то, что от неё ожидают.

# Тестирование методом «белого ящика»

Тестирование **с полным доступом к исходному коду** — проверяется внутренняя логика, структура и потоки выполнения программы.

## Что проверяется:

- Покрытие кода (statement, branch, path coverage)
- Корректность алгоритмов и условий
- Обработка ошибок внутри модулей

# Тестирование методом «белого ящика»

## Особенности:

- Требует знания языка программирования и архитектуры
- Часто выполняется разработчиками
- Ориентировано на **как работает система**

## Область применения:

- Модульное (unit) тестирование
- Интеграционное тестирование на уровне компонентов
- Поиск скрытых дефектов в логике кода

**Цель:** убедиться, что каждая часть кода работает корректно и покрыта тестами.

# Тестирование методом «серого ящика»

Гибридный подход: тестировщик имеет **частичное знание внутренней структуры** системы, но тестирует преимущественно через интерфейсы.

## Что проверяется:

- Взаимодействие между компонентами
- Логика обработки данных «под капотом», но через внешние вызовы
- Безопасность и производительность с учётом архитектуры

# Тестирование методом «серого ящика»

## Особенности:

- Баланс между функциональным и структурным тестированием
- Позволяет проектировать более эффективные тест-кейсы
- Часто используется при интеграционном и API-тестировании

## Область применения:

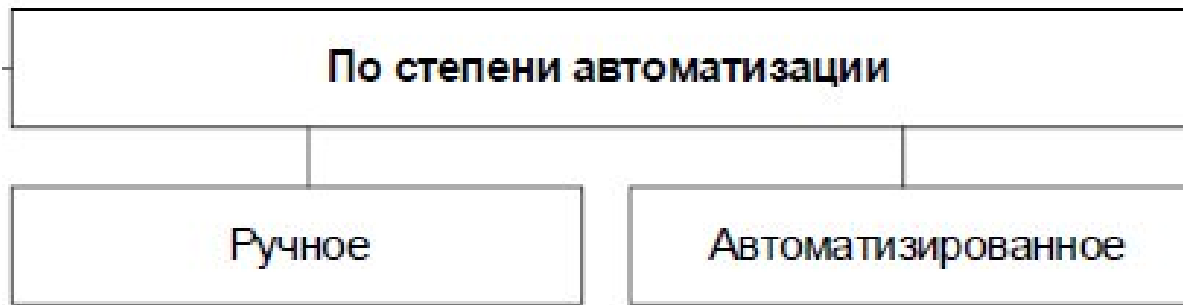
- Интеграционное и API-тестирование
- Тестирование безопасности (например, penetration testing)
- Сценарии, где важно понимать как данные проходят через систему

**Цель:** совместить преимущества «чёрного» и «белого» ящиков для глубокой и реалистичной проверки.

	Преимущества	Недостатки
<b>Метод белого ящика</b>	<ul style="list-style-type: none"> <li>• Показывает скрытые проблемы и упрощает их диагностику.</li> <li>• Допускает достаточно простую автоматизацию тест-кейсов и их выполнение на самых ранних стадиях развития проекта.</li> <li>• Обладает развитой системой метрик, сбор и анализ которых легко автоматизируется.</li> <li>• Стимулирует разработчиков к написанию качественного кода.</li> <li>• Многие техники этого метода являются проверенными, хорошо себя зарекомендовавшими решениями, базирующимися на строгом техническом подходе.</li> </ul>	<ul style="list-style-type: none"> <li>• Не может выполняться тестировщиками, не обладающими достаточными знаниями в области программирования.</li> <li>• Тестирование сфокусировано на реализованной функциональности, что повышает вероятность пропуска нереализованных требований.</li> <li>• Поведение приложения исследуется в отрыве от реальной среды выполнения и не учитывает её влияние.</li> <li>• Поведение приложения исследуется в отрыве от реальных пользовательских сценариев<sup>(146)</sup>.</li> </ul>

