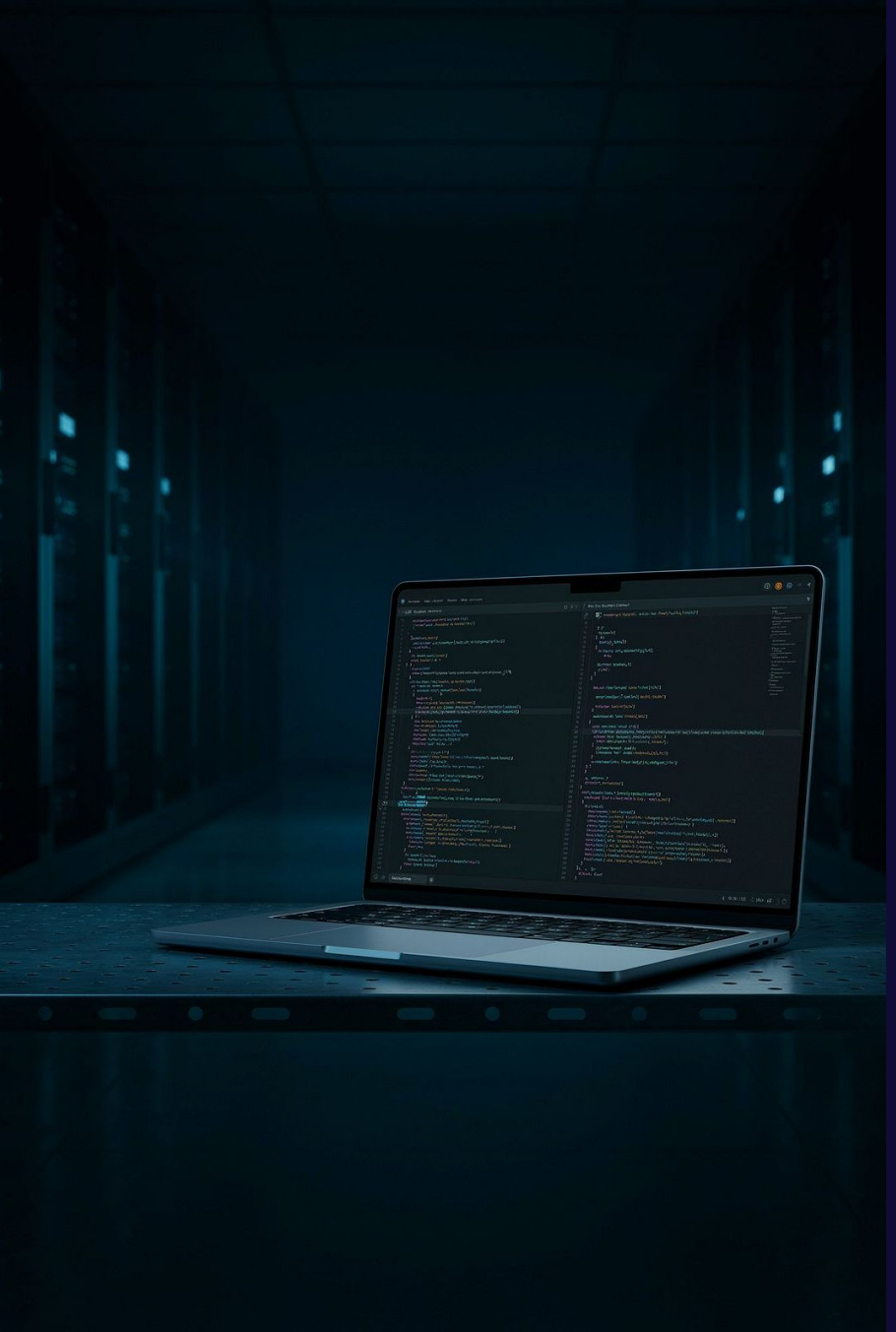


Оценка и улучшение репозитория программного проекта

Как понять, в каком состоянии находится программный проект, и привести его репозиторий в порядок

Дисциплина: Сопровождение, обслуживание и развитие программных проектов · 3 курс IT-колледжа · 2 академических часа

Проблемная ситуация

Представьте: вы присоединяетесь к уже существующему проекту. Открываете репозиторий — и видите следующее:

Файлы и папки

Десятки файлов, несколько папок с непонятными названиями, старые версии программы рядом с новыми.

Документация

README написан год назад. Часть файлов, которые в нём упоминаются, отсутствует в репозитории.

История Git

Коммиты называются «fix», «123», «работает», «final». Никто точно не знает, какая версия является актуальной.

⚠ **Вопрос к классу:** Можно ли сразу начинать изменять такой проект? — Нет. Сначала необходимо провести **аудит репозитория**.

Что такое репозиторий проекта

Репозиторий (от англ. *repository*) — это хранилище всей информации о программном проекте: исходного кода, истории изменений, документации и настроек. Управляется с помощью системы контроля версий **Git**. Размещается на платформе **GitHub** (или аналогичных).

Что хранится в репозитории

- Исходный код программы
- Конфигурационные файлы
- Документация (README, Wiki)
- Тесты и тестовые данные
- Ресурсы (изображения, шрифты)
- Зависимости (package.json, .csproj)
- История всех изменений
- Информация о версиях проекта

Git и GitHub — в чём разница?

Git — система контроля версий, установленная локально на компьютере. Отслеживает изменения файлов и позволяет откатываться к предыдущим состояниям.

