**Теория**

**1. Понятие репозитория проекта. Структура проекта.**

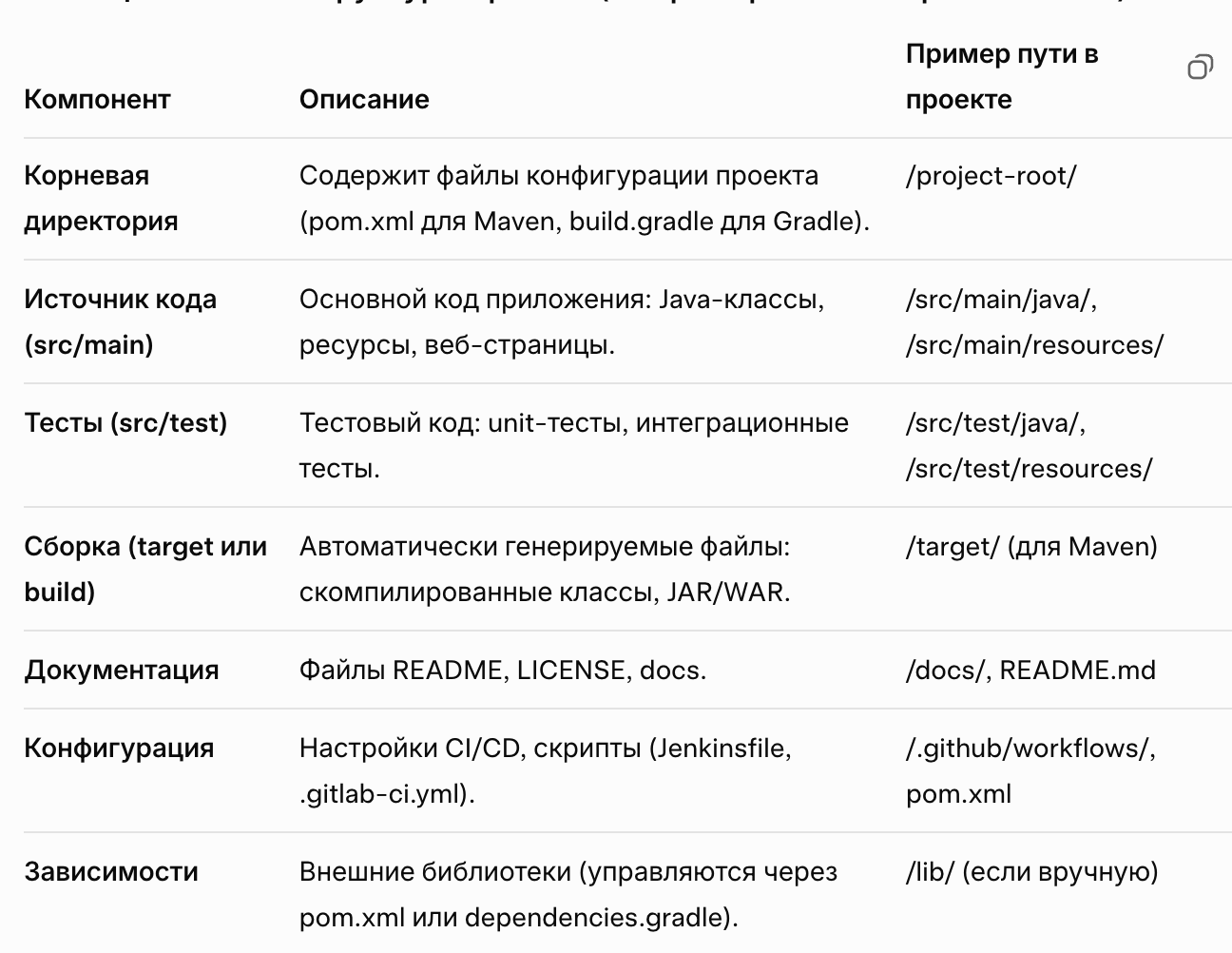
**Понятие репозитория проекта.**

Репозиторий проекта — это централизованное хранилище, где хранятся все файлы, код, документация и история изменений проекта по разработке программного обеспечения. Оно обеспечивает версионный контроль, совместную работу разработчиков и автоматизацию процессов сборки/развертывания. Основные типы: локальный (на машине разработчика) и удаленный (на серверах, например, GitHub, GitLab). Репозиторий позволяет отслеживать изменения через коммиты, ветки и слияния, предотвращая конфликты в коде.

**Схема репозитория проекта (основные компоненты):**

* **Рабочая директория (Working Directory):** Текущие файлы на локальной машине.
* **Индекс (Staging Area):** Временная область для подготовки изменений перед коммитом.
* **Локальный репозиторий (Local Repository):** Хранит историю коммитов локально.
* **Удаленный репозиторий (Remote Repository):** Синхронизируется с локальным для совместной работы.

Таблица типичной структуры проекта (на примере Maven-проекта в Java):



**Дополнительная схема workflow репозитория (Git-based):**

1. Инициализация: git init (локально) или клонирование git clone.
2. Добавление изменений: git add . → git commit -m "message".
3. Синхронизация: git push (на удаленный) → git pull (с удаленного).
4. Ветвление: git branch feature → git merge.

Эта структура облегчает автоматизацию и масштабирование проекта.

**2. Автоматизация бизнес-процессов.**

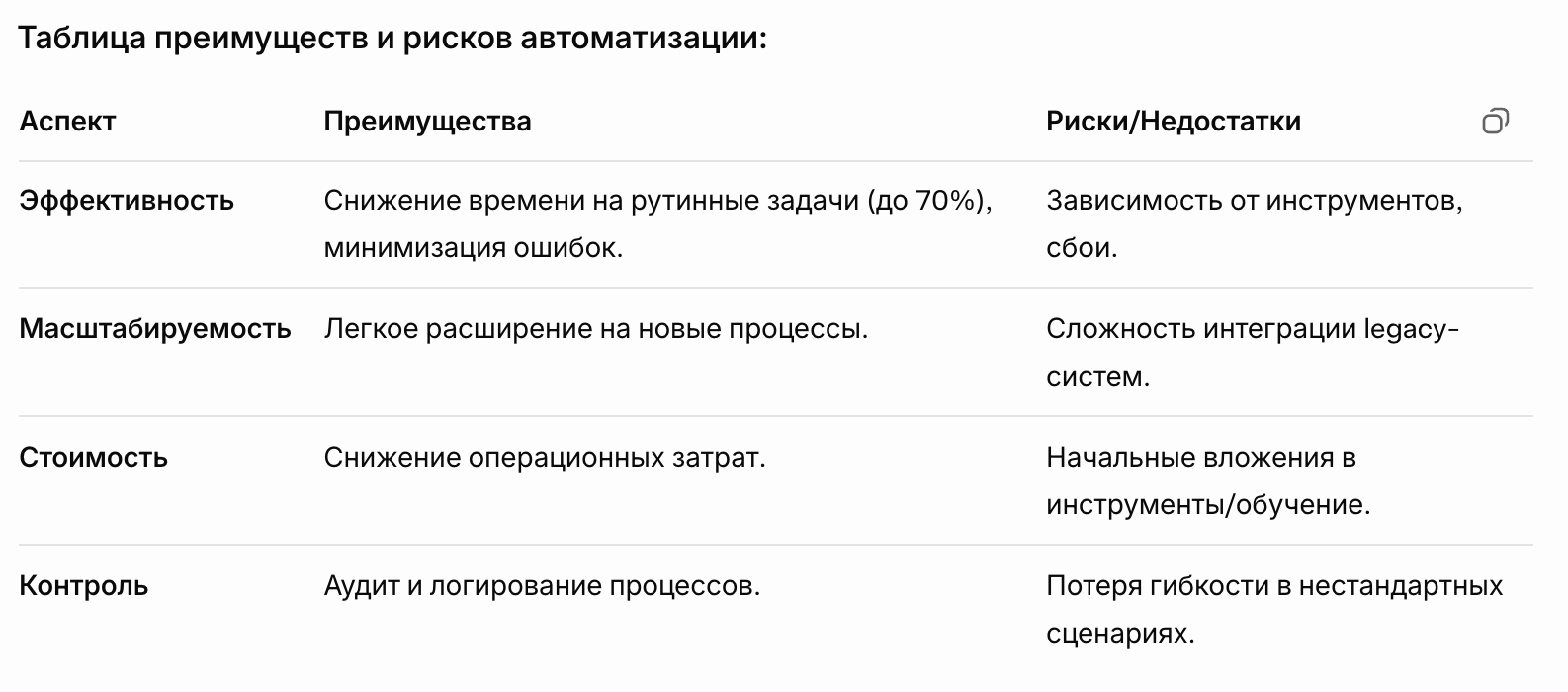
**Понятие автоматизации бизнес-процессов.**

Автоматизация бизнес-процессов (BPM — Business Process Management) — это использование инструментальных средств для моделирования, выполнения и мониторинга повторяющихся задач в организации, с целью повышения эффективности, снижения ошибок и оптимизации ресурсов. В контексте разработки ПО это включает интеграцию инструментов вроде BPMN (Business Process Model and Notation), workflow-систем (Camunda, Activiti) и CI/CD-пайплайнов (Jenkins, GitHub Actions).