	Преимущества	Недостатки
<b>Метод чёрного ящика</b>	<ul style="list-style-type: none"> <li>• Тестируемый не обязан обладать (глубокими) знаниями в области программирования.</li> <li>• Поведение приложения исследуется в контексте реальной среды выполнения и учитывает её влияние.</li> <li>• Поведение приложения исследуется в контексте реальных пользовательских сценариев<sup>{146}</sup>.</li> <li>• Тест-кейсы можно создавать уже на стадии появления стабильных требований.</li> <li>• Процесс создания тест-кейсов позволяет выявить дефекты в требованиях.</li> <li>• Допускает создание тест-кейсов, которые можно многократно использовать на разных проектах.</li> </ul>	<ul style="list-style-type: none"> <li>• Возможно повторение части тест-кейсов, уже выполненных разработчиками.</li> <li>• Высока вероятность того, что часть возможных вариантов поведения приложения останется не протестированной.</li> <li>• Для разработки высокоэффективных тест-кейсов необходима качественная документация.</li> <li>• Диагностика обнаруженных дефектов более сложна в сравнении с техниками метода белого ящика.</li> <li>• В связи с широким выбором техник и подходов затрудняется планирование и оценка трудозатрат.</li> <li>• В случае автоматизации могут потребоваться сложные дорогостоящие инструментальные средства.</li> </ul>

# **Классификация по степени автоматизации**



- Ручное тестирование — тест-кейсы выполняет человек.
- Автоматизированное тестирование — тест-кейсы частично или полностью выполняет специальное инструментальное средство.

# Ручное тестирование

## Основная идея:

Тестирование, при котором **человек вручную выполняет тест-кейсы**, взаимодействуя с приложением без использования специализированных автоматизированных скриптов.

## Что проверяется:

- Пользовательский интерфейс и удобство использования (UX)
- Логика работы функций «глазами пользователя»
- Нестандартные или творческие сценарии

# Ручное тестирование

## Особенности:

- Не требует написания кода или настройки инфраструктуры
- Гибкость: легко адаптироваться к изменениям
- Высокая когнитивная вовлечённость тестировщика

## Область применения:

- Исследовательское тестирование
- Приёмочное и юзабилити-тестирование
- Проекты с нестабильными требованиями или коротким сроком жизни

**Цель:** оценить качество с точки зрения реального пользователя и выявить то, что машина может упустить.

# Автоматизированное тестирование

## Основная идея:

Тестирование с использованием **специальных скриптов и инструментов**, которые выполняют проверки без прямого участия человека.

## Что проверяется:

- Регрессионные сценарии
- Корректность API, логики и данных
- Производительность и стабильность при повторяющихся запусках

# Автоматизированное тестирование

## Особенности:

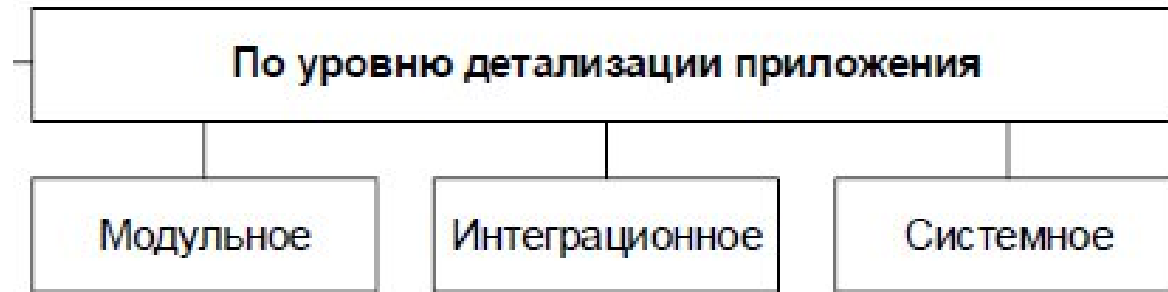
- Требует затрат на разработку и поддержку тестов
- Высокая скорость и воспроизводимость
- Идеально для частых и предсказуемых проверок

## Область применения:

- Регрессионное тестирование
- CI/CD-процессы (непрерывная интеграция и доставка)
- Тестирование на разных конфигурациях и объёмах данных

**Цель:** повысить эффективность, надёжность и скорость проверок за счёт повторного использования автоматизированных сценариев.