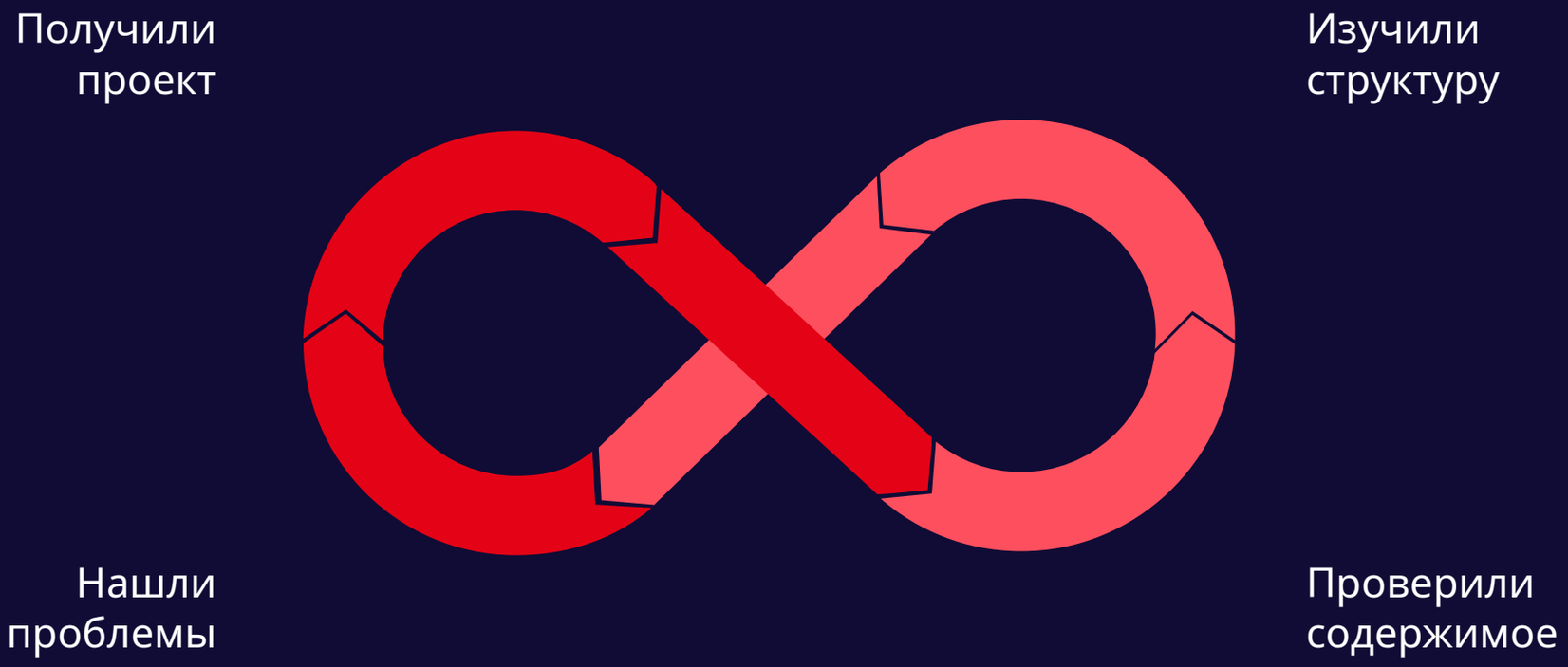
GitHub — веб-платформа для хранения Git-репозиториев в облаке. Добавляет инструменты совместной работы: Issues, Pull Requests, Projects, Actions.

Репозиторий — это **не только папка с кодом**, а полноценная среда для разработки и сопровождения проекта.

Что такое аудит репозитория

> **Аудит репозитория** — систематическая проверка структуры, содержимого, актуальности, документации и организации разработки проекта с целью определения его текущего состояния и выявления проблем.

Простыми словами: аудит — это когда вы **методично изучаете репозиторий**, как врач осматривает пациента, и составляете список того, что работает хорошо, а что требует лечения.



Аудит — это не разовое действие, а **циклический процесс**. После внедрения улучшений репозиторий снова проверяется, чтобы убедиться в корректности изменений.

Зачем проводить аудит

Ориентация в чужом проекте

Аудит позволяет быстро разобраться в структуре незнакомого репозитория — понять, где что находится и как проект организован.

Обнаружение лишнего

Находятся устаревшие файлы, временные копии, папки с непонятными названиями, которые засоряют репозиторий.

Снижение рисков

Выявляются пропущенные зависимости, конфиденциальные данные в коде, отсутствующие инструкции — это уменьшает количество ошибок при работе с проектом.

Улучшение совместной работы

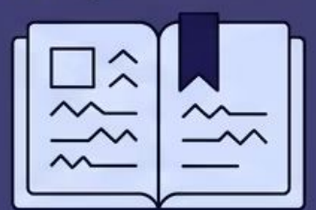
Упорядоченный репозиторий понятен всем участникам команды. Новые разработчики быстрее включаются в работу.

Основные направления оценки репозитория

Аудит охватывает сразу несколько аспектов проекта. Рассмотрим каждое направление:

1. Структура

2. Содержание



3. Документация

4. Актуальность

5. Git-история

6. Ветки

7. Качество файлов

8. Безопасность

9. Удобство разработки

Что такое структура проекта

Структура проекта — это способ организации файлов и папок в репозитории. От структуры зависит, насколько легко ориентироваться в проекте.

Основные понятия

- **Каталог (папка)** — контейнер для файлов и других каталогов
- **Подкаталог** — каталог внутри другого каталога
- **Корневой каталог** — главная папка репозитория
- **src** — исходный код программы
- **tests** — автоматические тесты
- **docs** — документация проекта
- **assets** — ресурсы (изображения, шрифты)
- **config** — конфигурационные файлы

Пример правильной структуры

```
project/
├── src/
│   ├── controllers/
│   ├── models/
│   └── views/
├── tests/
├── docs/
├── assets/
│   └── images/
├── README.md
├── .gitignore
└── LICENSE
```

Каждая папка имеет понятное название и чёткое назначение. Сразу понятно, где искать нужный файл.