**Схема этапов автоматизации бизнес-процессов:**

1. **Анализ и моделирование:** Определение процессов (AS-IS → TO-BE) с помощью диаграмм BPMN.
2. **Реализация:** Создание скриптов/кода для автоматизации (например, в Python с библиотекой Camunda).
3. **Исполнение:** Запуск процессов в runtime-окружении.
4. **Мониторинг и оптимизация:** Сбор метрик, корректировка на основе данных.
5. **Интеграция:** Связь с другими системами (API, базы данных).

Таблица инструментов для автоматизации бизнес-процессов: 

****

Автоматизация интегрируется с репозиториями (e.g., Git hooks для триггеров) для seamless разработки.

**3. Понятие артефакта и протокола проекта. Состав перечня артефактов и протоколов проекта.**

**Понятие артефакта проекта.**

Артефакт проекта — это любой tangible результат или продукт, созданный в процессе разработки ПО: код, бинарные файлы, документация, модели. В build-системах (Maven, Gradle) артефакт — это скомпилированный файл (JAR, WAR, DLL), который можно развернуть или использовать. Артефакты управляются репозиториями вроде Nexus или Artifactory.

**Понятие протокола проекта.**

Протокол проекта — это формализованный документ или запись, фиксирующая решения, встречи, изменения или коммуникации в проекте. Это может быть протокол встречи (meeting minutes), протокол тестирования или сетевой протокол (e.g., HTTP для API). В разработке — это стандарты обмена данными или логи процессов.

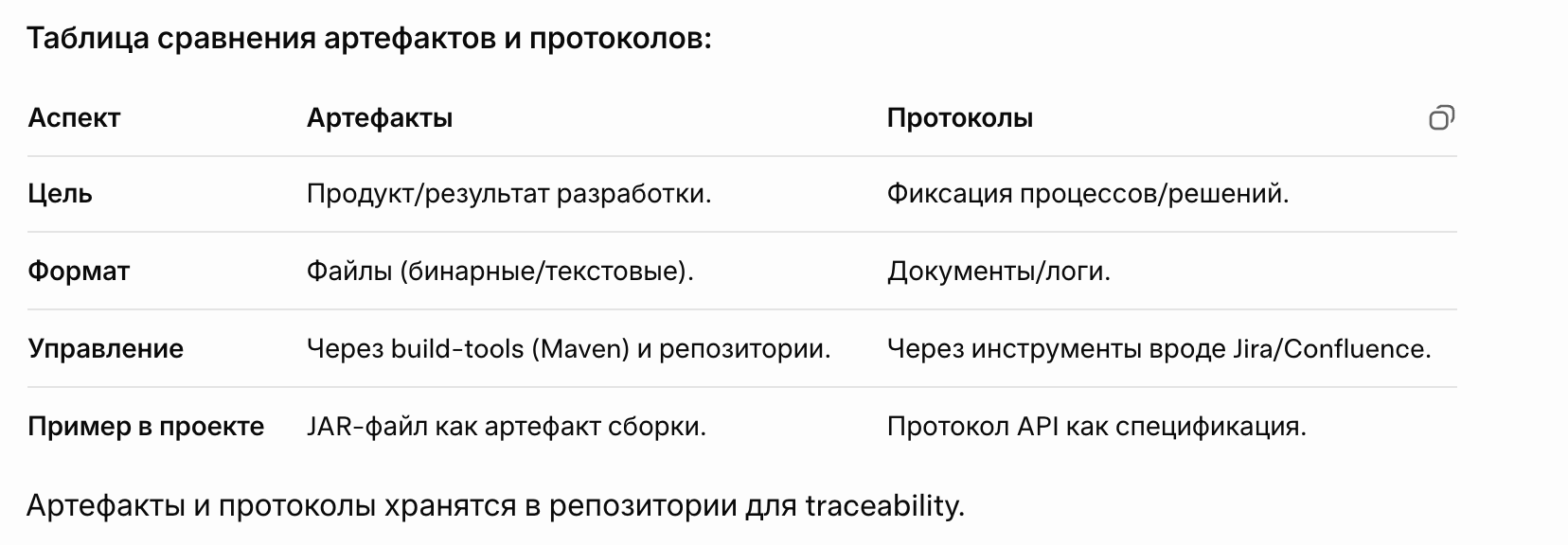
**Схема состава артефактов и протоколов:**

* **Артефакты:** Входные (требования) → Процессные (код, тесты) → Выходные (релиз).
* **Протоколы:** Планирование → Выполнение → Отчетность.

**Таблица состава перечня артефактов проекта (примеры):**







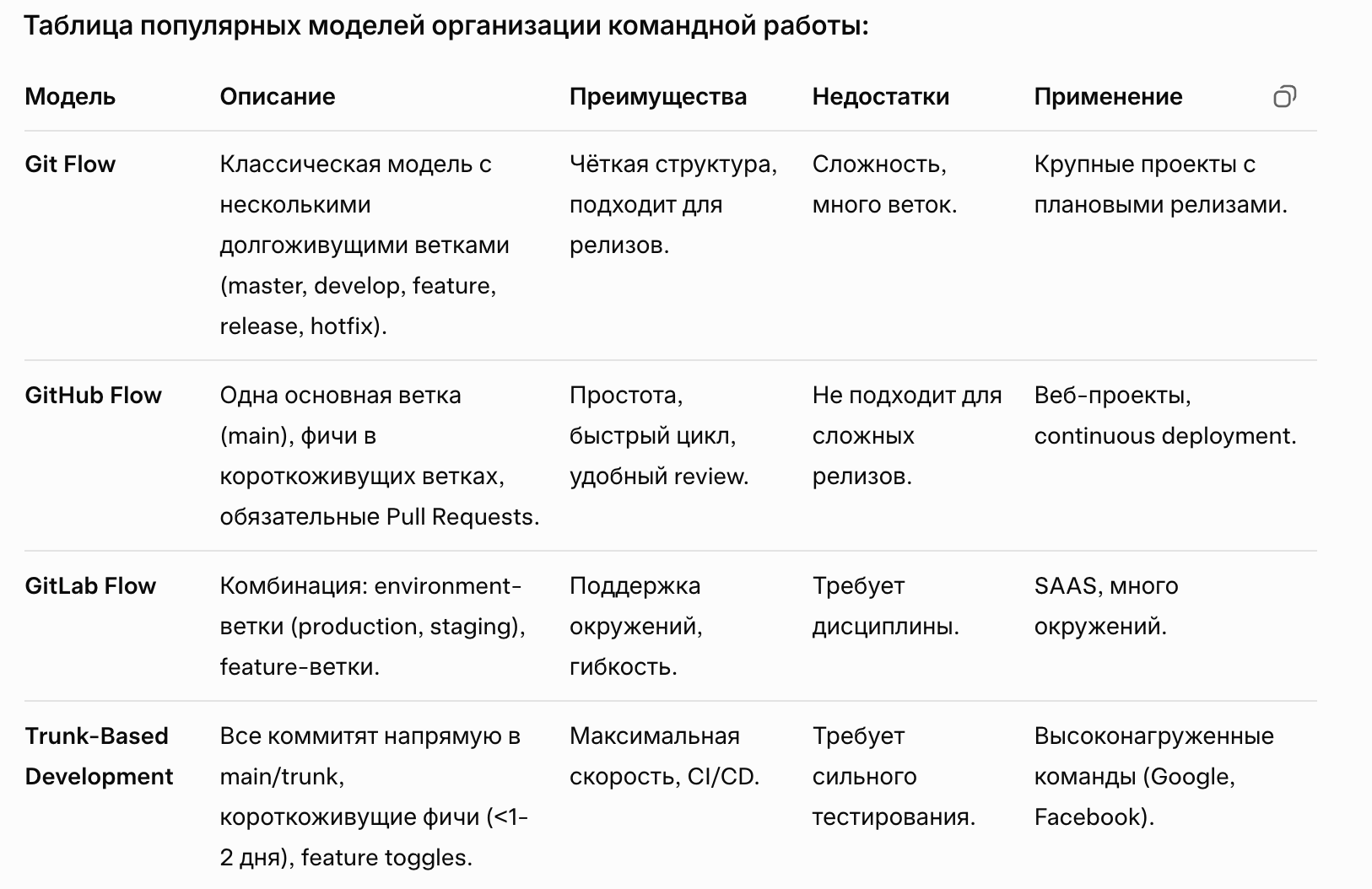
**4. Организация работы команды в системе контроля версий.**