# **Классификация по уровню детализации приложения**



- Модульное (компонентное) тестирование — проверяются отдельные небольшие части приложения.
- Интеграционное тестирование — проверяется взаимодействие между несколькими частями приложения.
- Системное тестирование — приложение проверяется как единое целое.

# Модульное тестирование

## Основная идея:

Проверка **отдельных модулей или компонентов** программы изолированно от остальной системы.

## Что проверяется:

- Корректность работы функций, классов, методов
- Обработка входных данных и генерация выходных
- Поведение в граничных и ошибочных ситуациях

# Модульное тестирование

## Особенности:

- Обычно выполняется разработчиками
- Часто автоматизируется (например, с помощью JUnit, pytest)
- Использует заглушки (mocks/stubs) для внешних зависимостей

## Область применения:

- Ранние этапы разработки
- Подтверждение корректности реализации логики
- База для последующих уровней тестирования

**Цель:** убедиться, что каждая «деталь» работает правильно сама по себе.

# Интеграционное тестирование

## Основная идея:

Проверка взаимодействия между модулями, компонентами или подсистемами после их объединения.

## Что проверяется:

- Передача данных между компонентами
- Корректность интерфейсов (API, протоколы обмена)
- Обработка ошибок на стыке модулей

# Интеграционное тестирование

## Особенности:

- Может выполняться как разработчиками, так и тестировщиками
- Применяются стратегии: «снизу вверх», «сверху вниз», «большой взрыв»
- Часто сочетает ручные и автоматизированные подходы

## Область применения:

- После завершения модульного тестирования
- При интеграции микросервисов, баз данных, внешних API
- Выявление проблем, невидимых на уровне отдельных модулей

**Цель:** убедиться, что части системы «понимают друг друга» и работают вместе без сбоев.

# Системное тестирование

## Основная идея:

Проверка **всей интегрированной системы целиком** как единого продукта в условиях, приближенных к реальным.

## Что проверяется:

- Соответствие функциональным и нефункциональным требованиям
- Поведение системы «из коробки»: установка, запуск, работа, восстановление
- Надёжность, производительность, безопасность, совместимость