Признаки хорошей структуры

Качественная структура репозитория — это не вкусовщина, а набор конкретных признаков, которые можно проверить.

Понятные названия

Имена каталогов и файлов однозначно отражают их содержимое: `src`, `tests`, `docs` — а не `new2`, `folder1`, `copy`.

Логичное разделение

Исходный код, тесты, документация и ресурсы находятся в разных папках — не перемешаны в одном месте.

Нет случайных файлов

В репозитории нет файлов вроде `123.txt`, `my_test.cs`, `kopiya.zip` — только то, что относится к проекту.

Структура соответствует проекту

Веб-проект имеет папки `frontend` и `backend`. Мобильное приложение — `android` и `ios`. Структура отражает реальную архитектуру.

Пример плохой структуры

Рассмотрим реальный пример хаотично организованного репозитория. Попробуйте самостоятельно определить проблемы, прежде чем читать объяснение.

❌ Проблемная структура

```
project/
├── new/
├── new2/
├── test/
├── old/
├── program_final/
├── program_final2/
├── copy/
├── main.cs
├── main_old.cs
├── main_new.cs
└── 123.txt
```

Выявленные проблемы

- **new, new2** — что «новое»? Чем отличаются?
- **old** — зачем хранить в репозитории? Для этого есть история Git
- **program_final / program_final2** — какая из них действительно финальная?
- **copy** — чья копия? Зачем?
- **main_old.cs, main_new.cs** — дублирование файлов кода
- **123.txt** — неизвестный файл с бессмысленным именем
- Нет **src/, tests/, docs/** — стандартных папок проекта

Такая структура — признак того, что разработчики использовали папки вместо веток Git. История изменений хранилась не в системе контроля версий, а в именах файлов.

Сравнение: плохая и хорошая структура

Принцип прост: **имя должно описывать содержимое, а не историю изменений**. История — задача Git, а не имён файлов.

❌ **Плохая организация**	✅ **Хорошая организация**
`new2/`	`src/` (исходный код)
`old/`	`archive/` или удалить (история — в Git)
`program_final2/`	`release/` с тегом версии
`test123/`	`tests/`
`doc1.txt`	`docs/installation.md`
`main_new.cs`	`main.cs` (одна актуальная версия)
`copy/`	Резервные копии не хранятся в репозитории

Понятные имена — это **уважение к другим разработчикам** (и к себе через полгода). Человек должен понять назначение папки, не открывая её.

Что необходимо проверить в репозитории

Анализ содержимого — это проверка каждой категории файлов на наличие, корректность и актуальность.

1

Исходный код

Актуален ли код? Нет ли дублирующихся файлов и версий?

2

README и документация

Есть ли README? Полный ли он? Соответствует ли текущему состоянию?

3

Конфигурация

Актуальны ли настройки? Нет ли паролей и токенов в открытом виде?

4

Зависимости

Есть ли файл зависимостей? Актуальны ли версии пакетов?

5

Тесты

Присутствуют ли тесты? Выделены ли они в отдельную папку?

6

Лишние файлы

Нет ли в репозитории временных файлов, бинарников, результатов сборки?

Лишние файлы в репозитории

Лишние файлы — это файлы, которые не являются частью исходного кода или документации, но случайно попали в репозиторий. Они увеличивают размер репозитория и создают путаницу.

Типичные лишние файлы

- `.exe`, `.dll` — скомпилированные программы
- `.tmp`, `.log` — временные и лог-файлы
- `*.bak` — резервные копии
- `.vs/`, `.idea/` — папки IDE
- `bin/`, `obj/` — папки сборки
- `node_modules/` — пакеты npm
- `.env` — файлы с паролями и токенами
- `Thumbs.db`, `.DS_Store` — системные файлы ОС

Почему они не должны быть в Git?

Скомпилированные файлы генерируются автоматически — каждый разработчик собирает их сам на своей машине.

Файлы IDE содержат личные настройки разработчика, не нужны другим.

.env с паролями — критическая угроза безопасности. Если такой файл попал в публичный репозиторий, пароли нужно немедленно менять.

Пакеты зависимостей восстанавливаются командой `npm install` или `dotnet restore`.

Файл .gitignore

.gitignore — это специальный файл в корне репозитория, который указывает Git, какие файлы и папки **не нужно отслеживать и добавлять в репозиторий**. Правильно настроенный `.gitignore` — признак качественного репозитория.

