Роль тестирования в процессе разработки на C#

1. Тестирование как часть разработки

Часто тестирование воспринимается как отдельный этап, который начинается после того, как разработка закончена. В этой модели разработчик пишет код, а затем передаёт его QA-инженеру для проверки.

В реальной инженерной практике такой подход плохо масштабируется и почти всегда приводит к проблемам.

Для разработчика тестирование — это не проверка чужого кода, а инструмент контроля качества собственного решения.

Автоматизированные тесты позволяют разработчику:

- проверить корректность бизнес-логики без ручного прогона приложения;
- быстро находить ошибки на раннем этапе;
- безопасно изменять и улучшать код;
- понимать, что именно должно работать и почему.

Преимущества автоматических тестов перед ручной проверкой:

- выполняется одинаково каждый раз;
- проверяет конкретное ожидаемое поведение;
- запускается за доли секунды;
- может быть запущен сотни раз без дополнительных усилий.

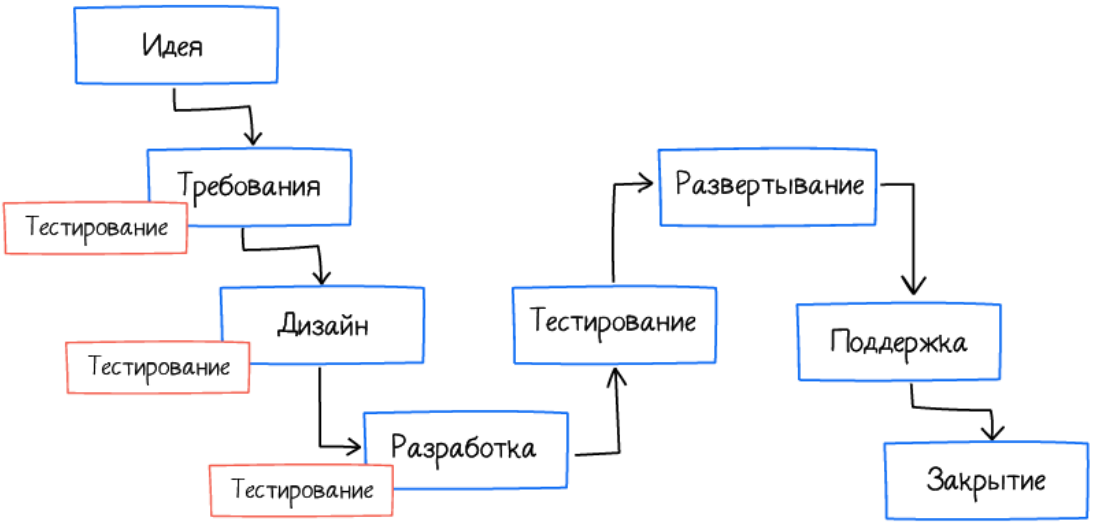
Место тестирования в жизненном цикле приложения

В инженерном процессе тестирование:

- не «после разработки»;
- не «перед релизом»;
- а **параллельно с написанием кода**.

Типичный рабочий цикл:

1. Формулируется требование или задача.
2. Пишется код бизнес-логики.
3. Пишутся тесты, проверяющие ожидаемое поведение.
4. Код дорабатывается и рефакторится.
5. Тесты гарантируют, что поведение не нарушено.



Таким образом, тесты становятся **частью кода**, а не внешним дополнением.

2. Понятие тестируемости кода

Тестируемость — это свойство кода, которое определяет, **насколько легко и надёжно его можно проверить автоматизированными тестами**.

Частая ошибка начинающих разработчиков — оценивать качество кода только по факту, что он:

- компилируется;
- запускается;
- «работает».

Пример:

- метод корректно добавляет задачу в todolist;
- UI отображает результат;
- пользователь доволен.

Но если:

- поведение невозможно проверить без запуска всего приложения;
- для теста нужна база данных;
- результат зависит от времени, файлов или сети;

то такой код сложно сопровождать.

Основные признаки тестируемого кода

Тестируемый код:

- принимает данные через параметры;
- возвращает результат явно;
- не полагается на скрытое состояние.