# Системное тестирование

## Особенности:

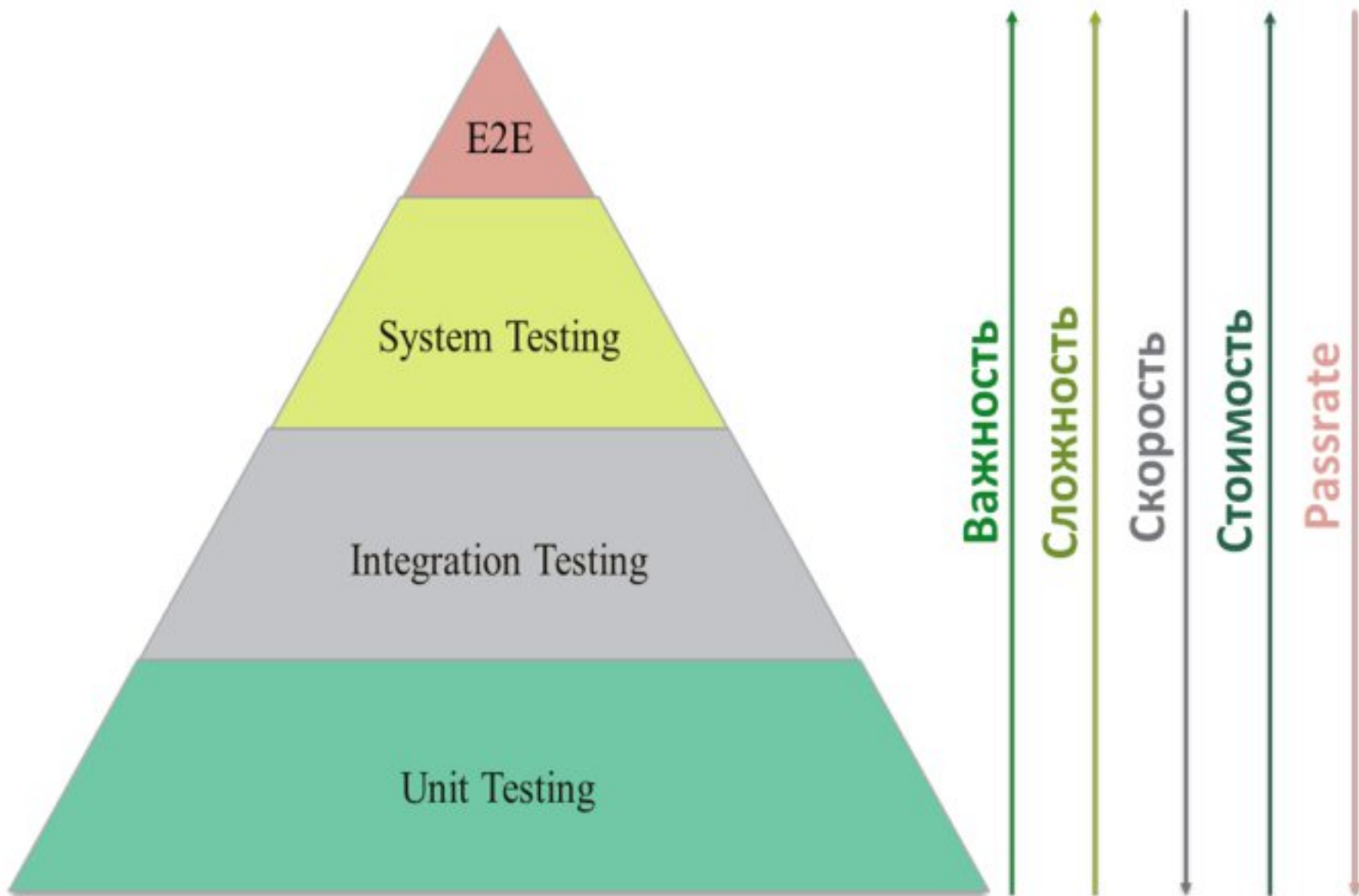
- Выполняется независимой тестовой командой (часто без доступа к коду)
- Основано на спецификациях и пользовательских сценариях
- Включает функциональное и нефункциональное тестирование

## Область применения:

- Финальный этап перед выпуском продукта
- Подготовка к приёмочному тестированию заказчиком
- Полная верификация готового решения

**Цель:** убедиться, что система в целом удовлетворяет ожиданиям пользователей и бизнеса.

# Пирамида тестирования



# Пирамида тестирования

## 1. Модульные (unit) тесты — основание

- Максимум автоматизированных тестов
- Быстрые, дешёвые, легко поддерживаемые
- Обеспечивают детальное покрытие логики

## 2. Интеграционные тесты — средний уровень

- Умеренное количество
- Проверяют взаимодействие компонентов
- Сложнее в написании и поддержке, чем unit-тесты

## 3. Системные / сквозные (end-to-end) тесты — вершина

- Минимум тестов из-за высокой стоимости и хрупкости
- Имитируют реальные сценарии пользователя
- Долгие и ресурсоёмкие, но критически важны для валидации

# Пирамида тестирования

## Зачем она нужна?

- Избегать «перевернутой пирамиды» (слишком много E2E, мало unit)
- Снижать стоимость сопровождения тестов
- Ускорять обратную связь при разработке

**Правило:** чем ниже уровень — тем больше должно быть тестов.

**Цель:** быстрая, надёжная и экономически эффективная верификация ПО.


# **Классификация по степени важности тестируемых функций**




- Дымовое тестирование (обязательно изучите этимологию термина — хотя бы в Википедии<sup>110</sup>) — проверка самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения.
- Тестирование критического пути — проверка функциональности, используемой типичными пользователями в типичной повседневной деятельности.
- Расширенное тестирование — проверка всей (остальной) функциональности, заявленной в требованиях.