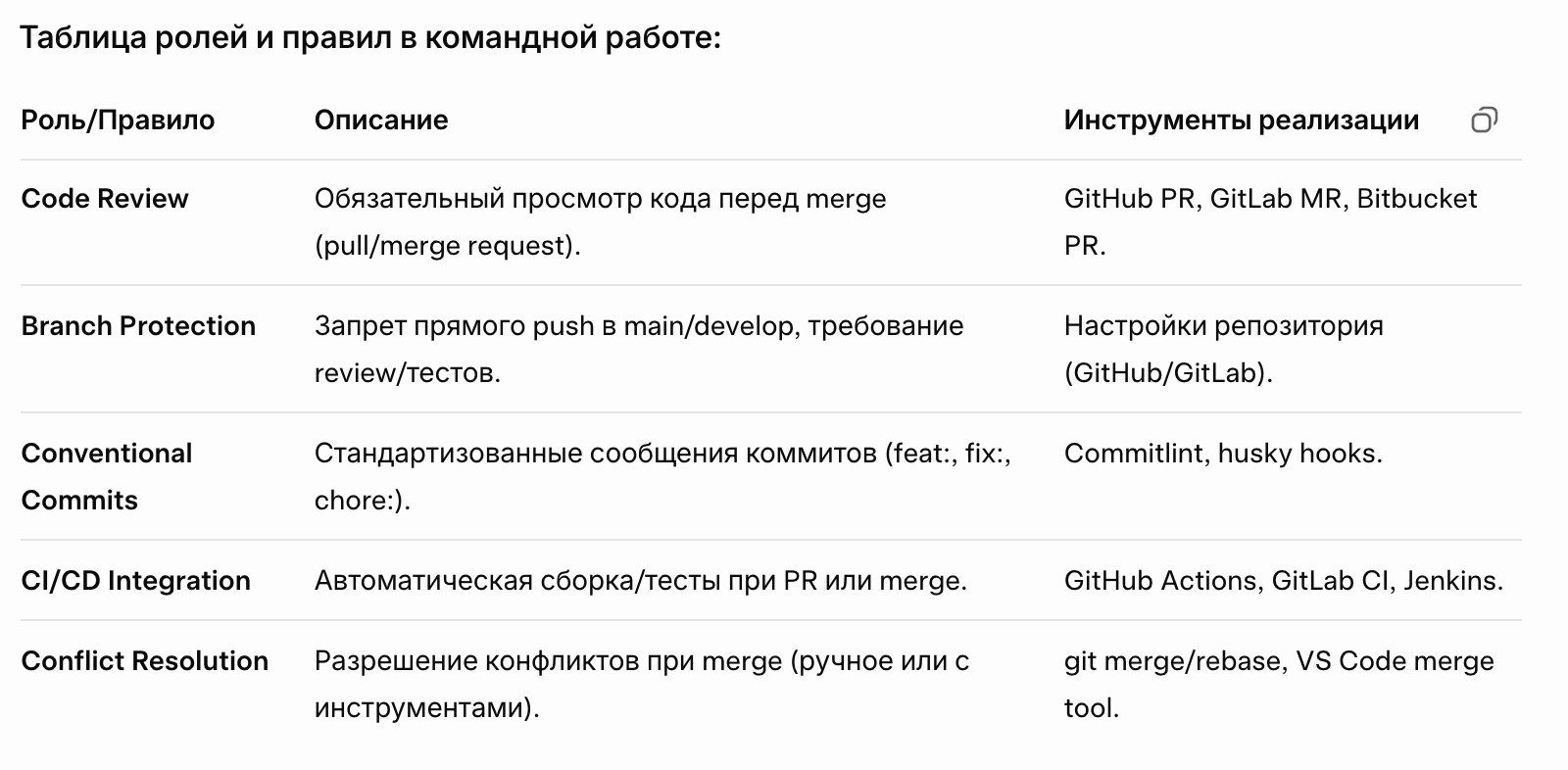
**Понятие организации работы команды в СКВ.**

Система контроля версий (СКВ, VCS) позволяет нескольким разработчикам одновременно работать над одним проектом без конфликтов, сохраняя историю изменений, обеспечивая откат и traceability. Основные принципы: ветвление (branching), слияние (merging), pull-requests/review, защита веток.

**Схема типичного workflow команды (Git Flow как пример):**

1. **main/master** — стабильная ветка (только релизные версии).
2. **develop** — ветка для интеграции фич.
3. **feature/** — ветки для новых функций (создаются от develop).
4. **release/** — ветки для подготовки релиза.
5. **hotfix/** — ветки для срочных исправлений (от main).





**Схема процесса внесения изменений в команде (GitHub Flow):**

1. git checkout -b feature/new-ui
2. Разработка → git commit
3. git push origin feature/new-ui
4. Создать Pull Request → Review → Approve
5. Merge в main → Автоматический деплой

**5. Особенности системы контроля версий Git.**

**Основные особенности Git (отличия от централизованных СКВ типа SVN):**



**6. Система контроля версий Git: создание проекта, локального и удаленного репозитория, фиксация и отправка коммитов.**

**Схема жизненного цикла работы с Git:**

1. **Создание проекта и локального репозитория**

* Вариант A (новый проект):

Bash

mkdir my-project

cd my-project

git init *# Создаёт .git директорию — локальный репозиторий*

touch README.md

git add README.md

git commit -m "Initial commit"

Вариант B (из существующего кода):

Bash

cd existing-project

git init

git add .

git commit -m "Initial commit"

2. Создание удалённого репозитория

- На платформе: GitHub → New repository → имя (без инициализации README).

- Локально связать:

Bash

git remote add origin https://github.com/username/my-project.git

git branch -M main *# Переименовать ветку в main (рекомендация)*

git push -u origin main *# Первый push с установкой upstream*

3. Клонирование существующего репозитория

Bash  git clone <https://github.com/username/my-project.git>

cd my-project      Схема типичного цикла разработки:

1. git pull — обновить локальную ветку.
2. git checkout -b feature/x — новая ветка.
3. Редактирование файлов.
4. git status → git add . → git commit -m "feat: добавлена функция X".
5. git push — отправить ветку.
6. Создать Pull Request на платформе.
7. После merge: git checkout main → git pull → удалить ветку git branch -d feature/x.

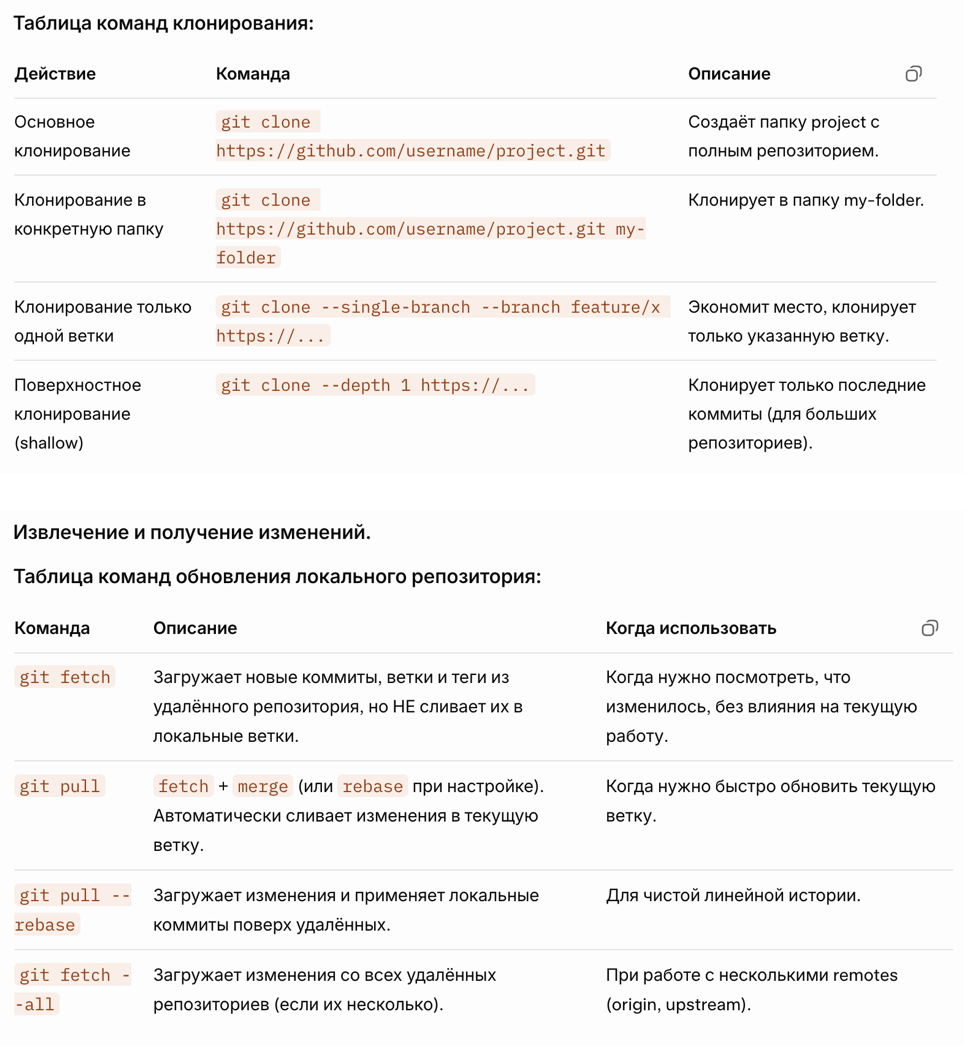
Эти команды и workflow обеспечивают безопасную и эффективную командную разработку.