Плохо тестируется код, который:

- читает глобальные переменные;
- обращается к внешнему состоянию без явного указания.

Чем яснее входы и выходы — тем проще тест.

Для одинаковых входных данных код должен:

- вести себя одинаково;
- возвращать одинаковый результат.

Если результат зависит от:

- текущего времени;
- порядка выполнения тестов;
- внешнего окружения;

то тесты становятся нестабильными и теряют ценность.

Чистые функции

Побочные эффекты:

- запись в базу данных;
- изменение файлов;
- сетевые вызовы;
- изменение глобального состояния.

Для улучшения тестируемости кода необходимо:

- изолировать побочные эффекты;
- выносить их за пределы бизнес-логики;
- явно управлять ими через зависимости.

Идеальный объект для unit-тестирования — это **чистые функции**, которые:

- принимают входные данные;
- возвращают результат;
- не зависит от окружения.

Такой код:

- легко тестируется;
- быстро выполняется;
- не требует сложной настройки.

Чем больше кода вынесено в такие компоненты, тем выше тестируемость проекта.

Зависимости как источник проблем

Большинство проблем с тестируемостью связано с зависимостями.

Плохо тестируемый код:

- использует `static`;
- читает конфигурацию напрямую;
- зависит от окружения.

Такие зависимости:

- сложно контролировать;
- сложно воспроизводить в тестах;
- часто приводят к нестабильным тестам.

Хорошо тестируемый код:

- принимает зависимости через конструктор;
- работает с интерфейсами;
- не создаёт зависимости самостоятельно.

Это позволяет:

- подменять реальные реализации тестовыми;
- изолировать тестируемый код.

Тестируемость и стоимость сопровождения

Код с высокой тестируемостью:

- быстрее изменяется;
- реже ломается;
- проще расширяется.

Код с низкой тестируемостью:

- требует больше ручной проверки;
- накапливает технический долг;
- становится «страшным» для изменений.

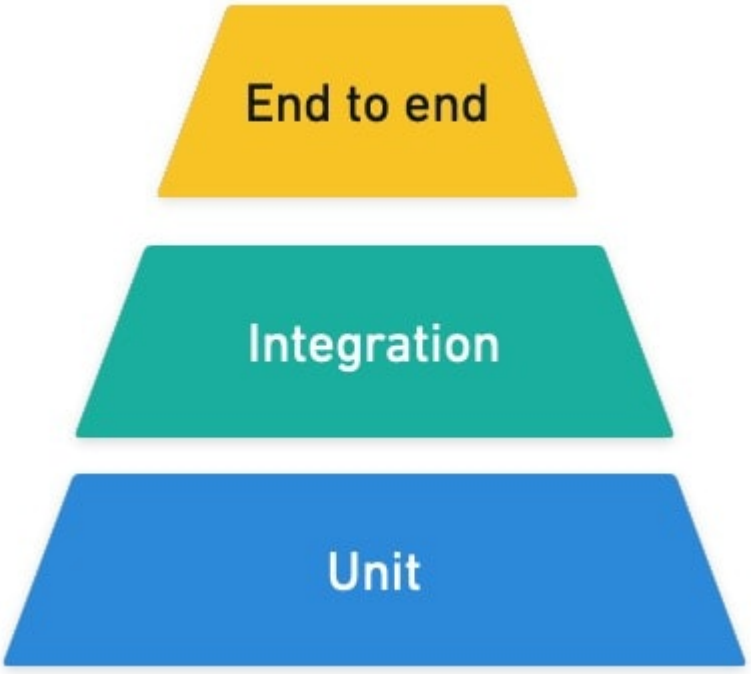
3. Типы тестов

Тесты решают **разные задачи**, работают на **разных уровнях системы** и имеют **разную стоимость**:

- по времени выполнения;
- по сложности написания;
- по сложности поддержки.

Чаще всего используется модель **пирамиды тестирования**:

- **Unit-тесты** — основание пирамиды
- **Integration-тесты** — средний слой
- **End-to-End тесты** — вершина



Чем выше уровень:

- тем тестов меньше;
- тем они медленнее;
- тем дороже их поддержка.

Unit-тесты

Unit-тест проверяет:

- один класс;
- один метод;
- одно бизнес-правило.