# Дымовое тестирование

---

1. **Печники (XIX век):** Собирали печь, закрывали заслонки, топили и смотрели, чтобы дым шел только в дымоход. Если дым валил отовсюду — проблема в конструкции.
2. **Радиоэлектроника (XX век):** Инженеры подавали напряжение на новую плату на секунду и проверяли на перегрев. Сильный нагрев или дым указывали на грубую ошибку, что означало негодность платы. 

## В тестировании ПО:

- Это быстрая, поверхностная проверка, которая подтверждает, что самые основные функции программы работают.
- Цель — не найти все баги, а быстро отсеять "плохие" сборки, не тратя время на глубокое тестирование, если в основе уже есть критические проблемы. 

Таким образом, название отражает идею предотвращения катастрофы (дыма) с помощью самой первой и простой проверки. 

# Дымовое тестирование

## Основная идея:

Быстрая проверка **основных функций системы** после сборки, чтобы убедиться, что приложение «вообще работает» и готово к дальнейшему тестированию.

## Что проверяется:

- Запуск приложения
- Основные пользовательские сценарии (например: вход, создание записи, навигация)
- Отсутствие критических сбоев

# Дымовое тестирование

---

## Особенности:

- Минимальный набор тестов (часто < 10–15 кейсов)
- Выполняется вручную или автоматически сразу после деплоя
- Не проверяет детали — только «работает / не работает»

## Область применения:

- После каждой новой сборки (в CI/CD)
- Перед началом регрессионного или системного тестирования
- Быстрый «фейл-фаст»: если дым не проходит — сборка отклоняется

**Цель:** убедиться, что система стабильна настолько, чтобы её можно было тестировать дальше.

# Тестирование критического пути

---

## Основная идея:

Проверка **ключевых бизнес-сценариев**, без которых продукт теряет смысл (например: оформление заказа, оплата, отправка сообщения).

## Что проверяется:

- Основные пользовательские цепочки действий
- Корректность обработки данных в типичных условиях
- Интеграция между основными модулями

# Тестирование критического пути

---

## Особенности:

- Фокус на «счастливом пути» — без ошибок и исключений
- Часто покрывается как ручными, так и автоматизированными тестами
- Имеет высший приоритет при регрессии

## Область применения:

- Регрессионное тестирование
- Подготовка к релизу
- Приёмочные проверки перед демонстрацией заказчику

**Цель:** гарантировать, что самое важное в продукте работает безупречно.

# Расширенное (глубокое) тестирование

---

## Основная идея:

Детальная проверка **всех возможных сценариев**, включая граничные значения, ошибочные ситуации, альтернативные потоки и нефункциональные аспекты.

## Что проверяется:

- Негативные сценарии (некорректный ввод, отказы сервисов)
- Граничные и экстремальные условия
- Юзабилити, безопасность, производительность, локализация и др.