**7. Система контроля версий Git: клонирование удаленного репозитория, извлечение и получение изменений, совместная параллельная разработка, работа с ветвями.**

**Клонирование удалённого репозитория.**

Клонирование — создание полной локальной копии удалённого репозитория со всей историей коммитов, ветками и тегами.



**Совместная параллельная разработка.**

Разработчики работают в изолированных ветках, периодически синхронизируются с общей веткой (main/develop).

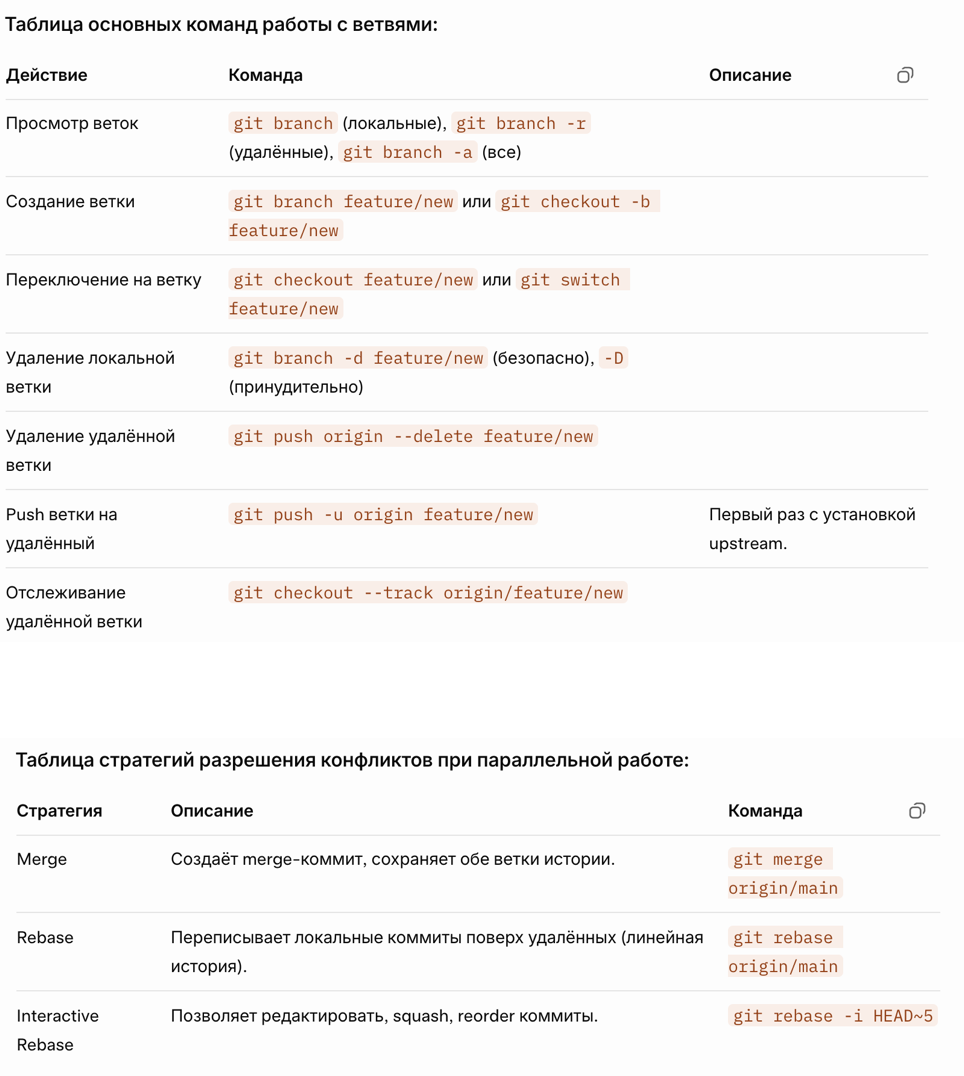
**Схема типичного процесса параллельной разработки:**

1. Клонирование: git clone ...
2. Создание feature-ветки: git checkout -b feature/login
3. Разработка и локальные коммиты.
4. Периодическое обновление: git fetch origin → git rebase origin/main (или merge).
5. Push своей ветки: git push -u origin feature/login
6. Создание Pull Request → Code Review → Merge в main.
7. После merge: обновить локальную main и удалить ветку.

Работа с ветвями (branches).

Схема жизненного цикла ветки в команде:

feature → разработка → push → PR → review → merge в main → delete (локально и удалённо).



1. **Понятие отладки программных продуктов.**

Определение отладки.

Отладка (debugging) — процесс обнаружения, локализации и исправления ошибок (багов) в программном коде, возникающих на этапах разработки, тестирования или эксплуатации. Цель — добиться корректного поведения программы в соответствии с требованиями.

**Схема жизненного цикла ошибки:**

1. **Обнаружение** — программа ведёт себя неверно (краш, неверный результат, исключение).
2. **Воспроизведение** — создание условий, при которых ошибка проявляется стабильно.
3. **Локализация** — поиск места в коде, вызывающего ошибку.
4. **Анализ** — понимание причины (логическая ошибка, неверное состояние, race condition и т.д.).
5. **Исправление** — изменение кода.
6. **Верификация** — повторное тестирование (регрессионные тесты).

****

**Отладка vs Тестирование:**

* Тестирование — поиск наличия ошибок.
* Отладка — поиск причины и места ошибки после её обнаружения.

1. **Инструменты отладки программных продуктов.**

****

**Схема типичного процесса отладки в IDE:**

1. Установка breakpoint (клик по строке или F9).
2. Запуск в debug-режиме (Shift+F9 в IntelliJ).
3. Остановка на breakpoint → осмотр переменных (Variables view).
4. Шаговое выполнение: Step Over (F8), Step Into (F7), Resume (F9).
5. Evaluate expression — выполнение кода в runtime.
6. Исправление → HotSwap (если поддерживается) или перезапуск.

**Дополнительные приёмы отладки:**

* Print-debugging (логи/printf) — быстро, но засоряет код.
* Rubber duck debugging — объяснение кода "уточке".
* Binary search по коммитам (git bisect) — поиск коммита, сломавшего код.

Эти инструменты и подходы позволяют эффективно локализовать и устранять ошибки на всех этапах разработки.

1. **Понятие ручного и автоматизированного тестирования программных продуктов.**

**Ручное тестирование (Manual Testing)**

Ручное тестирование — процесс выполнения тестовых сценариев человеком без использования автоматизированных инструментов. Тестировщик вручную вводит данные, выполняет действия в приложении и проверяет результаты на соответствие ожидаемым.