Он выполняется:

- быстро (миллисекунды);
- изолированно;
- без внешних зависимостей.

Unit-тесты обычно:

- не используют реальную БД;
- не делают сетевых вызовов;
- не зависят от файловой системы.

Если логика зависит от:

- от БД;
- файлов;
- внешнего API;

то такие зависимости подменяются моками (заменители реальных сервисов) с целью:

- изолировать тестируемый код;
- проверить только его поведение;
- исключить побочные эффекты.

Integration-тесты

Integration-тест проверяет **взаимодействие нескольких компонентов**:

Он запускается:

- медленнее unit-тестов;
- с реальной или тестовой инфраструктурой.

Integration-тесты оправданы, когда нужно проверить:

- работу с API внешних сервисов;
- реальные SQL-запросы;
- конфигурацию приложения.
- корректность работы с файлами

End-to-End (E2E) тесты

E2E-тест проверяет приложение **как единое целое**:

- от входа пользователя;
- до конечного результата.

E2E-тесты нужны, чтобы убедиться, что:

- основные пользовательские сценарии работают;
- система корректно интегрирована целиком.

E2E-тесты:

- самые медленные;
- самые хрупкие;
- самые дорогие в поддержке.

Поэтому:

- их должно быть немного;
- они покрывают только критические сценарии;
- они не заменяют unit- и integration-тесты.

4. Инструменты тестирования в .NET

В экосистеме .NET есть устоявшийся набор инструментов, которые используются в большинстве коммерческих проектов, и которые отвечают за:

- запуск тестов;
- отчёты о результатах;
- интеграцию с IDE и CI.

На данный момент существует 3 основных фреймворка для тестирования:

- MSTest
- NUnit
- xUnit

xUnit — де-факто стандарт в современных .NET-проектах.

Так же существуют библиотеки для создания моков.

Моки используются для:

- подмены реальных зависимостей;
- изоляции тестируемого кода;
- контроля поведения внешних компонентов.

Типичные зависимости для моков:

- БД;
- файлы;
- внешние API;
- логирование.

Moq — самая популярная библиотека моков в .NET.

Позволяет:

- создавать фейковые реализации интерфейсов;
- настраивать возвращаемые значения;
- проверять, что методы были вызваны.

5. Структура тестового проекта в .NET

Тесты — это такой же код, как и production-код.
В .NET тесты **всегда** размещаются в отдельном проекте.

Причины:

- тестовый код не попадает в production-сборку;
- зависимости для тестов не смешиваются с основными;
- упрощается настройка и запуск тестов.

Типичная схема решения:

- `TodoList.App` — основной проект
- `TodoList.Tests` — тестовый проект

Важно:

- тестовый проект ссылается на основной;
- основной проект **не знает** о существовании тестов.

Хорошая практика — **зеркалить структуру основного проекта**.

Если в основном проекте есть папки:

- `Services`
- `Repositories`
- `Domain`

то в тестовом проекте должна быть похожая структура.

Пример:

- `TodoList.Domain`
  - `Services`
  - `Entities`
- `TodoList.Domain.Tests`
  - `Services`
  - `Entities`

Есть практическое правило: **каждый production-класс имеет свой тестовый класс.**

Преимущества:

- тесты не превращаются в «свалку»;
- проще поддерживать соответствие;
- легче видеть, что не покрыто тестами.

Структура внутри тестового класса

Каждый тест должен иметь чёткую структуру:

1. **Arrange** — подготовка данных и зависимостей
2. **Act** — выполнение действия
3. **Assert** — проверка результата

Имя теста должно описывать поведение, а не реализацию.

Распространённый формат:

- `MethodName_WhenCondition_ShouldExpectedResult`

Пример:

- `CompleteTask_WhenTaskIsActive_ShouldMarkAsCompleted`
- `AddTask_WhenTitleIsEmpty_ShouldThrowException`

Даже без тела теста по имени уже понятно, что проверяется.