Пример .gitignore для C# / .NET

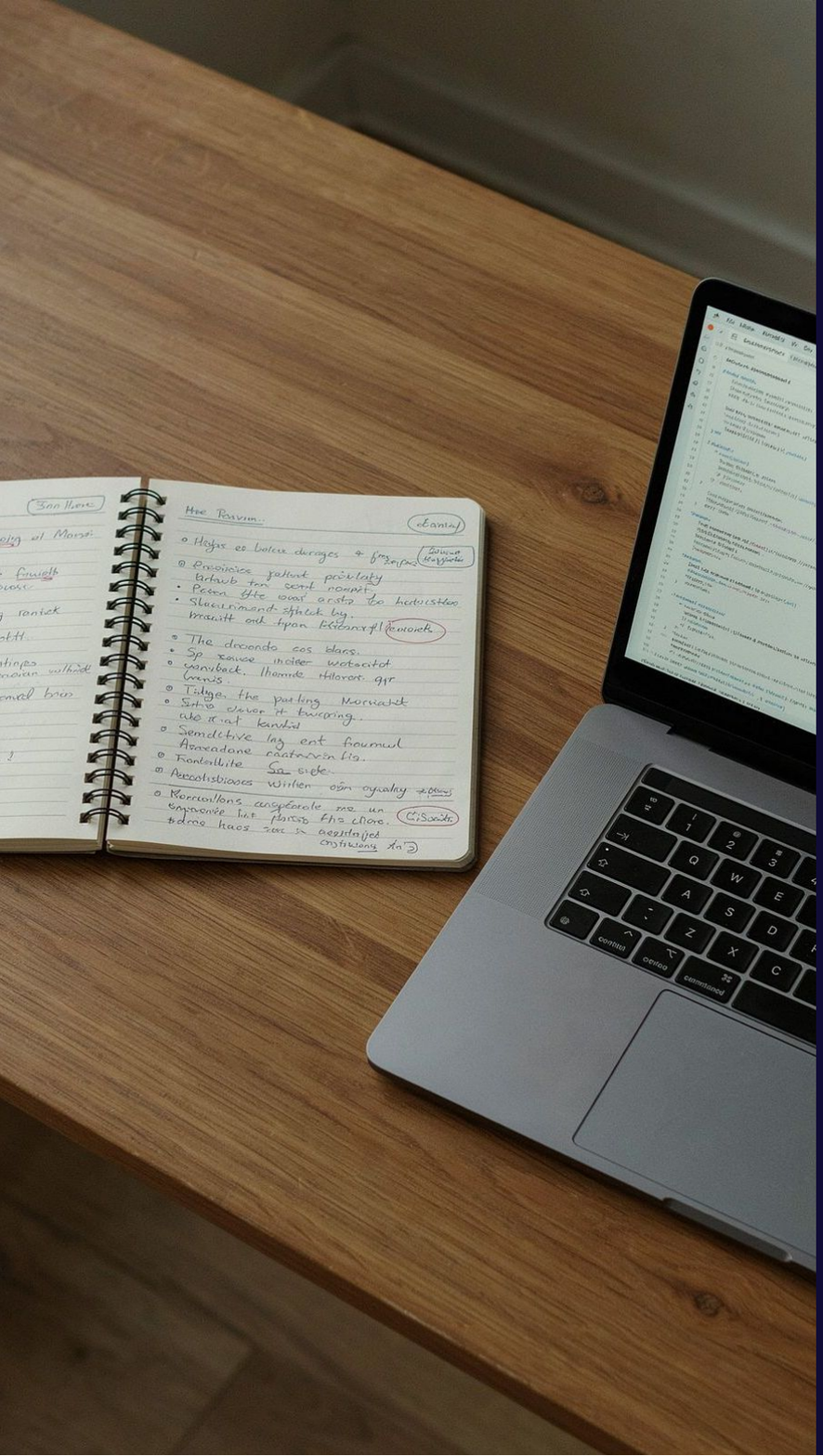
```
# Папки сборки
bin/
obj/

# Настройки Visual Studio
.vs/
*.user

# Переменные окружения
.env

# Логи
*.log

# Системные файлы
.DS_Store
Thumbs.db
```

Как работает .gitignore

Git читает этот файл и **автоматически исключает** указанные файлы и папки из отслеживания. Даже если вы выполните `git add .`, исключённые файлы не попадут в коммит.

Шаблоны для популярных языков и сред можно взять на сайте **gitignore.io** или прямо при создании репозитория на GitHub.

Если в репозитории нет `.gitignore` — это всегда проблема, которую нужно исправить в первую очередь.



Что такое README

README.md — главный документ репозитория. Это первое, что видит человек, открывший проект на GitHub. Хороший README позволяет за 5 минут понять всё необходимое о проекте.

Что это за проект?

Краткое описание: цель, область применения, для кого предназначен.

Как установить?

Пошаговые инструкции: зависимости, команды установки, конфигурация.

Как запустить?

Конкретные команды запуска, возможные параметры, примеры использования.

Какие технологии?

Язык, фреймворк, СУБД, версии — чтобы понять, что нужно установить.

Хорошая структура README

Покажем пример правильно структурированного README.md. Все разделы логичны и отвечают на конкретный вопрос читателя.

Структура **README.md**

```
#
```

Плохой README: пример и анализ

Рассмотрим реальный пример плохого README и разберём, что в нём не так.

❌ Плохой README

```
# Проект

Это наш проект. Для запуска
откройте программу. Всё работает.

Сделано командой.
```

Что отсутствует

- ❌ Нет описания — *что делает программа?*
- ❌ Нет технологий — *какой язык, версия .NET?*
- ❌ Нет инструкции установки — *что нужно установить?*
- ❌ Нет команды запуска — *как именно «открыть»?*
- ❌ Нет требований — *какая ОС, какой runtime?*
- ❌ Нет структуры — *где что находится?*
- ❌ «Всё работает» — не является документацией

Такой README бесполезен. Разработчик, получивший этот проект, вынужден тратить часы на разбирательство вместо работы.

Актуальность документации

Документация, которая не соответствует текущему состоянию проекта, **хуже отсутствия документации** — она вводит разработчика в заблуждение.

Пример устаревшей документации

```
# Требования
- .NET 6.0
- PostgreSQL 13

# Запуск
dotnet run --framework net6.0
```

⬇️ Но в файле проекта:

```
<TargetFramework>net8.0
</TargetFramework>
```

Последствия несоответствия

Разработчик устанавливает .NET 6, запускает команду из README — получает ошибку. Тратит время на поиск проблемы, хотя её нет: просто документация устарела.

Что нужно проверять

- Версии технологий и фреймворков
- Команды установки и запуска
- Описание структуры каталогов
- Скриншоты интерфейса
- Ссылки на внешние ресурсы

Правило: каждый раз, когда меняете код — обновляйте документацию.