**Автоматизированное тестирование (Automated Testing)**

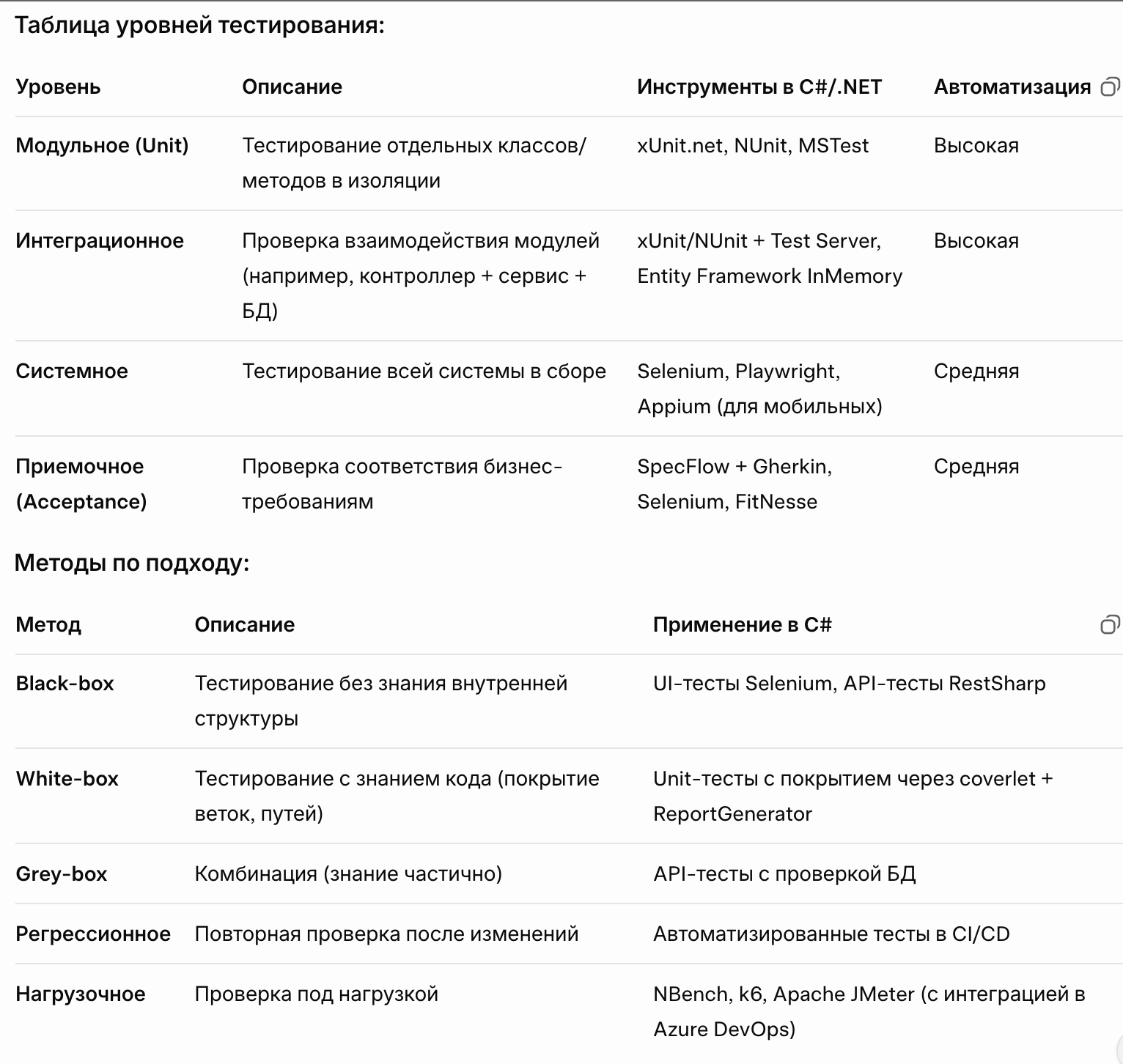
Автоматизированное тестирование — процесс выполнения тестов с помощью специальных инструментов и скриптов, написанных на языках программирования (включая C#). Тесты запускаются автоматически, результаты проверяются без вмешательства человека.



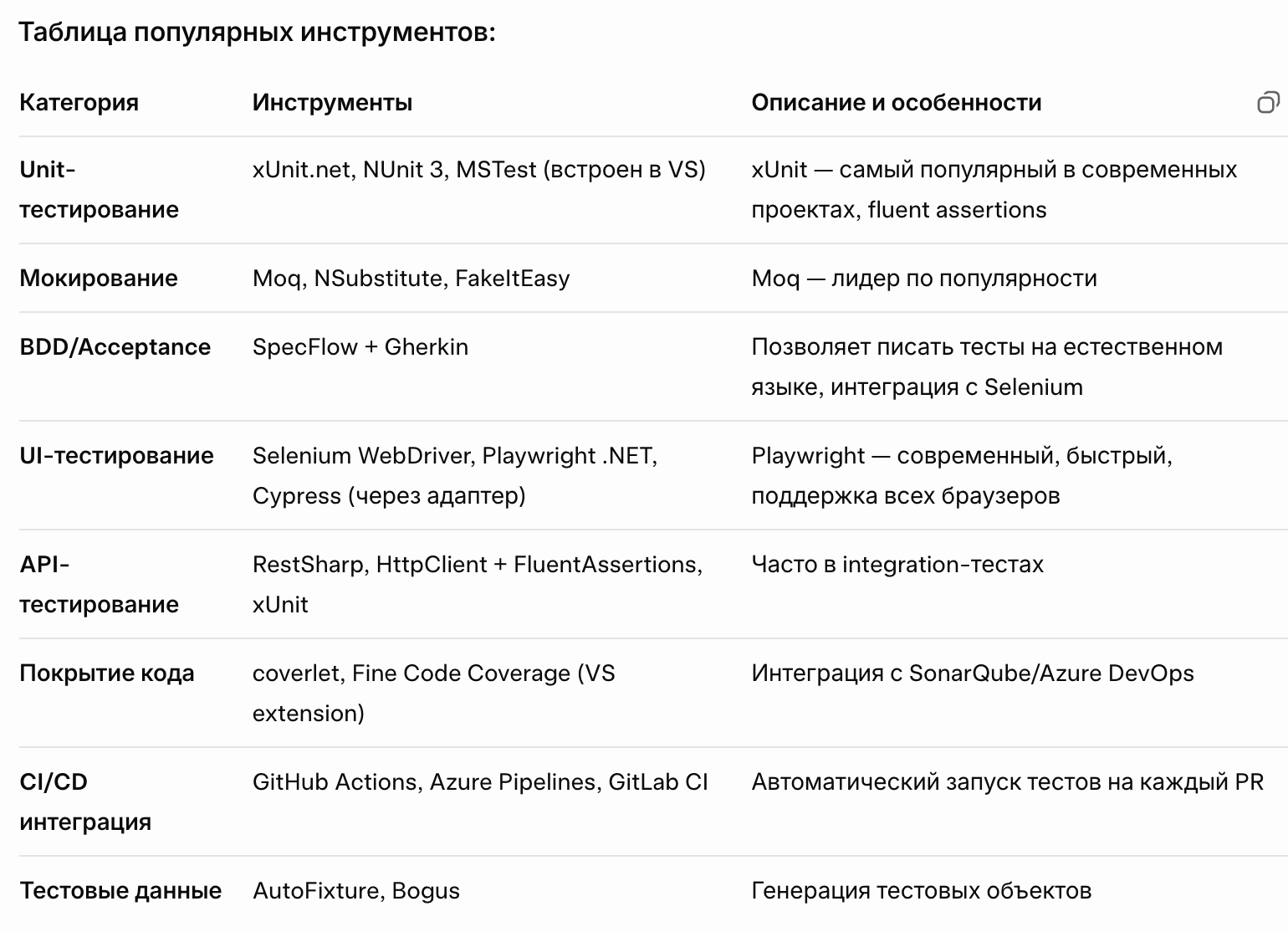
**Схема места тестирования в жизненном цикле .NET-проекта:**

1. **Unit-тесты** → почти всегда автоматизированные (xUnit, NUnit, MSTest).
2. **Integration-тесты** → чаще автоматизированные.
3. **UI-тесты** → смесь (ручные для exploratory, автоматизированные для регрессии).
4. **Acceptance-тесты** → часто ручные или BDD (SpecFlow).
5. **Exploratory/Usability** → только ручные.
6. **Методы и средства организации тестирования программных продуктов.**

Методы тестирования (уровни):



Средства организации тестирования в .NET:



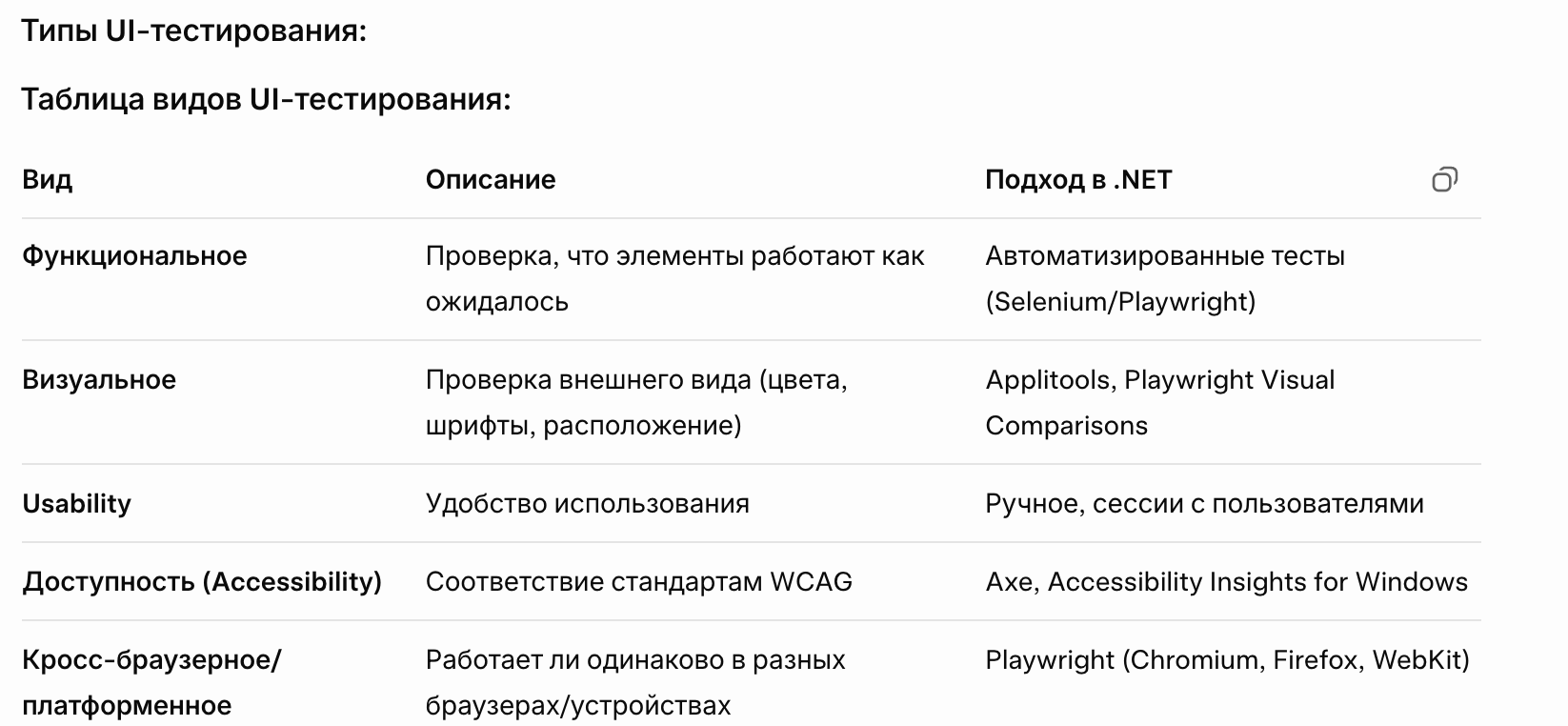
**Схема организации тестирования в типичном .NET-проекте:**

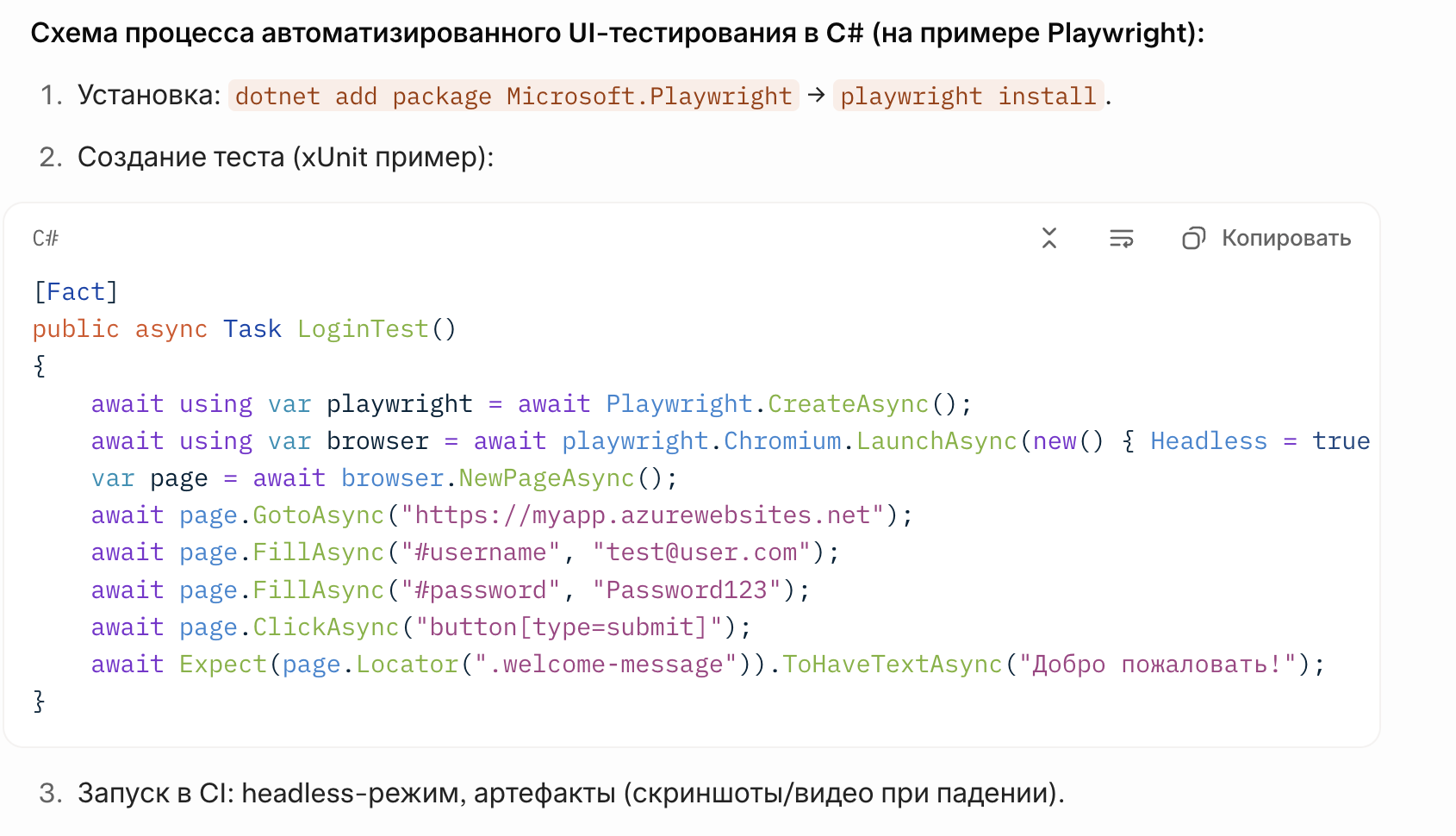
1. **Test Explorer в Visual Studio/Rider** — запуск и отладка всех тестов.
2. **dotnet test** — CLI-команда для запуска в CI.
3. **Пирамида тестирования:** много unit-тестов → меньше integration → мало UI-тестов.
4. **CI-пайплайн:** build → unit tests → integration tests → publish artifacts.
5. **Тестирование интерфейса пользователя программных продуктов.**

Понятие UI-тестирования.

Тестирование пользовательского интерфейса — проверка корректности отображения, поведения элементов управления, навигации и взаимодействия пользователя с приложением (WPF, WinForms, Blazor, ASP.NET MVC/Core, MAUI).

Типы UI-тестирования:

****

****

1. Запуск в CI: headless-режим, артефакты (скриншоты/видео при падении).

**Рекомендации по UI-тестированию в .NET-проектах:**

* Использовать Page Object Model (POM) для поддержки тестов.
* Минимизировать количество UI-тестов (они хрупкие и медленные).
* Комбинировать: основная логика — unit/integration, критичные пользовательские пути — автоматизированные UI.
* Для Blazor предпочтительнее bUnit (компонентное тестирование) вместо полноценного браузерного UI-тестирования.

Эти подходы обеспечивают высокое качество интерфейса пользователя в приложениях на C#.

1. **Функциональное тестирование программных продуктов.**

Понятие функционального тестирования. Функциональное тестирование — вид тестирования, при котором проверяется соответствие поведения программы заданным функциональным требованиям (спецификациям). Тестируется «что делает система», а не «как она это делает» (в отличие от белого ящика). Основное внимание уделяется входным данным, действиям пользователя и ожидаемым выходным результатам.



**Схема процесса функционального тестирования:**

1. **Анализ требований** → создание тест-кейсов.
2. **Подготовка тестовых данных**.
3. **Выполнение тестов** (ручное или автоматизированное).
4. **Сравнение фактического и ожидаемого результата**.
5. **Регистрация дефектов** (Azure DevOps, Jira).
6. **Регрессионное тестирование после исправления**.

****

1. **Тестирование интеграции программных продуктов.**

Понятие тестирования интеграции. Тестирование интеграции — проверка взаимодействия между отдельными модулями, компонентами или сервисами системы после их модульного тестирования. Цель — выявить ошибки на стыках: неверный формат данных, неправильная последовательность вызовов, проблемы с транзакциями и т.д.