# Расширенное (глубокое) тестирование

---

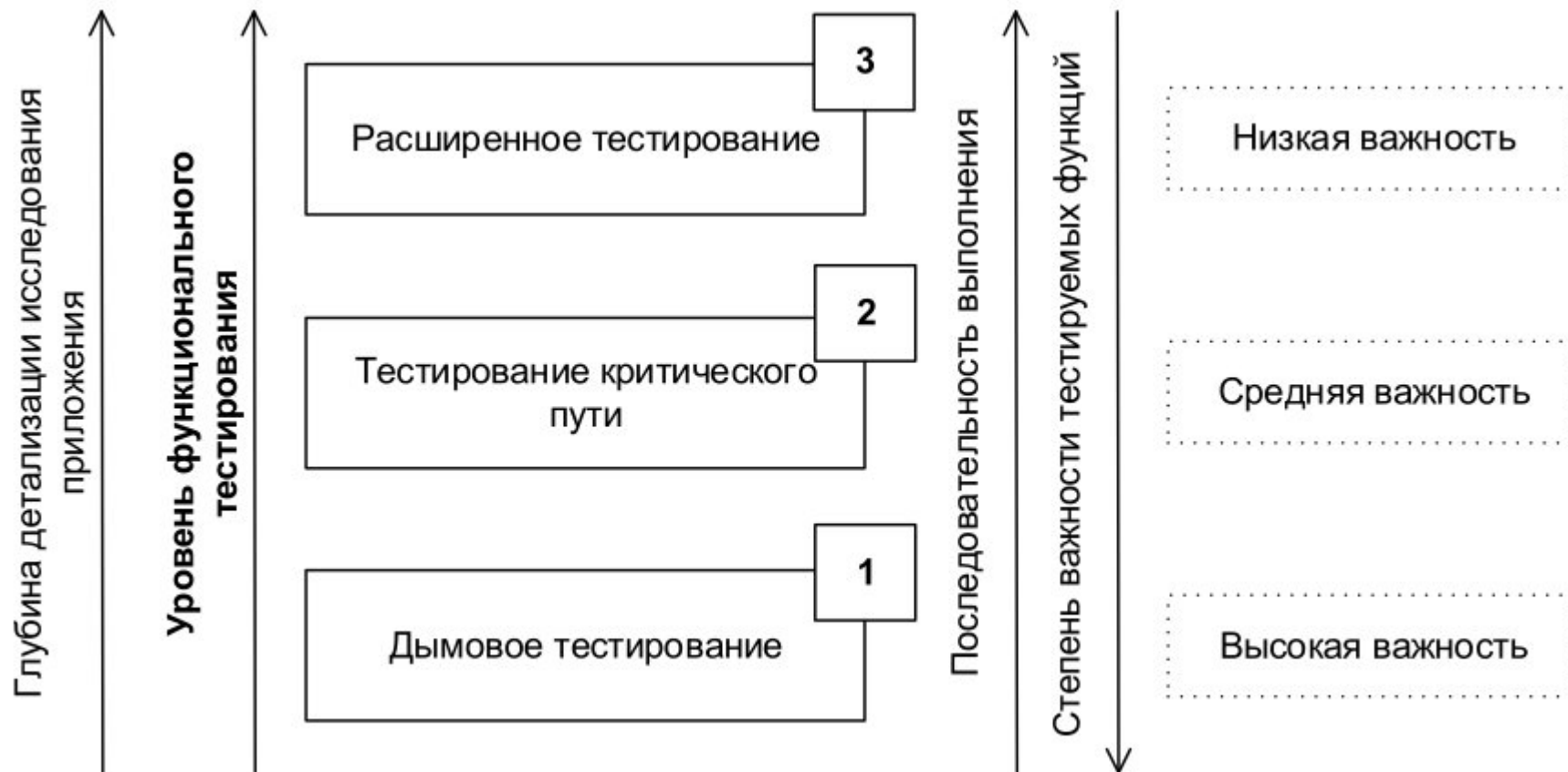
## Особенности:

- Наиболее трудоёмкий и длительный этап
- Включает exploratory testing, нагрузочное, стрессовое и другие виды
- Может выявлять скрытые и нетривиальные дефекты