Что означает актуальность репозитория

> **Актуальная информация** — это информация, которая точно соответствует текущему состоянию проекта: коду, зависимостям, структуре и поведению программы.

Устаревшая информация в репозитории — одна из самых частых причин проблем при сопровождении проекта.

1 Версия программы

В README написано v1.2, а в коде — v2.0. Какой версии верить?

2 Инструкции запуска

Команда из README возвращает ошибку, потому что параметры изменились.

3 Зависимости

Указана библиотека версии 2.0, а проект использует 4.1 с другим API.

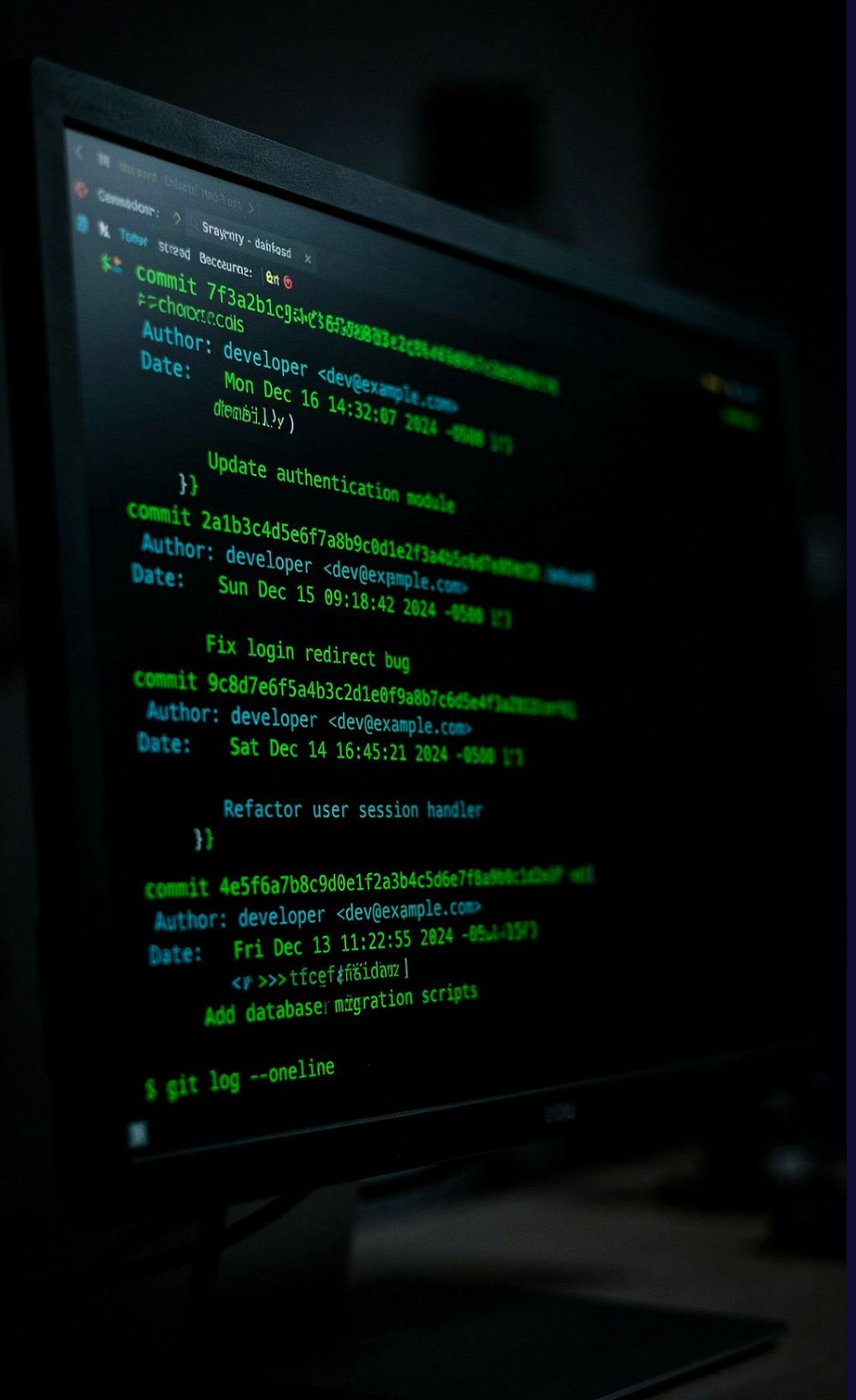
4 Ссылки и изображения

Ссылки ведут на несуществующие страницы. Скриншоты показывают старый интерфейс.

Как искать устаревшую информацию

Поиск устаревшей информации — это пошаговый процесс сравнения документации с реальным состоянием проекта.

Главный метод: не читайте документацию изолированно — **сравнивайте** её с кодом, файлами конфигурации и реальным поведением программы. Если README говорит одно, а проект делает другое — документация устарела.



История изменений Git

Коммит (commit) — это зафиксированное состояние репозитория в определённый момент времени. Каждый коммит содержит:

Что содержит коммит

- **Снимок файлов** — все изменённые файлы на момент коммита
- **Автор** — кто сделал изменение
- **Дата и время** — когда было зафиксировано
- **Сообщение (commit message)** — описание изменения
- **Хэш** — уникальный идентификатор коммита
- **Ссылка на предыдущий коммит** — связь с историей

Зачем нужна история Git при аудите

История коммитов позволяет **восстановить хронологию** проекта: когда появились те или иные функции, кто работал над проектом, насколько активна разработка.

По качеству сообщений коммитов можно судить о **культуре разработки** в команде. Плохие сообщения — признак отсутствия стандартов.

Команда `git log --oneline` показывает историю в компактном виде.