На следующих лекциях мы подробнее познакомимся как писать автоматические тесты на C#

Практическое задание

Что нужно сделать:

1. **Откройте ваш todolist-проект**, который вы разрабатывали в прошлом семестре.
2. **Создайте в корне проекта папку** `TestCases`.
3. В папке `TestCases` **создайте файл** `TestCases.md`.
4. В этом файле опишите **набор тест-кейсов** для проверки всего функционала приложения.
5. Для **каждого тест-кейса** обязательно укажите:
   - название тест-кейса;
   - описание проверяемого действия;
   - последовательность действий пользователя;
   - ожидаемый результат;
   - скриншоты выполнения теста в работающей программе.
6. **Скриншоты**:
   - делайте скриншоты консольного приложения;
   - сохраняйте их в папку `TestCases/Screenshots`;
   - вставляйте ссылки на изображения в `TestCases.md`.
7. **Проверьте приложение вручную**, строго следуя написанным тест-кейсам.
8. Убедитесь, что:
   - тест-кейсы понятны;
   - по ним может проверить программу другой человек.
9. **Сделайте commit и push** всех изменений в репозиторий.

Требования к структуре тест-кейсов

Каждый тест-кейс должен быть оформлен **по единому шаблону**.

Пример структуры одного тест-кейса

```
### TC-01 Создание нового пользователя

**Описание:**
Проверка корректного создания нового пользователя в системе.

**Предусловия:**
Приложение запущено. Пользователь не авторизован.

**Последовательность действий:**
1. Запустить приложение.
```

2. Выбрать команду `create user`.
3. Ввести имя: `Ivan`.
4. Ввести пароль: `12345`.

Ожидаемый результат:
- Пользователь `Ivan` успешно создан.
- В консоли отображается сообщение об успешном создании пользователя.
- Пользователь сохранён в файле данных.

Скриншоты:
- `Screenshots/TC-01-step1.png`
- `Screenshots/TC-01-step2.png`

Обязательные группы тест-кейсов

В файле `TestCases.md` **должны быть тест-кейсы для следующих функций** в зависимости от вашей реализации, набор тест-кейсов может быть немного другим.

---

1. Пользователи и авторизация

- Создание нового пользователя
- Авторизация пользователя
- Ошибка авторизации
- Выход из текущего пользователя
- Повторная авторизация после выхода
- Ввод данных пользователя
- Корректное отображение данных пользователя

---

2. Добавление задач (add)

- Добавление задачи с корректными данными
- Добавление задачи с пустым названием
- Добавление задачи с длинным текстом
- Добавление многострочных задач
- Добавление задачи со служебными символами (`/` , `;` , `|` , `\` , `:` и т.д.)
- Проверка отображения добавленной задачи (`read`)

---

3. Просмотр задач (view)

- Просмотр списка задач
- Отображаются только задачи текущего пользователя
- Корректное отображение при разных флагах
- Корректное отображение длинных задач
- Корректное отображение для многострочных задач
- Корректное отображение для задач со служебными символами
- Просмотр задач при пустом списке

---

4. Обновление задач (update / status)

- Обновление текста задачи
- Обновление статуса задачи
- Попытка обновить несуществующую задачу
- Проверка отображения обновлённого статуса (`read`)

---

5. Удаление задач (delete)

- Удаление существующей задачи
- Попытка удалить несуществующую задачу
- Проверка, что задача удалена из отображения

---

6. Поиск задач (search)

- Поиск задачи по части строки
- Поиск по первым символам
- Поиск по последним символам
- Поиск с определенной даты
- Поиск до определенной даты
- Поиск с сортировкой результатов
- Поиск с сортировкой результатов по убыванию
- Поиск top n элементов
- Поиск с комбинацией условий
- Поиск при отсутствии совпадений

---

7. Работа с файлами

- Корректное сохранение данных после:
  - добавления пользователя;
    - добавления задачи;
    - обновление задачи;
    - удаления задачи;
    - обновления статуса.
- Корректная загрузка данных при повторном запуске приложения
- Проверка целостности данных в файле

---

8. Парсинг строк

- Корректный парсинг строк из файла
- Обработка строк со служебными символами
- Обработка пустых или повреждённых строк
- Поведение программы при ошибке парсинга

---

9. Undo / Redo

- Отмена последнего действия (undo)
- Повтор отменённого действия (redo)