****

****

1. **Инструментарии анализа качества программных продуктов в среде разработки.**

Понятие анализа качества кода. Анализ качества — процесс автоматической проверки исходного кода на наличие потенциальных ошибок, нарушений стандартов, уязвимостей безопасности, дублирования, сложности и поддерживаемости.

****

****

**Схема интеграции анализа качества в процесс разработки .NET:**

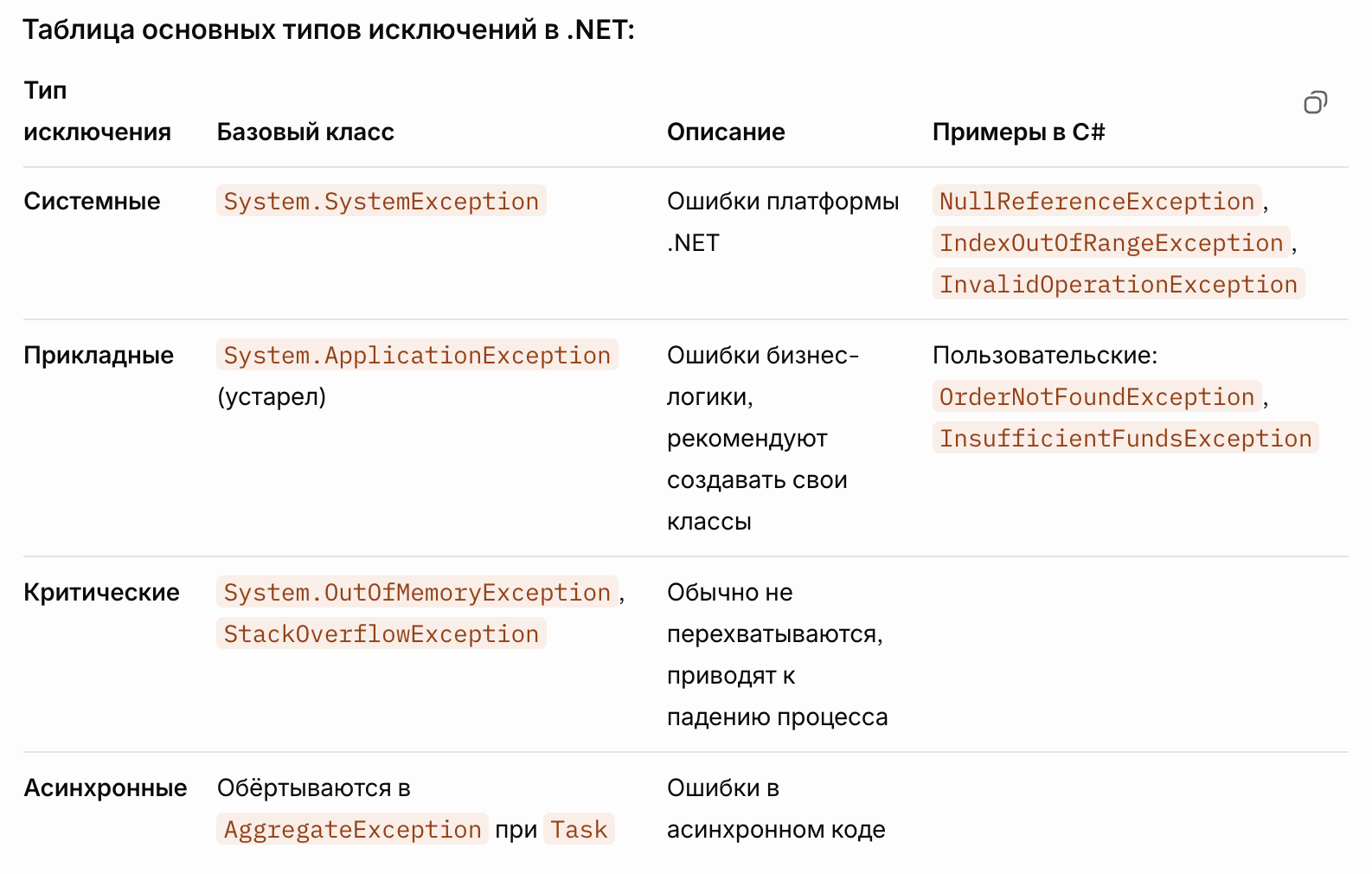
1. **Локально в IDE** — Roslyn/ReSharper подсвечивают проблемы в реальном времени.
2. **При сборке** — dotnet build запускает встроенные анализаторы (предупреждения как ошибки через <TreatWarningsAsErrors>true</TreatWarningsAsErrors>).
3. **В CI/CD** — GitHub Actions / Azure Pipelines:
   * Запуск dotnet test --collect:"XPlat Code Coverage".
   * SonarCloud scan.
   * Quality Gate: блокировка merge при низком качестве.

**Рекомендуемые практики для .NET-проектов:**

* Установить NuGet-пакеты: SonarAnalyzer.CSharp, Microsoft.CodeAnalysis.NetAnalyzers.
* Добавить .editorconfig для унификации стиля кода.
* Настроить Quality Gate: покрытие ≥ 80%, Maintainability Rating A, нет критических уязвимостей.

1. **Понятие исключительной ситуации. Обработка исключительных ситуаций.**

**Понятие исключительной ситуации (Exception).** Исключительная ситуация — это отклонение от нормального хода выполнения программы, возникающее во время runtime (выполнения). Оно сигнализирует о проблеме: неверные данные, отсутствие ресурса, сбой оборудования, логическая ошибка и т.д. В C# исключения представлены классами, наследующимися от System.Exception.

****

**Схема механизма обработки исключений в C#:**

1. **Возбуждение (throw)** — throw new Exception("Сообщение");
2. **Перехват (catch)** — блок catch
3. **Очистка (finally)** — выполняется всегда
4. **Проброс вверх по стеку** — если не обработано, передаётся вызывающему методу

Исключительная ситуация в C# — это любое отклонение от нормального выполнения программы, которое возникает во время работы. Например, деление на ноль, попытка обратиться к null-объекту, выход за пределы массива, отсутствие файла или проблемы с сетью.

В .NET все исключения наследуются от класса System.Exception. Есть две большие группы:

* системные исключения (System.SystemException) — это NullReferenceException, IndexOutOfRangeException, InvalidOperationException и другие;
* прикладные — это наши собственные исключения, например, OrderNotFoundException или InsufficientFundsException.

Обработка исключений строится вокруг трёх ключевых блоков: try, catch и finally.

* try — это блок, где мы пишем код, который может вызвать проблему;
* catch — ловим конкретное исключение и обрабатываем его;
* finally — выполняется всегда, даже если было исключение, — здесь обычно закрываем ресурсы.

C#

try

{

*// опасный код*

}

catch (FileNotFoundException ex)

{

*// обработка именно этой ошибки*

}

catch (Exception ex)

{

*// ловим всё остальное*

}

finally

{

*// всегда выполнится*

}

Важные правила:

* Ловить только то, что реально можем обработать.
* Никогда не писать catch (Exception) без логирования и проброса.
* В асинхронном коде используем await Task, а не .Result, чтобы не создавать deadlock.
* В ASP.NET Core часто ставят глобальный обработчик через middleware UseExceptionHandler.

1. **Методы и способы идентификации сбоев и ошибок.**

Идентификация сбоев — это обнаружение, где именно программа ведёт себя неправильно, и почему.

Основные способы:

1. Логирование — самый важный. Пишем в лог всё: входные параметры, состояние, исключения. Используем Serilog или Microsoft.Extensions.Logging. В продакшене смотрим логи в Seq, Application Insights или Azure Monitor.
2. Отладка в IDE — Visual Studio или Rider. Ставим точки останова, смотрим переменные, стек вызовов, шаговое выполнение.
3. Анализ стека вызовов — Exception.StackTrace показывает, откуда пришла ошибка.
4. Мониторинг в продакшене — Application Insights, OpenTelemetry. Автоматически ловят исключения, показывают, какой запрос, какой пользователь, сколько времени.
5. Дампы памяти — если приложение падает, собираем дамп (dotnet-dump) и анализируем в WinDbg или Visual Studio.
6. Профилирование — dotMemory, PerfView — ищут утечки памяти и медленные места.
7. Health Checks — проверяем, жив ли сервис, БД, диск и т.д.
8. Автоматические тесты — unit и integration — ловят ошибки ещё на этапе сборки.
9. **Ошибки системных компонентов и способы их выявления.**

Системные компоненты — это всё, что не написано нами: ОС, .NET Runtime, IIS/Kestrel, база данных, сеть, диск, внешние API.