Хорошие и плохие сообщения коммитов

Сообщение коммита — это **объяснение, что и зачем было изменено**. Оно должно быть понятно другому разработчику (или вам самим через полгода) без изучения кода.

❌ Плохие сообщения	✅ Хорошие сообщения
fix	Fix authorization error on login page
123	Add user registration form with validation
update	Update database connection settings for production
работает	Remove unused configuration files from root
final	Release v2.1.0: add PDF export feature
asdfgh	Fix #42: incorrect date format in reports

Хорошее сообщение начинается с **глагола в настоящем времени** (Fix, Add, Update, Remove, Refactor) и отвечает на вопрос «что сделано?». Длина — до 72 символов. Если нужно объяснить «почему» — добавляют описание ниже.

Анализ активности проекта

Раздел **Insights → Contributors** на GitHub показывает статистику активности. Анализ помогает понять реальное состояние проекта.

Дата последних коммитов

Если последний коммит был год назад — проект, вероятно, заброшен или завершён. Это важно понимать перед началом работы.

Частота изменений

Регулярные коммиты говорят об активной разработке. Резкое прекращение — о проблемах или завершении проекта.

Авторы изменений

Сколько людей работало над проектом? Есть ли один «знаток», без которого никто не разберётся?

Какие файлы изменялись

Если один файл менялся сотни раз — возможно, там накопились проблемы. Если папка никогда не трогалась — возможно, она устарела.

Отсутствие новых коммитов само по себе не означает плохого состояния. Проект может быть завершён и стабилен.

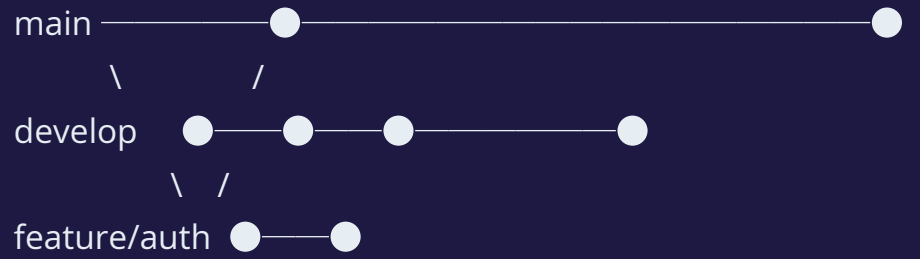
Ветки Git

Ветка (branch) — это независимая линия разработки. Ветки позволяют работать над новой функциональностью, не затрагивая основной код.

Стандартные ветки

- **main** (или master) — основная ветка, стабильный рабочий код
- **develop** — ветка активной разработки
- **feature/название** — разработка новой функции
- **bugfix/название** — исправление конкретной ошибки
- **release/v1.2** — подготовка к выпуску версии
- **hotfix/название** — срочное исправление в production

Схема ветвления



Изменения разрабатываются в отдельных ветках и сливаются в `develop` через Pull Request. Стабильный код попадает в `main` только после проверки.

По количеству и состоянию веток можно судить об организации разработки в команде.

Проблемы веток

Плохая организация веток — один из наиболее частых признаков запущенного репозитория.

Десятки старых веток

Ветки `feature/login`, `feature/login-v2`, `test-new` существуют месяцами. Неясно, какие из них были влиты в основную ветку, а какие заброшены.

Непонятные названия

Ветки `my-branch`, `test`, `fix-2` не дают никакой информации о том, что в них разрабатывалось.

Коммиты напрямую в main

Разработчики коммитят прямо в основную ветку без ревью. Любая ошибка сразу попадает в production-код.

Незавершённые слияния

Ветки с конфликтами слияния, которые никто не разрешил. Код существует в «подвешенном» состоянии.

При аудите нужно просмотреть список веток, определить актуальные и предложить удаление устаревших (после проверки, что они были влиты или не нужны).

Чек–лист аудита репозитория

Используйте эту таблицу как рабочий инструмент при проведении аудита. Пройдитесь по каждому пункту и зафиксируйте результат.

Объект проверки	Что проверяем	Возможная проблема
Структура каталогов	Логичность именования папок	Хаотичное размещение файлов, дубли
README	Полнота и актуальность	Нет инструкции запуска, устарела
Коммиты Git	Качество сообщений	«fix», «123», «работает»
Ветки	Актуальность и организация	Десятки старых неудалённых веток
Исходный код	Актуальность, дубли	Файлы «_old», «_final2», «copy»
.gitignore	Наличие и настройка	В репозиторий попадают bin/, .env
Документация	Соответствие проекту	Устаревшие версии технологий
Безопасность	Наличие токенов, паролей	Файл .env с паролями в репозитории

Классификация найденных проблем

Не все проблемы одинаково важны. Прежде чем исправлять, нужно расставить приоритеты — начинать всегда с критических.

1

🟢 Незначительные

Не мешают работе, но снижают качество. Пример: непоследовательный стиль именования файлов, отсутствие описания в некоторых коммитах.

2

🟡 Средние

Ухудшают качество репозитория. Пример: устаревший README, отсутствие .gitignore, нелогичная структура папок.

3

🟠 Серьёзные

Сильно усложняют сопровождение. Пример: отсутствие документации вообще, десятки хаотичных веток, нет инструкции запуска.

4

🔴 Критические

Проект невозможно запустить или использовать. Пример: пароли в репозитории, отсутствие ключевых файлов, сломанные зависимости.

Нельзя исправлять всё без плана

> Главная ошибка начинающего разработчика: увидел проблему — сразу начал удалять и переименовывать файлы. Через час проект перестаёт запускаться.

Правильный подход требует **системности**. Сначала анализ — потом действия.