## Область применения:

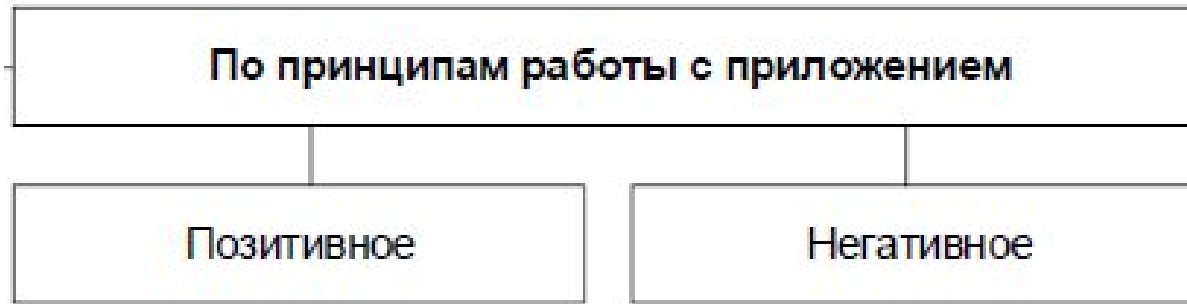
- Финальные этапы тестирования перед крупным релизом
- Проекты с высокими требованиями к надёжности (финансы, медицина)
- Поддержка качества на уровне превосходства, а не просто работоспособности

**Цель:** убедиться, что система устойчива ко всему — даже к тому, чего «не должно быть».



Эти три уровня образуют **логическую последовательность фокусировки**: от «работает ли вообще?» → «работает ли главное?» → «работает ли всё и везде?». Такой подход помогает эффективно распределять ресурсы и минимизировать риски на каждом этапе разработки.

# **Классификация по принципам работы с приложением**



- Позитивное тестирование — все действия с приложением выполняются строго по инструкции без никаких недопустимых действий, некорректных данных и т.д. Можно образно сказать, что приложение исследуется в «тепличных условиях».
- Негативное тестирование — в работе с приложением выполняются (некорректные) операции и используются данные, потенциально приводящие к ошибкам (классика жанра — деление на ноль).



Внимание! Очень частая ошибка! Негативные тесты НЕ предполагают возникновения в приложении ошибки. Напротив — они предполагают, что верно работающее приложение даже в критической ситуации поведёт себя правильным образом (в примере с делением на ноль, например, отобразит сообщение «Делить на ноль запрещено»).

# Позитивное тестирование

---

## Основная идея:

Проверка того, как система ведёт себя при корректных, ожидаемых входных данных и действиях пользователя.

## Что проверяется:

- Соответствие функциональности требованиям
- Корректная обработка «правильных» сценариев (например: ввод валидного email, успешный платёж)
- Ожидаемый результат при стандартном использовании

# Позитивное тестирование

---

## Особенности:

- Подтверждает, что система **делает то, что должна**
- Чаще всего используется при первоначальной верификации функций
- Лежит в основе тестов «критического пути» и дымового набора

## Область применения:

- Приёмочное и системное тестирование
- Демонстрации заказчику
- Базовое покрытие в автоматизированных регрессионных тестах

**Цель:** убедиться, что приложение корректно работает в штатных условиях.

# Негативное тестирование

---

## Основная идея:

Проверка того, как система реагирует на некорректные, неожиданные или ошибочные действия и данные.

## Что проверяется:

- Обработка неверного ввода (пустые поля, спецсимволы, превышение лимитов)
- Поведение при отсутствии интернета, сбоях сервисов, недоступности БД
- Защита от некорректного использования (например: SQL-инъекции)

# Негативное тестирование

---

## Особенности:

- Выявляет уязвимости, сбои и плохой UX
- Требуется творческого подхода и понимания возможных сценариев злоупотребления
- Часто пропускается при нехватке времени, но критически важен для надёжности

## Область применения:

- Тестирование безопасности и отказоустойчивости
- Подготовка к production-эксплуатации
- Расширенное и стрессовое тестирование

**Цель:** убедиться, что система **не ломается** и **корректно реагирует** на всё, что может пойти не так.

# Позитивное и негативное тестирование

---

Эти два подхода **дополняют друг друга**: позитивное тестирование показывает, что система работает, а негативное — что она **не сломается**, когда пользователь (случайно или намеренно) выйдет за рамки «правильного» поведения. Вместе они формируют целостную картину качества ПО.

# **Классификация по целям и задачам**

# Функциональное тестирование

---

## Основная идея:

Проверка того, **соответствует ли система заявленным функциональным требованиям** — то есть *что* делает система.