Типичные ошибки:

* Нет доступа к файлу → IOException, UnauthorizedAccessException
* Диск переполнен → DiskFullException
* Таймаут сети → HttpRequestException, TaskCanceledException
* Проблемы с базой данных → SqlException, DbUpdateException
* Утечка памяти → OutOfMemoryException
* Deadlock в асинхронном коде → AggregateException

Способы выявления:

1. Логирование зависимостей — включаем EF Core logging, HttpClient logging, видим все запросы и ошибки.
2. OpenTelemetry / Application Insights — собирают трассировки зависимостей (dependency tracking). Видно: сколько времени на запрос к SQL, какой код ответа от внешнего API.
3. Health Checks — проверяем БД, диск, внешние сервисы автоматически.
4. Polly — библиотека для retry, circuit breaker — автоматически обрабатывает временные сбои.
5. Мониторинг метрик — CPU, память, количество соединений, количество 5xx ошибок.
6. Windows Event Log — для IIS и Windows-сервисов.
7. Crash dumps — собираем автоматически при падении (Procdump или Azure).
8. Тестовые окружения — staging с реальными БД и сервисами, чтобы ловить проблемы заранее.

Главное — не просто ловить ошибку, а сразу видеть: какой запрос, кто его вызвал, сколько времени прошло, какой код ответа. Тогда исправить можно быстро.

1. **Перечень документов о результатах тестирования.**

При тестировании программных продуктов, особенно в проектах на C# и .NET, результаты фиксируются в стандартных документах. Основные из них:

* **Тест-план** (Test Plan) — описывает стратегию тестирования, уровни, ресурсы, сроки.
* **Тест-кейсы** (Test Cases) — детальные шаги: идентификатор, описание, предусловия, шаги выполнения, ожидаемый результат, фактический результат, статус (Passed/Failed/Blocked).
* **Отчёт о результатах тестирования** (Test Report или Test Summary Report) — итоговый документ: сколько тестов выполнено, сколько пройдено, сколько дефектов, процент покрытия, критичные баги.
* **Журнал дефектов** (Bug Report или Defect Report) — для каждого бага: ID, описание, шаги воспроизведения, severity (критичность), priority, скриншоты, логи, статус (New, Open, Fixed, Closed).
* **Матрица трассируемости требований** (Requirements Traceability Matrix) — показывает, какие требования покрыты какими тест-кейсами.
* **Отчёт о покрытии кода** (Code Coverage Report) — генерируется coverlet или SonarQube, показывает процент покрытых строк и веток.
* **Протокол приёмочного тестирования** (Acceptance Test Protocol) — подписывается заказчиком.

В командах часто используют Azure DevOps Test Plans, Jira + Xray или TestRail для хранения этих документов.

1. **Процедура проведения инспекции кода модулей проекта.**

Инспекция кода — это формальный или неформальный процесс проверки кода коллегами перед слиянием в основную ветку.

Процедура обычно такая:

1. Разработчик завершает задачу, пишет тесты, коммитит в feature-ветку.
2. Создаёт Pull Request (PR) в GitHub, GitLab или Azure DevOps.
3. Назначает ревьюеров (минимум 1–2 коллеги, часто senior или владелец модуля).
4. Ревьюеры проверяют:
   * Соответствие требованиям и архитектуре.
   * Читаемость, стиль кода (соответствие .editorconfig, Roslyn Analyzers).
   * Наличие тестов и покрытия.
   * Потенциальные баги, утечки ресурсов, безопасность.
   * Производительность и отказоустойчивость.
5. Ревьюеры оставляют комментарии: обязательные (blocking) и необязательные.
6. Разработчик отвечает, вносит правки, добавляет новые коммиты.
7. После одобрения (Approve) — merge в основную ветку (часто с squash или rebase).
8. Автоматические проверки: CI/CD запускает build, тесты, анализаторы — если падают, merge блокируется.

В .NET-командах часто включают правила ветки: обязательный review, успешный CI, минимум один approve.

1. **Обработка исключений. Объявление исключений. Привести пример (представить код).**

Объявление исключения — это создание собственного класса исключения, когда стандартных не хватает. Обычно наследуемся от Exception или ApplicationException (хотя сейчас рекомендуют просто от Exception).

Пример:

C#

*// Объявляем своё исключение*

public class OrderNotFoundException : Exception

{

public OrderNotFoundException() : base() { }

public OrderNotFoundException(string message) : base(message) { }

public OrderNotFoundException(string message, Exception inner)

: base(message, inner) { }

*// Часто добавляем дополнительные свойства*

public int OrderId { get; }

public OrderNotFoundException(int orderId, string message = "Заказ не найден")

: base(message)

{

OrderId = orderId;

}

}

Использование:

C#

if (order == null)

{

throw new OrderNotFoundException(orderId: 123, "Заказ с таким ID отсутствует");

}

1. **Обработка исключений. Выбрасывание исключений. Привести пример (представить код).**

Выбрасывание (throw) — это генерация исключения в коде.

Примеры:

C#

*// Простое выбрасывание нового исключения*

if (string.IsNullOrEmpty(userName))

{

throw new ArgumentException("Имя пользователя не может быть пустым", nameof(userName));

}

*// Переброс того же исключения (сохраняем стек)*

try

{

File.ReadAllText("config.json");

}

catch (FileNotFoundException ex)

{

logger.LogError(ex, "Конфиг не найден");

throw; *// переброс без потери стека*

}

*// Выбрасывание нового с сохранением внутреннего*

try

{

int.Parse("не число");

}

catch (FormatException inner)

{

throw new ArgumentException("Неверный формат числа", inner);

}

1. **Обработка исключений. Перехват исключений. Привести пример (представить код).**

Перехват — это catch-блоками ловим исключение.

Пример:

C#

try

{

var config = File.ReadAllText("settings.json");

var data = JsonSerializer.Deserialize<Config>(config);

return data;

}

catch (FileNotFoundException ex)

{

logger.LogWarning(ex, "Файл настроек не найден, используем значения по умолчанию");

return new Config(); *// дефолтные настройки*

}

catch (JsonException ex)

{

logger.LogError(ex, "Ошибка парсинга JSON");

throw new InvalidOperationException("Невозможно загрузить конфигурацию", ex);

}

catch (Exception ex) *// ловим всё остальное*

{

logger.LogCritical(ex, "Неожиданная ошибка при загрузке конфигурации");

throw;

}

finally

{

logger.LogInformation("Завершение загрузки конфигурации");

}

1. **Обработка исключений. Получение информации об исключениях. Привести пример (представить код).**

В catch мы получаем объект Exception, у которого много полезных свойств.

C#

try

{

*// какой-то код*

}

catch (Exception ex)

{

*// Основные свойства*

Console.WriteLine($"Тип исключения: {ex.GetType().Name}");

Console.WriteLine($"Сообщение: {ex.Message}");

Console.WriteLine($"Стек вызовов: {ex.StackTrace}");

*// Если есть внутреннее*

if (ex.InnerException != null)

{

Console.WriteLine($"Внутреннее исключение: {ex.InnerException.Message}");

}

*// Дополнительные данные (если добавляли)*

if (ex.Data.Count > 0)

{

foreach (var key in ex.Data.Keys)

{

Console.WriteLine($"{key}: {ex.Data[key]}");

}

}

*// Для отладки в Visual Studio*

System.Diagnostics.Debugger.Break(); *// остановка при отладке*

}

Ещё полезно: ex.Source, ex.TargetSite, ex.HelpLink.

1. **Обработка исключений. Когда следует использовать исключения. Привести пример (представить код).**

Исключения следует использовать только для **исключительных**, **неожиданных** ситуаций — ошибок, от которых программа не может продолжить нормальную работу без вмешательства.

Не используем исключения для управления потоком (вместо if).

Правильно — исключения:

C#

*// Правильно: файл действительно может отсутствовать по независящим причинам*

public string LoadConfig()

{

try

{

return File.ReadAllText("app.config");

}

catch (FileNotFoundException ex)

{

throw new ConfigurationException("Конфигурационный файл не найден", ex);

}

}

*// Правильно: неверный формат — ошибка данных*

public int ParseAge(string input)

{

if (!int.TryParse(input, out int age))

{

throw new FormatException("Возраст должен быть числом");

}

if (age < 0 || age > 150)

{

throw new ArgumentOutOfRangeException(nameof(age), "Недопустимый возраст");

}

return age;

}

Неправильно использовать исключения:

C#

*// Плохо: поиск элемента — нормальная ситуация, если не найден*

try

{

var user = users.First(u => u.Id == id);

}

catch (InvalidOperationException) *// Sequence contains no matching element*

{

return null; *// вместо этого лучше FirstOrDefault()*

}

Правило: если ситуация предсказуема и обрабатывается в обычном потоке — используем return null, bool-результат, TryParse-паттерн. Исключения — только для реальных ошибок.