❌ Неправильный подход

- Увидел проблему
- Сразу удалил «лишние» файлы
- Переименовал папки
- Проект перестал работать
- Непонятно, что сломалось

✅ Правильный подход

- Провёл полный анализ
- Составил список проблем
- Расставил приоритеты
- Создал резервную точку (ветку)
- Внёс изменения по плану
- Проверил результат

⚠ Никогда не удаляйте файлы без создания резервной ветки Git. Удалённый файл без коммита восстановить невозможно.

План улучшения репозитория

Покажем типовой план внесения улучшений в существующий репозиторий. Порядок шагов важен — не меняйте его произвольно.

01

Создать резервную точку

Выполнить `git branch backup/before-audit` — зафиксировать текущее состояние перед любыми изменениями.

02

Создать рабочую ветку

`git checkout -b improve/repository-structure` — все изменения делаем в отдельной ветке, не трогая main.

03

Удалить лишние файлы

Убрать временные файлы, бинарники, копии. Настроить .gitignore, чтобы они не вернулись.

04

Исправить структуру

Создать правильные папки, переместить файлы, переименовать по стандарту.

05

Обновить документацию

Переписать README, исправить версии, проверить все инструкции.

06

Commit → Push → Pull Request

Зафиксировать изменения, отправить на GitHub, создать PR для проверки.

Улучшение структуры: до и после

Рассмотрим конкретный пример реорганизации репозитория. Обратите внимание: каждый шаг имеет обоснование.

❌ До улучшения

```
project/
├── old/
│   └── main_v1.cs
├── new/
│   └── main_v2.cs
├── 123/
├── files/
├── test123/
├── main.cs
├── main_old.cs
├── program.exe
└── settings.txt
```

✅ После улучшения

```
project/
├── src/
│   └── Main.cs
├── tests/
│   └── MainTests.cs
├── docs/
│   └── architecture.md
├── assets/
├── README.md
├── .gitignore
└── LICENSE
```

Что было сделано

- Удалены папки old/, new/, 123/, files/
- Старые версии файлов — в истории Git
- Бинарный program.exe исключён в .gitignore
- settings.txt → переименован и помещён в docs/

Улучшение содержания репозитория

После исправления структуры необходимо проработать **содержимое** — убедиться, что каждый файл актуален, полон и корректен.

Удалить лишние файлы

Убрать временные файлы, бинарники (`.exe`, `.dll`), старые копии, папки сборки. Настроить `.gitignore`.

Обновить документацию

Переписать устаревшие инструкции, актуализировать версии технологий, исправить нерабочие ссылки.

Добавить отсутствующее

Если нет `README.md` — создать. Нет `LICENSE` — добавить. Нет инструкции запуска — написать.

Привести к единому стилю

Имена файлов должны следовать одному принципу: либо `kebab-case`, либо `PascalCase` — но не вперемешку.

Обновить зависимости

Проверить, актуальны ли версии пакетов в `package.json` или `.csproj`. Устаревшие — обновить или заменить.

Улучшение функциональности GitHub

GitHub предоставляет мощные инструменты для организации работы над проектом. При аудите стоит проверить, используются ли они.

Issues

Задачи, ошибки, предложения. Каждая задача — отдельный Issue с описанием и ответственным.

Pull Requests

Запрос на слияние ветки. Позволяет проводить ревью кода перед добавлением изменений.

Projects

Канбан-доска для управления задачами. Организует работу команды по спринтам.

Actions

Автоматизация: автосборка, тесты при коммите, деплой. CI/CD прямо в GitHub.

Releases

Оформленные релизы с номером версии, описанием изменений и прикреплёнными файлами.

Wiki

Встроенная вики-система для расширенной документации, не помещающейся в README.

GitHub Issues: задачи из аудита

Проблемы, найденные при аудите, следует оформить как **Issues** на GitHub. Так задачи не потеряются и за ними можно следить.

Пример хорошего Issue

Issue #15

Заголовок:
Update README installation instructions

Описание:
Текущая инструкция по установке в README.md
указывает .NET 6.0 и команду:
  dotnet run --framework net6.0

Проект использует .NET 8.0 (см. .csproj).
Инструкция вызывает ошибку у новых участников.

Что нужно сделать:
- Обновить версию .NET в разделе Требования
- Обновить команды установки и запуска
- Проверить другие разделы на актуальность

Метки: documentation, good first issue

Почему Issues важны

Оформление задач как Issues даёт несколько преимуществ:

- **История** — видно, когда проблема была найдена и исправлена
- **Ответственность** — можно назначить исполнителя
- **Обсуждение** — команда может обсудить решение в комментариях
- **Связь с кодом** — Issue можно закрыть через коммит: `Fixes #15`
- **Планирование** — Issues объединяются в Projects для управления

> Шаблоны Issues можно настроить в папке `.github/ISSUE_TEMPLATE/` — тогда все задачи будут оформляться единообразно.

Pull Request: безопасное внесение изменений

Pull Request (PR) — это запрос на слияние изменений из одной ветки в другую. PR позволяет коллегам проверить изменения до их добавления в основную ветку.

Почему PR безопаснее прямого коммита

- Изменения проверяются до попадания в main
- Можно обсудить подход до слияния
- Ошибки обнаруживаются до production
- История изменений структурирована

PR при аудите репозитория

Все улучшения, найденные при аудите, вносятся **не напрямую в main**, а через отдельные ветки и Pull Requests. Например: `improve/update-readme`, `improve/add-gitignore`, `improve/restructure-folders`.

Проверка результата после улучшений

После внесения всех изменений необходимо провести **повторную проверку** — убедиться, что улучшения не нарушили работу проекта.

1 Проверить запуск проекта

Следовать инструкции из обновлённого README с чистой машины или чистой папки. Если не запускается — инструкция всё ещё неверна.