## Что проверяется:

- Корректность выполнения бизнес-логики
- Работа пользовательских сценариев (например: авторизация, поиск, оформление заказа)
- Обработка входных данных и генерация ожидаемых выходов

# Функциональное тестирование

---

## Особенности:

- Основано на спецификациях, user stories, use cases
- Может выполняться как вручную, так и автоматически
- Часто использует методы «чёрного ящика»

## Область применения:

- Системное и приёмочное тестирование
- Верификация требований на всех этапах разработки
- Основа регрессионного тестирования

**Цель:** убедиться, что система делает **всё, что от неё требуется**, и ничего лишнего.

# Нефункциональное тестирование

---

## Основная идея:

Проверка **качественных характеристик системы** — то есть *как* она это делает.

## Что проверяется:

- Производительность (скорость, время отклика, пропускная способность)
- Надёжность, отказоустойчивость, восстанавливаемость
- Безопасность, удобство использования (UX), совместимость, масштабируемость

# Нефункциональное тестирование

---

## Основные виды:

- Нагрузочное и стрессовое тестирование
- Тестирование безопасности (penetration testing)
- Юзабилити- и локализационное тестирование
- Тестирование совместимости (браузеры, ОС, устройства)

## Особенности:

- Часто требует специализированных инструментов (JMeter, OWASP ZAP, BrowserStack и др.)
- Результаты измеряются количественно (время, количество ошибок, % успешных транзакций)
- Критически важно для production-готовности

**Цель:** убедиться, что система работает **надёжно, быстро, безопасно и удобно** в реальных условиях.

# Функциональное и нефункциональное тестирование

---

Эти два вида тестирования **взаимодополняют друг друга**:

- **Функциональное** отвечает на вопрос «*работает ли?*»,
- **Нефункциональное** — на вопрос «*насколько хорошо работает?*».

Игнорирование любого из них может привести к выпуску продукта, который либо не решает задачу, либо решает её плохо.

# Инсталляционное тестирование

---

## Основная идея:

Проверка корректности **установки, настройки, обновления и удаления** программного обеспечения в различных целевых средах.

## Что проверяется:

- Успешная установка ПО на разные ОС, устройства, конфигурации
- Корректность настройки параметров (файлы конфигурации, права доступа)
- Работоспособность после обновления или отката версии
- Полное и безопасное удаление без «следов» в системе

# Инсталляционное тестирование

---

## Особенности:

- Требуется тестирование на реальных или виртуальных окружениях
- Особенно важно для десктопных, мобильных и embedded-приложений
- Часто включает проверку зависимостей и совместимости

## Область применения:

- Перед релизом дистрибутива
- При выпуске патчей и обновлений
- В проектах с локальной установкой ПО у конечного пользователя

**Цель:** убедиться, что пользователь сможет **установить и начать использовать** продукт без технических проблем.

# Регрессионное тестирование

---

## Основная идея:

Проверка того, что **внесённые изменения (исправления, новые функции)** не нарушили уже существующую функциональность.

## Что проверяется:

- Работоспособность ранее протестированных функций
- Отсутствие побочных эффектов после изменений в коде
- Стабильность критических и часто используемых сценариев

# Регрессионное тестирование

---

## Особенности:

- Выполняется после каждого изменения в кодовой базе
- Часто автоматизируется для повышения скорости и надёжности
- Может включать как функциональные, так и нефункциональные аспекты

## Область применения:

- На всех этапах активной разработки
- В рамках CI/CD-процессов
- Перед каждым релизом или деплоем в production

**Цель:** гарантировать, что «исправив одно, мы не сломали другое».

# Инсталляционное и регрессионное тестирование

---

Эти два вида тестирования решают разные, но важные задачи:

- **Инсталляционное** — обеспечивает корректный старт работы с продуктом,
- **Регрессионное** — защищает стабильность продукта на протяжении всего его жизненного цикла.

Оба критически важны для доставки качественного и надёжного ПО конечному пользователю.