2 Проверить основные функции

Убедиться, что переименование файлов и папок не нарушило пути в коде и конфигурации.

3 Проверить полноту репозитория

Не были ли случайно удалены нужные файлы? Все ли зависимости на месте? Корректна ли структура?

4 Проверить историю и ветки

Понятны ли сообщения новых коммитов? Удалены ли устаревшие ветки? Влит ли PR корректно?

Формула качественного репозитория

Подведём итог. Качественный репозиторий — это не случайность, а результат осознанной работы по каждому направлению.

Понятная структура

Логичные имена папок, разделение кода, тестов и документации

Актуальный код

Только одна актуальная версия каждого файла, без «_old» и «_final2»

Актуальная документация

README соответствует коду, инструкции работают, версии актуальны

Чистая Git-история

Понятные сообщения коммитов, актуальные ветки, использование PR

Правильные настройки

Настроен .gitignore, нет конфиденциальных данных, правильный .editorconfig

= Репозиторий, который удобно сопровождать

ПРАКТИЧЕСКОЕ ЗАДАНИЕ

Кейс: аудит репозитория

Вам передан репозиторий проекта. Проведите его первичный аудит на основе следующего описания:

Описание репозитория

- README последний раз обновлялся **2 года назад**
- В корне репозитория присутствует папка **bin/**
- Есть файл **.env** с настройками подключения к БД
- Ветки называются **new**, **new2**, **test123**
- Последний коммит называется **«final»**
- В корне проекта находятся **40 файлов** без папок
- Отсутствует **инструкция запуска**
- Тесты лежат **вместе с исходным кодом**

Задание

Определите минимум **7 проблем** репозитория, используя знания занятия. Для каждой проблемы:

1. Назовите проблему
2. Объясните, почему это проблема
3. Присвойте приоритет (критическая / серьёзная / средняя / незначительная)
4. Предложите конкретный способ исправления

Запишите результаты в виде таблицы в тетради. Время: 10 минут.

РАЗБОР КЕЙСА

Возможный разбор задания

Проблема	Почему проблема	Приоритет	Исправление
Файл `.env` в репозитории	Пароли доступны всем, кто видит репозиторий	**Критическая**	Удалить, сменить пароли, добавить .env в .gitignore
Папка `bin/` в репозитории	Бинарники не должны храниться в Git	Серьёзная	Удалить, добавить bin/ в .gitignore
README устарел на 2 года	Инструкции, вероятно, не работают	Серьёзная	Обновить версии, проверить команды запуска
Нет инструкции запуска	Новый разработчик не может запустить проект	Серьёзная	Добавить раздел «Запуск» в README
40 файлов в корне без папок	Невозможно ориентироваться в проекте	Средняя	Создать src/, tests/, docs/, распределить файлы
Ветки new, new2, test123	Непонятное назначение, засоряют репозиторий	Средняя	Проверить и удалить ненужные ветки
Коммит «final»	Не понять, что было изменено	Незначительная	Использовать осмысленные сообщения впредь
Тесты вместе с кодом	Нарушает стандарт организации проекта	Незначительная	Переместить тесты в папку tests/

Итог занятия: что вы теперь умеете

После изучения темы вы освоили полный цикл работы с репозиторием программного проекта — от первичного осмотра до внедрения улучшений.

Анализ

Проводить аудит структуры, содержимого, документации и Git-истории репозитория

→ **Аудит репозитория**

Систематическая проверка структуры, содержимого, документации и организации проекта

Оценка

Определять актуальность информации, классифицировать проблемы по приоритету

→ **Классификация проблем**

Критические → серьёзные → средние → незначительные. Исправлять в этом порядке

Улучшение

Составлять план изменений, безопасно внедрять их через ветки и PR, проверять результат

→ **Безопасное внедрение**

Ветка → изменения → коммит → PR → проверка → слияние. Никогда напрямую в main

Тест по теме занятия

Ответьте на вопросы самостоятельно. Правильные ответы будут разобраны после.

1. Что такое аудит репозитория?

А) Удаление ненужных файлов из проекта
Б) Систематическая проверка структуры, содержимого и организации проекта
В) Создание резервной копии репозитория

2. Что показывает структура проекта?

А) Только имена авторов коммитов
Б) Организацию файлов и каталогов в репозитории
В) Количество строк исходного кода

3. Для чего используется README?

А) Для хранения паролей и настроек
Б) Для описания проекта, инструкций установки и запуска
В) Для хранения результатов компиляции

4. Для чего нужен файл .gitignore?

А) Для хранения конфиденциальных данных проекта
Б) Для исключения ненужных файлов из отслеживания Git
В) Для документирования истории изменений

5. Что означает актуальность документации?

А) Документ создан на текущей неделе
Б) Документация соответствует текущему состоянию проекта
В) Документ содержит более 10 страниц

6. Что содержит коммит Git?

А) Только сообщение об изменении
Б) Снимок файлов, автора, дату и сообщение
В) Архив всего репозитория

7. Как выглядит качественное сообщение коммита?

А) «fix», «update», «работает»
Б) «Add user registration form with validation»
В) Длинное описание на несколько абзацев

8. Для чего создаётся отдельная ветка Git?

А) Для резервного копирования всего репозитория
Б) Для независимой разработки без влияния на основную ветку
В) Для хранения результатов тестирования

9. Что такое Pull Request?

А) Команда для скачивания репозитория
Б) Запрос на слияние изменений из одной ветки в другую
В) Инструмент для создания резервных копий

10. Что необходимо сделать после внедрения улучшений?

А) Удалить резервную ветку и закрыть проект
Б) Провести повторную проверку: запуск, функции, документация
В) Сразу начать следующий этап разработки